Slack Stealing

Bin Zaprunnizam   
Muhammad Amirul Hakimi Hochschule Hamm Lippstadt Lippstadt, Germany [muhammad-amirul-hakimi.bin-zaprunnizam@stud.hshl.de](mailto:muhammad-amirul-hakimi.bin-zaprunnizam@stud.hshl.de)

*Abstract*— Every year, the pace of technological change accelerates, and new technologies emerge at a faster rate than ever before. The expense and risk of attaining improved production quality are steadily rising. The introduction of embedded technology not only lowered cost and risk, but also increased quality. The real-time operating system (RTOS) allows the user to make the most of his or her 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

# Introduction

Over the last decade, 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 case: (1) static or fixed priority algorithms, such as rate monotonic and deadline monotonic algorithms [2, 31], and (2) dynamic priority algorithms, such as the earliest deadline algorithm [2]. Both 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. While real-time applications typically consist of a set of hard deadline periodic tasks, hard deadline aperiodic tasks can emerge from a variety of sources, 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 [6]. Unlike the challenge of scheduling only periodic tasks, the joint scheduling problem is challenging because it is frequently the case that no single method can plan all tasks. When all of the task timing requirements can't 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. Because 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 on-line 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.

The slack-stealing technique is based on Joseph and Pandya's [9] accurate schedulability study for fixed priority algorithms used to schedule periodic task sets, which was afterwards elaborated on by a few other writers. Davis, Tindell, and Burns [lo] extended [7] to a wider set of scheduling issues and proposed a method for calculating the slack quantities needed by the algorithm. Both [lo] and [6] proposed approximations to the full slack theft approach that are less computationally expensive while still delivering good results. Tia, Liu, and Shankar [ll] recently proposed a method for soft aperiodic processes that improves on the slack stealing strategy. [7] defines the slack theft method as a greedy algorithm in which all available slack must be used to service aperiodic right away. When the class of algorithms is expanded to include non-greedy algorithms, [ll] shows that this greedy technique is not optimal. They show that if certain lower priority jobs are completed before the available slack is used, the total slack available during specified intervals can be more than that accessible to the slack theft algorithm. Soft aperiodic jobs should benefit from more slack, allowing for faster response times. Furthermore, the non-greedy approaches of [ll] may provide for the guarantee of a hard aperiodic, whereas the slack stealing strategy presented in this study may not.

# BackGround scheduling

Many operating systems that support dynamic task activation allow the ongoing work to be interrupted at any time, allowing a more critical activity to take over the processor without having to wait in the ready queue. The ongoing job is halted and inserted into the ready queue in this instance, while the CPU is assigned to the most critical ready task that has just arrived. Preemption is the process of suspending a running job and placing it in the ready queue. The concepts provided above are schematically illustrated in Figure 2.1. Preemption is significant in dynamic real time systems for three reasons [SZ92]:

* Tasks performing exception handling may need to preempt existing tasks so that responses to exceptions may be issued in a timely fashion.
* When tasks have different levels of criticality (expressing task importance), preemption permits executing the most critical tasks as soon as they arrive.
* Preemptive scheduling typically allows higher efficiency, in the sense that it allows executing a real-time task sets with higher processor utilization.

When there are no periodic instances available to execute, the easiest way to handle a group of soft aperiodic tasks in the existence of periodic tasks is to schedule them in the background. The main disadvantage of this approach is that for large periodic loads, the response time of aperiodic requests might be too long for 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.

* Hard: A real-time task is said to be hard if missing its deadline may cause catastrophic consequences on the system under control.
* Firm: A real-time task is said to be firm if missing its deadline does not cause any damage to the system, but the output has no value.
* Soft: A real-time task is said to be soft if missing its deadline has still some utility for the system, although causing a performance degradation.

Chart

Description automatically generated

Figure 5.1 shows an example in which RM schedules two periodic activities while two aperiodic tasks run in the background. Because background scheduling has no effect on the execution of periodic tasks, the guaranteed test remains unchanged in the presence of aperiodic requests.

Diagram

Description automatically generated

The simplicity of background scheduling is its main advantage. Two queues are required to implement the scheduling mechanism, as illustrated in Figure 5.2: one for periodic tasks (with a higher priority) and one for aperiodic requests (with a lower priority). The two queueing techniques are independent of one another and can be implemented using separate algorithms, such as RM for periodic workloads and FCFS for aperiodic requests. Only when the periodic queue is empty are tasks accepted from the aperiodic queue. 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 is

* *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*

The DS algorithm creates a periodic task (typically with a high priority) for servicing aperiodic queries as the Polling Server. Unlike polling, however, DS keeps its capacity if no requests are queued when the server is started.

* *Priority Server*

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 does, however, differ from DS in the way capacity is preserved. PE, unlike DS, 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 (SS) algorithm, which improves the average response time of aperiodic jobs without lowering the utilization bound of the periodic task set. The SS 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. SS, on the other hand, varies from DS in how it refreshes its capacity. SS 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*

Another aperiodic service methodology is the Slack Stealing algorithm, which promises significant response time advantages over prior service methods (PE, DS, and SS). 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 aperiodic 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.

# Model slack staling algorithm

The primary notion underlying slack stealing is that there is usually little benefit in completing periodic duties early. As a result, when an aperiodic request arrives, the Slack Stealer takes all available slack from periodic jobs and utilizes it to fulfill aperiodic requests as quickly as feasible. If no aperiodic requests are outstanding, RM will generally plan periodic work. We recall that if c*i*(*t*) is the remaining computation time at time *t*, the slack of a task τ*i* is



A picture containing timeline

Description automatically generated

Figure 5.20 shows the behavior of the Slack Stealer on a set of two periodic tasks, τ1 and τ2, with periods T1 = 4, *T2 = 5* and execution times *C1 = 1, C2 = 2*. Figure 5.20a shows the schedule produced by RM when no aperiodic tasks are processed, whereas Figure 5.20b illustrates the case in which an aperiodic request of three units arrives at time *t = 8* and receives immediate service. In this case, a slack of three units is obtained by delaying the third instance of τ 1 and τ2. For example, since U1 = 1/4 and U2 = 2/5, the P factor for the task set is P = 7/4; hence, the maximum server utilization, according to Equation (5.24) is



A picture containing chart

Description automatically generated

## Breadth First Search Algorithm

This means that, even with *Cs = 1*, the shortest server period that can be set with this utilization factor is *Ts = [Cs/Us] = 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 schedule an aperiodic request *Ja(ra, Ca)* using the Slack Stealing technique, we must first find the earliest time t at which at least *Ca* units of slack are available. The computation of the slack is carried out through the use of a slack function *A(s, t),* which returns the maximum amount of computation time that can be assigned to aperiodic requests in the interval *[s, t]* without compromising the schedulability of periodic tasks.

Figure 5.21 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 5.22 depicts how the slack function *A(s, t)* changes for the same periodic task set at time *s = 6*.

A picture containing diagram

Description automatically generated

Fig. 9. Map for Task 1

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.

### 

# Optimality in hard real time

### 

Lehoezky and Ramos-Thuel (8] proved that the slack stealing algorithm can be the basis for constructing some strongly optimal scheduling algorithms for jointly scheduling hard periodic and soft aperiodic. Ramos-Thuel and Lehoczky (14] showed that, unfortunately, no such strong optimality is possible for the hard aperiodic case unless the sets of periodic tasks under consideration permit some algorithm to meet successfully all of the deadlines in each of the aperiodic task sets. Otherwise, any algorithm will be faced with a situation in which it will not be able to do all offered work. Such a situation requires that a choice be made as to which tasks to process. This choice implies that no scheduling algorithm will be able to dominate all other algorithms on the task sets under consideration, and this, in turn, prevents any algorithm from being strongly optimal. The conventional view of optimality for hard aperiodic scheduling is that a hard aperiodic scheduling algorithm is optimal if it is guaranteed to schedule any periodic task set which can be feasibly scheduled by some other scheduling algorithm; that is, if a hard aperiodic task set is feasible by a hard aperiodic scheduling algorithm, it is also feasible if scheduled by the optimal algorithm. This defines a strong sense of optimality which is very narrowly applicable, because it is limited to the class of hard aperiodic task sets which are schedulable. There are many hard aperiodic task sets which fall outside this very restricted class (e.g., fault recovery

operations), and we are interested in hard aperiodic scheduling problems in which it may be impossible feasibly to schedule the aperiodic tasks by any algorithm. We next provide definitions of strong optimality for the hard periodic case. In addition to the given set of periodic tasks, consider any sequence of aperiodic jobs J = {Jk, k ≥ 1). Any scheduling algorithm, call it 2, will result in a subset of the aperiodic tasks, C\* (J), whose deadlines are met. Definition 1': A scheduling algorithm X dominates a scheduling algorithm Y on an aperiodic task set J if and only if CV (J) C C\* (J).

As discussed above and by Ramos-Thuel and Lehoczky [14], no algorithm will be strongly optimal unless the class of algorithms, C, is very severely restricted or the collection of aperiodic task sets, P, is very small, and includes only task sets which can be completely scheduled by a single algorithm. As soon as an aperiodic task set is included which cannot be completely scheduled by any algorithm, strong

optimality is lost. As long as no algorithm can schedule every periodic and aperiodic task that arrives, there can be no strong optimality. Given that strong optimality is rarely achievable, it is natural to weaken the concept in the way that was done in the soft aperiodic scheduling case, that is by computing a univariate performance measure, M, and finding an algorithm that minimizes this measure over all periodic task sets, as discussed in Definition 4. For example, we might consider performance measures such as the long-run fraction of periodic tasks that are scheduled unsuccessfully, the amount of aperiodic work for which the deadlines are not met relative to the amount of work offered, or some weighted average of the uncompleted work where the weights are chosen to reflect

the importance of each task. Much research remains to be done to find optimal scheduling algorithms for useful classes of algorithms with respect to interesting performance measures.

# Uppaal Implimentation

# Conclusion

The slack theft algorithm, a flexible approach for concurrently servicing hard deadline periodic jobs with soft and/or hard deadline periodic tasks, was reviewed in this chapter. For always determining both the slack available and the worst-case completion time of each instantiation of each periodic job, a new technique is introduced, which is a modest variant of the approach introduced by Davis, Tindell, and Burns (4). Algorithms for leveraging the slack stealing method to service soft or hard aperiodic workloads are provided. There is a huge discussion of optimality. We show that for the soft periodic scheduling problem, some strong optimality results are possible, whereas for the hard periodic scheduling problem, no strong optimality results are possible unless the periodic task sets being considered are severely restricted so that every task can be scheduled by some algorithm. Extensions are mentioned, emphasizing the slack stealing method's versatility and generality.

# Refrences

1. Dominey-Howes D. (2007) Geological and historical records of tsunami in Australia. Marine Geology 239: 99-123 doi:10.1016/j.margeo.2007.01.010
2. S. Schrödl and S. Edelkamp, Heuristic Search: Theory and Applications. Morgan Kaufmann Publishers, 2011
3. I. Ahmad, 40 Algorithms Every Programmer Should Know. [S.l.]: Packt Publishing, 2020.
4. E. Magid, T. Tsubouchi, E. Koyanagi and T. Yoshida, "Building a Search Tree for a Pilot System of a Rescue Search Robot in a Discretized Random Step Environment", *Journal of Robotics and Mechatronics*, vol. 23, no. 4, pp. 567-581, 2011. Available: 10.20965/jrm.2011.p0567.