Thread Carefully: Preventing Starvation in the ROS 2 Multithreaded Executor

Harun Teper, *Student Member, IEEE*, Daniel Kuhse, *Student Member, IEEE*, Mario Günzel, Georg von der Brüggen, Falk Howar, and Jian-Jia Chen, *Senior Member, IEEE*

***Abstract*—The robot operating system 2 (ROS 2) is a widely used collection of tools and libraries for building robot applications. It is designed to be flexible and easy to use when creating complex robot systems with many interacting components.Since its alpha version release in 2015, ROS 2 provides two options in a multithreading operating system, namely the single-threaded executor and the multithreaded executor. The single-threaded executor is starvation-free by design (i.e., every task is eventually executed) even in over-utilized systems, since the set of eligible task instances (called wait set) is only refilled once all the task instances in the wait set are executed. The multithreaded executor extends this mechanism to multiple threads that manage the wait set collaboratively. While intuitively this extension preserves the starvation-free property, and analyses for the multithreaded executor even build upon this assumption, the multithreaded executor has not been shown to be starvation-free.In this work, we examine the mechanism of the multithreaded executor in ROS 2 and demonstrate that it is prone to starvation, i.e., some tasks may never be executed even in under-utilized systems. This indicates risks for multithreaded executors in the current ROS 2 design and further leads to counterexamples to the state-of-the-art response-time analyses by Jiang et al. (RTSS 2022) and Sobhani et al. (RTAS 2023). We propose a minimal change in the software architecture of the ROS 2 multithreaded executor to enable starvation- and deadlock-free behavior. We empirically test that we prevent starvation in concrete ROS 2 system configurations, and show that our solution incurs a negligible overhead using the autoware reference benchmark. Moreover, we prove that our solution is starvation- and deadlock-free using formal proofs and model checking.**

***Index Terms*—Formal verification, multithreading, operating systems, robot programming.**

I. Introduction

In RECENT years, the complexity of robot systems has increased as more functionalities are incorporated. Such systems must be safe and reliable as well as efficient and scalable, which is a challenging tradeoff during the system design.

The robot operating system 2 [7] (ROS 2) was released in 2017 as the successor of ROS [8]. ROS 2 is a collection of tools and libraries for creating modular robot systems composed of many components, which are interconnected through the data distribution services (DDSs), a middleware standard for real-time communication in distributed systems. The ease of deployment is a key property of ROS 2, aiming to provide a structured solution to create complex robot systems.

In ROS 2, tasks are scheduled by so-called executors. An executor determines the set of eligible task instances, denoted as *wait set*, at a specific time point, called *polling point*, and the task instances in the wait set are then executed in the related *processing window*. The goal is to utilize the resources of the system whenever there are eligible tasks.

As the ROS 2 system is hosted by an operating system (OS) (e.g., Linux) the multitasking and multithreading features of the OS can also be adopted. There are two options (and also a hybrid of them) offered by ROS 2 to utilize multithreading of M threads since the release of its alpha version in 2015.

1) *Single-Threaded Executor:* M executors are created, each with one thread. The M executors *independently* manage their wait sets, polling points, and processing windows. Each task must be designated to one of the M executors. That is, tasks do not migrate between executors (threads).
2) *Multithreaded Executor:* One executor with M threads is created. At run time, the M threads *cooperatively* and *recurringly* determine one wait set at the global polling points to be processed in processing windows. The tasks assigned to the executor can be executed in any of the M threads, i.e., they can be executed on different threads.

The default ROS 2 single-threaded executor has been extensively studied, e.g., in response time analyses for the ROS 2 executor using DAG-based models [2], [3] and processing chains [11] as well as for end-to-end latency semantics [12], [13]. Furthermore, novel executor designs have been developed, e.g., PiCAS [4], the real-time executor [15], the rclc executor [10], and a dynamic-priority-based executor [1].

Compared to the default single-threaded executor, the multithreaded executor offers the advantages of synchronous polling points among the tasks and load balancing; yet, response-time analyses for the multithreaded executor are rather limited. In 2022, Jiang et al. [6] proposed the first

Manuscript received 13 August 2024; accepted 13 August 2024. Date of current version 6 November 2024. This work was supported in part by the German Federal Ministry of Education and Research (BMBF) in the Course of the 6GEM Research Hub under Grant 16KISK038, and in part by the Project (PropRT) that has received funding from the European Research Council (ERC) through the European Union's Horizon 2020 Research and Innovation Programme under Agreement 865170. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Jian-Jia Chen.)*

Harun Teper, Daniel Kuhse, Mario Günzel, Georg von der Brüggen, and Falk Howar are with the Department ofComputer Science, TU Dortmund University, 44227 Dortmund, Germany (e-mail: harun.teper@tu-dortmund.de; daniel.kuhse@tu-dortmund.de; mario.guenzel@tu-dortmund.de; georg.von-der-brueggen@tu-dortmund.de; falk.howar@tu-dortmund.de).

Jian-Jia Chen is with the Department of Computer Science, TU Dortmund University, 44227 Dortmund, Germany, and also with Lamarr Institute, 44227 Dortmund, Germany (e-mail: jian-jia.chen@tu-dortmund.de).

Digital Object Identifier 10.1109/TCAD.2024.3446865

response-time analysis. They showed that the latencies of some tasks assigned to the multithreaded executor may be higher compared to those on a single-threaded executor. Furthermore, they highlighted that tasks may be removed from the wait set. In 2023, Sobhani et al. [9] provided an analysis for constrained and arbitrary deadline systems with multithreaded executors and added priority-driven enhancements to improve response times.

Our Contributions: We examine the mechanism of the multithreaded executor in ROS 2, which has not been significantly changed since the ROS 2 alpha in 2015. Our contributions are as follows.

1) We show that the ROS 2 multithreaded executor is *prone to starvation*, i.e., a task is never executed even after an instance of it has already been put into the wait set. We provide concrete ROS 2 system configurations that suffer from starvation. These examples indicate the risks of multithreaded executors in current ROS 2 systems.
2) We propose design changes to the multithreaded executor to solve the starvation problem efficiently with minimal changes to the mechanism of the multithreaded executor. We empirically tested that the proposed design prevents starvation in all the system configurations in which we previously observed starvation. Moreover, we used the autoware reference system [14] benchmark as a real-world example to show that our solution incurs a negligible overhead. Furthermore, we formally prove starvation freedom and use model checking and formal proofs to ensure the proposed design is deadlock-free.
3) With the possibility of starvation in the existing ROS 2 multithreaded executor, we provide a counterexample to the response-time analyses by Jiang et al. [6] and Sobhani et al. [9] for the multithreaded executor. We note that we do not provide any updated response-time analysis in this article, as our focus is to ensure that the multithreading software architecture is starvation-free.

II. System Model

This section provides an overview of the ROS 2 system model considered throughout this article. We use ROS 2 Humble, the latest LTS release, as the reference version.

An ROS 2 system includes a set of $|\mathcal{N}|$ nodes, denoted as $\mathcal{N} = \{n_1, n_2, \ldots, n_{|\mathcal{N}|}\}$. Each node has a set of tasks, and each task is uniquely assigned to one node.

Tasks can be grouped into two categories: 1) time-triggered tasks, called timers and 2) event-triggered tasks, called subscriptions. Each task is associated with a function, called callback, which is executed when the task is scheduled. We use the terms task and callback interchangeably in this article. A timer is defined as a tuple $\text{tmr}_i = (T_i, C_i)$, where T_i is the period and C_i is the worst-case execution time (WCET) of the timer's callback. The utilization U_{tmr_i} of a timer is defined as C_i/T_i. A subscription is defined as a tuple $\text{sub}_i = (C_i)$, where C_i is the WCET of the subscription's callback.

Each task has an activation status that determines whether the executor samples a task instance for execution. Tasks with an active activation status are referred to as activated tasks. For subscriptions, the activation status is active if and only if the subscription buffer contains a message. The activation status of timers is activated at every integer multiple of the timer period since its start. The activation status for timers is deactivated whenever the executor selects the timer for execution.

ROS 2 provides a publish-subscribe communication mechanism through a DDS middleware for task communication. Any task can publish messages to a topic. These messages are received by and written to the buffers of the subscriptions subscribing to the topic. In ROS 2, subscriptions only subscribe to a single topic. We assume that one system core is exclusively dedicated to managing the OS level interrupts and DDS middleware.

Each ROS 2 system has a set of executors that are responsible for the execution of the tasks of the nodes. Each node is assigned to exactly one executor, and each task is indirectly assigned to an executor through the assignment of its node. Each task has a unique priority among the tasks assigned to an executor. For simplicity of presentation, when two tasks are assigned to the same executor, we assume that the task with the smaller index has higher priority. An ROS 2 executor can have one or multiple threads. All threads share the same set of nodes and manage the execution of the tasks assigned to the executor. For the simplicity of presentation, we assume that each thread is exclusively assigned to one core of the system.

ROS 2 uses callback groups to control the concurrent execution of callbacks of a node. Each callback in the node is assigned to one callback group. Moreover, each callback group has a type, either *reentrant* or *mutually exclusive*.

1) *Reentrant callback groups* allow the concurrent execution of callbacks that are in the group. Moreover, even the same callback can be executed in parallel by multiple threads if the schedule allows for it.
2) In contrast, *mutually exclusive callback groups* guarantee that at most one callback from the callback group can be executed at each point in time. Thus, other threads cannot sample and execute a callback from the same callback group until the currently executed one is finished.

III. Multithreaded Scheduling and Starvation

In this section, we introduce the ideas behind the general executor design in ROS 2, considering the default executor provided by ROS 2 Humble. While ROS 2 uses the same implementation for the single-threaded and the multithreaded executor, we start by explaining the design ideas and behavior for the single-threaded scenario and then extend to the multithreaded scenario. Afterward, we discuss how the multithreaded executor design can cause starvation for systems with mutually exclusive callback groups.

A. ROS 2 Executor Design

This section explains the underlying design philosophy behind the scheduling mechanism in the ROS 2 executor. We start with the general philosophy, based on the single-threaded executor, and see which problems arise when the performance

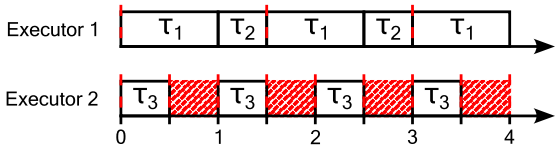

Fig. 1. Over-utilized and under-utilized single-threaded executor schedules. The red dashed lines indicate the polling points. In the schedule for Executor 2, a polling point is initiated at time 0.5 and the thread keeps polling throughout the dashed red idle interval until a new instance of τ_3 is eligible at time 1.

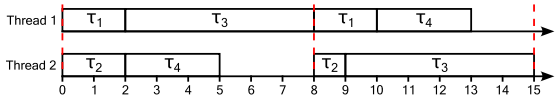

Fig. 2. Online load balancing when polling points are as late as possible.

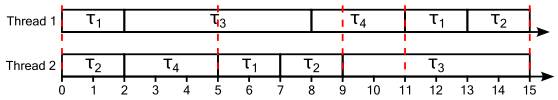

Fig. 3. Online load balancing when polling points are as early as possible.

is enhanced with the multithreaded executor. Technical details are discussed in Section IV while this section focuses on high-level explanations and concepts.

In ROS 2, tasks are uniquely assigned to and scheduled by *executors*. These executors are conceptually different from classical real-time schedulers (like earliest-deadline-first and static-priority scheduling for periodic and sporadic tasks), where each activated task instance is directly eligible to be scheduled. In contrast, an ROS 2 executor determines the set of eligible task instances, denoted as the *wait set*, at a specific time instance, called *polling point*, and the task instances in the wait set are then executed in the related *processing window*. Specifically, when a thread idles, the highest-priority task instance is removed from the wait set and scheduled. If no task instance in the wait set can be scheduled when a thread idles, the system continues with another polling point, followed by the related processing window, and so on.

The schedule in the processing window is work-conserving, nonpreemptive, and based on a given task-related priority order. At polling points, the executor collects at most one instance of each eligible task. Timers are eligible if they were activated since the final time they were selected for execution, and subscriptions are eligible if their buffer is nonempty.

This scheduling concept has advantages for the system specification. First, timers can be specified to be executed as frequently as possible (i.e., by setting the period to 0) without any knowledge about the system load. Second, this suggests that tasks are guaranteed to be executed every time they are eligible regardless of the system configuration (i.e., even when the system is overloaded, there is no starvation and every task is executed in the processing window following its activation). However, it should be noted that the timer period may not reflect its execution frequency, as the processing window may be longer than the timer period for overloaded systems.

For the remainder of this section, we call an executor *overloaded* if $\sum_{\tau_i} U_{\text{tmr}_i} > M$, and *underloaded* if $\sum_{\tau_i} U_{\text{tmr}_i} < M$, where M is the number of threads of the executor.

Example 1: For the overloaded single-threaded executor in the top row of Fig. 1 (with two timers $\tau_1 = (1, 1)$ and $\tau_2 = (1, 0.5)$), both tasks are executed and there is no starvation (which would occur under static-priority scheduling if τ_1 has higher priority). However, no task is executed as often as intended (on average every 1 time unit) due to overload.

Multiprocessor systems allow load balancing to avoid such overloaded scenarios. For instance, if there is an additional underutilized executor with only one assigned timer $\tau_3 = (1, 0.5)$ (like Executor 2 in Fig. 1), τ_2 could be moved to that executor, and all the tasks would be executed as often as intended. That is, the load balancing is done offline by assigning tasks to executors based on their WCET and period.

ROS 2 supports online load balancing in multiple processor scheduling with the multithreaded executor. After a polling point, the multithreaded executor schedules the task instances in the wait set by assigning the highest-priority task instance in the wait set to a thread as soon as a thread idles.

The multithreaded executor also solves another potential problem that can arise when using multiple single-threaded executors. For instance, we can observe in Fig. 1 that the polling points of the executors are not synchronous, even if all timer periods are equal. Yet, synchronicity might be necessary, for example, to ensure that all the tasks in a processing window use the sensor data that is collected at the same point in time.

There are two obvious options for the realization of multithreaded executors. One is to start the next polling point *as late as possible*, i.e., when the current processing window finishes. Another is to start the next polling point *as soon as possible*, i.e., immediately when one thread idles.

Example 2: Design Option 1 - as Late as Possible: Fig. 2 shows two consecutive processing windows for a set of four timers [$\tau_1 = (5, 2)$, $\tau_2 = (5, 2)$, $\tau_3 = (5, 6)$, and $\tau_4 = (5, 3)$] scheduled by a multithreaded executor with two threads. In the first window, all the task instances execute according to their WCET. In the second window, τ_2 finishes its execution early (i.e., after 1 time unit) and τ_3 can be scheduled on Thread 2 at time 9, resulting in a shortened processing window.

The example in Fig. 2 shows that idle time can still occur at the end of the processing window if the next polling point only starts once all threads idle. Yet, this idle time can also be utilized if a polling point is started as soon as a thread idles and there is no eligible task instance left in the wait set (e.g., at time 5 or 13 in Fig. 2).

Example 3: Design Option 2 - as Soon as Possible: Fig. 3 shows the updated schedule from Fig. 2 when the polling points start as early as possible. At the polling point at time 5, only instances of τ_1, τ_2, and τ_4 are added to the wait set, since an instance of τ_3 is executing at time 5. (Similarly, τ_4 is not added to the wait set at time 9.) Note that, the threads never idle as long as task instances are eligible at each polling point.

Although it seems that the as-late-as-possible design option is pessimistic, it ensures that all the task instances in the wait set collected at a polling point are executed in the subsequent processing window. The as-soon-as-possible design option

however starts to collect a new wait set even when some task instances in the old wait set have not been fully executed yet. As a result, processing windows may overlap with each other. For instance, in Fig. 3, the instance of τ_3 executed from 2 to 8 is part of the first processing window (starting at time 0) while the parallel executed instance of τ_1 is part of the second processing window (starting at time 5).

Note that, we presented these two options as if ROS 2 users would be able to choose between them. However, ROS 2 only provides the as-soon-as-possible option for multithreaded executors since its alpha version in 2015.

B. Starvation in the Multithreaded Executor

The previously described design of the ROS 2 multithreaded executor maximizes the throughput and is, at first glance, starvation-free. The main reason is that polling is only initiated when there are no more task instances in the wait set that can be scheduled. Therefore, intuitively, all the task instances have started executing (and may be finished) when a new polling point is initiated. However, the ROS 2 multithreaded executor design can result in starvation for mutually exclusive callback groups. In the following, we detail this problem based on examples, while a detailed analysis of the ROS 2 executor is discussed in Section IV and a fix is provided in Section VI.

Assigning callbacks to a mutually exclusive callback group ensures task instances of these callbacks cannot be executed concurrently. This feature, for instance, guarantees data consistency for shared variables. If the highest-priority task instance in the wait set is part of the same mutually exclusive callback group as a task instance that is currently executing, it is blocked by the executing task instance and ignored during the scheduling process; the instance with the second-highest priority is scheduled instead (if it is not part of the same mutually exclusive callback group as a task instance that is currently executing, etc.). Meanwhile, scheduling the highest-priority callback (and all other callbacks that cannot be executed due to the mutually exclusive callback groups) is postponed until the blocking task instance finishes executing.

To ensure high throughput, the ROS 2 multithreaded executor initiates a polling point once a thread idles and no task instance can be scheduled. In this situation, some task instances may still be part of the wait set (i.e., not yet scheduled), since they are blocked from executing due to a mutually exclusive callback group. However, at the start of a polling point, ROS 2 clears the wait set (that is, ROS 2 removes all the task instances from the wait set). Afterward, it fills the wait set with all task instances from groups that are not blocked. This process potentially adds additional task instances from unblocked groups to the wait set that have a higher priority than the instances currently blocked due to mutual exclusion, which itself can postpone executing these blocked instances. Even more, at each polling point, instances of tasks that are currently executing or blocked by an executing callback (due to a mutually exclusive callback group) are not added to the wait set. They may be added to the wait set at a later polling point; yet, since the callbacks are executed according to the same static-priority order in each processing window, these task instances may again not be executed due to higher-priority task instances in the same mutually exclusive callback group that were added at the polling point. This process may continue indefinitely, and the task instance is never executed.

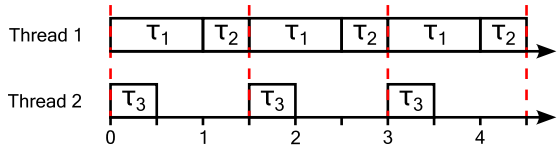

Fig. 4. Polling as late as possible guarantees starvation freedom even for over-utilized callback groups.

Throughout this article, we refer to a callback group as *over utilized* if $\sum_{\tau_i} U_{\text{tmr}_i} > 1$, *fully utilized* if $\sum_{\tau_i} U_{\text{tmr}_i} = 1$, and *under utilized* if $\sum_{\tau_i} U_{\text{tmr}_i} < 1$. Since the callbacks in a callback group must be sequentially executed, it is impossible to execute the callbacks as often as expected according to their periods when the callback group is over utilized.

We now provide examples showing that the multithreaded executor in ROS 2 is indeed not starvation-free. These examples cover different scenarios to show that starvation does not only happen in corner cases but in a wide range of scenarios. We first consider the most intuitive scenario when the callback group is over utilized. Second, we show that starvation also can happen when the callback group is under utilized. Third, we show that recurrent activation of only one callback in a mutually exclusive callback group is sufficient to lead to the starvation of other callbacks in the group; even if the overall system utilization is very low. We implemented[1] and tested all the example systems, and the described behavior corresponds to the actual behavior of the multithreaded executor.

Starvation With Over-Utilized Callback Groups: We show a starvation example, where we configure the system such that a callback group is over utilized while the whole system can be either under utilized or over utilized.

Example 4: Consider a multithreaded executor with two threads and three timers $\tau_1 = (1, 1)$, $\tau_2 = (1, 0.5)$, and $\tau_3 = (1, 0.5)$, where $g = \{\tau_1, \tau_2\}$ is a mutually exclusive callback group. If the polling point is initiated after all the instances finish executing, this system can be scheduled without starvation, as shown in Fig. 4. Specifically, τ_1 is scheduled at time 0 since it has the highest priority and, due to the mutually exclusive callback group with τ_1, τ_2 cannot be scheduled and τ_3 takes priority. In the ROS 2 multithreaded executor, a polling point is initiated as soon as Thread 2 idles (i.e., at time 0.5), and at the related polling point, the wait set is cleared. However, τ_2 is not added back to the wait set since τ_1 is executing. At time 1, τ_1 and τ_3 release a new task instance, the executor collects these two instances as well as an instance of τ_2 and adds them to the wait set. The same schedule as before repeats indefinitely, as shown in Fig. 5.

The above example works for any WCET of τ_2, so we have an example for under utilized ($C_2 < 0.5$), fully utilized

[1] https://github.com/tu-dortmund-ls12-rt/ROS2-MT-Starvation-Examples

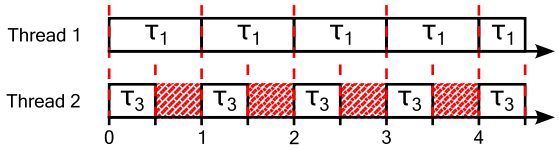

Fig. 5. Starvation for τ_2 in the multithreaded executor when the mutually exclusive group $g = \{\tau_1, \tau_2\}$ is over utilized.

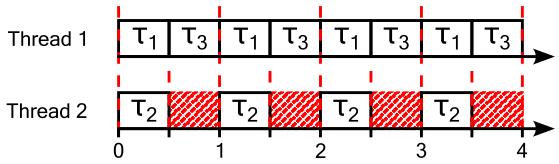

Fig. 6. Starvation for τ_4 in the multithreaded executor when the mutually exclusive group $g = \{\tau_3, \tau_4\}$ is under utilized.

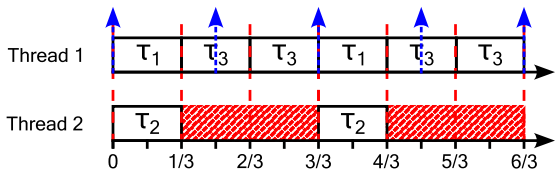

Fig. 7. Starvation for τ_4 in the multithreaded executor for the mutually exclusive callback group $g = \{\tau_3, \tau_4\}$ due to the frequent activation of τ_3.

($C_2 = 0.5$), and over-utilized systems ($C_2 > 0.5$). However, the callback group $g = \{\tau_1, \tau_2\}$ is over utilized if $C_2 > 0$.

Starvation With Under-Utilized Callback Groups: While it is rather intuitive that starvation can happen when the system or a callback group is over utilized, starvation also occurs when both the system and the callback groups are under utilized.

Example 5: Consider four timers $\tau_1 = (1, 0.5)$, $\tau_2 = (1, 0.5)$, $\tau_3 = (1, 0.5)$, and $\tau_4 = (1, 0.01)$, where $g = \{\tau_3, \tau_4\}$ is a mutually exclusive callback group. Since τ_1 and τ_2 occupy both threads in parallel, τ_4 is removed from the wait set once τ_3 starts executing, as shown in Fig. 6.

Activation-Induced Starvation: Even if the overall system load is very low, starvation can occur due to the frequent activation of one task in the group.

Example 6: Consider a system with four timers $\tau_1 = [1, (1/3)]$, $\tau_2 = [1, (1/3)]$, $\tau_3 = [(1/2), (1/3)]$, and $\tau_4 = (5, 0.01)$, where $g = \{\tau_3, \tau_4\}$ is a mutually exclusive callback group. Since τ_1 and τ_2 occupy both processors in parallel, τ_3 is only started at time $(1/3)$. Multiple polling points are initiated between time $(1/3)$ and time $(2/3)$ since Thread 2 idles (with the first of these polling points removing the instance of τ_4). At time $(2/3)$, the polling point adds an instance of τ_3 (which was activated at time $(1/2)$ but not yet added to the wait set as another task instance is executing) and an instance of τ_4 to the wait set. The instance of τ_3 is assigned to Thread 1 since it has higher priority than the instance of τ_4. Thus, τ_4 is starved, as shown in Fig. 7.

Example 6 can easily be extended to show starvation in a system with utilization close to 1, even if $M \to \infty$. Specifically, assume M tasks $\tau_1 = \tau_2 = \ldots = \tau_M = (X, \varepsilon)$ where X is arbitrarily large and $\varepsilon > 0$ is arbitrarily small. In addition, assume two tasks $\tau_{M+1} = (X, X - \varepsilon)$ and $\tau_{M+2} = (X, \varepsilon)$ in a mutually exclusive callback group $g = \{\tau_{M+1}, \tau_{M+2}\}$. As a result, τ_{M+2} will never be executed.

Summary: Starvation can occur in the ROS 2 multithreaded executor in a variety of situations; even if the system load is small and when the mutually exclusive callback groups are under utilized. All provided examples also work when replacing the starved timer with a subscription.

IV. Structure of the ROS 2 Executor

In this section, we discuss details of the ROS 2 executor mechanism, while the underlying concepts have been introduced in Section III-A. The same implementation is utilized for both the single-threaded and multithreaded executors; thus; part of the functionality is not necessary in the single-threaded executor. We provide an overview of the mechanism in the flowchart in Fig. 8. Instead of providing details on every state, we elaborate on the progress of the executor and its threads for processing windows and polling points.

The flowchart in Fig. 8 consists of nine abstract states, marked by gray boxes. Each rounded rectangle represents an operation that the executor performs. Blocks in the same color access or modify the same resource, such as a mutex, a variable, or a data structure (excluding the wait set). Uncolored boxes indicate the operation is not resource specific. The parallelograms represent if-else blocks, where the nonfilled circles indicate the path taken if the condition is true and the filled circle the path if the condition is false. One main aspect of the ROS 2 executor design is that the wait set is shared among all the threads of an executor. Hence, ROS 2 uses multiple shared resources to ensure the data consistency in the wait set.

A. Processing Windows

We first explain how the task instances are executed when starting from State (*1-Idle*). The sequence of states (or path) <(*1-Idle*),(*2-Init*),(*3-Sample*),(*7-Take*),(*8-Run*),(*9-Finish*)> corresponds to a thread executing a task instance during a processing window. If there is no task instance in the wait set that can be executed, a polling point is initiated in (*3-Sample*) (which we discuss later when detailing polling points).

The general sequence of operations for the *single-threaded executor* is: starting from (*1-Idle*), the thread determines which task instance to execute (*3-Sample*), removes it from the wait set (*7-Take*), executes this task instance (*8-Run*), and returns to (*1-Idle*). (*2-Init*), (*4-Status*), (*6-Return*), and (*9-Finish*) are only relevant in the multithreaded executor.

For the *multithreaded executor*, the general process is similar, but it must be ensured that 1) there are no consistency issues in the wait set and 2) task instances of callbacks in a mutually exclusive group are not executed at the same time.

1) To ensure consistency, the *wait set mutex* (orange) prevents concurrent access to the wait set. If the mutex is locked and owned by a thread, other threads that try to acquire the lock are blocked until the current thread releases the lock. Consequently, the *wait set mutex* is locked in (*2-Init*) [i.e., before the wait set is accessed to determine which task instance to execute in

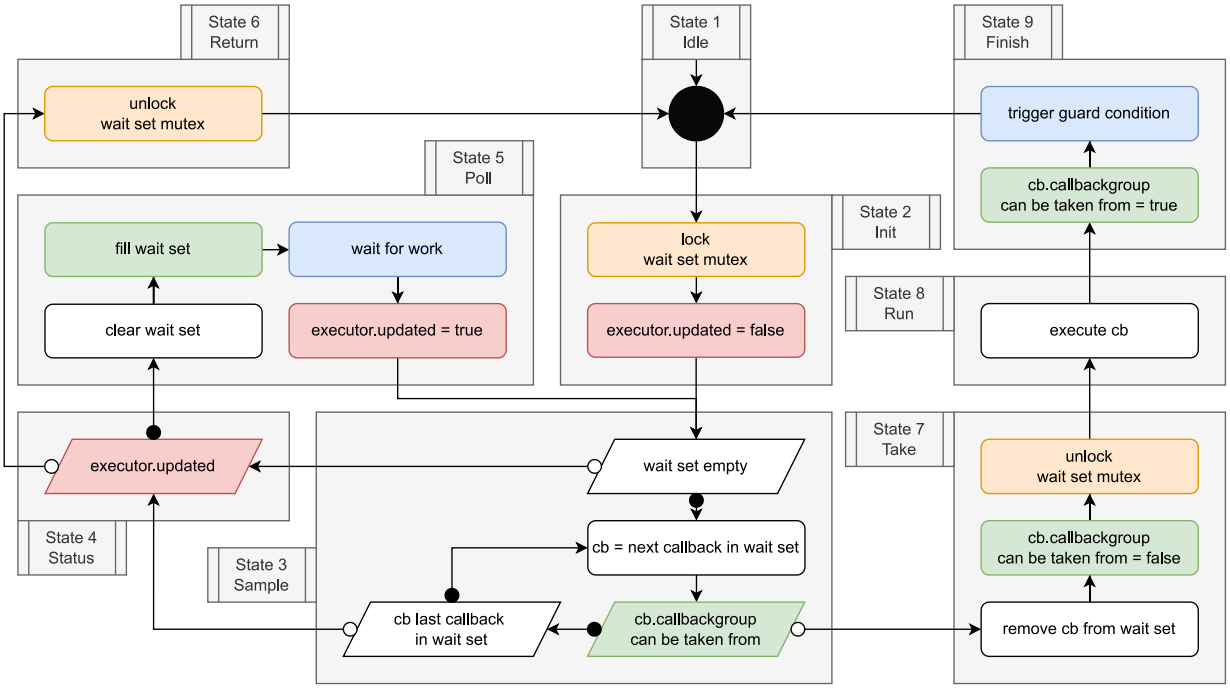

Fig. 8. ROS 2 Executor. A thread goes through <(*1-Idle*),(*2-Init*),(*3-Sample*),(*7-Take*),(*8-Run*),(*9-Finish*)> when executing a task instance during the processing window. If no task instance can be taken from the wait set in (*3-Sample*), the thread initiates a polling point via <(*3-Sample*),(*4-Status*),(*5-Poll*),(*3-Sample*)>, followed either by executing a task instance (i.e., continuing to (*7-Take*)) or returning to idle via (*6-Return*) (i.e., if no task instance is added to the wait set).

(*3-Sample*)]. Thus, only one thread initiates a polling point [<(*3-Sample*), (*4-Status*), (*5-Poll*), (*3-Sample*)>] or removes a task instance from the wait set to schedule it [<(*3-Sample*), (*7-Take*)>] at each point in time. All threads can execute task instances [i.e., be in (*8-Run*)] simultaneously.

2) To ensure that task instances in a mutually exclusive callback group are not executing in parallel, each callback group has a *can_be_taken_from* flag (green), indicating if a task instance of a callback in the group is being executed. Specifically, in (*3-Sample*), if a task instance has the highest priority in the wait set and *can_be_taken_from* of its group is false, the task instance is ignored, and the thread checks for the task instance with the next lower priority. Otherwise, if the task instance has the highest priority in the wait set and *can_be_taken_from* is true, the task instance is chosen for execution in (*3-Sample*). After it is removed from the wait set, *can_be_taken_from* is set to false in (*7-Take*). Note that, in (*7-Take*) the flag is set before the *wait set mutex* is unlocked to ensure that only one callback of the group can be executed at any time. After the task instance is executed in (*8-Run*), *can_be_taken_from* is set to true in (*9-Finish*), which allows polling and scheduling callbacks from the group again.

B. Polling Points

So far, we discussed how task instances are scheduled during processing windows in the single-threaded and in the multithreaded executor. If a thread finds no task instance to schedule in (*3-Sample*), the thread initiates a polling point via <(*4-Status*), (*5-Poll*), (*3-Sample*)>. This situation can occur if i) the wait set is empty (the only situation possible in the single-threaded executor) or 2) all task instances in the wait set are part of callback groups and one callback in each of these groups is currently executing. In this case, the thread may check multiple task instances in the inner loop of (*3-Sample*), each in a mutually exclusive group with a currently executing task instance, before progressing to (*4-Status*). Hence, at the time of reaching (*5-Poll*), the wait set may contain multiple blocked task instances due to mutual exclusion.

At each polling point, to update the wait set, a thread loops through <(*3-Sample*), (*4-Status*), (*5-Poll*), (*3-Sample*)>. The *executor.updated* flag (red) tracks whether the wait set was updated since the *wait set mutex* was set and ensures the *wait set mutex* is unlocked either in (*6-Return*) or (*7-Take*).

We specify the purpose of *executor.updated* later and, for now, assume *executor.updated* is false. In this situation, a thread entered (*3-Sample*) from (*2-Init*), could not find any task instance to schedule, and continued to (*4-Status*) and (*5-Poll*).

State 5 (Poll): Since the main problems arise here, we take a closer look at the polling operation in (*5-Poll*).

(A) The wait set is *cleared*. If the polling point was initiated because the wait set was empty, nothing changes. Otherwise, task instances blocked due to mutually exclusive callback groups are removed from the wait set.

(B) The wait set is *filled* with instances of all eligible callbacks, i.e., instances of callbacks whose group flag *can_be_taken_from* is set to true when polling. If a running callback is part of a mutually exclusive callback group, no instance of any callback in this group is added to the wait set. In particular, a callback can block itself from being added to the wait set.

(C) The thread *waits for work*. Specifically, it waits until one of the three following conditions becomes true.
  (i) A callback in the wait set is activated. This condition can be true directly when the wait set is filled — if at least one timer or subscription is already activated.
  (ii) A callback executing in another thread finishes and triggers the guard condition in (*9-Finish*).
  (iii) The executor-specific timeout value elapses (which is by default infinite in ROS 2 Humble).

  Afterward, the executor removes all the nonactivated task instances from the wait set.
(D) The *executor.updated* flag is set to true.

While (A), (B), and (D) are rather straightforward, we now take a closer look at the three conditions in (C) *wait for work*.

If *condition (i)* is triggered in *wait for work*, the wait set is not empty after all the nonactivated task instances are removed. Since not all the task instances in the wait set are in mutually exclusive groups with a task instance currently running (and therefore not blocked by their group), at least one task instance can be scheduled, and the thread continues to (*7-Take*).

If *conditions (ii)* or *(iii)* are triggered, the wait set is empty after all the nonactivated task instances are removed. Thus, the thread leaves (*3-Sample*) via (*4-Status*) and, since *executor.updated* is true, continues to (*6-Return*) and (*1-Idle*). Specifically, *executor.updated* ensures that a thread cannot enter (*5-Poll*) twice after acquiring the *wait set mutex*.

C. Starvation

The starvation issues result from the mutually exclusive callback groups in combination with the way polling points are performed. Specifically, task instances are removed from the wait set when callbacks of the same group are executing and (in some cases) only added back together with higher-priority task instances of the same group — which block them again.

The common property of Examples 4–6 is that one blocked callback from a mutually exclusive group is removed from the wait set. Later on, the removed callback is only added back once the other callback in the group has finished executing. Hence, the higher-priority callback is added as well and can starve the lower-priority callback indefinitely.

V. Starvation and Response Time Analysis

In this section, we present a counterexample to the response-time analyses of the ROS 2 multithreaded executor by Jiang et al. [6] and Sobhani et al. [9]. We demonstrate that these analyses return a bounded value even when the actual response time of a task is unbounded.[2] We note that Jiang et al. [6] discuss how the tasks in the wait set are changed by the executor and observed that a task may be postponed multiple times due to blocking by higher-priority tasks when the task is repeatedly removed. They concluded that the multithreaded executor may not always result in better performance than the single-threaded executor due to this behavior. However, it is not mentioned that this may result in starvation and the observation is not included in the analysis provided in [6].

We again consider the system in Example 6 and Fig. 7, with three timers $\tau_1 = [1, (1/3)]$, $\tau_2 = [1, (1/3)]$, $\tau_3 = [(1/2), (1/3)]$, and one subscription $\tau_4 = (0.01)$, where $g = \{\tau_3, \tau_4\}$ is a mutually exclusive callback group and tasks τ_1 and τ_2 are in individual mutually exclusive callback groups. The system is managed by a multithreaded executor with two threads.

Fig. 7 shows that task τ_4 is never executed. However, we show that the analyses by Jiang et al. [6] and Sobhani et al. [9] provide a bounded response time for the task τ_4.

Both analyses are intended for chains, which are sequences of tasks. We consider three chains $\Gamma_1 = (\tau_1)$, $\Gamma_2 = (\tau_2)$, and $\Gamma_3 = (\tau_3, \tau_4)$ identical to the callback groups, and focus on Γ_3 for the counterexamples.

A. Response-Time Analysis by Jiang et al. [6]

According to Theorem 1 by Jiang et al. [6], given a chain and its constrained deadline (with respect to the period of the first task in the chain), the response time is bounded if the workload L to execute the full chain is less than the deadline.

We directly apply the results by Jiang et al. [6] to the example system above, with a deadline of 0.5 for the chain Γ_3. Using [6, Th. 1], we determine that the workload corresponds to $L = (1/3)$ plus the workload of task τ_4. Furthermore, $L + \text{WCET}(\tau_4) \approx 0.334 + 0.01 = 0.344$ is less than the deadline of 0.5, and the response time of task τ_4 is bounded according to Theorem 1 by Jiang et al. [6]. However, as starvation occurs for task τ_4, the response time of τ_4 is unbounded, contradicting the analysis, as shown in Fig. 7.

B. Response-Time Analysis by Sobhani et al. [9]

According to Theorem 5 by Sobhani et al. [9], the response time of a chain is upper bounded, if the demand bound function $dbf(\Delta)$ is less than the supply bound function $sbf(\Delta)$ for any time interval Δ [where $dbf(\Delta)$ is the total amount of work that is required to be executed in Δ and $sbf(\Delta)$ is the total amount of work that can be executed in Δ].

We directly apply the result by Sobhani et al. [9] to the example system above and choose the time interval $\Delta = 1$. The supply bound function for this system is $sbf(\Delta) = 2$ according to [9, Definition 1]. The demand bound function is $dbf(\Delta) = (4/3)$ according to [9, Th. 5]. Therefore, the $dbf(\Delta)$ for Γ_3 is less than the $sbf(\Delta)$ of the multithreaded executor, and the response time of task τ_4 is bounded according to Theorem 5 by Sobhani et al. [9]. However, as shown in Fig. 7, starvation occurs for the task τ_4, and the response time of τ_4 is unbounded, contradicting the analysis.

VI. Starvation-Free Multithreaded Executor

In this section, we propose a design for the multithreaded executor that is starvation-free, tries to maximize the throughput, and keeps the general structure of the original design. We introduce each new part of our solution step-by-step, highlighting what changes and which issues still need to be

[2] Jiang et al. [6] and Sobhani et al. [9] have been informed about this bug in July 2024, and they intend to fix their analyses and make new versions available online.

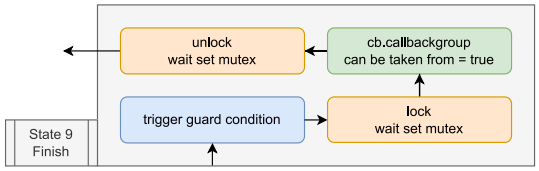

Fig. 9. Mutex lock proposal to prevent callback group flag race conditions.

addressed until we reach the final design. Finally, we show that our design prevents starvation in all the system configurations from Section III-B in which we previously observed starvation.

As mentioned in Section IV-C, starvation occurs because blocked callbacks are removed from the wait set and only added back together with higher-priority callbacks of the same group. This leads to the situation where a callback is never executed, since it is blocked indefinitely by these higher-priority callbacks.

A. Intuitive Starvation-Free Approach

An intuitive approach to avoid starvation may be as follows.

1) *Preventing Callback Removal:* We modify *clear wait set* to not remove any blocked callbacks from the wait set. Hence, no additional polling is required to add them back later. Preventing removal is not sufficient to avoid starvation due to a race condition for the callback group flag. Specifically, another thread in (*9-Finish*) may set a callback group's flag to true, right before the thread in (*5-Poll*) clears the wait set, which leads to removing the callbacks of the group from the wait set.
2) *Prevent Callback Group Flag Race Condition:* To prevent this, we add a critical section to (*9-Finish*), which guards the callback group flag. The updated (*9-Finish*) is depicted in Fig. 9. The existing *wait set mutex* protects the callback group flag. We trigger the guard condition before we lock the mutex. Otherwise, a thread in *wait for work* that holds the *wait set mutex* would be blocked indefinitely as it is waiting for the guard condition to be triggered (when the wait set is empty).

This intuitive solution is not deadlock-free. Assume a multithreaded executor with two threads that manages a callback group with two tasks, τ_1 and τ_2. The following sequence of events leads to a deadlock.

1) The wait set only includes τ_1. Both threads are idle.
2) Thread 1 executes τ_1, blocking the callback group.
3) Thread 2 acquires the *wait set mutex*, starts polling, and cannot add τ_2 to the wait set, as the callback group is blocked. Thus, the wait set is empty when reaching *wait for work*. Thread 2 is blocked and waits for a guard condition to be triggered.
4) Thread 1 finishes executing τ_1 and triggers the guard condition in (*9-Finish*), notifying thread 2.
5) Thread 2 continues and returns to the idle state via <(*5-Poll*),(*3-Sample*),(*4-Status*),(*6-Return*),(*1-Idle*)>, releasing the *wait set mutex*.
6) Thread 2 moves to (*2-Init*), acquires the *wait set mutex* and moves from (*3-Sample*) to (*4-Status*) because the wait set is empty. As the callback group is blocked, the wait set remains empty after *fill wait set*. In *wait for work*, it waits for a guard condition to be triggered.
7) Thread 1 tries to lock the *wait set mutex* in (*9-Finish*).

Now, Thread 2 is blocked, waiting for the guard condition to be triggered by Thread 1 while holding the *wait set mutex*. On the other hand, Thread 1 cannot acquire the *wait set mutex* in (*9-Finish*), as it is held by Thread 2, resulting in a deadlock.

B. Our Improved Deadlock-Free Solution

To avoid a deadlock for the critical section in (*9-Finish*), we propose to use an additional mutex for the callback group flag. We depict the new structure of the flowchart in Fig. 10.

Compared to the original design, *clear wait set* in (*2-Init*) is replaced with the *conditional clear wait set*, which only removes nonblocked callbacks from the wait set, as described in *preventing task removal*. Furthermore, the states (*2-Init*), (*5-Poll*), (*7-Take*), and (*9-Finish*) are modified. They now include a new mutex, called *notify mutex*, which is responsible for protecting multiple threads from accessing operations that access or modify the callback group flags.

This changes polling points, i.e., a thread looping through <(*3-Sample*), (*4-Status*), (*5-Poll*), (*3-Sample*)>, as follows.

1) The *notify mutex* is locked in (*2-Init*). Other threads are blocked from entering (*9-Finish*) until it is unlocked.
2) The executor determines the state of the wait set and callback group flags in (*3-Sample*).
3) If the thread moves to (*5-Poll*), the state of the callback group flags is unchanged since being in (*3-Sample*). Hence, *clear wait set* and *fill wait set* are executed based on the state of the callback group flags in (*3-Sample*).

In (*7-Take*), the *notify mutex* is either already unlocked if the thread was in (*5-Poll*) since the final *wait set mutex* lock. If the thread was not in (*5-Poll*), the *notify mutex* is locked, and the *notify mutex* is unlocked in (*7-Take*).

Preventing Busy Waiting: We need to account for one more technical detail in our solution. Specifically, during polling, if a blocked callback is in the wait set, the *wait for work* operation immediately unblocks, as the blocked callback in the wait set is already activated. This can lead to busy waiting, as a thread would constantly initiate and finish a polling point, without changes in the state of the wait set and the callback groups.

We prevent this by modifying the *conditional clear wait set* operation and the *fill wait set* operation. Instead of waiting for all the callbacks in the wait set during *wait for work*, we only wait for the callbacks that are added by the previous *fill wait set* operation. These callbacks are not blocked, and only an activation of one of these callbacks can lead to a change in the wait set. Likewise, only when the guard condition is triggered, *wait for work* unblocks, as this is the earliest point at which one of the previously blocked callbacks can be executed.

C. Evaluation and Outlook

We *implemented and tested* our proposed design using all the systems described in Section III-B for which we encountered starvation with the default executor. We ran each system five times for 10 min and did not observe any

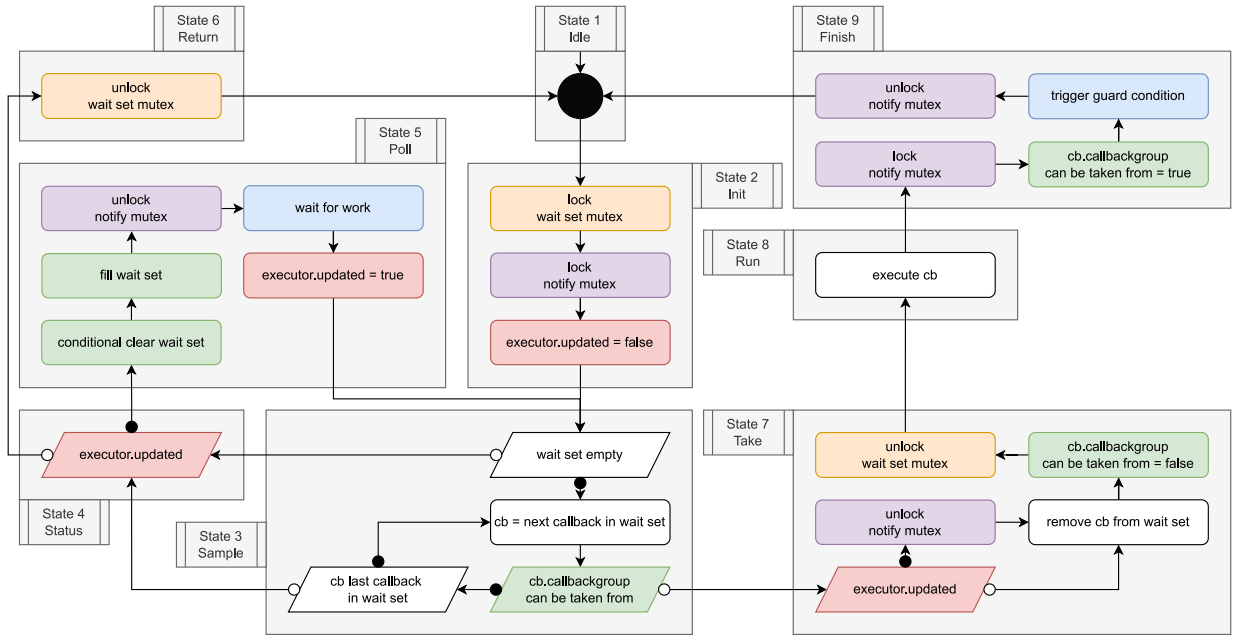

Fig. 10. Proposed multithreaded executor fix, using mutex locks for callback group flags and notifications.

TABLE I
OVERHEAD COMPARISON WITH THE AUTOWARE REFERENCE SYSTEM

Type	Mean [ms]	Std [ms]	99th [ms]
Baseline	0.215	0.063	0.399
Ours	0.216	0.060	0.376

starvation, i.e., all tasks were executed. However, due to the amount of work per callback group and the resulting blocking time, not all timers were executed as often as specified by their period.

Furthermore, we ran the autoware reference system [14] benchmark to evaluate the overhead of our proposed design. We ran the system on Ubuntu 22.04 running ROS 2 Humble, using an AMD Ryzen 5900x with 32 GB of RAM.

Table I shows the combined results of ten 10-min runs, including the mean, standard deviation, and 99th percentile. The overhead of our proposed design is comparable to the baseline of the original multithreaded executor of ROS 2.

We note that our proposed design is only one possible solution to provide starvation freedom. There are other feasible solutions, each with its own behavior. For example, in our design, the callback group flags are not necessarily guarded by the *notify mutex* in (*3-Sample*), if the mutex has been unlocked in (*5-Poll*). Furthermore, different solutions may lead to different amounts of blocking time for callback group tasks. In this article, we do not focus on such design choices, but rather on what is necessary to ensure starvation freedom. *For the future redesign of the multithreaded executor, such design choices need to be evaluated*, including the overhead for polling, as well as the consequences for the overall performance of the system, e.g., the additional blocking time for callback groups and the latency of the system.

In summary, we have proposed a new design of the ROS 2 multithreaded executor, which keeps the general structure of the original design, and tested it to be starvation-free. We prove that our design is deadlock-free and starvation-free in Section VII. Our design is more efficient than the design option as late as possible mentioned in Section III, as threads immediately update the wait set if no work is available.

VII. PROPERTIES OF THE EXECUTOR EXTENSION

In this section, we prove that our executor model based on Fig. 10 in Section VI-B is deadlock-free and starvation-free. We first describe how threads interact with the operations as follows.

1) At each time, a thread *executes* an operation, *waits*, or *progresses* to the next operation.
2) *Execution:* Usually, an operation takes a certain time to be executed by the thread, and afterward the thread is eligible to progress to the next operation. An exception is *wait for work* (discussed below).
3) *Wait:* When the thread is eligible to progress to an operation that locks a mutex, but the mutex is already owned by another thread, then the thread *waits*. Similarly, the thread waits at *wait for work* until either a callback in the wait set is activated, the timeout runs out, or another thread executes *trigger guard condition*.
4) *Progress:* When a thread moves from one operation to the next one, we say that the thread makes *progress*.

In the following, we prove deadlock freedom and starvation freedom regardless of the timeout value of the *wait for work* operation, i.e., either a callback is activated or *trigger guard condition* is executed within a finite amount of time.

Furthermore, we make the following *assumptions*.

1) The execution of any operation takes only finite time > 0.
2) The system has finitely many threads (at least one).

3) The system has finitely many callbacks (at least one).

First, we prove properties for the design with two mutexes.

Lemma 1: For the proposed executor, the following properties hold.

(i) A thread that owns the *wait set mutex* must be in states (*2-Init*)–(*7-Take*).
(ii) A thread that owns the *notify mutex* must be in states (*2-Init*)–(*7-Take*) or in (*9-Finish*).
(iii) At any time, at most one thread is in (*2-Init*)–(*7-Take*).
(iv) At any time, at most one thread is in (*9-Finish*). If there is one, it holds the *notify mutex*.
(v) A thread in *fill wait set* owns both mutexes.

Proof: (i) and (iii) hold because a thread has to lock the *wait set mutex* when it enters (*2-Init*), and unlocks the *wait set mutex* when it leaves states (*6-Return*) or (*7-Take*). Similarly, (iv) holds because a thread entering (*9-Finish*) locks the *notify mutex*, and unlocks only when leaving (*9-Finish*).

For (ii), we need to show that a thread that owns the *notify mutex* cannot be in (*1-Idle*), (*6-Return*), and (*8-Run*). We observe that a thread in (*4-Status*) or (*7-Take*) that locks the *notify mutex* must have the parameter *executor.updated* = *true*. Therefore, it can only progress to states (*6-Return*) or (*8-Run*) if the thread does not own the *notify mutex*. Consequently, it also reaches (*1-Idle*) only if it does not own the *notify mutex*.

For (v), when a thread enters (*5-Poll*), *executor.updated* must be set to *true*. Therefore, at that time, it must own the *notify mutex*. Moreover, the thread must hold the *wait set mutex* because when reaching (*5-Poll*), the thread must have locked the mutex in (*2-Init*) without unlocking afterward. Hence, the thread holds both mutexes. ■

A. Deadlock Freedom

In this section, we show that our proposed executor design for the multithreaded executor is deadlock-free. That is, no deadlock state (specified below) can occur.

Definition 1 (Deadlock State): A set of threads is in a deadlock state if every thread in the set is waiting and can only be released by another thread in the set.

We observe that a thread ξ can only wait for another thread ϕ in one of two cases.

1) Thread ξ waits for a mutex (*wait set mutex* or *notify mutex*) that is held by ϕ.
2) Thread ξ is executing *wait for work* and no callback was added by ξ during *fill wait set*. In that case, ξ cannot be released by a callback activation, but waits for ϕ to enter *trigger guard condition*.

We first derive some properties for the *notify mutex* and the *wait for work* operation, before proving deadlock freedom.

Lemma 2: A thread ξ holding the *notify mutex* cannot wait for another thread.

Proof: We prove this lemma by contradiction. Assume that ξ waits. Since ξ already holds the *notify mutex*, it can only wait for the *wait set mutex* or be in *wait for work*.

If it waits for the *wait set mutex*, then the thread is in (*1-Idle*). This contradicts Lemma 1. If it is blocked by *wait for work*, then it has unlocked the *notify mutex* previously, which contradicts that ξ is holding the *notify mutex*. ■

Next, we show that *wait for work* cannot wait indefinitely.

Lemma 3: If thread ξ adds no callback during *fill wait set*, then there must be another thread ϕ in (*8-Run*).

Proof: Since ξ runs *fill wait set*, it owns both mutexes by Lemma 1 (v). Therefore, all other threads must be in either (*1-Idle*) or (*8-Run*), by Lemma 1 (iii) and (iv).

If there is no thread in (*8-Run*), then no callback group can be blocked. That means that ξ must have reached *fill wait set*, as the wait set is empty. As the system has at least one callback (by assumption) that is not in the wait set, it must be added by *fill wait set*. Since ξ adds no callback during *fill wait set*, we conclude that there must be another thread in (*8-Run*). ■

We can now show that the system is deadlock-free.

Theorem 1: The proposed executor is deadlock-free.

Proof: We prove this theorem by contradiction. Assume there is a set of threads $\{\xi_i\}$ waiting for each other that can only be released by other threads in the set. We consider one of these threads ξ_1 specifically and distinguish different cases.

Case 1: ξ_1 waits for the *notify mutex*. The *notify mutex* must be owned by another thread in the set $\{\xi_i\}$, which we denote by ξ_2. By Lemma 2, ξ_2 does not wait, which contradicts the assumption that all threads in $\{\xi_i\}$ wait.

Case 2: ξ_1 waits in *wait for work*. By Lemma 3 there exists another thread in state (*8-Run*) that can potentially release ξ_1. We denote that thread by ξ_2. Since $\xi_2 \in \{\xi_i\}$ this thread waits as well. More specifically, ξ_2 waits for the *notify mutex*. We have already shown in Case 1 that the existence of such a thread in the thread set leads to a contradiction.

Case 3: ξ_1 waits for the *wait set mutex*. Hence, the *wait set mutex* is owned by another thread in $\{\xi_i\}$, say ξ_2. ξ_2 can either wait for the *notify mutex* or it can wait in *wait for work*. Cases 1 and 2 show that both scenarios lead to a contradiction.

In conclusion, we have shown that no deadlock state can occur, i.e., the system is deadlock-free. ■

We also applied a model checker to verify that the proposed multithreaded executor is deadlock-free. We created a model of the proposed executor design in Promela, the specification language of the SPIN model checker [5] that checks LTL properties (linear temporal logic) for models of concurrent software. Technically, we model every state in the state machine of the executor and the transitions between the states with *labels* and *goto* statements in a Promela process and instantiate n of these processes to analyze the behavior for n concurrent threads. We model the mutexes using atomic statements and message channels of capacity one (i.e., one thread can take the token while the others block and are notified once the token is put back into the channel). Moreover, we model the wait set state using a global Boolean variable that tracks if any callback group flag is set to false. When this is the case, the *wait for work* operation may block a thread indefinitely, until another thread triggers the guard condition. We encoded deadlock freedom as an LTL property and used SPIN to verify the property for sets of 2 and 3 threads.

B. Starvation Freedom

In this section, we show that our proposed executor design for the multithreaded executor is starvation-free.

Definition 2 (Starvation Freedom): Every callback that is activated is eventually executed by a thread.

To achieve this, we need the additional *assumption* that the mutexes are *fair* in the following sense: if a thread ξ is waiting to lock a mutex, the mutex can only be locked finitely many times by other threads before ξ locks the mutex.

For the system to be starvation-free, it is fundamental that any thread makes progress eventually. Otherwise, threads will wait indefinitely, and task instances may potentially starve. This is formalized in Lemma 6. The following two lemmas help us reach Lemma 6.

Lemma 4: Every thread can only make a finite amount of progressions before returning to (*1-Idle*).

Proof: There are only finitely many operations in the system. Therefore, if a thread progresses infinitely often without return to (*1-Idle*), then it must execute some operation infinitely often. However, since the only loops are in states (*3-Sample*)–(*5-Poll*), all other states (and therefore all operations inside those states) are only visited once. Moreover, the *executor.updated* variable ensures that states (*3-Sample*)–(*5-Poll*) are visited at most twice. The number of visits of the operations in (*3-Sample*) is bounded by two times the number of callbacks, which is finite. ■

Lemma 5: Whenever there are activated callbacks, eventually, *one* thread makes progress.

Proof: We prove this lemma by contradiction. To that end, we assume that after time t, no thread makes progress indefinitely. This can happen only in two cases as follows.

(i) The system reaches a deadlock.
(ii) A thread waits in *wait for work* indefinitely for an outside activation.

Case (i) cannot occur due to Theorem 1. Therefore, we only consider Case (ii).

To that end, assume that there is a thread ξ in *wait for work*. By Lemma 1, all other threads can only be in states (*1-Idle*), (*8-Run*), or (*9-Finish*). If another thread were in (*8-Run*) or (*9-Finish*), that thread could make progress. Therefore, the possible scenario is that ξ is in *wait for work* and all other threads are in (*1-Idle*).

Since ξ cannot progress, this means that all other states must have been at state (*1-Idle*) already when ξ was executing *fill wait set*. Otherwise, another thread would have passed *trigger guard condition* and ξ could progress.

Since all other threads have been in (*1-Idle*) when ξ was running *fill wait set*, no group was blocked and all callbacks are added to the wait set. Therefore, at time t a newly added callback is activated and ξ can progress. This contradicts the assumption that ξ cannot progress. ■

Lemma 5 differs from deadlock freedom as it considers the cases in which a thread is not waiting for another thread, but is stuck in *wait for work* and can only be released due to a callback activation. The following lemma further shows that *every* thread eventually makes progress, whereas the previous lemma only shows progress for *at least one* thread.

Lemma 6: Whenever there are active callbacks, *every* thread makes progress, eventually.

Proof: We prove this lemma by contradiction. To that end, we assume that a thread ξ never makes progress after time t.

Case 1: ξ is in states (*2-Init*)–(*7-Take*) or (*9-Finish*). Then, ξ owns a lock. Since another thread always processes, there are only finitely many threads, and by Lemma 4, after finitely many progressions every other thread is waiting for the lock held by ξ. At that time, no thread can make any progress, which contradicts Lemma 5.

Case 2: ξ is in (*1-Idle*) or (*8-Run*). In both cases, ξ waits for a mutex lock. We have shown, in Case 1, that the thread ϕ holding the lock progresses eventually. Therefore, ϕ unlocks the mutex eventually. By assumption, the mutex is fair, meaning that only finitely many other threads can lock and unlock the mutex before ξ eventually progresses. ■

Since every thread makes progress eventually, we know that every thread returns to state (*1-Idle*) eventually.

Proposition 1: Eventually, every thread returns to the idle state.

Proof: Since every thread makes some progress eventually by Lemma 6, and each thread can only make finitely many progressions before returning to (*1-Idle*) by Lemma 4, every thread returns to the idle state eventually. ■

Due to the preceding proposition, each thread runs in cycles, i.e., follows a path that starts and ends in (*1-Idle*) and takes a finite amount of time. We denote such a *thread cycle* by γ. In the following lemmas, we examine the impact of a thread cycle on the wait set.

Lemma 7: When the wait set contains callbacks of group g when thread ξ enters (*2-Init*) during a thread cycle γ, then ξ does not add a callback of group g to the wait set during γ.

Proof: If there are callbacks of group g in the wait set when ξ enters (*2-Init*) during γ, then those callbacks are still in the wait set when ξ is in (*3-Sample*) during γ. From (*3-Sample*), ξ can only add callbacks to the wait set if it reaches (*5-Poll*). However, since there are callbacks of group g in the wait set, the wait set is not empty. Therefore, ξ can only reach (*5-Poll*) if all callbacks are blocked (*can_be_taken_from = false*). In that case, g is blocked as well and no callbacks can be added during *fill wait set* by ξ. ■

Lemma 8: If thread ξ does not reach (*5-Poll*) in thread cycle γ, it removes one callback from the wait set.

Proof: If thread ξ does not reach (*5-Poll*) in thread cycle γ, then it executes *remove cb from the wait set* in (*7-Take*). ■

We prove starvation freedom in two steps. First, we show that every activated callback is added to the wait set (Lemma 10). Second, we show that every callback in the wait set is eventually executed (Lemma 9). For convenience in the proof, we first show the second step.

Lemma 9: Each group is *eventually* fully removed from the wait set for execution.

Proof: We prove this lemma by contradiction. To that end, we assume that there exists a group g and a time point t_0, such that after t_0 there are always callbacks of g in the wait set. Let $t_1 < t_2 < \dots$ be the time points after t_0 where threads enter (*2-Init*). Because of Proposition 1, there are infinitely many such time points. Furthermore, let $\xi_1, \xi_2, \dots$ be the corresponding threads. Note that, duplicates may be possible, i.e., $\xi_i = \xi_j$ for $i \neq j$ is allowed.

By Lemma 7, no callbacks in g are added after t_1. Let t_i be the time point when the minimal number of callbacks of

g is in the wait set. By Proposition 1, every thread finishes the execution of callbacks of g and returns to (*1-Idle*). Hence, there is a time t_j where the minimal number of callbacks of g is in the wait set and g is not blocked. After t_j, no thread can reach (*5-Poll*) as the wait set is not empty and g is not blocked. By Lemma 8, at each time point $t_j, t_{j+1}, \ldots$ a thread $\xi_j, \xi_{j+1}, \ldots$ removes one callback from the wait set. Thus, as the number of callbacks is finite, the wait set is emptied. This contradicts the assumption and proves the lemma. ■

Lemma 10: Eventually, every active callback is added to the wait set.

Proof: We prove this lemma by contradiction, i.e., a callback *cb* of a group g is active but never added to the wait set.

By Lemma 9, all active callbacks in g are removed from the wait set eventually. Moreover, by Proposition 1 every execution of callbacks in g finishes, and the group g gets unblocked, eventually. We call that time point t_x.

After t_x, if a thread would enter (*5-Poll*), then *cb* would be added to the wait set. Therefore, after t_x, no thread enters (*5-Poll*). By Lemma 8, each thread cycle after t_x removes one callback from the wait set. Eventually, all callbacks are removed from the wait set, and a callback has to enter (*5-Poll*). That thread then adds callback *cb* to the wait set. This contradicts our assumption. ■

We now have all the tools to prove starvation freedom.

Theorem 2: The proposed executor is starvation-free.

Proof: When a callback is activated, Lemma 10 shows that it is added to the wait set eventually. After it is added to the wait set, Lemma 9 shows that it is eventually removed for execution. The executing thread eventually finishes its cycle due to Proposition 1. Then, the callback is executed.

We have shown that any activated callback is eventually executed. This shows that the system is starvation-free. ■

We note that we are not able to additionally apply a model checker to verify starvation freedom.

VIII. Conclusion

We explore the general design of the ROS 2 executor and show that the ROS 2 multithreaded executor is prone to starvation by providing concrete ROS 2 system configurations. We further show that the existing response-time analyses for the ROS 2 multithreaded executor are flawed, as they provide finite response times for at least one of these configurations.

We proposed minimal design changes to the software architecture of the ROS 2 multithreaded executor to solve the starvation problem. We empirically test that our design changes prevent starvation in all the previously affected system configurations while incurring a negligible overhead. Furthermore, we prove our design is starvation- and deadlock-free.

References

[1] A. A. Arafat, S. Vaidhun, K. M. Wilson, J. Sun, and Z. Guo, "Response time analysis for dynamic priority scheduling in ROS2," in *Proc. ACM/IEEE Design Autom. Conf.*, 2022, pp. 301–306.

[2] T. Blass, D. Casini, S. Bozhko, and B. B. Brandenburg, "A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance," in *Proc. Real-Time Syst. Symp. (RTSS)*, 2021, pp. 41–53.

[3] D. Casini, T. Blass, I. Lütkebohle, and B. B. Brandenburg, "Response-time analysis of ROS 2 processing chains under reservation-based scheduling," in *Proc. Euromicro Conf. Real-Time Syst.*, 2019, pp. 1–23.

[4] H. Choi, Y. Xiang, and H. Kim, "PiCAS: New design of priority-driven chain-aware scheduling for ROS2," in *Proc. Real-Time Embedded Technol. Appl. Symp. (RTAS)*, 2021, pp. 251–263.

[5] G. J. Holzmann, "The model checker spin," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[6] X. Jiang, D. Ji, N. Guan, R. Li, Y. Tang, and Y. Wang, "Real-time scheduling and analysis of processing chains on multi-threaded executor in ROS 2," in *Proc. Real-Time Syst. Symp. (RTSS)*, 2022, pp. 27–39.

[7] "ROS 2: Humble," Open Robotics. 2023. [Online]. Available: https://docs.ros.org/en/humble

[8] M. Quigley et al., "ROS: An open-source robot operating system," in *Proc. ICRA Workshop Open Source Softw.*, 2009, pp. 1–6.

[9] H. Sobhani, H. Choi, and H. Kim, "Timing analysis and priority-driven enhancements of ROS 2 multi-threaded executors," in *Proc. Real-Time Embedded Technol. Appl. Symp. (RTAS)*, 2023, pp. 106–118.

[10] J. Staschulat, I. Lütkebohle, and R. Lange, "The RCLC executor: Domain-specific deterministic scheduling mechanisms for RoS applications on microcontrollers: Work-in-progress," in *Proc. Int. Conf. Embedded Softw. (EMSOFT)*, 2020, pp. 18–19.

[11] Y. Tang et al., "Response time analysis and priority assignment of processing chains on ROS2 executors," in *Proc. Real-Time Syst. Symp. (RTSS)*, 2020, pp. 231–243.

[12] H. Teper, T. Betz, G. von der Brüggen, K.-H. Chen, J. Betz, and J.-J. Chen, "Timing-aware ROS 2 architecture and system optimization," in *Proc. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, 2023, pp. 206–215.

[13] H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, and J.-J. Chen, "End-to-end timing analysis in ROS2," in *Proc. Real-Time Syst. Symp. (RTSS)*, 2022, pp. 53–65.

[14] "Autoware reference system," RoS-Realtime Working Group, 2022. [Online]. Available: https://github.com/ros-realtime/reference-system

[15] Y. Yang and T. Azumi, "Exploring real-time executor on ROS 2," in *Proc. Int. Conf. Embedded Softw. Syst.*, 2020, pp. 1–8.