

Sporadic server scheduling in Real-Time operating systems

Andrew Zakhary
Electronics Engineering
Hochschule Hamm-Lippstadt
Hamm, Germany
2210009

Abstract—Real Time Operating Systems are time bounded operating systems that rely on executing their tasks within a pre-specified period of time. Periodic tasks are predictable and so their schedulability is relatively simple, however aperiodic tasks make the scheduling process much more complex. As a result of this complexity many scheduling algorithms have been introduced. In this paper we will take a look at one of them, namely Sporadic Server. A full analysis of the history and background of it, Scheduling analysis and an example implementation in C++ is also introduced.

Index Terms—RTOS, Scheduling, Sporadic Server

I. INTRODUCTION

The term "Real time operating systems (RTOS)" describes operating systems where the execute of the tasks have to take into consideration the time constraints. These types of operating systems are used in time-critical applications as energy generation facilities, medical devices, motor drivers, etc. Real time systems have some features that set them apart from other operating systems. These features include:

- **Determinism:** RTOS is deterministic by nature. This means that tasks must and will be executed within a pre-specified deadline. Unlike other operating systems where tasks are only processed sequentially.
- **Schedulability:** RTOS are designed to use algorithms that decide in what order can the processes run while maintaining that every task is done before their deadline. The ability of doing this is called being "schedulable". These algorithms could be split further into two groups:
 - **Preemptive:** These are scheduling algorithms that can interrupt the running task to run another task instead that it deemed with a higher priority. Examples of these algorithms are Round-Robin, Priority Scheduling and Longest remaining Job First.
 - **Non-preemptive:** these and algorithms that run each task to their end without interrupting and only then can proceed with the next task. Examples of these are Shortest Job First, Longest Job First and First Come First Served.

These RTOSs can also be split into different categories depending on the application where it's used into different categories as follows:

- **Hard:** Hard RTOS are systems where failure in meeting the deadline in one of the results could lead to catas-

trophic failures in the systems that may be even deadly. This could be seen in systems as aircraft landing systems.

- **Firm:** Firm RTOS are systems where the failure in meeting one of the deadlines doesn't cause any harm by itself but the result would be useless to the system. Example of this could be found in industrial automation where occasional misses of the system could be tolerated but if they occurred frequently enough the whole system could fail.
- **Soft:** Soft RTOS are systems where the failure in meeting one of the deadlines doesn't lead to any harm but is undesirable. This could be related to performance or quality of service in applications like streaming services where a delay in the system could result in a worse experience for the user but the system would remain functional.

As a result of these varying application constrains a lot of different methods of scheduling tasks inside the OS have been developed. In this paper we will review one of them, namely Sporadic Server (SS) scheduling which is built on the Fixed-Priority Server concept.

II. BACKGROUND

A. Fixed-Priority Server

tasks in an RTOS could be seen as two categories as follows:

- **Periodic tasks:** These are tasks that occur regularly according to predictable intervals. Sensor data acquisition is an example of this as readings from the sensor come at a regular interval.
- **Aperiodic tasks:** These are tasks that unpredictable and can occur at any time during execution. Event-driven changes as user inputs or button presses are an example of this as they can happen at any time and cannot be predicted.
- **Sporadic tasks:** These are a special type of Aperiodic tasks that occur irregularly but still have a pre-known minimum time interval between their arrival time. These could be seen in interrupt handlers that occur irregularly but not more frequently than a pre-known rate.

Since RTOSs require to guarantee the schedulability of tasks assuming worst-case conditions [1], trying to handle aperiodic and periodic tasks makes guaranteeing this before running

the system (i.e. offline guarantee) impossible. On the other hand handling periodic and sporadic tasks could be guaranteed assuming knowing the minimum arrival time of these sporadic tasks. This would allow the realization of all types (i.e. Hard, Soft and Firm) RTOS types.

Handling period and aperiodic tasks could be guaranteed during the running of the OS (i.e. online guarantee) by checking each aperiodic task first as an individual and seeing if it could be scheduled. As a result of this check the algorithm either queues the task to be executed or skips it. As a result this allows only the realization of Firm RTOS.

Fixed-Priority Server scheduling algorithms try to schedule the set of periodic tasks according to a fixed-priority system as Rate-Monotonic (RM) algorithm (in case of SS).

B. Rate Monotonic

Rate Monotonic is a preemptive scheduling algorithm used to schedule periodic tasks. RM organizes tasks so that the priority is inversely proportional to the period of this task. As the periods are fixed it follows that the priorities are also fixed. This makes the RM a fixed priority scheduling algorithm.

As an example, assume a set of tasks as shown in the Table I to be scheduled using RM. First a schedulability check needs

Process (P)	Execution Time (C)	Period (T)
P_1	3	20
P_2	2	5
P_3	2	10

TABLE I
TASKS TO BE SCHEDULED WITH RM

to be made for the tasks to be executed. This is done using the formula 1 as proven by Liu and Layland [2] where n is the number of processes to be scheduled and C and T are the execution and period time respectively.

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \quad (1)$$

Using this equation it is seen that Equation 2 is indeed less than Equation 3. This means that these 3 tasks are schedulable under RM.

$$U = \frac{3}{20} + \frac{2}{5} + \frac{2}{10} = 0.75 \quad (2)$$

$$n(2^{1/n} - 1) = 3(2^{1/3} - 1) = 0.7977 \quad (3)$$

The LCM of the periods (20,5,10) is calculated which is 20 which would be the scheduling time. Then the priority is set, as mentioned before it's inversely proportional to the period so this means the priorities are set as $P_2 > P_3 > P_1$. The Fig. 1 shows how the period and the execution time relate. As an example P_3 will be executed every 10 units of time and each time with execution time of 2 units.

The algorithm starts then by running the task with the task with the highest priority (i.e. P_2) and executes it for 2 units of time uninterrupted. Then P_3 is executed for another 2 units of time uninterrupted. After that it's P_1 's turn to run but it can

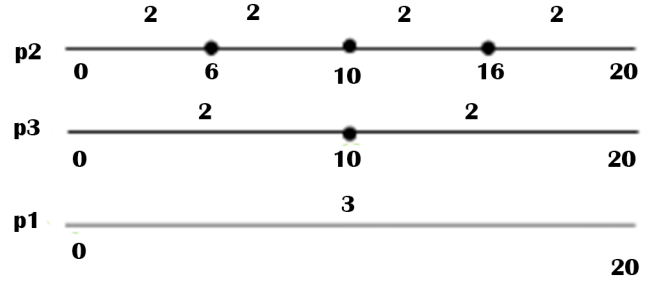


Fig. 1. Tasks with their period and execution time denoted

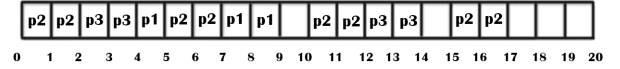


Fig. 2. Period of execution using RM algorithm

only run for 1 unit until it gets preempted and P_2 is executed again for another 2 units as it's period has restarted. Since P_3 's period has not restarted yet and it was completely executed the algorithm re-executes P_1 as it has 2 units of time remaining. Now, all tasks are completely run so nothing is going to be executed until $T = 10$. Then the period of P_2 and P_3 restarts so P_2 will run first until $T = 12$ and then P_3 will run until $T = 14$. After that, again all tasks are executed in full so the algorithm doesn't run anything until $T = 15$ after which the period of P_2 is restarted and it runs until $T = 17$ and then stays without running anything until $T = 20$. At this point the whole period of 20 units is passed and the pattern repeats. This process could be visualised in Fig. 2

III. SPORADIC SERVER

A. Server mechanism

The sporadic server was introduced by Sprunt, Sha and Lehoczky from Carnegie Mellon University in 1989. The SS algorithm tries to handle aperiodic tasks by creating a high priority task that serves the incoming aperiodic task. This server would have a capacity C_s that is used up whenever an aperiodic task is executed. This capacity is only replenished after a it has been consumed by a period of time T_s . The working of SS algorithm contains also some terminologies [3] that could be explained as follows:

- P_s This represents the priority of the task currently being executed by the system.
- P_i This represents the priority of a specific task. Priorities are organized in an ascending order. So P_1 has higher priority than P_2 .
- **Active** A description of a priority of a specific task. The task with priority P_i is said to be "active" if P_s is equal to or higher than P_i .
- **Idle** A description of a priority of a specific task, similar to "active" but an opposite. The task with priority P_i is said to be "active" if P_s is lower than P_i .
- **RT** Represents the replenishment time for the server. This is where the capacity of the server is restocked.

- **RA** Represents the replenishment amount. this is the amount the capacity is replenished by after RT time.

The replenishment time RT is set whenever the SS is *active* and is has capacity ($C_s > 0$). let the time where the SS started being *active* is T_A then $RT = T_A + T_s$. The replenishment amount RA is set whenever the SS is switching from *active* to *idle*. let the time when the SS switches from *active* to *idle* is T_I then RA is the capacity consumed in the time frame $[T_A, T_I]$.

B. High priority SS example

Given the set of tasks shown in Table II it is noticeable that according the RM rules mentioned earlier the priorities could be organized as follows $P_{ss} > \tau_1 > \tau_2$. The schedulability could also be tested according to Equation 1 to be $U = \frac{1}{6} + \frac{2}{10} + \frac{6}{14} = 0.7952 < 0.7797$

Since, however the SS is only executed when there's an aperiodic task, the scheduler must start by executing τ_1 . This is executed for only one unit of time and then preempted when the aperiodic request #1 is received as shown in Fig. 3. At this time ($T=1$) the SS switches from *idle* to *active* ($T_A = 1$) and executes the request for another one time unit. The capacity is dropped by 1 during this period so RA is set to 1 and since $RT = T_A + T_s$ it follows that $RT = 1 + 5 = 6$. After this τ_1 continues the remaining 1 unit of time and then the scheduler executes τ_2 for 5 units of time until this is also preempted by aperiodic request #2. Again, the SS is *active* at $T=8$ and executes for 1 unit of time. As mentioned before $RT = 8 + 1 = 9$ and $RA = 1$. After that τ_2 continues executing for the remaining 1 unit of time of it's execution time. τ_1 is not active again is its period is repeating ($T=10$) so it executes again for 2 units of time. After that the scheduler is free for 2 units of time as both τ_1 and τ_2 are both executed to completion in this period and there are no aperiodic requests. The scheduler would then continue executing τ_2 when its period repeats at $T=14$ until $T=20$ where it would execute the higher priority τ_1 .

Process (P)	Execution Time (C)	Period (T)
SS	1	5
τ_1	2	10
τ_2	6	14

TABLE II
HIGH-PRIORITY EXAMPLE OF SS

C. Schedulability

Even though the sporadic server algorithm relies on the RM algorithm, it violates [4] one of the basic assumptions laid out by Liu and Layland in their analysis of RM which states "If a periodic task which has the highest priority is ready for execution, it must execute". In SS however the execution of the periodic task can be deferred in favor of the aperiodic request served by the SS. To prove that this deferral process doesn't affect the schedulability of the system a theorem was introduced by Sprunt, Sha and Lehoczky [3].

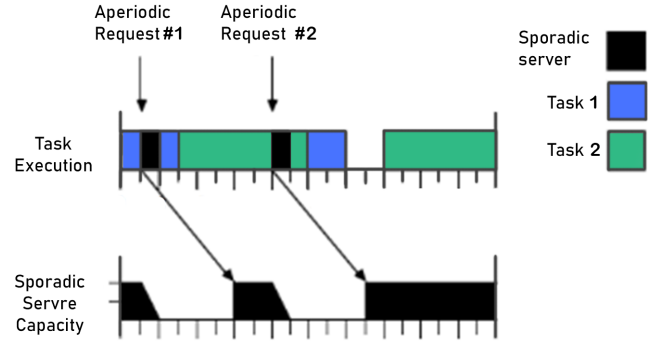


Fig. 3. SS with high priority example

The theory claims "A periodic task set that is schedulable with a task T_i is also schedulable if T_i is replaced by a Sporadic Server with the same period and execution time." To prove this theory an SS is assumed with capacity C_s and period T_s . The SS become active at T_A and switches to *idle* at T_I . The behaviour of the SS could be categorised into three scenarios:

- 1) **No capacity is consumed** this happens if no aperiodic request arrives and this means the capacity stays at C_s and is ready to execute during the period $[T_A, T_I + T_s]$. In this case the SS is equivalent to a periodic task with execution time C_s and period T_s whose arrival time is delayed from T_A to T_I . Since delaying the arrival time of a periodic task doesn't affect schedulability, it follows that an SS with such a behaviour shouldn't either.
- 2) **C_s is completely consumed during $[T_A, T_I]$** The SS replenished happens after $T_A + T_s$ with the full C_s amount. In this case the SS is identical to a periodic task with execution time C_s and period T_s .
- 3) **C_s is partially consumed** Assume the amount C_x is consumed. This means that $C_y = C_s - C_x$ are remaining to be used up in the period $[T_I, T_I + T_s]$ while C_x will be replenished at $T_A + T_s$. This is equivalent to two process T_x that is released at T_A and T_y where it's release is delayed to T_I , and as discussed in case 1 this doesn't affect schedulability.

as derived by Buttazzo [4] a set of periodic tasks with utilization factor U_p scheduled alongside an SS with utilization factor U_s is schedulable with RM if:

$$U_p \leq n \left[\left(\frac{2}{U_s + 1} \right)^{\frac{1}{n}} - 1 \right] \quad (4)$$

for larger n , it is schedulable if:

$$U_p \leq \ln \frac{2}{U_s + 1} \quad (5)$$

D. Implementation in C++

We can simulate the operation of the SS algorithm with C++ code. the code starts with some class definitions. Here two classes are defined, Task and *SporadicServer*. The task class follows the logic explained before, however there's also an arrival time attribute which in this implementation would be defined as 0 for all periodic tasks and non-zero for aperiodic

ones depending on their arrival time. The *SporadicServer* class also follows the logic explained before however since it's important to know when we switch from *active* to *idle* and vice versa, both the last and second to last states are saved in order to compare and know if we switched cases

```

class Task {
public:
    int id;
    // Time required to complete the task
    int execution_time;
    // Period of the task
    int period;
    // Next deadline of the task
    int next_deadline;
    // Remaining time to complete the current
    // instance of the task
    int remaining_time;
    int arrival_time;
    bool isPeriodic;
    // constructor function
    Task(int id, int exec_time, int period, int
        arrival_time, bool isPeriodic)
        : id(id), execution_time(exec_time),
          period(period),
          next_deadline(period),
          remaining_time(exec_time),
          arrival_time(arrival_time),
          isPeriodic(isPeriodic) {}
    // resetting function once period repeats
    void reset() {
        if (isPeriodic) {
            remaining_time = execution_time;
            next_deadline += period;
        }
    };
};

class SporadicServer{
public:
    int capacity;
    int period;
    int replenishmentAmount;
    int replenishmentTime;
    // to check transistion between Idle to
    // Active
    bool lastStateActive;
    bool secondLastStateActive;

    //Constructor function
    SporadicServer(int capacity, int period)
        : capacity(capacity), period(period),
          replenishmentAmount(0),
          replenishmentTime(period),
          lastStateActive(false),
          secondLastStateActive(false) {}
};

```

After the classes are defined a function for scheduling the tasks is defined. First Thing this function is doing is selecting the finding the task to be executed. It does this by getting the task with the lowest priority and non-zero remaining time. If a valid task is found it is sent to the *execute_task* function otherwise we assume the queue is idle. In this case the time is incremented, SS set to inactive and the last state is checked. If the state switched from active to inactive the *RA* is calculated

```

void schedule_tasks(std::vector<Task>&
    tasks_full, int
    total_time, std::vector<Task>&
    tasks, SporadicServer& server) {
    //sorting tasks for lowest period with
    // non-zero remaining time
    auto selected_task =
        std::min_element(tasks.begin(),
            tasks.end(), comparater);
    if (selected_task != tasks.end() &&
        selected_task->remaining_time > 0) {
        execute_task(*selected_task,
            current_time, server);
    }
    // No valid task is selected, Queue is
    // Idle
    else {
        current_time++;
        std::cout << "Idle at time = "
            << current_time << std::endl;
        server.secondLastStateActive =
            server.lastStateActive;
        server.lastStateActive = false;
        // Setting RA for SS
        if (server.secondLastStateActive
            && !server.lastStateActive) {
            server.replenishmentAmount =
                1 - server.capacity;
        }
    }
}

```

The *schedule_tasks* function also checks if the replenishment time arrives and replenishes SS with the required *RA*. It also uses the *next_deadline* attribute to reset the period and check if any tasks missed their deadlines.

```

// Replenishing SS if time is equal to
// RT
if (current_time == server.replenishmentTime
    && server.replenishmentAmount) {
    server.capacity +=
        server.replenishmentAmount;
    std::cout << "server replinshed with "
        << server.replenishmentAmount
        << " at time " << current_time
        << "\n";
    server.replenishmentAmount = 0;
}
// Resetting tasks and notifying about
// missed deadlines if one happens
for (auto& task : tasks) {
    if (current_time >=
        task.next_deadline) {
        if (task.remaining_time > 0) {
            std::cout << "Task " <<
                task.id << " missed its
                deadline at time " <<
                current_time << "\n";
        }
        task.reset();
    }
}
}

```

The *execute_task* checks first if the task to be executed is periodic. If so, the task's *remaining_time* is simple decremented and the current time incremented. Similar to what happened in the idle queue case, the state of the SS is set to

Idle and a check for *RA* is made.

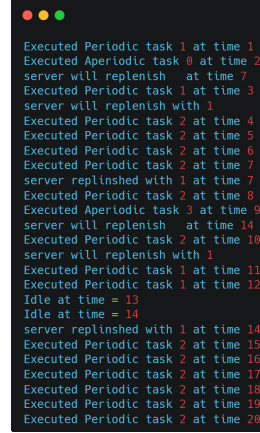
If the task is aperiodic however, both the *remaining_time* and the *capacity* for SS are decremented while incrementing the current time. In contrast to previous step the SS is set to *active* and if the SS was *idle* before the *RT* is calculated.

```
void execute_task(Task& task, int&
    current_time, SporadicServer& server) {
    if(task.isPeriodic){
        task.remaining_time--;
        current_time++;
        server.secondLastStateActive
        =server.lastStateActive;
        server.lastStateActive=false;
        if (server.secondLastStateActive
            &&!server.lastStateActive){
            server.replenishmentAmount=
                1-server.capacity;
        }
    }
    else{
        task.remaining_time--;
        current_time++;
        server.capacity--;
        server.secondLastStateActive =
            server.lastStateActive;
        server.lastStateActive=true;
        if (!server.secondLastStateActive
            &&server.lastStateActive){
            server.replenishmentTime =
                current_time+server.period;
        }
    }
}
```

All of this is put set in motion with the *main* function as shown below. Four tasks are declared, two of them are periodic and two of them are aperiodic. Similar to the example given at Fig.3 The arrival time is also set to 1 and 8 for task 0 and task 3 respectively. The SS is also defined with capacity 1 and period of 5. The total run time of the queue is set to 20 units of time but this can be set to any number needed. The output of the code can be seen in Fig.4 which exactly matches the example given at Fig.3

```
int main() {
    std::vector<Task> tasks = {
        Task(0, 1, 5,1,false), // Aperiodic
        Task 0 with execution time 1 and
        period 5
        Task(1, 2, 10,0,true), // periodic Task
        1 with execution time 2 and period
        10
        Task(2, 6, 14,0,true), // periodic Task
        2 with execution time 3 and period
        14
        Task(3, 1, 5,8,false) // Aperiodic Task
        3 with execution time 1 and period 5
    };
    std::vector<Task> tasks_temp;
    SporadicServer server =
        SporadicServer(1,5);
    // Total time to run the scheduler
    int total_time = 20;
```

```
    schedule_tasks(tasks,
        total_time,tasks_temp, server);
    return 0;
}
```



```
Executed Periodic task 1 at time 1
Executed Aperiodic task 0 at time 2
server will replenish at time 7
Executed Periodic task 1 at time 3
server will replenish with 1
Executed Periodic task 2 at time 4
Executed Periodic task 2 at time 5
Executed Periodic task 2 at time 6
Executed Periodic task 2 at time 7
server replenished with 1 at time 7
Executed Periodic task 2 at time 8
Executed Aperiodic task 3 at time 9
server will replenish at time 14
Executed Periodic task 2 at time 10
server will replenish with 1
Executed Periodic task 1 at time 11
Executed Periodic task 1 at time 12
Idle at time = 13
Idle at time = 14
server replenished with 1 at time 14
Executed Periodic task 2 at time 15
Executed Periodic task 2 at time 16
Executed Periodic task 2 at time 17
Executed Periodic task 2 at time 18
Executed Periodic task 2 at time 19
Executed Periodic task 2 at time 20
```

Fig. 4. Code implementation output

IV. CONCLUSION

Sporadic Server algorithm has proven itself a solid solution for scheduling in real time operating systems since its introduction in the 80s of the last century. It offers an elegant way to try to handle aperiodic tasks in a simple way specially for sporadic aperiodic tasks. The algorithm however can be demanding in the aspect of overhead if non-sporadic aperiodic requests are needed to be handled where this would introduce complexity in accepting or rejecting the request depending on its schedulability. The Sporadic Server algorithm can also lead to under-utilization of system resources if a too-high of a capacity is given where this would affect the utilization of the periodic tasks to be scheduled. Priority selection can also be tricky in the system design where selecting the right priority for the SS is crucial to optimize performance of the system. On the other hand however, if these challenges are overcome, SS offers a good solution that can also be scaled up depending on the need. Multiple sporadic servers [5] can be implemented to handle different types of aperiodic tasks to achieve better performance.

REFERENCES

- [1] Li Zhou, Hong Li, Weimin He, Chengshuo Zhang, and Zhu Wang. Scheduling non-periodic tasks using sporadic server in autosar operating system. In *2010 IEEE International Symposium on Industrial Electronics*, pages 315–321, 2010.
- [2] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.
- [3] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-time Systems*, 1(1):27–60, 1989.
- [4] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, volume 24 of *Real-Time Systems Series*. Springer US, New York, NY, 3rd ed. 2011 edition, 2011.
- [5] Goran Martinovic, Miran Karic, and Drago Zagar. Simulation results on multiple sporadic server mechanism. In *2007 29th International Conference on Information Technology Interfaces*, pages 627–632, 2007.