Priority Exchange

Mofifoluwa Ipadeola Akinwande
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
mofifoluwa-ipadeola.akinwande@stud.hshl.de

***Abstract*—In real-time systems, the coherent execution of periodic and aperiodic tasks is important to ensure system reliability and predictability. Methods like polling tasks and background processing help with the execution of aperiodic tasks, but can lead to rather long wait and response times. Aperiodic service algorithms such as the Priority Exchange(PE) algorithm, loosely based on the same concepts as the polling task and compatible with Rate Monotonic Scheduling(RMS) algorithm, help to reduce the wait time of aperiodic tasks while maintaining the deadlines of periodic tasks. This paper will explore the PE algorithm in more detail and offer a comparative analysis with other scheduling techniques, showing its effectiveness in real-time systems.**

I. Introduction

The ability of a real-time system to be predictable, that is meet strict timing constraints, is what separates it from normal general-purpose systems. Unlike conventional systems, where "fairness" ensures tasks are scheduled in a way that they all share processing time and resources equally, real-time systems require deadline adherence, where some tasks have to be executed with absolute certainty.

For this reason, scheduling is a very important factor in designing real-time systems. There are two main ways to satisfy the predictabilty requirements required of a real-time system. The Cyclically Executives and the Priority Scheduling Methods. While cyclically executives are more general they are usually more expensive to achieve as they require extensive verification and testing. On the other hand, priority-based preemptive scheduling algorithms have been extensively studied and two main algorithms were derived: the deadline-driven scheduling methods, where the tasks with the earliest deadlines are given higher priority and those with later deadlines are given the lowest priority, and rate monotonic scheduling that assigns higher priority to tasks with shorter periods [1].

In rate monotonic scheduling algorithm, more focus is given to periodic tasks which occur at regular intervals and have hard timing constraints than to aperiodic tasks that do not have a fixed time of arrival and usually have softer deadlines. In this algorithm, aperiodic tasks are scheduled using polling servers or background processing. In a polling server setup, a periodic task checks at fixed intervals for aperiodic task requests. If any are found, it executes them. In background processing, the aperiodic tasks only run when the CPU is idle. The limitation of these two approaches is that the wait and response times for the aperiodic request are rather long [2].

To significantly reduce wait times, aperiodic service algorithms like Deferrable server and Priority Exchange were proposed. These methods aim to reduce the wait and response times for aperiodic tasks while still meeting the hard deadlines of periodic tasks. This paper will focus on the Priority Exchange algorithm - exploring its derivation, how it works, benefits, etc. [3]

II. Background

In real-time systems, periodic tasks are those that recur at fixed intervals and typically have hard timing constraints that have to be met. There are multiple ways to schedule these tasks in a system, but the Rate Monotonic Scheduling(RMS) algorithm -which assigns highest priority to tasks with the shortest periods- has proven to be one of the most efficient methods. According to Liu and Layland, for any given set of n periodic tasks to be guaranateed schedulable by RMS, the total CPU utilization

$$U \;=\; \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{1}$$

must satisfy

$$U \;\leq\; U_{\mathrm{LL}}(n) = n\big(2^{1/n} - 1\big) \tag{2}$$

where C_i is the worst–case execution time and T_i the period of task i [1].

In the worst-case scenario as n approaches infinity, the bound converges to 0.693. Earlier studies show that though some other scheduling algorithms can have worst-case bounds as high as 1.00, RMS is more practical [4]. One of the reasons is that it offers a way to efficiently schedule aperiodic tasks - which describes tasks with no fixed time of arrival and softer or no deadlines.

The two general methods of scheduling aperiodic tasks under the RMS algorithm are by using the polling server method or background servicing [2].

In the polling server method, a periodic task is created that checks if there are any aperiodic requests and executes them if there are. The downside to the polling server method is that if an aperiodic request is made after the cycle of the polling task is over, it has to wait for the next cycle, that is, no time is reserved for aperiodic requests. In background service, aperiodic tasks run when the CPU is idle, and if the load of the periodic tasks is high, then very little time is left for aperiodic requests. Both these methods mean that the wait time(the time it takes for a task to execute after a request is

made), and the response time(the time it takes from request to full execution of the task) of aperiodic tasks are usually long.

To further explain these methods and contextualize their effects on aperioidc tasks, consider two periodic tasks A and B with their execution and period times as shown in Figure 1.

Tasks	Execution Time	Period
Task A	2 seconds	6 seconds
Task B	4 seconds	10 seconds

Fig. 1. Execution Time and Period of Tasks A and B

For this example, both tasks are assumed to be ready for execution at time 0. Additionally, in the polling server method, a polling task with an execution time of 1 second and period of 4 seconds is created.

In the background servicing example, as seen in Figure 2, both periodic tasks execute according to their fixed priority schedule. The aperiodic request that arrives at time 5 must wait until the processor becomes idle. Therefore, it is not executed until time 8 resulting in a response time of 4 seconds. Similarly, the next aperiodic request at time 11 is delayed and does not receive any processor time here. This shows that while the deadlines of the periodic tasks are preserved, the response times for aperiodic requests can become long. In more complex cases with multiple periodic tasks and tighter scheduling, aperiodic tasks might not get any processor time at all.

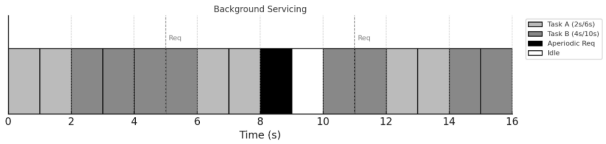

Fig. 2. Background servicing

As seen in Figure 3, Task A begins execution at time 0 for 2 seconds, despite the polling task having the highest priority due to the absence of no aperiodic request. Following Task A, Task B executes for 4 seconds. At time 5, an aperiodic request is made; however, since the polling task's period is over, Task B continues to execute. At time 8, the polling server becomes active again and the aperiodic request is finally executed before Task B continues with its execution. Another aperiodic request is made at time 9, and doesn't get executed until the next polling task period becomes active at time 12. This results in response times of 4 seconds for both requests.

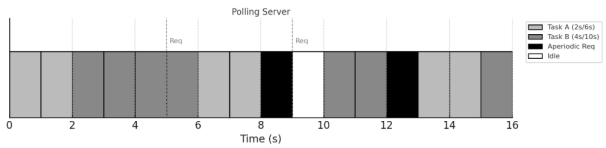

Fig. 3. Polling server

Although this approach also guarantees some CPU time for aperiodic tasks and preserves periodic task deadlines, the improvement over background servicing is only marginal. Aperiodic requests may still experience delays depending on when they arrive within the polling cycle.

III. Priority Exchange Algorithm

The Priority Exchange(PE) algorithm, proposed by Lehoczky, Sha, and Strosnider, was developed in order to properly serve aperiodic tasks by reducing their wait and response times, while maintaining the strict deadlines of period tasks [3]. It operates similarly to a polling server in that a high-priority task is created to service an aperiodic request. However, it differs from a polling server in that unused service time is not lost but accumulated at all priority levels.

Initially, the high-priority aperiodic server is replenished with its execution capacity. If there are any aperiodic requests pending at the onset, they are serviced immediately using the server's budget. If no aperiodic requests are present, a priority exchange occurs with the highest priority periodic task. In this exchange, the server's current higher-priority execution time is given to the periodic task, and the server accumulates an equivalent amount of aperiodic time at the priority level of the periodic task.

If an aperiodic request later arrives while credit is available, the request immediately preempts periodic tasks at the priority level where the credit was stored, using only the accumulated budget. If the periodic task completes without an aperiodic request being made, the credit is lost.

To better illustrate this mechanism, consider this example. Two periodic tasks A and B with execution times and periods as shown in Figure 4, are created along with a Priority Exchange server with an execution time of 1 second and 5 seconds. Fig. 5 shows how the tasks are scheduled, and two aperiodic requests are made, first at $t = 6$ and $t = 12$

Entity	Execution Time	Period
Priority Exchange Server	1 second	5 seconds
Task A	2 seconds	6 seconds
Task B	3 seconds	13 seconds

Fig. 4. Task and server configuration for the PE example.

As seen in Figure 5, Task A is scheduled first because no aperiodic requests are pending. It completes its execution, and Task B is subsequently scheduled. The first aperiodic request arrives at $t = 6$ and is serviced immediately. As illustrated in Figure 6, this is possible because a priority exchange occurred at $t = 1$, during which the server's budget was exchanged with Task A, allowing aperiodic time to accumulate at that priority level. Although Task A is ready to execute again at that level, the aperiodic request preempts it using the accumulated credit. The next aperiodic request arrives at $t = 12$, but since no credit is available at that point, it must wait until the Priority Exchange server is replenished at $t = 15$, when it is finally executed.

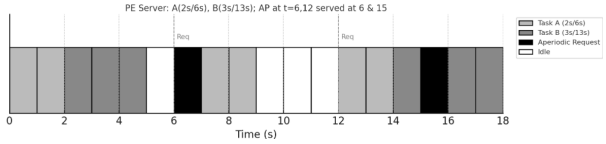

Fig. 5. Schedule of tasks and aperiodic requests under PE.

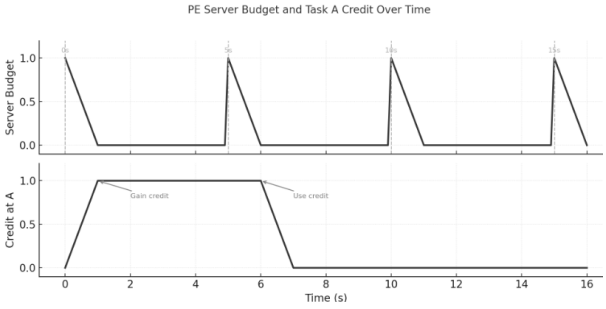

Fig. 6. Server and task-level credit accumulation timeline.

Discussion: A caveat of the Priority Exchange algorithm can be seen in the example presented earlier. At $t = 12$, an aperiodic request arrives but must wait an additional 3 seconds before it is finally serviced, which resembles the behavior of a polling server. This happens because PE stores credit only when an exchange occurs. If no exchange takes place, then no credit is available, and the request must wait.

In contrast, the Deferrable Server (DS)—another aperiodic service algorithm, unlike Priority Exchange—retains its budget throughout the server period, allowing it to service requests immediately upon arrival, as long as they occur within the server's active period. As a result, wait and response times under DS can be shorter. An improvement on the PE algorithm, the Extended Priority Exchange (EPE) algorithm, proposed by Sprunt, addresses this issue by allowing accumulated credit to migrate across all priority levels, increasing the chances that an aperiodic request can be serviced promptly [2].

What PE sacrifices in worst-case response time, it makes up for in schedulable utilization. Let $U_s = C_s/T_s$ represent the server utilization and U_p the total utilization of periodic tasks. According to the asymptotic analysis by Lehoczky, Sha, and Strosnider, the system remains schedulable if:

$$U_p \leq \ln(U_s + 1) - U_s \quad \text{(for PE) [3]} \tag{3}$$

$$U_p \leq \ln(2U_s + 1) - 2U_s \quad \text{(for DS) [3]} \tag{4}$$

The DS bound is tighter because a Deferrable Server can cause two consecutive bursts of execution within the same window, leading to shorter periodic utilization. Therefore, Priority Exchange allows a higher periodic workload to be scheduled compared to Deferrable Server, though sometimes at the cost of slightly delayed aperiodic response.

IV. Verification

Since the Priority Exchange algorithm is compatible with Rate Monotonic Scheduling, which is a static form of scheduling—meaning priorities and scheduling decisions are handled at compile time—it can be modeled as one big finite state machine. This makes it suitable for verification of its timing properties. For the example task illustrated in figure 4, several temporal properties of the corresponding timed automata are verified using the model-checking tool UPPAAL [5].

In UPPAAL, four templates are created: `TaskA`, `TaskB`, `PE` (for the Priority Exchange server), and `aperiodicReq`. The three periodic tasks — `TaskA`, `TaskB`, and `PE` — are each modeled using four distinct states:

- **Idle**: The task is inactive and outside of its period.
- **Ready**: The task has been released and is awaiting execution.
- **Running**: The task is currently executing.
- **Done**: The task has finished execution within its current period.

The main variables declared and used across the system are:

- **`credit_A` / `credit_B`**: Integer variables used to store the aperiodic credit accumulated at the priority levels of Task A and Task B, respectively.
- **`pe_server_flag`**: An integer flag that indicates whether an aperiodic request is currently pending and awaiting service.
- **`req_ap`**: A broadcast channel used to signal to the system that an aperiodic request has been made.
- **`giveA` / `giveB`**: Broadcast channels used by the PE server to model priority. If no aperiodic request is pending, the server "gives" execution permission to Task A or Task B.
- **`readyA` / `readyB`**: Boolean flags that indicate whether Task A or Task B has been released and is ready to run.
- **`period_active`**: A boolean used internally by the PE server to track whether it is currently within its execution window for the current period.

A. PE Server template

As seen in Figure 7, the Priority Exchange (PE) server is modeled as a periodic task with four states: *Idle*, *Ready*, *Running*, and *Done*. Unlike the standard periodic tasks, it conditionally either services an aperiodic request or exchanges its execution time with the lower-priority periodic tasks.

- **Idle → Ready**: This transition occurs at the beginning of each server period. The clock is reset (`clk := 0`), and a flag such as `period_active` may be set to indicate that the server is within its execution window.
- **Ready → Running (Servicing Request)**: If an aperiodic request is pending (`pe_server_flag == 1`), the server transitions to the *Running* state. This transition is guarded by the condition `pe_server_flag == 1`, and includes an update `pe_server_flag := 0` to clear the request. It also synchronizes via the channel `req_ap?` with the `aperiodicReq` template, representing the servicing of the aperiodic request.
- **Ready → Done (Give Credit to Task A)**: If no request is pending and `readyA == true`, the server gives its time to Task A. This transition updates `credit_A := credit_A + 1` and synchronizes with `giveA!`.

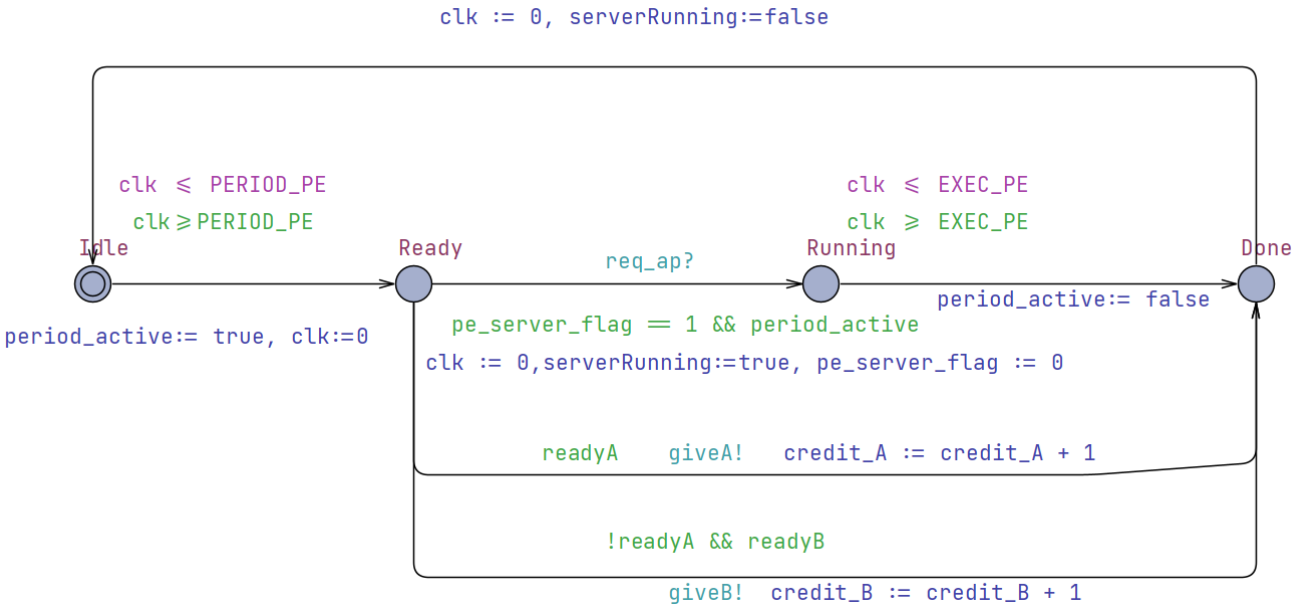

Fig. 7. UPPAAL template for the Priority Exchange (PE) server.

- **Ready → Done (Give Credit to Task B)**: If Task A is not ready but `readyB == true`, a similar credit exchange occurs, updating `credit_B := credit_B + 1` and synchronizing with `giveB!`.
- **Running → Done**: This transition marks the end of the server's activity, either after servicing a request or completing a credit exchange. It may optionally clear flags such as `period_active`.
- **Done → Idle**: This final transition marks the end of the server's current cycle and resets it for the next period.

B. Temporal Properties

To verify the correctness of a scheduling algorithm—particularly in the context of real-time systems—it is essential to ensure that certain temporal properties hold. In this work, a set of such properties was formulated and verified using UPPAAL to confirm the correctness and reliability of the modeled system behavior. Table 1 shows some of these properties and the corresponding verification description inputted in UPPAAL. While all the properties shown described are important, the most important one is "Deadline compliance" which describes that all periodic tasks meet their deadlines.

TABLE I
VERIFIED TEMPORAL PROPERTIES IN UPPAAL

Property Type	Description
Deadlock Freedom	The system should never reach a state where no transitions are possible. `A[] not deadlock`
Mutual Exclusion	Only one periodic task should be running at any given time. `A[] (TaskA.Running imply !TaskB.Running)`
Request Responsiveness	If an aperiodic request is made, the PE server should eventually service it. `A[] (pe_server_flag == 1 imply req_served)`
Task Completion	Each task that becomes ready should eventually complete. `A[] (readyA imply TaskAInst.Done)`
Priority Respect	Task B should only run if Task A is not running. `A[] (TaskBInst.Running imply TaskAInst.Idle || TaskAInst.Ready)`
Execution Time Bound	Each task must respect its worst-case execution time. `A[] (TaskAInst.Running imply TaskAInst.clk <= EXEC_A)`

REFERENCES

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] B. Sprunt, "Aperiodic task scheduling for real-time systems," Ph.D. dissertation, Carnegie Mellon University, 1990.

[3] J. P. Lehoczky, L. Sha, and J. B. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in *Proc. IEEE Real-Time Systems Symposium*, 1987, pp. 261–270.

[4] J. P. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proc. IEEE Real-Time Systems Symposium*, 1989, pp. 166–171.

[5] M. Spuri and G. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems," in *Proc. IEEE Real-Time Systems Symposium*, 1996.