IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 27, 2020, accepted August 5, 2020, date of publication August 10, 2020, date of current version August 21, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3015385

Non-Blocking Synchronization Between Real-Time and Non-Real-Time Applications

ALEJANDRO PEREZ RUIZ, MARIO ALDEA RIVAS, AND MICHAEL GONZÁLEZ HARBOUR

Software Engineering and Real-Time Group, University of Cantabria, 39005 Santander, Spain

Corresponding author: Alejandro Perez Ruiz (perezruiza@unican.es)

This work was supported in part by the Spanish Government and FEDER funds (AEI/FEDER, UE) under Grant TIN2017-86520-C3-3-R (PRECON-I4).

ABSTRACT Real-time systems where applications with timing requirements coexist with applications without timing constraints are increasingly common. Furthermore, the processors used in desktops, smart phones or embedded devices are mostly multi-core, allowing the execution of applications in parallel. This article presents a set of non-blocking synchronization mechanisms to share data between real-time and non-real-time applications executing in different cores of a shared memory multi-core system. Four typical producer/consumer scenarios have been explored; a) shared data object with real-time reader, b) shared data object with real-time writer, c) shared queue with real-time writer, and d) shared queue with real-time reader. For these scenarios we have developed different non-blocking protocols where the execution of the real-time application is always prioritized over the execution of the non-real-time part. In this way, the real-time applications never have to repeat their operations and, consequently, their execution times are bounded. Furthermore, to reduce the overhead caused by the copies of the information used in the non-blocking algorithms, we have imposed the limitation of a single real-time reader or a single real-time writer in the algorithms developed. Finally, we have evaluated the response times of the developed protocols on a multi-core device with the Android operating system.

INDEX TERMS Android, multi-core, non-blocking synchronization, real-time, wait-free.

I. INTRODUCTION

In recent years, the evolution of processors has been carried out by increasing the number of cores. As evidence of this evolution, most major chip manufacturers implement multi-core processors. This type of processors are used in all kinds of computing devices, from desktop computers or smartphones to embedded systems. In such devices, various processes or threads can be executed simultaneously in the different cores to sharply reduce the computation times. With parallel and concurrent programing, the applications can be divided into several threads to be executed in different cores and thus maximize the performance of the system.

When there are different processes or threads running simultaneously in a multi-core device there is a need to have inter-process communication (IPC) mechanisms. These mechanisms are used for transferring data between processes. There are different techniques to carry out this communication, for example shared memory, message passing, remote procedure calls (RPC), etc [1]–[3]. When these techniques are used, a synchronization mechanism is necessary to deal with the possible concurrent accesses to the shared data.

Since real-time applications are usually concurrent, developers are faced with the need to use synchronization among the different threads. In addition, it is becoming more usual to find systems that allow the execution of heterogeneous applications in the same computational device. This means that in the same system there may be real-time applications coexisting with applications without real-time requirements. Evidence of this is that there is a rising interest in adapting general-purpose operating systems such as Android in systems where real-time applications can be executed [4]–[6] together with general-purpose applications obtained from very large repositories and stores, or developed under ecosystems populated of libraries built for different purposes.

The associate editor coordinating the review of this manuscript and approving it for publication was Noor Zaman.

The aim of this work is to provide synchronization protocols to share data between real-time and non-real time applications that run in different cores on the same shared memory multi-core device. The protocols guarantee that read/write operations on the shared data performed by real-time threads will have bounded response times even in the case of concurrent access with the non-real-time threads. In order to achieve this objective, the protocols are non-blocking to avoid locks that could increase the response time of real-time applications and introduce unbounded delays or even priority inversion problems. In addition, our non-blocking protocols ensure that real-time applications will always execute read or write operations on shared data with guarantees of not having to repeat the operation due to the interference of the non-real-time part of the system, which could lead to unbounded response times.

Four typical producer/consumer scenarios have been explored in this work, a) shared data object with real-time reader, b) shared data object with real-time writer, c) shared queue with real-time writer, and d) shared queue with real-time reader.

These scenarios have a series of requirements. They must allow data sharing between real-time and non-real-time apps running on a multi-core processor. Moreover, they have to ensure that the real-time part should never be interrupted by the non-real-time part in order to have bounded response times. They must also offer the possibility of sharing one or more data items. To share several data items, a queue structure is used.

One of our objectives is that the protocols developed have small overheads, therefore, despite the fact that there are previous studies [9], [10] that offer wait-free solutions with multiple readers or writers, we self-impose the limitation of a single writer or a single real-time reader to significantly reduce overheads. By limiting the number of real-time readers or writers, we use fewer copies of the information.

To satisfy the previous scenarios, we present the following protocols: (1) SDrtR (shared data with real-time reader), (2) SDrtW (shared data with real-time writer), (3.1) QrtWo (queue with real-time writer and overwrite), (3.2) QrtWc (queue with real-time writer and clearing) and (4) QrtR (queue with real-time reader).

These protocols are intended for shared memory multi-core systems with real-time and non-real-time applications running on different cores. In this article, we also present an evaluation of these protocols in an Android device where real-time and non-real-time applications can coexist using core isolation mechanisms described in a previous work [4].

The remainder of this article is organized as follows. In Section II we present existing related works on non-blocking synchronization algorithms and give the motivation that led to the realization of this study. The description of the environment where our non-blocking synchronization protocols are applicable is carried out in Section III. Section IV describes the five non-blocking synchronization protocols developed. In Section V we describe our proposed solution to run real-time applications on an Android operating system based on Linux. In Section VI we describe and evaluate the inter-process communication mechanisms more frequently used in Linux-based operating systems. Section VII contains the evaluation of the five protocols developed on an Android device. Section VIII gives our conclusions.

II. RELATED WORK AND MOTIVATION

A recurring problem in real-time environments is how to share data between different tasks while maintaining data consistency and avoiding priority inversions. For this reason, during the last decades different techniques have been explored to achieve this objective. Many different options exist to access shared resources using synchronization mechanisms. Two types of mechanisms are blocking and non-blocking synchronization.

The most commonly used synchronization mechanisms are blocking synchronization or locks. This traditional approach uses synchronization primitives such as mutexes, semaphores or critical sections, which ensure that specific sections of code are executed under mutual exclusion. If a process tries to acquire a lock that is already held by another process, it will be blocked until the lock is released. However, as described in other studies [11], [12], this type of mechanism can cause undesirable situations such as low throughput in case of heavily contended lock, priority inversion or deadlock. Although priority inversion or deadlock may be avoided or mitigated with some algorithms, low throughput may occur in multi-core systems due to the serialization introduced in the execution of protected operations. Non-blocking synchronization mechanisms are presented as a solution to the problems associated with blocking algorithms. In addition, non-blocking techniques can improve the overall performance of the system, as shown in some studies [13]–[15].

The traditional approach to non-blocking implementations distinguishes three types of algorithms: *obstruction-free*, *lock-free*, and *wait-free* [16], [17]. Obstruction-free provides the weakest non-blocking progress guarantee. A synchronization algorithm is obstruction-free if, from any point after which it executes in isolation, it finishes in a finite number of steps [18]. Lock-free algorithms guarantee system-wide progress, but individual threads may have a possibility of starvation (even permanent starvation). Wait-free is the strongest non-blocking guarantee of progress, thus all threads in the system have a bound on the number of steps to complete every operation. A wait-free algorithm is always lock-free, but lock-free may not necessarily be wait-free. Wait-free algorithms and their associated data structures are even harder to design and implement than lock-free ones.

There are several studies that propose different implementations of non-blocking shared data structures and techniques in a satisfactory manner. Therefore, we are identifying some of the most relevant ones. It is known that any data structure can be transformed into a lock-free one [19]. Proof of this is that there are numerous implementations to obtain efficient lock-free data structures such as the non-blocking queue implementation [20] using a singly linked list and

double-word compare-and-swap operations for the head and tail pointers, hash tables [21] or lock-free concurrent dictionaries [22]. These solutions use lock-free synchronization algorithms, and therefore they do not offer full guarantees that some executions do not have to be repeated until the data is consistent. This means that when a process tries to access a shared resource the execution times are not bounded. Moreover, there are developments such as the Tervel framework [10] for implementing non-blocking libraries. This framework tries to be an unification for methodologies and techniques to enable the implementation of wait-free algorithms. This development provides an alpha release for developers, thanks to the utilization of memory reclamation, descriptors and progress assurance constructs. This framework needs a system that supports C++11 and all the algorithms that use it are based on descriptor-based methodologies that introduce significant space and time overhead.

None of the previous algorithms have been designed to be used in environments where there are real-time applications alongside non-real-time applications, where real-time threads need to have bounded time without unbounded repetitions to access shared resources.

Non-blocking mechanisms for real-time systems have become a research topic that attracts extensive interest. There is a work [23] that describes an implementation of a lock-free framework for uniprocessor systems, based on a multi-word compare-and-swap primitive (MWCAS) that allows simultaneous access to shared data. Another work [24] describes a non-blocking protocol for read/write buffers with support for multiple readers. The writer never has to repeat the operation (wait-free). At the end of their operation, readers always check if the shared data has been modified during the reading. If a modification has occurred, the reader has to repeat the operation. Another paper [9] describes a non-blocking solution for the read/write buffers with support for multiple writers on multiprocessor real-time systems.

The work presented in [23] uses the MWCAS atomic instruction to generate lock-free implementations, however our goal is to obtain wait-free behaviour for the real-time threads. The solutions presented in [9] and in [24] are valid for mixed-criticality systems where there are a real-time writer and a non-real-time reader. As this is one of the scenarios that we study in this work we have chosen the solution proposed in [24] because of its implementation simplicity.

Despite all the existing studies on non-blocking synchronization, we have not found a practical study that explores the different producer/consumer scenarios in a multi-core system where real-time and non-real-time applications can coexist sharing data. Thus, in our approach we seek that the response times of real-time applications are bounded and without significant overhead when operating with shared resources.

During the last few years, real-time systems have gone from executing closed applications in embedded systems with a specific purpose to being part of powerful, open and interconnected execution environments. However, the increase in power is no longer associated with the speed of processors, and is currently linked to a greater number of cores. As multi-core systems become more common in industrial environments, a natural evolution is to integrate more and more functionality into a single system. In many cases it is desirable to use a general purpose operating system, such as Linux, in which non-real-time and real-time applications can coexist together to make the best possible use of all the features of the system. Thus, in a previous work [4] we have presented a proposal where soft real-time applications are executed in a set of isolated cores from the rest of non-real-time applications on devices with an Android/Linux operating system. The isolation of the real-time components in a specific set of cores makes it possible to eliminate interferences that the non-real time components might cause in the response times.

The protocols presented in this article are targeted to multi-core systems with shared memory, where real-time and non-real-time applications are executed in different cores. For this kind of systems we need to have data structures that can be shared and for which access or modification in the real time part is performed with bounded times and wait free, thus without retries.

The following section presents a description of the environment where our four non-blocking scenarios are being used.

III. APPROACH TO THE PROBLEM

A. ARCHITECTURAL ENVIRONMENT

The proposed solutions in this work for the synchronization of real-time and non-real-time applications are designed to be used in multi-core systems with shared memory.

We propose the use of non-blocking synchronization protocols in applications where real-time and non-real-time threads are executed in different cores. In this way, we have guarantees that the response times of the threads with real-time requirements are never affected due to sharing data with non-real-time applications. In our solution we only focus on synchronization between real-time and non-real-time applications, and thus the synchronization mechanism is asymmetric: it has real-time and non real-time operations. Real-time applications that use our algorithm to access or modify a shared resource are guaranteed to advance in a finite number of steps, since they do not have locks or loops. Therefore, the execution times are bounded. On the other hand, response times can be affected by the interferences that are caused by the used operating system, the interruptions or tasks with higher priority, but in no case by the use of our algorithms.

The communication between applications of the same type (real-time or non-real time) can be done using the classic mechanisms of mutual exclusion. The non-blocking synchronization algorithms presented in this article only allow one real-time thread to act as reader or writer, and therefore a classic mechanism of mutual exclusion could be used to synchronize all real-time readers or writers that want to communicate with non-real-time applications. Fig. 1 illustrates

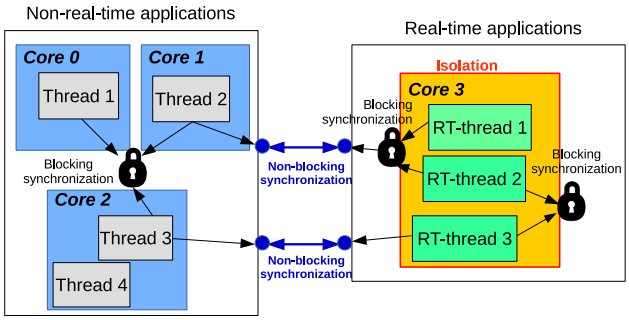


FIGURE 1. Proposal for the synchronization between real-time and non-real time applications.

this situation in a system with four cores; three of them are used to execute non-real-time applications while the other one (*Core 3*) is isolated to allow the execution of applications with real-time requirements. When the synchronization is between a real-time thread and a non-real-time thread, one of the non-blocking mechanisms that we propose in this work can be used directly (see *Thread 3* and *RT-thread 3* in Fig. 1). However, if several real-time threads need to be synchronized with any of the non-real-time threads, the real-time threads can use a traditional mutual exclusion mechanism using a real-time protocol [7], [8] to serialize the access to the non-blocking synchronization algorithm. Since the non-blocking synchronization is wait-free, it can be used from inside a protected operation or critical section performed with a mutex locked (this is illustrated in Fig. 1 as a blocking synchronization used between *RT-thread 1* and *RT-thread 2*). When non-blocking protocols are used to synchronize the real-time part with the non-real-time part, we are guaranteed that there are no problems such as priority inversion [7], since no thread is involved in a blocking wait. The real-time threads never have to wait, and the non-real-time threads only have busy waits. As mentioned above, when we combine our non-blocking protocol with other blocking mechanisms, we must use real-time protocols (such as priority inheritance or priority ceiling) that avoid priority inversion.

B. NON-BLOCKING SYNCHRONIZATION PROBLEM

Non-blocking synchronization algorithms are designed in such a way that they do not require a critical section. This kind of algorithms operate with a local copy of the shared data. When a writer thread wants to update any shared data, it attempts to copy its local value to the shared one. This operation will fail if it overlaps another read or write operation on the same data performed by another thread. In such case, one or both operations must be repeated.

Real-time applications must have time guarantees in the access to shared resources, so that this kind of applications should never have to repeat a read or write operation. However, applications that do not have timing requirements could repeat the read or write operations as many times as necessary until the data is consistent.

We have identified four commonly used producer/consumer scenarios:

- **SDrtR (shared data with real-time reader)**: Shared data with one non-real time writer and one real-time reader.
- **SDrtW (shared data with real-time writer)**: Shared data with one real-time writer and multiple non-real time readers.
- **QrtW (queue with real-time writer)**: Circular queue with one real-time writer and one non-real time reader. In this situation we have identified two possible behaviours when the queue is full and new data needs to be written:
  - **QrtWo (queue with real-time writer and overwrite):** When the queue is full, the new data overwrites the oldest data.
  - **QrtWc (queue with real-time writer and clearing):** When the queue is full and there is new data to enqueue, all the old data is discarded.
- **QrtR (queue with real-time reader)**: Circular queue with one non-real-time writer and one real-time reader.

As an example of a real-time writer thread, we can think of an application where a thread is used to obtain information from a sensor, process the data, perform some control action, and finally display that information in a graphic interface for the user. The sensor needs to be read periodically, from a real-time thread, which processes the data, performs control actions based on it, and writes the information into a synchronization object for its display in the non-real-time part of the application where the graphical user interface (GUI) is located. Depending on the application requirements, it is possible that the GUI is only interested in the latest available data item, in which case we would use the SDrtW pattern, or it may require processing the full series of data, in which case one of the QrtW patterns would be used. Another example where our algorithms could be used is the arrival of data through a communication network accessible from the non-real-time part. The data is passed to the real-time part for processing. Again, it could be interesting to collect only the last available data, or to have a queue with various data items that must be processed.

These kind of scenarios have a limitation in the number of readers and writers; this limitation makes the overhead low, since in the protocols developed in this work the number of copies of the information is a maximum of two. In addition; it would be possible to share data with more readers or writers if classic mutual exclusion mechanisms are used, as we describe in the Subsection III-A.

C. PROPERTIES OF THE SOLUTION

Our non-blocking algorithms for synchronization and data sharing must satisfy two properties:

- **Real-time part without blocking or repetitions:** In a system where there are real-time applications, it is really important to have bounded response times. In our case, where the real-time thread synchronizes with

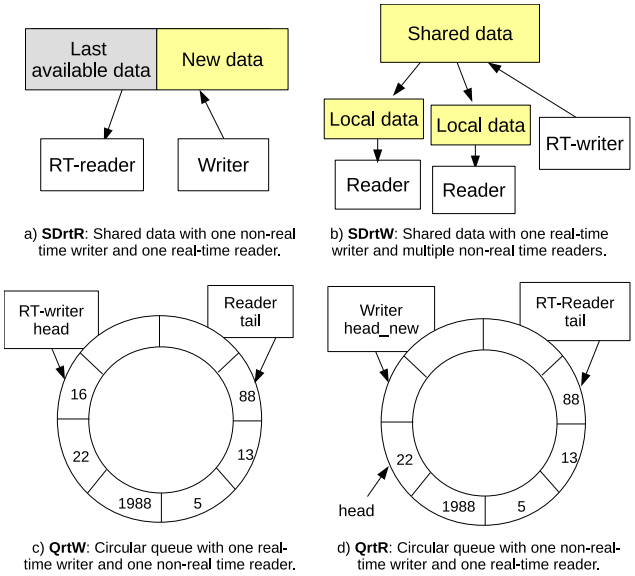
non-real-time threads whose response times are potentially unbounded, we propose wait-free synchronization where the real-time part does not repeat write or read operations. Thus, worst case execution times for the real-time applications will not be negatively affected by contention in the resources shared with non-real-time applications.

- **Integrity of the data:** It is necessary that the solution guarantees the integrity of the data in both the writing and the reading operations. Due to the asymmetric nature of our synchronization primitives non-real-time applications will be the ones that have to verify in each of their operations wheather the data is valid or not, and may need to wait or repeat operations to ensure their successful completion.

In the next section we develop the four non-blocking protocols for the scenarios depicted in Section III-B.

IV. NON-BLOCKING PROTOCOLS

To provide synchronization mechanisms that can be used in the scenarios described in the Section III-B we have developed five solutions for reader-writer and queueing non-blocking synchronization. Figure 2 shows the non-blocking synchronization patterns developed in this work for the different scenarios. For illustration purposes the queues hold integer data.



a) **SDrtR**: Shared data with one non-real time writer and one real-time reader.
b) **SDrtW**: Shared data with one real-time writer and multiple non-real time readers.
c) **QrtW**: Circular queue with one real-time writer and one non-real time reader.
d) **QrtR**: Circular queue with one non-real-time writer and one real-time reader.

FIGURE 2. Non-blocking patterns developed.

A. SDrtR: SHARED DATA WITH ONE NON-REAL-TIME WRITER AND ONE REAL-TIME READER

We describe a protocol in which there is one reader with real-time requirements and a non-real-time writer. The reader always has access to the last valid written data without the need to block or repeat the operation. On the other side, in some situations the writer needs to wait until there is no conflict with the reader who is accessing the data.

For the correct operation of the protocol a double buffer is used. Therefore, the writer never modifies data when the reader is accessing the buffer. To carry out the development of a non-blocking protocol with a double buffer, the elements in Fig. 3 have been used. The double buffer and the pointer to the index value (*pt_reader_index*) to identify the buffer that is currently being read are stored in a shared memory region available to both the reader and the writer starting at the address pointed by *pt_base*. The index value can identify a buffer using two values: 0 or 1 for the first one and 2 or 3 for the second one. The use of two different values is necessary to know if the reader is accessing the buffer. When the value of the index is odd, the reader is accessing the data and when it has an even value this indicates that it is not reading.

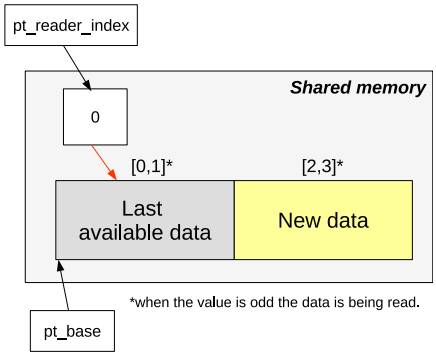


FIGURE 3. Shared elements used in the SDrtR protocol.

The detailed non-blocking protocol for writing and reading data using a double buffer is shown in the following listing:

For the write operation we have a local variable called *writer_index* which is an index to identify the buffer where the new data must be written. To obtain the value of this local variable, the content of the *pt_reader_index* pointer is shifted right by one bit (which is the same as performing the integer division between 2), and the subtraction of 1 minus the previous value is performed to flip the value and identify the buffer that is not being used for reading (see line 17 in Listing 1). In line 19 of Listing 1 the local variable called *non_reading_value* is used to stored the even value (used when the reader is not accessing the data) of the current buffer used for the reader. In line 20, *pt_next* is a pointer to the buffer in which the new data is written. After the data is stored in the buffer, we obtain the index with an even value that points to the newly written data (see line 22 in Listing 1). To change the shared pointer *pt_reader_index* an atomic operation called *CAS (compare and swap)* is used. This operation only changes the content of *pt_reader_index* with the value of *pt_reader_new_index* if it is equal to the value of *non_reading_value*. In case this comparison is not fulfilled, meaning that the reader is currently using the buffer, the writer has to busy-wait until it is satisfied (see Lines 23-24 in Listing 1). Finally, with *pt_has_data* we specify that some data has been written for the first time.

The real-time reader always gets the last correct data available and it performs two atomic operations. The first one

```
1  data_size : unsigned
2  pt_base : pointer to shared memory area
       of data_size*2 bytes
3  volatile pt_reader_index : pointer to
       shared memory area of unsigned
4  volatile pt_has_data : pointer to shared
       memory area of unsigned
5
6  initialization (d_size: unsigned)
7     data_size := d_size
8
9     pt_base := allocate data_size*2 bytes
          in the shared memory
10
11    volatile pt_reader_index : allocate
          unsigned to store an index := 0
12
13    volatile pt_has_data : allocate
          unsigned := 0
14
15 //Non-real-time operation
16 write (pt_data: pointer)
17    writer_index := 1 -
          (*pt_reader_index>>1)
18    //*pt_reader_index & 0xFFFFFFFE =
          //(*pt_reader_index>>1)*2
19    non_reading_value := *pt_reader_index
          & 0xFFFFFFFE
20    pt_next : pointer to next buffer
          (pt_base + writer_index*data_size)
21    copy data_size bytes from *pt_data to
          *pt_next*data_size
22    reader_new_index := writer_index*2
23    while (!CAS(pt_reader_index,
          non_reading_value,
          reader_new_index))
24       //Busy wait
25    endwhile
26    *pt_has_data := 1
27
28 //Real-time operation
29 read (pt_data: pointer) return integer
30    if (*pt_has_data == 0)
31       return NO_DATA
32    endif
33    //Atomic operation
34    fetch_and_add (pt_reader_index, 1)
35    pt_actual : pointer to current buffer
          (pt_base +
          (*pt_reader_index>>1)*data_size)
36    copy data_size bytes from *pt_actual
          to *pt_data*data_size
37    //Atomic operation
38    fetch_and_sub (pt_reader_index, 1)
39    return SUCCESS
```

Listing 1. Pseudocode for shared data with non-real-time writer and real-time reader.

called *fetch_and_add* (see Line 34 in Listing 1) increases the value of *pt_reader_index* by one unit to set it to an odd value (indicating that the reader is accessing the data). The second atomic operation called *fetch_and_sub* decrements the content of *pt_reader_index* to indicate with an even value that the data is no longer being read (see Line 38).

We have identified the instructions in Listing 1 where the shared pointer called *pt_reader_index* is accessed or modified by the writer and the reader. These instructions are all those that could cause an inconsistency in the data if they are not atomically modified. The instructions for the write operation are the following:

- Line 17: Read *pt_reader_index*
- Line 19: Read *pt_reader_index*
- Line 23: Read and modify *pt_reader_index*

For the read operation we have identified the following instructions:

- Line 34: Read and modify *pt_reader_index*
- Line 35: Read *pt_reader_index*
- Line 38: Read and modify *pt_reader_index*

To verify the integrity of the data we have used the same reasoning described in Kopetz's work [24]. Seven possible temporal relations between a real-time read and a write operations are illustrated in Fig. 4.

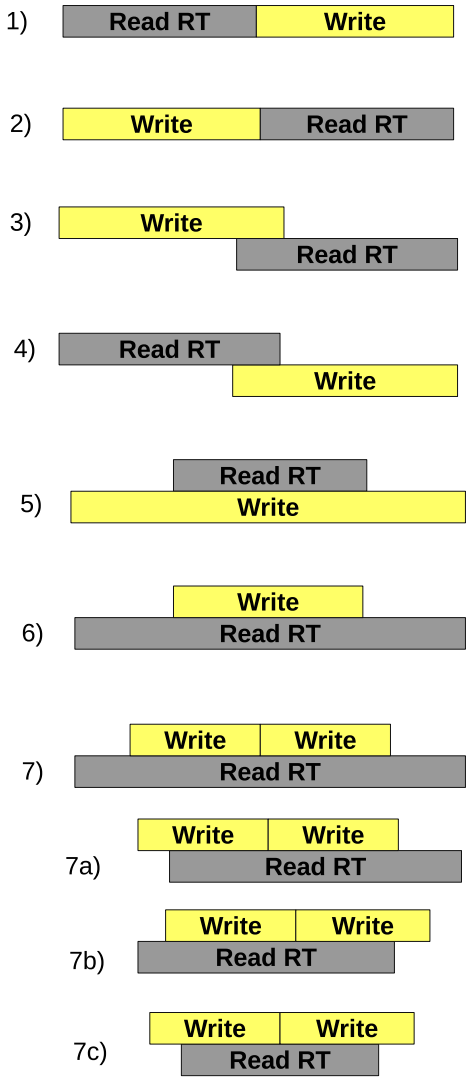


FIGURE 4. Temporal relations between a real-time read and a write operation.

The integrity of the data is ensured when the operations are not concurrent as in relations (1) and (2) of Fig. 4. In relations (3) and (4) the writer does not end until the reader changes the content of the *pt_reader_index* pointer to an even value (line 38) because in line 23 the writer performs a *CAS* atomic operation to verify that the *pt_reader_index* has a correct value and then it modifies the pointer. In relation (5) the writer can finish the operation because the reader has set the content of the *pt_reader_index* pointer to an even value, to indicate that it is no longer accessing the data. In the relation (6) the writer necessarily has to wait for the real-time reader to finish with the atomic operation in line 38 in order to complete its operation. Finally, in relations (7), (7a), (7b) and (7c) the writer can not complete the first operation until the reader changes the content of *pt_reader_index* to an even value (line 38 in Listing 1). Therefore, in all the temporal relationships described in Fig. 4 the integrity of the data is ensured and the real-time reader always ends in a finite number of steps since each operation is performed sequentially without loops or interruptions from the non-real-time part with which it is synchronized.

B. SDrtW: SHARED DATA WITH ONE REAL-TIME WRITER AND MULTIPLE NON-REAL-TIME READERS

A solution for having a writer on a data structure that can be read by multiple readers has been described previously in [24]. The algorithm described in the referred work fits perfectly our requirements, consequently we do not develop a new one, but we describe it in this article for completeness.

In their study, the authors describe the solution to be used in real-time systems where both readers and writer have timing requirements. Readers have to check at the end of each reading whether the writer has modified the data structure during this reading or not. To detect if the writer has carried out any modification to the shared data, an unsigned global counter is used. This counter is incremented just before modifying the shared data, setting it to an odd value, and then after the modification the global counter is incremented again to obtain an even value (see lines 14 and 17 in Listing 2). If the reader verifies that the shared data has not been modified, by checking the global counter (see line 25 in Listing 2), the reading operation will be finished; otherwise the operation must be repeated. For this reason the authors do a timeliness analysis for the write and read operations. Firstly, they assume that the duration of a read and write operation is the same, in this way they determine that an interference originated by the writer could cause up to three retries in the read operation. Secondly, they assume that a minimum time between succesive write operations into the shared data is known, if this time cannot be guaranteed it is not possible to determine the worst case number of interferences. With this assumption it is possible to bound the maximum execution time of a read operation considering the retries.

This algorithm fits perfectly the *SDrtW* scenario described in Subsection III-A, where the writer is the one with real-time requirements while the readers do not have such timing requirements. In our approach, we cannot make assumptions about the non-real-time part because we have no control over the interruptions that could occur in the system. Therefore, we have used the algorithm adding the initialization function with the necessary variables for its implementation in the C language. The protocol for writing and reading data based on [24] is given in the following Listing:

```
1  volatile counter : pointer to shared
       memory area of an integer
2  data_size : unsigned
3  pt_base : pointer to shared memory area
       of data_size bytes
4
5  initialization (d_size: unsigned)
6     counter := 0
7
8     data_size := d_size
9
10    pt_base := allocate data_size bytes in
          the shared memory
11
12 //Real-time operation
13 write (new_value: pointer)
14    local_counter := *counter
15    *counter := local_counter + 1
16    copy data_size bytes from *new_value
          to *pt_base*data_size
17    *counter := local_counter + 2
18
19 //Non-real-time operation
20 read (pt_data: pointer)
21    loop
22       counter_begin := *counter
23       copy data_size bytes from *pt_base
             to *pt_data *data_size
24       counter_end := *counter
25       if (counter_end == counter_begin
             and counter_begin is even)
26          break;
27       endif
28    endloop
```

Listing 2. Pseudocode for shared data with real-time writer and multiple non-real-time readers.

In the work where the previous protocol was described [24], correctness is demonstrated and it is proved that readers always obtain uncorrupted data.

C. QrtW: CIRCULAR QUEUE WITH A REAL-TIME WRITER AND A NON-REAL-TIME READER

Two protocols have been developed for a circular queue with a real-time writer and a non-real-time reader. The real-time writer can enqueue elements without blocking or repeating the operation. When the queue is full and a new element is enqueued two different behaviours have been considered. One overwrites the oldest element and the other one clears the queue. Therefore, these two protocols have been developed:

- **QrtWo (Queue real-time writer with overwrite):** If the queue is full and there is a new element to enqueue, the new element overwrites the oldest element.

- **QrtWc (Queue real-time writer with clearing):** If the queue is full and there is new element to enqueue, all old data is discarded.

The protocol called *QrtWo* uses a queue with an empty entry to help in determining when the queue is full or empty (see Fig. 5). There are also two variables that determine the head and the tail of the queue. Therefore, when the queue is full this condition is satisfied: *(tail+2)%queue_size == head* and for an empty queue this condition is met *(tail+1)%queue_size == head*.

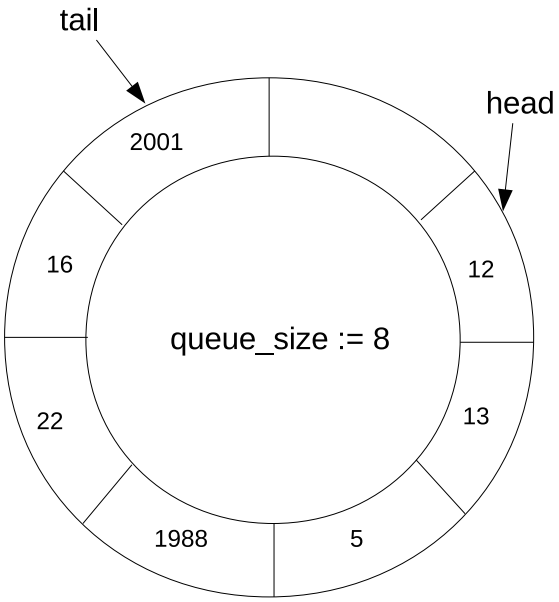


FIGURE 5. **Graphical representation of the shared variables for the protocol called *QrtWo*. In this scenario the queue is full since *(tail + 2)% queue_size == head*.**

For the correct operation of this protocol, both the head and the tail are in shared memory. In addition, the head stores three values inside. We consider the head as a 32-bit integer where the least significant bit (called *overwrite*) indicates whether an overwriting has occurred by the real-time writer, the second least significant bit (called *changing*) indicates whether the writer is modifying the head index value, and finally the 30 most significant bits are used to store the index into the array containing the queue. This is illustrated in Fig. 6.

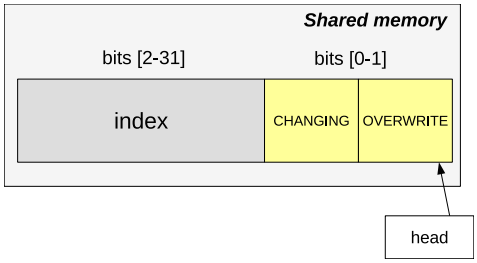


FIGURE 6. **Elements used inside the head index for the QrtWo protocol.**

The detailed protocol called *QrtWo* is shown in the following listing:

```
constant OVERWRITE = 0x01
constant CHANGING = 0x02
queue_size : unsigned
data_size : unsigned
volatile tail : allocate unsigned in
    shared memory
volatile head : allocate unsigned in
    shared memory
pt_queue : pointer to shared memory area
    of data_size*queue_size bytes


initialization (d_size: unsigned, q_size:
    unsigned)
   volatile queue_size := q_size

   data_size := d_size

   volatile tail := queue_size-1

   volatile head := 0

   pt_queue := allocate
       data_size*queue_size bytes in
       shared memory


// Real-time operation
enqueue (q_elem: pointer)
   integer unsigned current_head := *head

   integer unsigned local_head :=
       current_head | CHANGING
   integer unsigned head_not_overwrite :=
       current_head & ~(OVERWRITE)
   integer unsigned head_overwrite :=
       current_head | OVERWRITE
   // pt_queue[*tail+1] := *q_elem
   copy data_size bytes from *q_elem to
       *pt_queue +
       ((*tail+1)%size)*data_size
   //Check if the queue is full and set
       //changing to 1
   if ((*tail+2) % queue_size ==
       local_head>>2 && (CAS(head,
       head_not_overwrite, local_head) ||
       CAS(head, head_overwrite,
       local_head)))
      integer unsigned new_head =
          ((local_head>>2)+1) % queue_size
      new_head := (new_head<<2) |
          OVERWRITE
      new_head := new_head & ~(CHANGING)
      CAS(head, local_head, new_head)
   endif
   *tail = (*tail+1) % queue_size;


// Non-real-time operation
dequeue (q_elem: pointer) return integer
   if ((*tail+1) % queue_size == *head>>2)
      return EMPTY
   bool success := false
   integer unsigned new_head, local_head
   do
```

Listing 3. **Pseudocode for the QrtWo (Queue real-time writer with overwrite) protocol.**

```
47        local_head := *head
48        local_head := local_head &
              ~(OVERWRITE+CHANGING)
49        // *q_elem := pt_dequeue[head]
50        copy data_size bytes from *pt_queue
              + (local_head>>2)*data_size to
              *q_elem
51        new_head := ((local_head>>2)+1) %
              queue_size
52        new_head := (new_head<<2) &
              ~(OVERWRITE+CHANGING)
53        //Atomic operation
54        success := CAS(head, local_head,
              new_head)
55        if (success == false)
56           local_head := *head
57           int not_changing_head :=
                 local_head & ~(CHANGING)
58           new_head = local_head &
                 ~(OVERWRITE+CHANGING)
59           //Atomic operation
60           CAS (head, not_changing_head,
                 new_head)
61        endif
62     while(success==false)
63     return SUCCESS
```

Listing 3. ***(Continued.)*** **Pseudocode for the QrtWo (Queue real-time writer with overwrite) protocol.**

In this protocol, the real-time method called *enqueue* copies the data item into the empty slot in the queue and then always checks whether or not the queue is full. In case it is full, the value of the bit called *changing* is set to one to indicate to the non-real-time operation (*dequeue*) that the head index is going to be incremented (see line 31 in listing 3). When the modification is complete, the *changing* bit is set to 0 and the *overwrite* bit is set to 1 to indicate that the head index has been modified.

The non-real-time method called *dequeue* has to verify, after reading the corresponding element pointed to by the *head*, that it has not been overwritten and that it is not being changed by the real-time operation (see lines 54-55 in Listing 3). If any change is detected in the *overwrite* or *changing* bits in the *head*, the dequeue operation is repeated.

In the case of the shared variable called *tail*, it is not necessary to modify it atomically because only the real-time method can modify it to indicate that a new element has been added to the queue.

In our protocol, apart from having the shared queue, there are two shared variables for which we must verify how they are modified and accessed to determine the correctness of the algorithm. These shared variables are *tail* and *head*.

In the *enqueue* method the shared variables are accessed or modified in the following lines of the Listing 3:

- Line 24: Read *head*.
- Line 29: Read *tail* and copy data item into the queue.
- Line 31: Read *tail* and modify *head*.
- Line 35: Modify *head*.
- Line 37: Read and modify *tail*.

The *dequeue* method performs the following accesses and modifications of the shared variables:

- Line 42: Read *tail* and *head*.
- Line 47: Read *head*.
- Line 50: Read data item from queue.
- Line 54: Modify *head*.
- Line 56: Read *head*.
- Line 60: Modify *head*.

All modifications of the previous shared variables are made through the atomic operation called *CAS (compare and swap)*. In addition, the fact of having three pieces of information in the shared variable *head* means that we can modify them atomically in a single operation. This helps us to guarantee that inconsistent data is not produced during the modification of these variables.

To demonstrate the correctness and consistency of the circular queue data we have identified some scenarios where the operations could have conflicts:

- A possible temporal sequence of operations is illustrated in Fig. 7. In this case the queue is empty and just initialized. When it comes to obtaining data from an empty queue through the *dequeue* method, the writer has to enqueue a new data item and then it modifies the *tail* to indicate that there is at least one new data item. In this case, the temporal sequence of the operations is carried out without causing any potential conflict in the shared data.

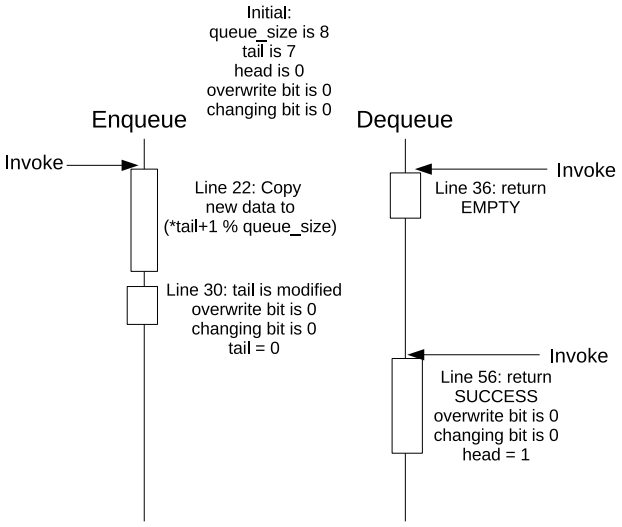


FIGURE 7. ***QrtWo*: Temporal sequence of possible operations when the queue has no elements and has just been initialized.**

- Fig. 8 shows a temporal sequence where a queue is full and a new element is enqueued. In the first invocation of the *dequeue* method, the data is not available because the *changing* bit indicates that the writer is modifying the *head* value. When the writer finishes the modification of the *head* value, the *changing* bit is 0 and this indicates that the head is accessible by the reader. Next, the reader needs two iterations to get the data. In the iteration 3

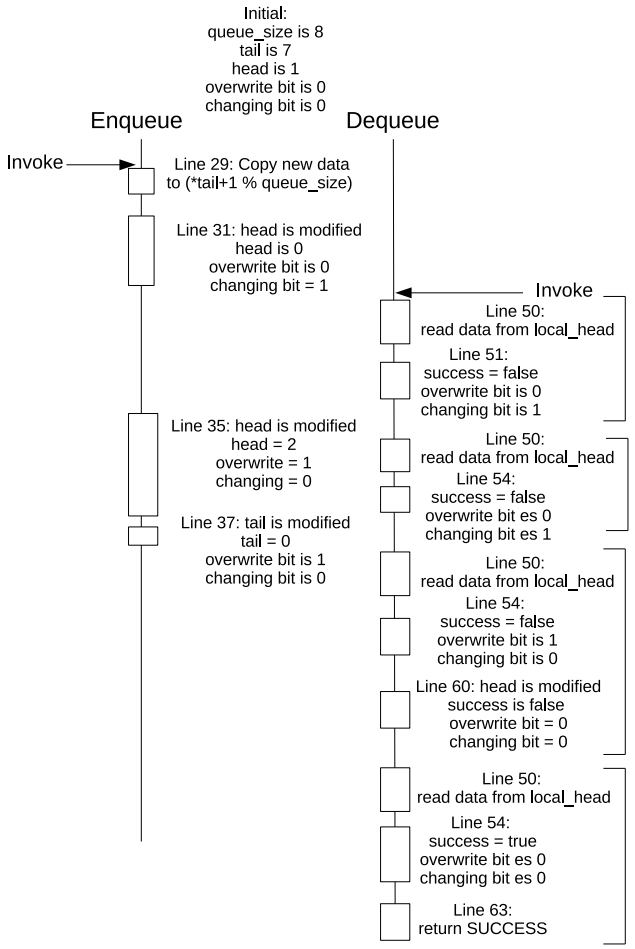


FIGURE 8. ***QrtWo*: Temporal sequence of possible operations when the queue is full and a new element is enqueued.**

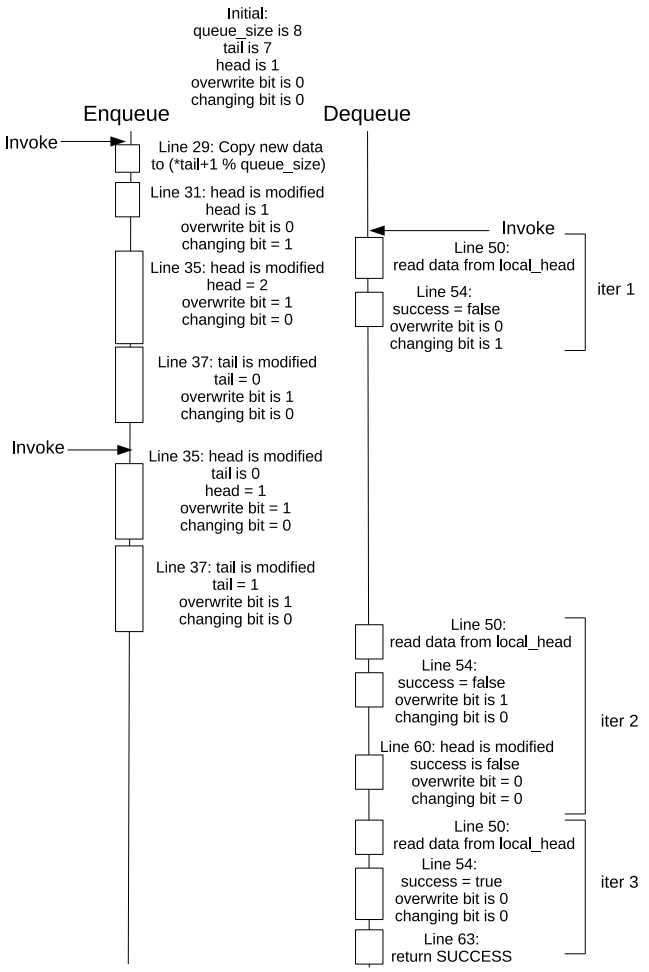


FIGURE 9. ***QrtWo*: Temporal sequence of possible operations when the queue is full and two new elements are enqueued.**

of Fig. 8 it detects an overwrite through the *overwrite* bit. The fourth iteration (see iter 4 in Fig. 8) with the *overwrite* and *changing* bits with a value of 0 allows obtaining the data pointed to by the *head* index. In this sequence of operations the race conditions have been avoided by the *overwrite* and *changing* bits and by the use of atomic operations.

- Fig. 9 illustrates a scenario where the queue is full and two elements are enqueued. During the enqueue of the elements the reader tries to obtain the data pointed to by the *head* but in the first iteration it is unsuccessful because the *changing* bit has a value of 1. When the real-time writer finishes enqueueing the second element, the *changing* bit changes to 0 and that is when the reader can access the data pointed to by the *head*. In this case, the reader has been prevented from accessing an element that was being modified by the real-time writer, therefore, no race condition occurs.

When the queue is not full, the real-time writer and the reader do not have potential race conditions since each one modifies or accesses a different element of the queue. Instead, the above three scenarios represent temporal sequences of operations where without the use of the control bits (*changing* and *overwrite*) and atomic operations race conditions could occur generating corrupted data in the queue.

Another much simpler protocol has been developed where, if the queue is full and there is new data to enqueue, all the enqueued elements are discarded. This protocol has been called *QrtWc* and it is described in detail below:

This protocol controls the possible conflicts between the real-time reader and the non-real-time writer using a counter. When the writer is enqueueing a new element it increases the counter, setting it to an odd value to indicate that it is modifying the data, and when it finishes the modification, it increases the counter again setting it to an even value (see lines 21 and 25 in Listing 4). In addition, in this queue it is not necessary to have a free slot and, therefore, to determine if it is empty we have to check that the following condition is met *(*tail+1) % queue_size == head*. In this algorithm the *head* is only modified by the non-real-time *dequeue* method, while the tail is only modified by the real-time writer (*enqueue*). This fact causes that when the *tail* reaches the *head*, queue is considered empty and as a consequence, all its elements are

```
1  queue_size : unsigned
2  data_size : unsigned
3  volatile tail : allocate unsigned in
       shared memory
4  volatile counter : allocate unsigned in
       shared memory
5  pt_queue : pointer to shared memory area
       of data_size*queue_size bytes
6
7  initialization (d_size: unsigned, q_size:
       usigned)
8     volatile queue_size := q_size
9
10    data_size := d_size
11
12    volatile tail := queue_size-1
13
14    volatile counter := 0
15
16    pt_queue : allocate
          data_size*queue_size bytes in
          shared memory
17
18
19 // Real-time operation
20 enqueue (q_elem: pointer)
21    *counter := *counter + 1
22    *tail := (*tail+1) % queue_size
23    // pt_queue[*tail+1] := *q_elem
24    copy data_size bytes from *q_elem to
          *pt_queue + (*tail%size)*data_size
25    *counter := *counter + 1
26
27 // Non-real-time operation
28 head : unsigned := 0
29 dequeue (q_elem: pointer) return integer
30    if ((*tail+1) % queue_size == head)
31       return EMPTY
32    loop
33       counter_begin := *counter
34       // *q_elem := pt_dequeue[head]
35       copy data_size bytes from *pt_queue
             + head*data_size to *q_elem
36       counter_end := *counter
37       exit when counter_begin ==
             counter_end and counter_begin is
             even
38    endloop
39    head := (head+1) % queue_size
40    return SUCCESS
```

Listing 4. Pseudocode for the QrtWc (Queue real-time writer with clearing) protocol.

discarded (see lines 30-31 in Listing 4). Moreover, the non-real-time reader can detect if there is any change in the queue data by checking if the counter has been modified, in which case it retries the reading operation (see line 37 in Listing 4). Fig. 10 illustrates a scenario where a new data item is enqueued while the *dequeue* method is reading a data item pointed to by the *head* value and, therefore, the reader has to repeat the operation until it does not detect a change in the counter and the counter has an even value.

There is a scenario where the *QrtWc* algorithm could provide an inconsistent data item. It is a really unlikely scenario

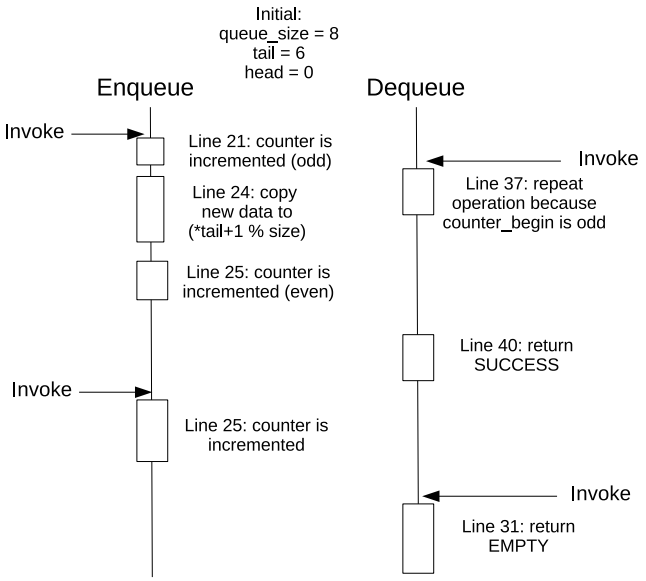


FIGURE 10. *QrtWc*: Temporal sequence of possible operations when the reader tries to dequeue an element and the writer is enqueueing.

to occur but it should be taken into account. If during the reading of a data item in the *dequeue* method, the invoker's thread is preempted it could happen that the writer might execute 2^{32} times, overflowing the unsigned integer that is used for the counter. In this case, when the reader resumes its execution it would detect that the read data item had not been modified but in reality it has changed and may be inconsistent. However, the described scenario is extremely unlikely. For instance, assuming very frequent writes every 100 microseconds, the reader would have to be preempted for at least 119 hours to produce the situation described above. This is totally unrealistic and therefore, it can be said that this situation is highly unlikely to happen.

The advantage of this protocol is that it is simpler than the previous one described in this work (*QrtWc*), however it has the disadvantage that many elements are discarded when the queue is full and overwrites occur.

D. QrtR: CIRCULAR QUEUE WITH A REAL-TIME READER AND A NON-REAL-TIME WRITER

In this case a queue with a real-time reader and a non-real-time writer has been implemented. The real-time reader can obtain data without blocking or repeating the operation, as long as there is data available. The writer will be able to enqueue data whenever there is no conflict with the real-time reader, in which case the writer will have to wait until the resource is no longer being used by the reader.

The queue is circular and when it is full, the writer will have to wait until the reader dequeues some element to free a slot. Moreover, our protocol uses an empty entry to avoid having a variable to count the number of items in the queue. As illustrated in Fig. 11, our solution uses two indexes, the *tail* to store the position of the last enqueued element and the *head* to address the element that will be read by the reader.

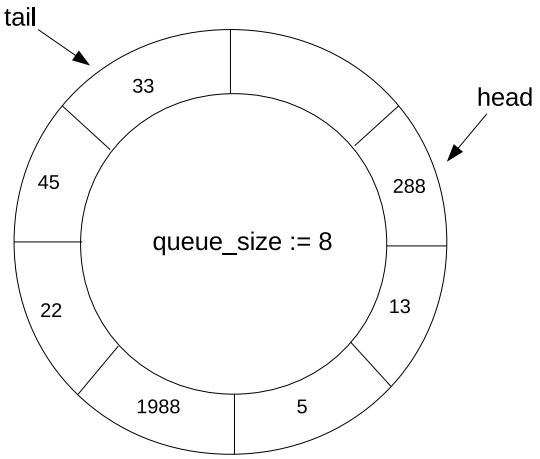


FIGURE 11. ***QrtR*: Graphical representation of the all the elements used in our protocol.**

The protocol developed for this type of queue is described below:

```
1  volatile queue_size : unsigned
2  head : allocate unsigned in shared memory
3  tail : allocate unsigned in shared memory
4  data_size : unsigned
5  pt_queue : pointer to shared memory area
       of data_size*queue_size bytes
6
7  initialization (d_size: unsigned, q_size:
       usigned)
8     volatile queue_size := q_size
9
10    head := 0
11
12    tail := queue_size-1
13
14    data_size := d_size
15
16    pt_queue := allocate
          data_size*queue_size bytes in the
          shared memory
17
18
19 // Non-real-time operation
20 enqueue (q_elem: pointer) return integer
21    if (*tail+2) % queue_size == *head)
22       return FULL
23    new_tail : unsigned := (*tail+1) %
          queue_size
24    // pt_queue[new_tail] := *data_elem
25    copy data_size bytes from *q_elem to
          *pt_queue + new_tail*data_size
26    *tail := new_tail
27
28 // Real-time operation
29 dequeue (q_elem: pointer) return integer
30    if ((*tail+1) % queue_size == *head)
31       return EMPTY
32    // *q_elem := pt_queue[head]
33    copy data_size bytes from *pt_queue +
          *head*data_size to *q_elem
34    *head := (*head+1) % queue_size
35    return SUCCESS
```

Listing 5. **Pseudocode for the *QrtR* (Queue real-time Reader) protocol.**

The method called *enqueue* performs a check at the beginning to determine whether the queue is full or not (see Line 21 in Listing 5). If the queue is full the method will return a constant indicating it. By using this information an external active wait could be programmed until the queue has a free space to enqueue a new element. The method called *dequeue* needs to check on line 30 in Listing 5 if the queue has elements available to dequeue. After this check an element is dequeued and the head increased.

In this protocol, to avoid conflicts between the writer and the real-time reader, overwriting data when the queue is full is disallowed. Therefore, the tail variable is only modified by the *enqueue* method and the head variable only by the *dequeue* method.

To demonstrate that the data read and written are consistent we have identified the four possible cases that can occur through Fig. 12. When the *enqueue* method has available slots there is no conflict (case 1 in Fig. 12). When there are no slots for new elements a full queue notification value is immediately returned (case 2 in Fig. 12). In case 3 of Fig. 12 there is no conflict because the reader (*dequeue* method) never has a new available element until the writer (*enqueue* method) has not finished writing the data (see line 19 in Listing 5). Finally, if the queue is empty the reader will immediately return a notification value.

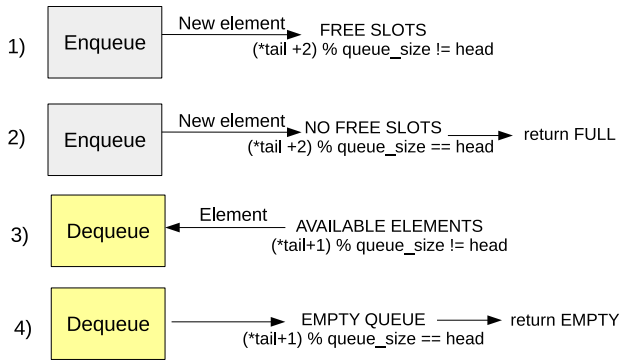


FIGURE 12. ***QrtR*: Situations that the *enqueue* and *dequeue* methods can find, depending on the state of the queue.**

With this protocol it is guaranteed that the reader has a bounded response time if there are items available in the queue.

V. RUNNING REAL-TIME APPLICATIONS ON ANDROID/LINUX

In a previous study [4] we have presented a mechanism to run soft real-time and non-real-time applications on an Android/Linux device. Applications with timing requirements are executed directly on an isolated core of the CPU, and the rest of the applications are run on the other cores. In this section we summarize the isolation mechanisms used in our previous study because they have been applied to test the non-blocking synchronization algorithms developed in this work.

The isolation mechanisms can be used on any Android/Linux device with a multi-core processor and Linux kernel version 2.6 or higher. The applications with timing requirements must be executed using one of the real-time policies offered by the Android/Linux kernel (SCHED_FIFO, SCHED_RR or SCHED_DEADLINE scheduling policies). Moreover, modern Linux kernels make most of the kernel code preemptable, except in the interrupt handlers and regions protected with spinlocks.

Despite all the features offered by the Android/Linux kernel, there are some issues that we have to solve if we want to have some reasonable predictability for applications with timing requirements:

- Interferences between real-time applications and other applications that may be running in the system.
- Interferences of interrupt handlers on the tasks of the real-time application.
- Dynamic frequency changes and automatic shutdown of cores.
- Limitations of the Bionic library (only in Android) for real-time applications.

To reduce the interferences between real-time applications and other applications in the system, we take advantage of multi-core processors and the mechanisms offered by Android/Linux kernel to isolate a core. The real-time applications will be executed directly on the Android/Linux kernel and the native libraries offered by the system. Figure 13 illustrates the architecture for our solution.

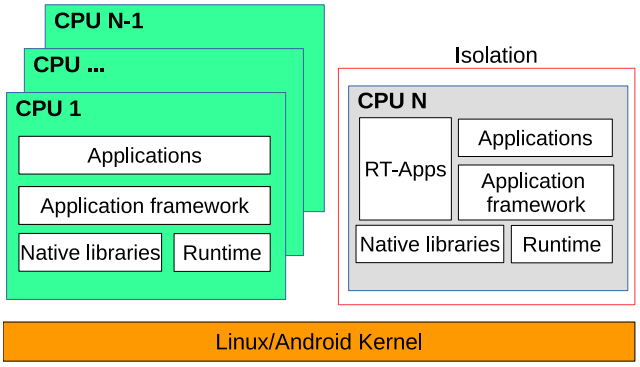


FIGURE 13. Solution proposed to execute real-time applications on Android/Linux.

Linux provides some mechanisms to isolate CPUs from the general activity of the scheduler. The most suitable for our purpose is the one called *cpuset*. Its aim is to restrict the number of processors and memory resources that a set of processes can use. With this mechanism we can move all the processes of the system to a specific *cpuset* and at the same time we can create another *cpuset* where only the real-time processes are assigned. Therefore, in the *cpuset* that is used for real-time applications, system applications cannot be run.

To avoid the arrival of interrupts to an isolated core, it is possible to assign certain interrupts to a core of the processor (or a set of cores). This functionality is called *SMP IRQ affinity* and allows us to control which cores handle the different interrupts that occur in the system. Not all interrupts are masked since there are some interruptions whose default affinity cannot be changed, for example those produced between the cores (*IPI-interprocessor interrupts*).

To achieve a greater degree of predictability in the execution times, it is necessary to set the frequency of the cores destined to execute applications with timing requirements.

If we are using Android it is necessary to take into account that this platform does not use the traditional glibc library, but uses the Bionic library. This library is developed by Google under BSD license to isolate Android applications from the effect of copyleft licenses, in order to allow the creation of proprietary user-space code in the application ecosystem. In addition, Bionic is much smaller than the traditional glibc. However, the Bionic library has some limitations to execute real-time applications, for example it does not have priority inheritance protocols for mutexes. To solve these limitations in a previous work [25] we have confirmed that it is possible to use the traditional glibc library in Android.

Applying all the mechanisms described above, we determined that substantial improvements are achieved in the interference of the OS and the non-real-time tasks on the execution of a simple task in an isolated core, compared to executing that same task in a non-isolated core. In our previous experiments, we found that the maximum jitter observed in a real-time thread is 250 microseconds executing during hours compared to the jitter of 1.5 seconds when no CPU isolation is used. This improvement is sufficient for soft real-time requirements.

In the next section we analyze the existing mechanisms to share memory in Linux/Android.

VI. SHARING DATA BETWEEN TASKS

If we want to share data between real-time and non-real-time tasks, we need to use inter-process communication (IPC) mechanisms. The most powerful and popular IPC mechanisms are pipes, sockets and shared memory.

It has been shown in another study [26] that shared memory is the fastest form of IPC mechanisms. The shared memory is a mechanism that provides simultaneous memory access to multiple processes or threads. This mechanism allows bidirectional communication between two or more processes. Therefore, the shared memory mechanism is the most suitable for exchanging data between processes located in different cores of the same host.

We have different alternatives to share memory between processes in Linux-based operating systems. There are essentially three, that we proceed to describe below.

A. MEMORY-MAPPED FILES

A mechanism to share data between processes is to use a file. In order to improve performance, this file can be mapped in virtual memory. The mapped file can be shared among various processes and can be accessed through an arithmetic pointer. We have done some tests measuring the

performance of this mechanism. After analyzing the results we have detected that every 4096 bytes written in the mapped memory segment there is a dump of the data to the file on disk. Therefore, there is a significantly increment of response time.

B. POSIX SHARED MEMORY

The POSIX interface provides a series of functions that allow having a shared memory area between different processes. The shared object can be mapped concurrently in the address spaces of several processes.

The POSIX functions for shared memory managment are not available natively on Android, because this operating system uses the Bionic library instead of the traditional glibc library. However, it is possible to use these functions if we use the library glibc in Android as we described in a previous work [25]. It is also necessary to disable the security layer called SELinux (Security-Enhanced Linux) which is used by default in Android. SELinux improves the protection, confines system services, controls access to application data and system logs and diminishes the effects of malicious software. From Android 4.3, SELinux is used to further define the boundaries of the Android application sandbox (the limits of the environment where the application can run), therefore we could compromise system security if we disactivate it temporarily. Consequently, we discourage using POSIX shared memory objects in the Android operating system.

C. TMPFS

The temporary file system, tmpfs, is a file system that keeps files in virtual memory. It is intended to appear as a mounted file system, but stored in volatile memory instead of in a persistent storage device. Since the files that use the tmpfs are held in the kernel's caches and not swapped out, access to them is significantly faster than access to a file on disk.

D. EVALUATION OF DATA SHARING MECHANISMS

We have done some tests in order to measure the response times to access an integer field shared by two processes using the three aforementioned mechanisms. In the tests we have used the Android operating system on a Nexus 5 phone that has an ARM Snapdragon 800 processor and 2GB of RAM.

The process on which the measurements were taken was running on an isolated core of the processor. To achieve the process isolation of the process, the solution described in section V has been followed. Table 1 shows the average access values and the worst case times for each of the mechanisms used. The worst case response times observed in the table are caused because the isolation is not perfect and the operating system can generate some interferences during the execution of the tests. The average-case response times show a more precise measurement of the actual overheads incurred by the synchronization primitives. As indicated by the table 1, the most suitable mechanisms for real-time environments are POSIX shared memory and tmpfs. Moreover tmpfs is natively supported in Android therefore our recommendation is to use this mechanism with this operating system.

TABLE 1. Measurements of access time to an integer using different shared memory mechanisms. They have been obtained through one million executions.

	Worst response time	Average
Memory-mapped files	59.532 us	0.501 ns
POSIX shared memory	23.803 us	0.472 us
tmpfs	15.886 us	0.417 us

VII. EVALUATION

A. TEST ENVIRONMENT

All tests are executed on the Android device Nexus 5. This smartphone has Android 6.0 with root privileges, a Qualcomm Snapdragon 800 (Quad-core) processor and 2GB of RAM. To carry out the tests, the mechanisms explained in Section V have been used: the cpuset mechanism has been activated, the affinity of the interrupts has been modified and the CPU frequency has been set to 2 GHz.

We have conducted several tests that consist of measuring the time to read or write one integer using the non-blocking protocols described in this work. The tests intended to measure real-time and non-real-time read or write times have always been run on an isolated CPU. Therefore, we have isolated two cores to execute in parallel the real-time and non-real-time parts of our protocols. We use the C programming language with a gcc cross-compiler for the ARM architecture to run directly on the kernel and using the libc standard library. In addition, as it is necessary to use shared memory in all the tests, the *tmpfs* mechanism has been selected, which has been shown in Section VI to be the most suitable for our purpose.

B. TESTING THE SHARED DATA NON-BLOCKING PROTOCOLS

In this subsection we present the results of the tests carried out to measure the time necessary to read or write a 32-bit integer using the protocols for simple data described in this article (see Subsections IV-A and IV-B). In these tests the average and the maximum response times for the real-time part have been measured, as well as for the non-real-time part. The latter has been measured for different number of iterations, since this part can be interfered by the real-time part.

Table 2 illustrates the obtained measurements for the *SDrtR* protocol described in Subsection IV-A. Similarly, Table 3 shows measurements for the *SDrtW* protocol.

In the tests carried out the reader and the writer have been run periodically every 100 microseconds and show that the jitter in real-time part, measured as the difference between the maximum and the average response times, is not increased by our protocol. The worst response times are due to the interferences produced by non-maskable interrupts and some kernel threads.

In the case of the non-real-time part, the response times are variable depending on the number of repetitions required to

TABLE 2. Measurements of read and write times for a 32-bit integer using the *SDrtR* protocol described in Subsection IV-A.

	Worst response time	Average	Standard deviation	Observations
Reader (real-time)	29.430 us	0.945 us	0.325 us	10^8
Writer (non-real-time without interferences)	1.250 us	0.662 us	0.171 us	10^8
Writer (non-real-time with 1 repetition)	0.938 us	0.937 us	0.004 us	7
Writer (non-real-time with 2 repetitions)	0.941 us	0.926 us	0.021 us	9
Writer (non-real-time with 3 repetitions)	1.286 us	0.941 us	0.032 us	104
Writer (non-real-time with 4 repetitions)	-	-	-	0

TABLE 3. Measurements of read and write times for a 32-bit integer using the *SDrtW* protocol described in Subsection IV-B.

	Worst response time	Average	Standard deviation	Observations
Writer (real-time)	54.951 us	0.624 us	0.111 us	10^8
Reader (non-real-time without interferences)	40.993 us	0.636 us	0.132 us	10^8
Reader (non-real-time with 1 repetition)	1.146 us	0.771 us	0.130 us	43
Reader (non-real-time with 2 repetitions)	1.354 us	1.146 us	0.143 us	90
Reader (non-real-time with 3 repetitions)	-	-	-	0

successfully complete the operation. In both Table 2 and 3, the tests where repetitions are considered have been obtained for executions of 10^8 iterations. Therefore, we can determine that repetitions occur infrequently and the impact over time is not very significant. In addition, both the average and worst response times are small. However, in Table 2 the non-real-time writer has a significantly lower jitter, this is because the atomic operations are the most time consuming, specifically in the hardware used it is about 0.1 microseconds each. Therefore, as the real-time reader has two atomic operations and the writer only one in the *SDrtR* algorithm, this causes that the longer the execution time, the greater the probability that non-maskable interruptions or kernel thread executions will occur in the system. The isolation mechanisms used in these tests are not perfect and we already measured in a previous work [4] that interferences of up to around 250 microseconds could occur, for this reason the worst case times are significantly greater than the average time. Our non-blocking algorithms do not cause those worst case response times; the average-case response times are a more precise measurement of their actual overheads.

Table 2 shows performances with up to 3 repetitions in the non-real-time writer, and Table 3 shows up to 2 repetitions in the non-real-time reader. In both cases they are the maximum number of repetitions that we have observed in our tests.

C. TESTING CIRCULAR QUEUES WITH NON-BLOCKING PROTOCOLS

We have measured the times to read and write data using the three protocols with the circular queues described in Subsections IV-C and IV-D. Table 4 and 5 show the read and write times for a queue of 32-bit integers using the protocols described in Subsection IV-C. In the same way, Table 6 shows the same measurements for the *QrtR* protocol described in Subsection IV-D.

TABLE 4. Measurements of read and write times for a queue of 32-bit integer using the *QrtWo* protocol described in Subsection IV-C.

	Worst response time	Average	Standard deviation	Observations
Writer (real-time)	15.365 us	0.647 us	0.103 us	10^8
Reader (non-real-time without interferences)	188.207 us	0.681 us	1.889 us	10^8
Reader (non-real-time with 1 repetition)	16.772 us	1.553 us	1.947 us	433
Reader (non-real-time with 2 repetitions)	1.927 us	1.1456 us	0.086 us	94
Reader (non-real-time with 3 repetitions)	-	-	-	0

TABLE 5. Measurements of read and write times for a queue of 32-bit integer using the *QrtWc* protocol described in Subsection IV-C.

	Worst response time	Average	Standard deviation	Observations
Writer (real-time)	13.221 us	0.619 us	0.104 us	10^8
Reader (non-real-time without interferences)	26.561 us	0.621 us	1.52 us	10^8
Reader (non-real-time with 1 repetition)	129.856 us	1.975 us	0.882 us	322
Reader (non-real-time with 2 repetition)	-	-	-	0

The worst case response times in Tables 4 and 5 are lower the more repetitions there are because as the number of observations decreases the probability of suffering an impact from non-masking interruptions or kernel threads is significantly less. Although the previous statement may seem contradictory, the number of observations with repetitions in the non-real-time parts is low, therefore it is highly unlikely that interferences caused by the system will occur in these time intervals. These interferences as we have already commented in section V take place because the isolation mechanisms used in Android/Linux cannot avoid all the interrupts and kernel threads.

TABLE 6. Measurements of read and write times for a 32-bit integer using the protocol called *QrtR* described in the Subsection IV-D.

	Worst response time	Average	Standard deviation	Test iterations
Reader (real-time)	11.874 us	0.891 us	0.199 us	10^8
Writer (non-real-time without interferences)	13.646 us	0.883 us	2.071 us	10^8

In the previous tables we can notice how the worst-case for non-real-time and real-time operations is large compared to the average times. In these protocols, the worst response times are due to interferences caused by kernel threads that run in the same cores, in fact the impact caused by these interferences is more significant than the repetition of the operations caused by the real time part. This occurs in this way because the *QrtR* protocol runs with a small number of instructions that consumes little CPU time compared to a non-maskable interrupt. In any case, we can say that real-time operations obtain response times within the expected jitter values, that are usable in most control applications with deadlines in the millisecond range.

D. SUMMARY OF THE EVALUATION

We have evaluated the non-blocking algorithms developed on a device that uses Android. The isolation mechanisms described in Section V have been used in the device to have an environment where real-time and non-real-time applications can coexist. In this way it has been possible to evaluate the response times for the real-time and non-real-time part of our non-blocking algorithms.

In all the tests carried out, we have observed that the average times are significantly low, since the algorithms have a small number of instructions. However, the worst case response times in tests with many iterations have been significantly longer than the average times. This is explained by the fact that the isolation mechanisms used are not perfect and some interference occurs in the system that affects the isolated core of the processor. Despite this, the real-time part of our algorithms has a finite number of steps, therefore we can guarantee that without external interferences, the response times are always bounded.

VIII. CONCLUSION

In this article, we consider mixed systems where real-time and non-real-time applications coexist. Five non-blocking protocols that cover basic cases to share data between the real-time and non-real-time parts have been developed. In these protocols, real-time applications have been prioritized over non-real-time applications, allow real-time applications to have small bounded execution times, even at the expense of making response times larger for the non-real-time counterparts. The real-time part never has to repeat operations, if there is a conflict, while the non-real-time part may have to repeat the operation until there is no potential inconsistency in the data. We have tested these algorithms and evaluated their performance in an implementation over an Android/Linux operating system where two cores were used in isolation to execute the real-time tasks. Our experiments have shown that response times are suitable for soft real-time applications that need to share data with non-real-time applications. Furthermore, all the protocols developed in this work could be used in hard real-time applications if the operating system where they run is designed to support this kind of applications.

We plan to extend this work in the future by adding new protocols over different data structures, as well as, using all of them in real environments to evaluate the ability for developers with no experience in non-blocking programming to adapt or implement solutions with non-blocking approaches.

REFERENCES

[1] Z. Xiurong, "The analysis and comparison of inter-process communication performance between computer nodes," *Manage. Sci. Eng., Comput. Sci. Can. Res. Develop. Center Sci. Cultures*, vol. 3, no. 3, pp. 162–164, 2011.

[2] K. Wright, K. Gopalan, and H. Kang, "Performance analysis of various mechanisms for inter-process communication," Dept. Comput. Sci., Operating Syst. Netw. Lab, Binghamton Univ., New York, NY, USA, 2007.

[3] S. Krishnaveni and D. Ruby, "Comparing and evaluating the performance of inter process communication models in Linux environment," in *Proc. Int. J. Trend Res. Develop. (IJTRD)*, Sep. 2016.

[4] A. P. Ruiz, M. A. Rivas, and M. G. Harbour, "CPU isolation on the Android OS for running real-time applications," in *Proc. 13th Int. Workshop Java Technol. Real-time Embedded Syst. JTRES*, Oct. 2015, pp. 1–7.

[5] Y. Yan, G. Gokul, K. Dantu, S. Y. Ko, L. Ziarek, and J. Vitek, "Can Android run on time? Extending and measuring the Android platform's timeliness," *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 6, pp. 1–26, 2019.

[6] I. Kalkov, D. Franke, J. F. Schommer, and S. Kowalewski, "A real-time extension to the Android platform," in *Proc. 10th Int. Workshop Java Technol. Real-time Embedded Syst. JTRES*, 2012, pp. 105–114.

[7] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.

[8] A. Easwaran and B. Andersson, "Resource sharing in global fixed-priority preemptive multiprocessor scheduling," in *Proc. 30th IEEE Real-Time Syst. Symp.*, Dec. 2009, pp. 377–386.

[9] P. Tsigas and Y. Zhang, "Non-blocking data sharing in multiprocessor real-time systems," in *Proc. 6th Int. Conf. Real-Time Comput. Syst. Appl. RTCSA*, Dec. 1999, pp. 247–254.

[10] S. Feldman, P. LaBorde, and D. Dechev, "Tervel: A unification of descriptor-based techniques for non-blocking programming," in *Proc. Int. Conf. Embedded Comput. Syst., Archit., Modeling, Simulation (SAMOS)*, Jul. 2015, pp. 131–140.

[11] V. Nazaruk and P. Rusakov, "Blocking and non-blocking process synchronization: Analysis of implementation," *Sci. J. Riga Tech. Univ. Comput. Sci.*, vol. 44, no. 1, pp. 145–150, Jan. 2011.

[12] D. Cederman, B. Chatterjee, N. Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, "A study of the behavior of synchronization methods in commonly used languages and systems," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, May 2013, pp. 1309–1320.

[13] P. Tsigas and Y. Zhang, "Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst. SIGMETRICS*, 2001, pp. 320–321.

[14] P. Tsigas and Y. Zhang, "Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies," in *Proc. 3rd Int. Workshop Softw. Perform. WOSP*, 2002, pp. 55–67.

[15] D. Cederman, A. Gidenstam, P. H. Ha, H. Sundell, M. Papatriantafilou, and P. Tsigas, "Lock-free concurrent data structures," in *Programming Multi-Core and Many-Core Computing Systems*, S. Pllana and F. Xhafa, Eds. Hoboken, NJ, USA: Wiley, 2014.

[16] M. L. Scott, "Shared-memory synchronization," *Synth. Lectures Comput. Archit.*, vol. 8, no. 2, pp. 1–221, Jun. 2013.
[17] Y. Zhang, "Non-blocking synchronization: Algorithms and performance evaluation," Ph.D. dissertation, TU Chalmers, Gothenburg, Sweden, 2003.
[18] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Mateo, CA, USA: Morgan Kaufmann, 2011.
[19] G. Barnes, "A method for implementing lock-free shared-data structures," in *Proc. 5th Annu. ACM Symp. Parallel Algorithms Archit. SPAA*, 1993, pp. 261–270.
[20] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proc. 15th Annu. ACM Symp. Princ. Distrib. Comput. - PODC*, 1996, pp. 267–275.
[21] K. Fraser, "Pratical lock-freedom," Comput. Lab., Univ. Cambridge, Cambridge, U.K., Tech. Rep. UCAM-CL-TR-579, 2004.
[22] H. Sundell and P. Tsigas, "Scalable and lock-free concurrent dictionaries," in *Proc. ACM Symp. Appl. Comput. SAC*, 2004, pp. 1438–1445.
[23] J. H. Anderson and S. Ramamurthy, "A framework for implementing objects and scheduling tasks in lock-free real-time systems," in *Proc. 17th IEEE Real-Time Syst. Symp.*, Dec. 96, pp. 92–105.
[24] H. Kopetz and J. Reisinger, "The non-blocking write protocol NBW: A solution to a real-time synchronization problem," in *Proc. Real-Time Syst. Symp.*, Dec. 1993, pp. 131–137.
[25] A. P. Ruiz, M. A. Rivas, and M. G. Harbour, "Real-time Ada applications on Android," *Revista Iberoamericana de Automática e Informáática Industrial*, vol. 16, no. 3, pp. 264–272, 2019.
[26] A. Venkataraman and K. K. Jagadeesha, "Evaluation of inter-process communication mechanisms," *Arquitecture*, vol. 86, p. 64, 2015.

ALEJANDRO PEREZ RUIZ received the B.Sc. degree in computer engineering and the M.Sc. degree in computer science from the University of Cantabria, Spain, in 2011 and 2012, respectively. His research interests include centered in real-time systems for mobile devices, Ada for real-time applications, and software engineering.

MARIO ALDEA RIVAS is currently an Associate Professor with the Electronics and Computers Department, University of Cantabria, Spain. He is the Main Developer of MaRTE OS, an operating system that has served as a platform to provide support for advanced real-time services. His research interest includes real-time systems, with a special focus on flexible scheduling, real-time operating systems, and real-time languages. He has been involved in several research and industrial projects related with real-time and embedded technologies.

MICHAEL GONZÁLEZ HARBOUR is currently a Professor with the Department of Computer Science and Electronics, University of Cantabria. He works in software engineering for real-time systems, and particularly in modeling and schedulability analysis of distributed real-time systems, real-time operating systems, and real-time languages. He is a co-author of *A Practitioner's Handbook on Real-Time Analysis*. He has been involved in several industrial projects using Ada to build real-time controllers for robots. He has participated in the real-time working group of the POSIX standard for portable operating system interfaces. He is one of the principal authors of the MAST suite for modelling and analysing real-time systems.