Resource Access Protocols

1st Sheikh Muhammad Adib Bin Sh Abu Bakar
Hochshule Hamm-Lippstadt
B.Eng. Electronic Engineering
Lippstadt, Germany
sheikh-muhammad-adib.bin-sh-abu-bakar@stud.hshl.de

***Abstract*—In hard real time system, it is important to ensure every single task not only to be successfully executed but to produce the right value on the right time. Thus, scheduling algorithm play a big role in handling various of task. It is common in real time system environment, some tasks share the same resource, which is one of the complicated part in concurrent operating system. Any concurrent operating system should utilize proper synchronization to assure mutual exclusion among competing activities, in order to ensure the predictability of the system, which is important in order to produce reliable system especially when safety is a crucial part. This is why resource access protocol one of the important topic in building a real time system. In this research paper, I will explain two resource access protocol for uni-processor that are developed under fixed priority assignment that are used to solved a certain problem that arise during two or more tasks sharing a resource. To ensure this paper is easy to understand, I also include basic information about real time system before going deep into resource access protocol.**

***Index Terms*—real time system, resource, protocol**

I. Introduction

Before we start, first we need to make sure the reader could understand the concept and terms used in this paper. Hence it is essential to explain them. We will start with the definition of the real time system..

Real Time System - A system whose response time and delay time are deterministic without uncertainty and non-reproducibility and has an internal configuration that makes the worst value predictable or makes it easy to produce an educated guess value [1].

A real-time system must have the ability to not only process data in a defined, predictable time frame but also ensure that critical tasks, such as safety-related workloads, are completed prior to less critical tasks [2].

So, the main key features that a real time system must have is the predictability of the system to ensure its capability to produce the right result in the right time . We will discuss more about predictability because this is one of the main reasons the need of resource access protocol, so that a task doesn't miss its deadline to produce the right result.

In the next section I will explain further important concept and terms within the scope of this paper in more detail.

Identify applicable funding agency here. If none, delete this.

II. Task Scheduling

In this section we will describe the concept and terms that are dominant in this research paper.

A. Task, Job, Thread and Process

Task, thread, job and process are the basic building block in task scheduling. Their definition are listed below according to [4]

- Task: a sequence of instructions that, in the absence of other activities, is continuously executed by the processor until completion.
- Job: an instance of a task executed on a specific input data.
- Thread: a task sharing a common memory space with other tasks.
- Process: a task with its private memory space, potentially generating different threads sharing the same memory space.

B. Scheduling Policy and Scheduling Algorithm

When a single processor has to execute a set of concurrent tasks – that is, tasks that can overlap in time – the CPU has to be assigned to the various tasks according to a predefined criterion, called a scheduling policy and the set of rules that, at any time, determines the order in which tasks are executed is called a scheduling algorithm [5].

Considering that the schedule algorithm handles more than one task, now we will extend the definition of a task correspond to their state according to [5]

- A task that could potentially execute on the CPU can be either in execution (if it has been selected by the scheduling algorithm) or waiting for the CPU (if another task is executing).
- A task that can potentially execute on the processor, independently on its actual availability, is called an active task.
- A task waiting for the processor is called a ready task, whereas the task in execution is called a running task.
- All ready tasks waiting for the processor are kept in a queue, called ready queue.

C. Classification of Scheduling Algorithms

In this subtopic we will only focus on important classification of scheduling algorithm that are related to the scope of this research paper.

- Preemptive: running task can be interrupted at any time.
- Non-preemptive: a task, once started is executed until completion
- Static: scheduling decisions are based on fixed parameters (off-line) .
- Dynamic: scheduling decisions are based on parameters that change during system evolution.
- Off-line : Scheduling algorithm is performed on the entire task set before start of system. Calculated schedule is executed by dispatcher.
- On-line : scheduling decisions are taken at run-time every time a task enters or leaves the system.
- Optimal : the algorithm minimizes some given cost function, alternatively : it may fail to meet a deadline only if no other algorithm of the same class can meet it.
- Heuristic : algorithm that tends to find the optimal schedule but does not guarantee to find it

D. Periodic and Aperiodic Task

A task can be periodic or aperiodic base on the way it is activated. Their definition are listed below according to [4]

- Periodic Task: a task in which jobs are activated at regular intervals of time, such that the activation of consecutive jobs is separated by a fixed interval of time, called the task period.
- Aperiodic Task: a task in which jobs may be activated at arbitrary time intervals.

The topic of this paper is limited to periodic tasks. Figure 1 shows an example of periodic task.

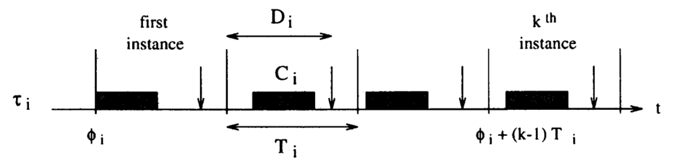

Fig. 1. periodic task [5]

E. Type of task constraint

Task constraints are used by the scheduling algorithm to set priority for each task. There are 3 main type of task constraint and they are:

- Timing constraint
- Precedence constraint
- Resource constraint

Next we will explain more detail about timing constraint and resource constraint. Precedence constraint is not covered in this paper.

1) Timing Constraint: One of the timing constraint that are essential in this paper is deadline. There is two type of deadline:

- Relative Deadline: the longest interval of time within which any job should complete its execution [4].
- Absolute Deadline (of a job): the time at which a specific job should complete its execution [4].

With time constraint we can characterize real time system into three category :

- Hard Real Time System - failed to met the deadline can cause catastrophic event
- Soft Real Time System - failed to met the deadline only cause the degradation of the system
- Firm Real Time System - failed to met the deadline only cause the production of useless output

Hard real time is the main focus in this research paper.

2) Resource Constraint: Another important constraint related to this topic is resource constraint.

According to [5] - resource is any software structure that can be used by the process to advance its execution. Typically, a resource can be a data structure, a set of variables, a main memory area, a file, a piece of program, or a set of registers of a peripheral device. A resource dedicated to a particular process is said to be private, whereas a resource that can be used by more tasks is called a shared resource.

Many shared resources do not allow concurrent access by competing processes in order to ensure data consistency, and instead require mutual exclusion. This means that if another task is within R modifying its data structures, a task cannot access R. R is referred to as a mutually exclusive resource in this scenario. A piece of code executed under mutual exclusion constraints is called a critical section. This can be illustrate in figure 2 where R is the mutually exclusive resource, τ_1 and τ_2 are two distinct task.

In figure 3 we can see the data inconsistency without mutual exclusion

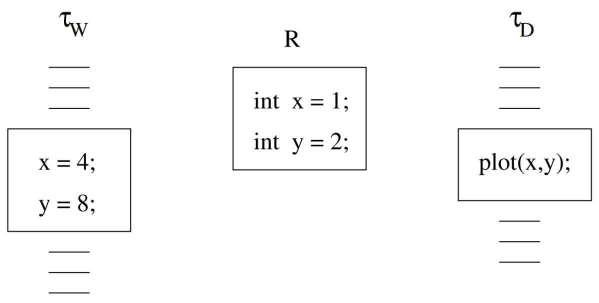

Fig. 2. Two tasks sharing a buffer with two variables. [5]

Semaphore is one of the very well known example of mechanism for synchronization provided by operating system. Each critical section must begin with wait(S) primitive and end with signal(s) primitive where s is a binary semaphore

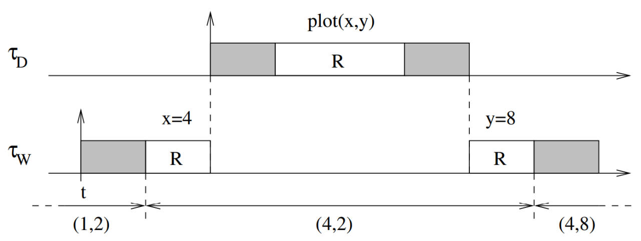

Fig. 3. Example of schedule creating data inconsistency. [5]

as shown in figure 4. If a task want to access a resource that currently accessed by another task, that task must wait until signal(S) is executed by the previous task as illustrated in figure 5 and the states of the task are shown in figure 6

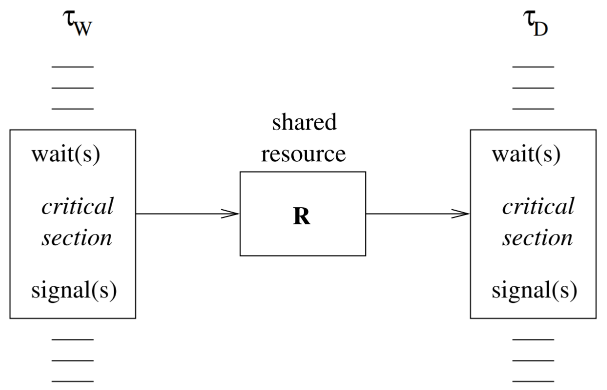

Fig. 4. Structure of two tasks that share a mutually exclusive resource protected by a semaphore. [5]

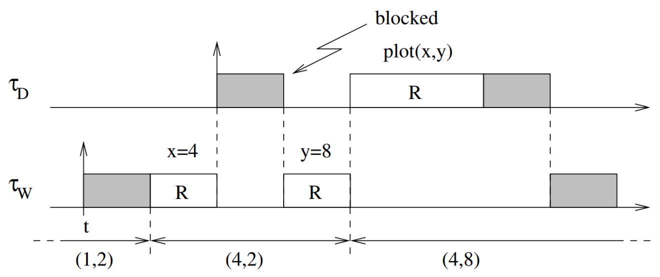

Fig. 5. Example of schedule when the resource is protected by a semaphore.. [5]

F. Schedulability

Before we considering to implement resource access protocol it is important to make sure that all task are schedulable or a schedule is feasible on the set of task. This can be done through scheduability test. This paper will not explain about scheduability test since all task already considered as schedulable. More detail about scheduability test can be found in [5].

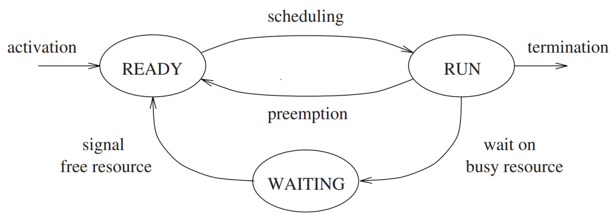

Fig. 6. Waiting state caused by resource constraints. [5]

III. Problem Definition: Invention Priority

In theory, a set of task that are schduable will be executed upon on its arriving time and preempted if a task with higher priority arrive. That will not be always true if we apply resource constraint. The problem arrive when a scheduler want to preempt a lower task that is currently accessing a mutually exclusive resource R and want to execute higher priority task that also want to use the resource. Since the lower priority task didn't yet produce signal(S) primitive where S is the binary semaphore, the higher priority task will be blocked. This phenomenon called priority invention. The implication from this phenomenon is that it will cause unbounded delay on execution of task with higher priority and reduce the predictability of the system. The priority invention is illustrated in figure 10 where τ_1 have the highest priority.

Here is where we need resource access protocol to make some adjustment to the scheduler so that the implication of invention priority could be reduce which we going to discuss in incoming sections.

IV. Resource Access Protocol

Now we will discus 2 main resource access protocol for periodic task which are:

- Non-Preemptive Protocol
- Highest Locker Priority

The first protocol is for non-preemptive-able scheduler and the second protocol is for preemptive-able scheduler. The following list are the important notation that I will use in explaining each protocol. The main idea about those protocol is to avoid priority invention, so that the blocking time experienced by the task that have highest priority by lower priority task will also be discussed for each protocol.

- B_i denotes the maximum blocking time task τ_i can experience.
- $z_{i,k}$ denotes a generic critical section of task τ_i guarded by semaphore S_k.
- $Z_{i,k}$ denotes the longest critical section of task τ_i guarded by semaphore S_k.
- $\delta_{i,k}$ denotes the duration of $Z_{i,k}$.
- $z_{i,h} \subset z_{i,k}$ indicates that $z_{i,h}$ is entirely contained in $z_{i,k}$.

V. Non-Preemptive Protocol

A. Definition

This protocol is a simple protocol and named non preemptive because it avoid any interruption on running task τ_j that accessing a resource $R_k$that guarded by , S_k. To reduce total blocking time experienced by the task τ_i that have highest priority, this protocol just increase the priority of the task τ_j that currently accessing the resource τ_j, so that the task will not be interrupted and can be done much faster.Without this protocol, the task that highest priority τ_i will interrupt the task τ_j that currently accessing the resource R_k even though the task cannot access the resource because it already guarded by S_k. The scheduler then switch back to the task before to finish its process, this switch context process could cause longer blocking time experienced by the highest priority task. After the taskτ_j finish accessing the resources, its priority will be back to its nominal priority P_j.These situations can be compared through figure 7 and 8 . So, the priority of the task $\tau_i$that currently accessing the resource is

$$p_i(R_k) = \max_h \{P_h\}$$

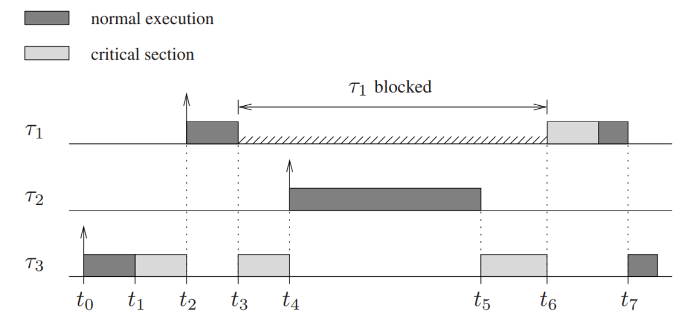

Fig. 7. An example of priority inversion.. [5]

B. Blocking time computation

The total of critical section of lower priority task τ_j blocking higher priority task τ_i is

$$\gamma_i = \{Z_{j,k} | P_j < P_i, k = 1, ..., m\} \text{ [5]}$$

Hence the total duration highest priority task is blocked is

$$B_i(R_k) = \max_{j,k} \{\delta_{j,k} - 1 | Z_{j,k} \in \gamma_i\} \text{ [5]}$$

C. Implementation Strategies

As state by [6] - All commercial RTOSs have a means for beginning and ending a critical section. Invoking this Scheduler operation prevents all task switching from occurring during the critical section. If we write our own RTOS, the most common way to do this is to set the Disable Interrupts bit on our processor's flags register. The precise details of this vary, naturally, depending on the specific processor.

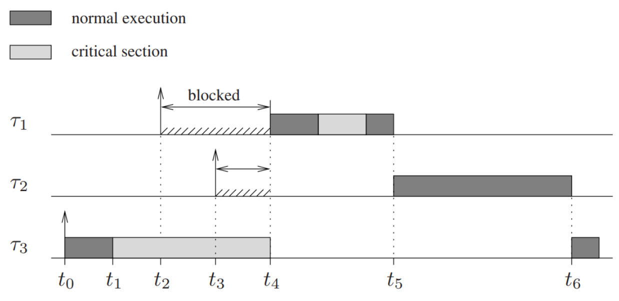

Fig. 8. Example of NPP preventing priority inversion. [5]

D. Sample Model

The following example model is totally taken from [6].

An example of the use of this pattern is shown in Figure 9. This example contains three tasks: Device Test (highest priority), Motor Control (medium priority), and Data Processing (lowest priority). Device Test and Data Processing share a resource called Sensor, whereas Motor Control has its own resource called Motor.

The scenario starts off with the lowest-priority task, Data Processing, accessing the resource that starts up a critical section. During this critical section both the Motor Control task and the Device Test task become ready to run but cannot because task switching is disabled. When the call to the resource is almost done, the Sensor.gimme() operation makes a call to the scheduler to end the critical section. The scenario shows three critical sections, one for each of the running tasks. Finally, at the end, the lowest-priority task is allowed to complete its work and then returns to its Idle state.

E. Problem Arise

As shown in figure 10, this protocol will block highest priority task τ_1 even though the task will not access the resource.This problem could be solved in the next protocol which is Highest Locker Priority (HLP) protocol.

VI. Highest Locker Priority

A. Definition

Highest Locker Priority (HLP) is the improvement of the previous protocol to allow the highest priority task τ_i that doesn't use resource R_k to interrupt the lower priority task τ_j that use the resource, R_k by limiting the raised priority of task τ_j. So,

$$p_i(R_k) = \max_h \{P_h | \tau_h \text{ uses } R_k\} \text{ [5]}$$

This dynamic priority then set back to its nominal value P_i when the task leave its critical section. The maximum

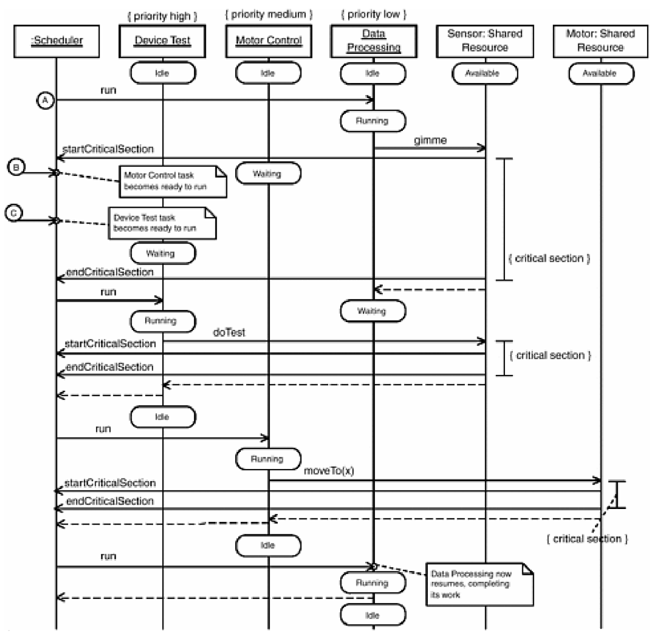

Fig. 9. Sample Model for Non-Preemptive Protocol [6]

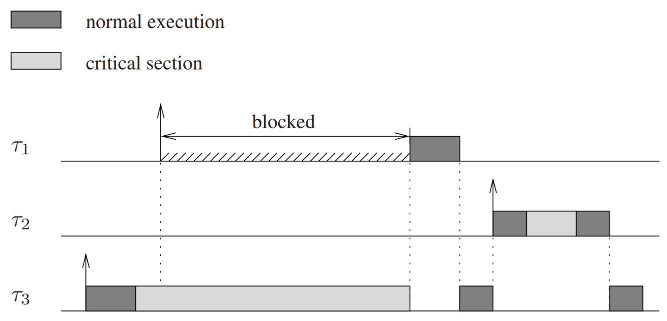

Fig. 10. Example in which NPP causes unnecessary blocking on τ_1 [5]

raised priority of task τ_j is called priority ceiling $C(R_k)$ and computed off-line. The maximum priority $C(R_k)$ of the tasks sharing R_k is the computed online such

$$C(R_k) \overset{def}{=} \max_h \{P_h | \tau_h \text{ uses } R_k\} \text{ [5]}$$

Since the priority of lower priority task τ_j is raised as soon as the task entering R_k, this protocol also known as Immediate Priority Ceiling. This protocol can be visualize as in figure **??** where task τ_1 have higest priority and task τ_3 is the first task arrive

B. Blocking Time Computation

So, total of critical section of lower priority task τ_j blocking higher priority task τ_i is reduced by adding new parameter as shown below.

$$\gamma_i = \{Z_{j,k} | P_j < P_i \text{ and } C(R_k) \geq P_i\} \text{ [5]}$$

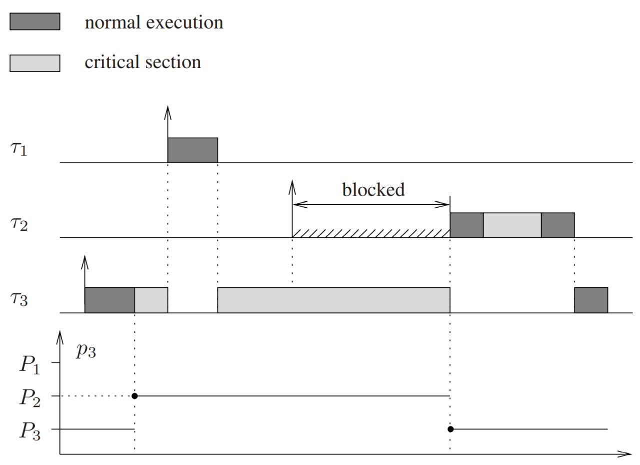

Fig. 11. Example of schedule under HLP, where $p3$ is raised at the level $C(R) = P_2$ as soon as τ_3 starts using resource R [5]

According to [5] - Under HLP, a task τ_i can be blocked, at most, for the duration of a single critical section belonging to the set γ_i and this theorem is proved by contradiction,

Below is the proof of the theorem according to [5].

Assuming that τ_i is blocked by two critical sections, $z_{1,a}$ and $z_{2,b}$. For this to happen, both critical sections must belong to different tasks (τ_1 and τ_2) with priority lower than P_i, and both must have a ceiling higher than or equal to P_i. That is, by assumption, we must have

$$P_1 < P_i \leq C(R_a)$$
$$P_2 < P_i \leq C(R_b)$$

Now, τ_i can be blocked twice only if τ_1 and τ_2 are both inside the resources when τ_i arrives, and this can occur only if one of them (say τ_1) preempted the other inside the critical section. But, if τ_1 preempted τ_2 inside $z_{2,b}$ it means that $P_2 > C(R_b)$, which is a contradiction. Hence, the theorem follows.

As shown in figure 11, τ_i can be block at maximum once, means that

$$B_i(R_k) = \max_{j,k} \{\delta_{j,k} - 1 | Z_{j,k} \in \gamma_i\} \text{ [5]}$$

We need to minus one unit of time because the lower priority task τ_j need to access R_k atleast one unit time earlier then τ_i to block it.

C. Implementation Strategies

According to [6] - Fewer RTOSs support the Highest Locker Pattern more than the basic Priority Inheritance Pattern. The implementation of this pattern is fairly straightforward, with the addition of priority ceiling attributes in the Shared Resource. When the mutex is locked, it must notify the Scheduler to elevate the priority of the locking task to that resource's priority ceiling.

D. Sample Model

The following example model is totally taken from [6].

In the example shown in Figure 12, there are four tasks with their priorities shown using constraints, two of which, Waveform Draw and Message Display, share a common resource, Display. The tasks, represented as active objects in order of their priority, are Message Display (priority Low), Switch Monitor (priority Medium Low), Waveform Draw (priority Medium High), and Safety Monitor (priority Very High), leaving priority High unused at the outset. Message Display and Waveform Draw share Display, so the priority ceiling of Display is just above Waveform Draw (that is, High).

The scenario runs as follows: First, the lowest-priority task, Message Display, runs, calling the operation Display.displayMsg(). Because the Display has a mutex semaphore, this locks the resource, and the Scheduler (not shown in Figure 7-11) escalates the priority of the locking task, Message Display, to the priority ceiling of the resource—that is, the value High.

While this operation executes, first the Switch Monitor and then the Waveform Draw tasks both become ready to run but cannot because the Message Display task is running at a higher priority than either of them. The Safety Monitor task becomes ready to run. Because it runs at a priority Very High, it can, and does, preempt the Message Display task.

After the Safety Monitor task returns control to the Scheduler, the Scheduler continues the execution of the Message Display task. Once it releases the resource, the mutex signals the Scheduler, and the latter 225 deescalates the priority of the Message Display task to its nominal priority level of Low. At this point, there are two tasks of a higher priority waiting to run, so the higher-priority waiting task (Waveform Draw) runs, and when it completes, the remaining higher-priority task (Switch Monitor) runs. When this last task completes, the Message Display task can finally resume its work and complete.

E. Problem Arise

As claimed by [5], despite the fact that this algorithm improve the previous algorithm, it still could produce some unnecessary blocking. This algorithm block a task at the time it attempt, before it actually require a resource [5]. It also says that - If a critical section is contained only in one branch of a conditional statement, then the task could be unnecessarily blocked, since during execution it could take the branch without the resource.

Another downside of this protocol according to [6] is that the deadlock could happen if a task that is currently accessing a mutually exclusive resource suspends it self.

VII. Conclusion

Acknowledgment

I would like to express my very great appreciation to Dr Stefan Henkler for his valuable and constructive suggestions during the planning and development of this research work. His willingness to give his time so generously has been very much appreciated.

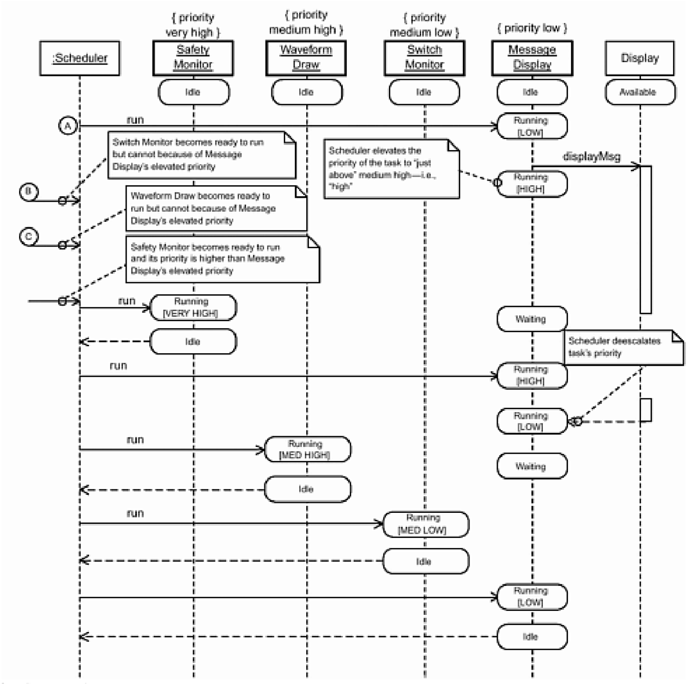
{ priority very high }
{ priority medium high }
{ priority medium low }
{ priority low }
:Scheduler
Safety Monitor
Waveform Draw
Switch Monitor
Message Display
Display
Idle
Available
A
B
C
run
Running [LOW]
Switch Monitor becomes ready to run but cannot because of Message Display's elevated priority
Scheduler elevates the priority of the task to "just above" medium high—i.e., "high"
displayMsg
Running [HIGH]
Waveform Draw becomes ready to run but cannot because of Message Display's elevated priority
Safety Monitor becomes ready to run and its priority is higher than Message Display's elevated priority
Running [VERY HIGH]
Waiting
Scheduler deescalates task's priority
Running [MED HIGH]
Running [MED LOW]

Fig. 12. Example of schedule under HLP [6]

References

[1] "IEEE Standard for a Real-Time Operating System (RTOS) for Small-Scale Embedded Systems," in IEEE Std 2050-2018 , vol., no., pp.1-333, 24 Aug. 2018, doi: 10.1109/IEEESTD.2018.8445674.
[2] "Real-time systems overview and examples," Intel. [Online]. Available: https://www.intel.com/content/www/us/en/robotics/real-time-systems.html. [Accessed: 04-May-2022].
[3] "Characteristics of real-time systems," GeeksforGeeks, 04-May-2020. [Online]. Available: https://www.geeksforgeeks.org/characteristics-of-real-time-systems/. [Accessed: 04-Apr-2022].
[4] "Terminology and notation," Terminology and Notation —. [Online]. Available: https://cmte.ieee.org/tcrts/education/terminology-and-notation/. [Accessed: 04-Apr-2022].
[5] G. C. Buttazzo, "Hard real-time computing systems: Predictable scheduling algorithms and applications". New York: Springer, 2011.
[6] B. P. Douglass, Real-time design patterns: Robust Scalable Architecture for real-time systems. Boston: Addison-Wesley, 2003.