

Sporadic Server

Shihab Ud Doula
Dept. Of Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
Email: shihab-ud.doula@stud.hshl.de

Abstract— *The sporadic server algorithm is an extension of the rate monotonic scheduling algorithm. Sporadic servers are tasks created to provide limited and usually high- priority service for other tasks, especially aperiodic tasks. The Sporadic Server (SS) overcomes the major limitations of other Resource Reservation Fixed Priority based techniques, but it also presents some drawbacks, mainly related to an increased scheduling overhead and a not so efficient behavior during overrun situations. In this paper we introduce and prove the effectiveness of an improved SS with reduced overhead and fairer handling of server overrun situations. We also talk about Deferrable servers as an extension of sporadic servers that provide additional flexibility in handling sporadic tasks [1].*

Keywords—Aperiodic test, sporadic server, algorithm, overruns, slack stealing

I. INTRODUCTION

The Sporadic Server algorithm is a technique proposed by Sprunt, Sha, and Lehoczky that allows for improving the average response time of aperiodic tasks without degrading the load limit of the periodic task set. The sporadic server algorithm creates a high-priority task to handle aperiodic requests and, like DS, maintains the server capacity at its high-priority level until an aperiodic request occurs. However, a sporadic server differs from a deferrable server in the way it replenishes its capacity. While deferrable servers and priority swaps regularly replenish their capacity to full at the beginning of each server period, the sporadic server replenishes its capacity only after it has been consumed by aperiodic task execution. Sporadic servers can be used to guarantee deadlines for aperiodic tasks with hard deadlines and to achieve significant improvements in average response times for aperiodic tasks with soft deadlines over query techniques. Sporadic servers also provide a mechanism to implement the period transformation technique, which can guarantee that a critical set of periodic tasks will always meet their deadlines during a temporary overload. Sporadic servers can also support error detection and containment in a real-time system by limiting the maximum execution time of a task and detecting attempts to exceed a certain limit. [2]

In the context of real-time systems, there are several types of sporadic servers that can be used to handle sporadic tasks efficiently. Here are a few commonly used types:

- Fixed-Priority Sporadic Server: In this type of sporadic server, each sporadic task is assigned a

fixed priority. The server schedules tasks based on their priorities, and higher priority tasks are given precedence over lower priority tasks. The fixed-priority sporadic server ensures that tasks with higher priorities meet their deadlines while providing best-effort scheduling for lower priority tasks.

- Earliest Deadline First (EDF) Sporadic Server: EDF is a popular scheduling algorithm used for sporadic tasks. In an EDF sporadic server, each task is assigned a deadline, and the server schedules tasks based on their earliest deadline. The task with the closest deadline is given the highest priority. EDF scheduling ensures that tasks with imminent deadlines are processed first, minimizing the possibility of missing deadlines.
- Slack Stealing Sporadic Server: Slack stealing is a technique used to improve the efficiency of sporadic servers. In a slack stealing sporadic server, when a task finishes before its deadline, the remaining slack time (the difference between the task's completion time and its deadline) can be "stolen" by other tasks with higher priority. This allows for better resource utilization and can improve the overall system performance.
- Deferrable Server: A deferrable server is a type of sporadic server that supports deferred processing. In this approach, when a sporadic task arrives, it can be deferred or delayed if the server is currently busy processing another task. The deferrable server provides a guarantee that all deferred tasks will eventually be processed within their deadlines. This flexibility allows for efficient utilization of server resources and can handle sporadic tasks with varying execution times.

These are just a few examples of sporadic server types. The choice of a particular type depends on the specific requirements and characteristics of the real-time system being designed. Different types of sporadic servers have different scheduling algorithms and resource allocation strategies to ensure timely processing of sporadic tasks while maximizing resource utilization.

II. TASK DESIGN OF SPORADIC SERVER [3]

A *sporadic server* is a periodic task created specifically for the purpose of executing sporadic tasks. A sporadic server, denoted by Φ_s is given by (P_s, E_s, θ, ρ) , where.

- (P_s, E_s) defines a periodic task with a period of p_s and an execution time of e_s , where e_s is also called the server's execution budget.
- $\theta = \{Si = (r_i, e_i, d_i) \mid 1 \leq i \leq k\}$ is a set to be populated with sporadic jobs released at run time; and
- ρ is a set of rules for regulating the operations of Φ_s .

Since a sporadic server is treated as a periodic task, it is straightforward to employ the RMA principle to schedule the sporadic server together with the other periodic tasks that are pertinent to the domain problem at hand.

III. REPLENISHMENT OF SPORADIC SERVER [3] [1]

Replenishment is a concept used in sporadic server scheduling algorithms to ensure that sporadic tasks are allocated sufficient processing time for their execution. It involves replenishing the budget or allocating additional resources to sporadic tasks at specific intervals to meet their timing requirements. The replenishment mechanism ensures that the sporadic server can handle incoming sporadic tasks within their specified deadlines. Here's how replenishment works in a sporadic server:

- **Initial Budget Allocation:** When a sporadic task is released, the sporadic server assigns an initial budget of processing time to that task. This budget represents the maximum amount of processing time that the task can utilize during its execution. The initial budget is typically based on the worst-case execution time of the task.
- **Budget Consumption:** As the sporadic task executes, it consumes its allocated budget of processing time. The server keeps track of the remaining budget for each task, which is updated as the task progresses. The task execution continues until either the entire budget is consumed, or the task completes its execution.
- **Replenishment Interval:** A replenishment interval is defined for the sporadic tasks. It specifies the time period at which the sporadic server evaluates the remaining budget of each task and replenishes it if necessary. The replenishment interval can be predetermined or dynamically determined based on

the characteristics of the tasks and the system requirements.

- **Budget Replenishment:** At each replenishment interval, the sporadic server checks the remaining budget of each sporadic task. If the task has consumed its entire budget or if its remaining budget falls below a certain threshold, the server replenishes the task's budget by allocating additional processing time. The amount of replenishment can be based on factors such as the task's execution progress, its importance, or other scheduling policies.
- **Deadline Guarantees:** The replenishment mechanism ensures that sporadic tasks are allocated sufficient budget or processing time to meet their deadlines. By replenishing the budget at specific intervals, the server avoids situations where tasks run out of budget and risk missing their deadlines.
- **Resource Constraints:** Replenishment takes into account any resource constraints, such as processor availability or system capacity. The server allocates additional processing time to sporadic tasks while considering the availability of resources to avoid overloading the system or causing resource contention.

Replenishment is a key aspect of sporadic server scheduling algorithms, such as the Sporadic Server Algorithm (SSA). It helps ensure that sporadic tasks receive the necessary resources and timing guarantees to meet their real-time requirements, even in situations where their execution times may vary or be uncertain.

Given is a code for replenishment for sporadic server algorithm:

```

next_replenishment = last_sporadic.start +
sporadic_template.period
    if first_lower_priority_occurence >
last_sporadic.start:
next_replenishment = first_lower_priority_occurence +
sporadic_template.period

```

next_replenishment is the time point (in cycles) for next replenishment.

last_sporadic is the last executed sporadic task instance. **sporadic_template** is common task template for the sporadic server filled with your initial inputs.

first_lower_priority_occurence is the time (in cycles) a periodic task is preferred instead of a sporadic execution due to high priority.

Sporadic Server

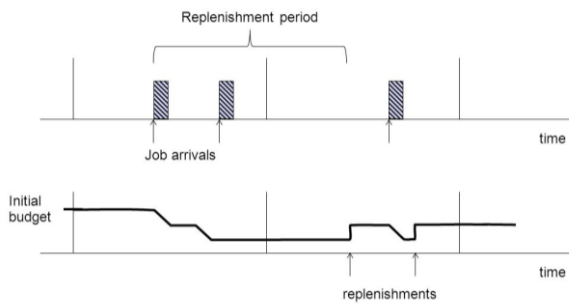


Figure 4: Sporadic Server Scheduling

IV. IMPLEMENTATION IN C++

An implementation in C++ code for sporadic server that shows the behavior of sporadic server in handling sporadic tasks was done in Visual Code Studio. The code is shown below:

```

1  #include <iostream>
2  #include <queue>
3  #include <cstdlib>
4  #include <ctime>
5
6  using namespace std;
7
8  // Structure to represent a sporadic task
9  struct Task {
10     int executionTime; // Worst-case execution time (WCET)
11     int interArrivalTime; // Minimum inter-arrival time
12     int deadline; // Relative deadline
13 };

```

In this part, we include the necessary header files (**iostream** for input/output, **queue** for the task queue, **cstdlib** for random number generation, and **ctime** for seeding the random number generator). We also define a structure **Task** to represent a sporadic task. The structure contains three properties: **executionTime** (worst-case execution time), **interArrivalTime** (minimum inter-arrival time), and **deadline** (relative deadline) of a task. This is an initial part of the code. The full code is provided in a separate file in GITHUB. The output

```

30
31 // Simulate sporadic server behavior
32 while (currentTime <= simulationTime) {
33     if (!taskQueue.empty()) {
34         Task currentTask = taskQueue.front();
35         taskQueue.pop();
36
37         // Check if task can be executed within its allocated budget
38         if (currentTask.executionTime <= currentTask.deadline - currentTime) {
39             cout << "Task executed. Execution Time: " << currentTask.executionTime
40                 << ", Deadline: " << currentTask.deadline << ", Result: Success" << endl;
41         }
42         else {
43             cout << "Task executed. Execution Time: " << currentTask.executionTime
44                 << ", Deadline: " << currentTask.deadline << ", Result: Missed Deadline" << endl;
45         }
46
47         currentTime += currentTask.interArrivalTime;
48     }
49     else {
50         // No tasks in the queue, advance the time
51         currentTime++;
52     }
53 }
54

```

Inside the **while** loop, we simulate the behavior of the sporadic server. The loop continues until the **currentTime** exceeds the **simulationTime**.

Inside the loop, we check if the **taskQueue** is not empty. If there are tasks in the queue, we retrieve the next task using **taskQueue.front()** and remove it from the queue using **taskQueue.pop()**.

We then check if the current task's execution time is less than or equal to the time remaining until its deadline (**currentTask.deadline - currentTime**). If it is, we consider the task executed within its allocated budget, and we print a success message indicating the execution time, deadline, and the result.

If the task's execution time exceeds the remaining time until the deadline, we print a message indicating that the task missed its deadline.

After handling the current task, we update the **currentTime** by adding the current task's inter-arrival time. This simulates the arrival of the next task in the future.

If there are no tasks in the queue, we simply advance the **currentTime** by one unit.

The full code is provided in a separate file in GITHUB. The output is given below.

```

Microsoft Visual Studio Debug
Task executed. Execution Time: 7, Deadline: 15, Result: Success
Task executed. Execution Time: 7, Deadline: 16, Result: Success
Task executed. Execution Time: 6, Deadline: 14, Result: Success
Task executed. Execution Time: 1, Deadline: 19, Result: Success
Task executed. Execution Time: 5, Deadline: 14, Result: Missed Deadline

C:\Users\shiha\source\repos\ss2\x64\Debug\ss2.exe (process 22580) exited with code 0.
Press any key to close this window . . .

```

The provided output corresponds to the execution of sporadic tasks using the sporadic server simulation code. Let's analyze each line of the output:

1.Task executed. Execution Time: 7, Deadline: 15, Result: Success

This line indicates that a task with an execution time of 7 units and a deadline of 15 units was successfully executed. The task was completed within its allotted execution time and met its deadline.

2. Task executed. Execution Time: 7, Deadline: 16, Result: Success

Like the previous line, this indicates the execution of another task with an execution time of 7 units and a deadline of 16 units. The task was also completed within its execution time and met its deadline.

3. Task executed. Execution Time: 6, Deadline: 14, Result: Success

This line represents the execution of a task with an execution time of 6 units and a deadline of 14 units. The task was successfully executed and completed within its allotted time, meeting its deadline.

4. Task executed. Execution Time: 1, Deadline: 19, Result: Success

This line corresponds to a task with an execution time of 1 unit and a deadline of 19 units. The task is executed successfully and completed within its execution time, meeting its deadline.

5. Task executed. Execution Time: 5, Deadline: 14, Result: Missed Deadline

The final line indicates the execution of a task with an execution time of 5 units and a deadline of 14 units. However, this task missed its deadline, indicating that it was not completed within the allotted execution time. The execution time exceeded the available time before the deadline, resulting in a missed deadline.

Overall, the output demonstrates the execution of sporadic tasks using the sporadic server simulation code. The tasks are assigned execution times and deadlines, and the code checks whether they are completed within their execution times and meet their deadlines. The "Success" or "Missed Deadline" result indicates whether each task was executed successfully or missed its deadline.

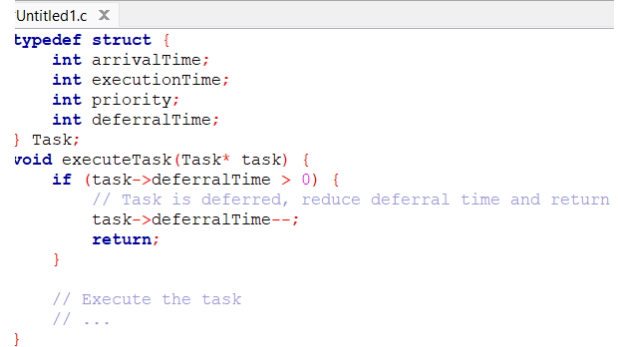
V. DEFERRABLE SERVER [4]

Deferrable servers and sporadic servers serve different purposes and can be used in different scenarios. While both are related to managing real-time tasks, they address different aspects of task scheduling and timing constraints. Sporadic servers are designed to handle sporadic or periodic tasks with deterministic arrival patterns and fixed execution times. They allocate a fixed amount of processing time to each task to ensure that it is completed within its specified deadline. Sporadic servers are useful when tasks have strict timing requirements and need to be serviced in a timely manner.

On the other hand, deferrable servers provide flexibility in handling tasks with uncertain arrival times or varying degrees of urgency. They allow tasks to defer their execution for a certain period if they are not critical or time sensitive. Deferrable servers are beneficial in scenarios where tasks can be postponed without violating their deadlines or system constraints.

Such as for Handling Uncertain Arrival Times: Sporadic servers assume known and deterministic arrival patterns for tasks. However, in certain scenarios, the exact arrival times of tasks may be uncertain or unpredictable. Deferrable servers can handle such situations by allowing tasks to defer their execution until their arrival or event occurs.

A c code function shows the structure to represent task with arrival time, execution time, priority, and deferral time where after a function is created for each task to execute them.



```
Untitled1.c X
typedef struct {
    int arrivalTime;
    int executionTime;
    int priority;
    int deferralTime;
} Task;

void executeTask(Task* task) {
    if (task->deferralTime > 0) {
        // Task is deferred, reduce deferral time and return
        task->deferralTime--;
        return;
    }

    // Execute the task
    // ...
}
```

Figure: 5

VI. FUTURE IMPLEMENTATION [3]

Although deferrable server acts as extension of sporadic server as it can handle uncertain time constraints, the future implementations of deferrable servers could involve advancements and enhancements to their functionality, adaptability, and performance. Some noteworthy mentions are:

- **Advanced Deferral Policies:** Future deferrable server implementations can incorporate more sophisticated deferral policies. These policies can consider factors such as task characteristics, system load, resource availability, and task dependencies to make intelligent decisions on task deferral. Machine learning techniques or predictive algorithms could be employed to dynamically adjust the deferral times based on historical data or system behavior.
- **Dynamic Deferral Time Calculations:** Rather than using fixed deferral times, future implementations could allow for dynamic calculation of deferral times based on the changing system conditions. For example, deferral times could be adjusted based on the current workload, available resources, or priority levels of tasks. This adaptability would enable better resource allocation and responsiveness.

- Adaptive Task Prioritization: Future deferrable servers could incorporate adaptive task prioritization mechanisms. The server can dynamically adjust task priorities based on their urgency, importance, or other criteria. This adaptive prioritization can ensure that critical or time-sensitive tasks are executed with minimal delay while still providing flexibility for less critical tasks.
- Energy-Efficient Deferral Strategies: With the increasing focus on energy efficiency, future deferrable server implementations can explore strategies to optimize energy consumption. This could involve intelligent decisions on task deferral based on the energy state of the system or the specific energy requirements of tasks. Energy-aware scheduling policies can be developed to minimize energy usage without compromising task deadlines. [1]

VII. SUMMARY AND CONCLUSION [5] [4]

An approach for sporadic server in aperiodic task handling has been presented in the paper. It is based on which tends to be more effective according to our research. Algorithms and charts with implementation results are described to understand the replenishment of sporadic server. As it requires deterministic behavior only, an extension of it called deferrable server is described and explained for real time system. The combination of the proposed mechanisms has multiple positive outcomes: from a fairer behavior in presence of overruns, to an improved stimulability and predictability of the system, to a reduced replenishment overhead. The effectiveness of the proposed solutions has been shown with simple examples and simulations.

VIII. REFERENCES

- [1] L. Abeni, "Real Time Operating Systems," *The Sporadic Server*.
- [2] D. Faggioli, M. Bertogna and F. Checconi, "Sporadic server revisited," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.
- [3] X. Fan, Real-Time Embedded Systems: Design Principles and Engineering Practices, 1st ed., USA: Newnes, 2015.
- [4] G. Martinović, M. Karić and I. Crnković, "Simulation and Evaluation of Multiple Sporadic Servers Rescheduling," *IFAC-PapersOnLine*, vol. 48, p. 270–275, 2015.
- [5] M. Stanovich, T. P. Baker, A.-I. Wang and M. G. Harbour, "Defects of the POSIX sporadic server and how to correct them," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010.
- [6] B. Sprunt and L. Sha, "Implementing sporadic servers in Ada," 1990.
- [7] G. C. Buttazzo, "Hard Real-Time Computing Systems," *Predictable Scheduling Algorithms and Applications*, 2011.
- [8] D. W. Ahmad, "Real-Time Systems," *Real-Time Systems*.