RTEMS Internals Manual

-how the kernel works-

author	**Luca Bonato**
version	**v1.0**
	June 21, 2016

Contents

1 | Introduction

This document aims to help in understanding the inner structure and inner working of the RTEMS kernel. This document is aimed to developers who are interested in discovering the kernel of RTEMS and that are not familiar with it: this manual should accompany them (step by step) by showing the most important features of the OS (namely scheduler, semaphores, messages and events).

The kernel of RTEMS is big, and in this document only a little subset of it is explained. This document focuses on the time-composable version of RTEMS produced at the University of Padua based on the work-in-progress RTEMS 4.11 at commit 40d24d54ab59fdb2e4133128bf184ec8935f3545 (April 2015). *The focus is on the SMP personality of the kernel (which is only a proof of concept in version 4.11), and specifically targets the fixed-priority scheduler and the RTEMS API (no POSIX).*

However, even if it is focused on a specific version of RTEMS, this document should reveal useful even for the master branch of RTEMS-SMP. Some discrepancies are to be expected, but the core concepts should still hold. Moreover, some topics are not considered because subjected to rapid change (e.g., consistency and synchronization inside the kernel) or because too platform specific (e.g., interrupts).

1.1 Time composable version

The modified version of RTEMS focuses on the 4 services listed in Table 1.1: the major changes are expected to be found on those managers[1]. With respect to the original version of RTEMS, those managers are not fully exploited: only a subset of their operations are assumed available to the user. In this way it is easier to provide a sort of guarantee on the timing behavior of the services since those services has been reduced, simplified and modified to this purpose (hardware limitations still applies, e.g., contention on the bus is intrinsically a source of variability in SMP systems).

1.2 Structure of the document

Each core functionality of RTEMS is addressed (mostly) as:

[1] The biggest changes relate to the refactoring of the implementation of MrsP (*Semaphore Manager* and the scheduler).

Table 1.1: Services modified for the RTEMS-SMP Time-Composable version.

Task Manager
Semaphore Manager
Message Manager
Event Manager

1. data structures used to implement it
   (a) brief explanation of the structure and its goals
   (b) visual representation of the structure
   (c) list and description of its fields
2. operations available
   (a) when and where the data structures are instantiated
   (b) invariants / possible state configurations of the system
   (c) interesting runtime behavior

Table 1.2 lists the topics that are discussed in this document. Other than the 4 main managers, the other topics deal with some core concepts or RTEMS. It is suggested to read the topics as they are presented: even if some concepts are strictly related, the structure of the document tries to describe all the topics in a linear and incremental way.

Table 1.2: Topics of this document.

Chains
Objects and Managers
Scheduler (Partitioned Fixed-Priority only)
Task Manager
Semaphore Manager (MrsP only)
Message Manager
Event Manager

In order to ease the understanding of the document, specific fonts and color are used.

- *Structure*: identifies data structures.
- `Structure.field`: identifies a specific field inside a structure. When the structure is obvious, it is omitted.
- `function`: identifies procedures. The signature of the procedures is omitted for clarity. Sometimes the name of a procedure is a hyperlink referencing where the procedure is described.
- `MACRO`: identifies a macro or a value of an enumeration.

A red background is used to highlight very important concepts.

A gray background is used to offer some examples, normally describing how some fields interact.

1.3 Acknowledgments

The research leading to this document has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 611085.

2 | Chains

Chains are used extensively inside the kernel of RTEMS. They are used to create ordered, mutable and dynamic lists of structures that are not assured to be contiguous in memory.

2.1 Data Structures

Chains are implemented as a double linked lists. The anchor point for the chain is *Chain_Control*. It embodies both the head and the tail of the chain. Moreover it points to both the first and the last element of the chain. The element of the chain is *Chain_Node*. It points to both the next and previous element of the chain.

2.1.1 Chain Node

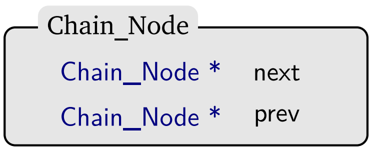

Figure 2.1: *Chain_Node*

Figure 2.1 depicts a *Chain_Node*.

`next`: points to the next node in the list.

`prev`: points to the previous node in the list.

> *The Chain_Node does not point to the complex structure that should be part of the chain: it must be part of it (be the first member)*. Being the first member, both the *Chain_Node* and the complex structure share the same address: it is then programmer responsibility to cast such address to the desired complex structure.

Since *Chain_Node*s reside inside the structures that must be listed, it is not necessary to dynamically allocate memory whenever an element must be inserted or removed from a list: the memory needed to insert/remove objects from a list is the memory occupied by the *Chain_Node* which is created at the same time as the object it represents.

2.1.2 Chain Control

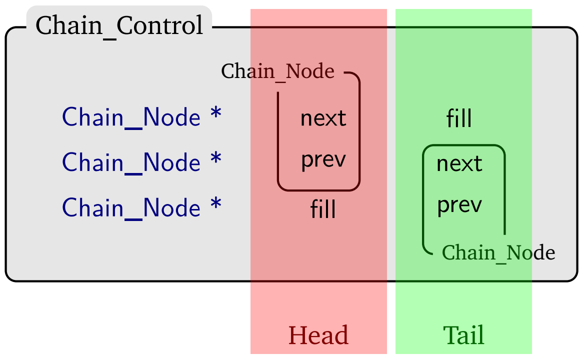

Figure 2.2: *Chain_Control*

Figure 2.2 depicts a *Chain_Control*. A `Chain_Control` is a `union` of two structures: `Head` and `Tail`. Both these structures contain 3 pointers to *Chain_Node*s, therefore they use up the same memory and each field in one structure correspond to another field in the other structure. The two fields other than `fill` are in effect a *Chain_Node* structure.

`fill`: unused. It serves as padding in order to overlay `Head.prev` with `Tail.next`.

`Node`: the chain node of the Head or Tail. In Figure 2.2 it is depicted as the two fields `next` and `prev`. Thanks to `fill`, `Head.prev` and `Tail.next` share same memory location: this memory location is set to `NULL` since Head has no predecessors and Tail has ho successors.

2.2 Usage

A chain can be modified in a protected or in a non-protected way. In case the chain is a shared resource (and therefore subject to data race) it must be modified in the protected way. How the protection is implemented is not discussed in this document.

3 | Managers and Objects

RTEMS tries its best to hide its internal structures to the user: it offers a vast interface for its services (grouped in *Managers*) while exposing very little details on its inner structure. This is accomplished by forcing the user to use *rtems_id*s, which are identifiers pointing to specific kernel objects upon which the services of a manager can be used. The user should never use the kernel structures directly: the *rtems_id* plays the same role of a pointer, but it is the manager of the service that translates the id to a specific kernel structure.

> As an example, if the user needs to lock a semaphore, he will use the service `rtems_semaphore_obtain` offered by the semaphore manager and he will supply the *rtems_id* of a previously created semaphore.

3.1 Data Structures

3.1.1 Objects Control

An *Object* in RTEMS can be defined as everything that a user can address with an *rtems_id*. Inside the kernel, the *rtems_id* is translated in order to retrieve a more complex structure, possibly a set of complex structures: this set of complex structures is indeed the real kernel object. There are lots of different kernel objects which are implemented in different way (e.g., a thread is a complete different structure than a semaphore). However, there is something in common in all these objects: they all have the structure *Object_Control* as their first member[1].

> The *Objects_Control* represents the super class of any RTEMS *Object*. *It must be the first member of any* Object, thus allowing any *Objects_Control* (the structure that any *Manager* uses to perform general operations) to be downcast to the real complex structure of the *Object*.

Figure 3.1 depicts a *Objects_Control*.

`Node`: the chain node used to place the object in a chain. The object can be inside in at most one chain at every instant.

`id`: the ID of the objects. This ID is created before the actual object it relates to. Strictly speaking, it identifies a particular memory region: the same ID can refer to differ-

[1] Recall what highlighted in Section 2.1.1: being the first member allows the *Object_Control* to be downcast to a more specific structure while it allows to treat different objects in the same way (for basic, common operations).

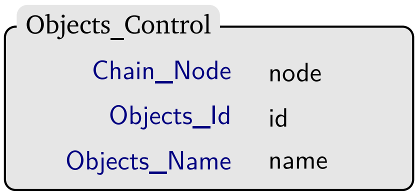

Figure 3.1: *Objects_Control*

ent objects during the life time of the system, given that those objects do not exists simultaneously. Refer to Section 3.1.3.

`name`: the name of the object. Depending on how RTEMS is compiled, it can be a 4 letters name (4 bytes) or an arbitrary length name (maximum length must be capped by the configuration of the system). It is used as a mnemonic identifier of the object: the object's *rtems_id* can be retrieved by supplying this name to `rtems_<manager>_ident`. If several objects of the same manager share the same name, one of them is retrieved. Retrieving an object by its name is discouraged. In order to retrieve the object, `rtems_<manager>_ident` performs a linear search over all the objects of the manager until one with the given name is found (the first encountered in case there are several objects with the same name).

3.1.2 Objects Id / rtems_id

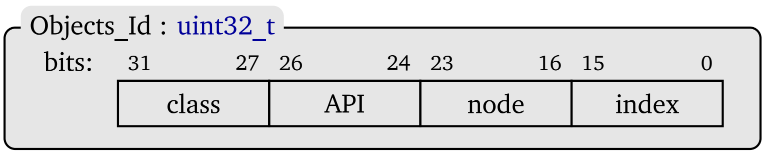

Figure 3.2: *Objects_Id* aka *rtems_id*

Figure 3.2 depicts a *Objects_Id*. This structure is available to the user as *rtems_id*.

`class`: identifies the manager which the object belongs to.

`API`: identifies the API used to create the object. There are 3 APIs:

1. Internals: used to manage kernel related structures. User never sees these objects.
2. Classic: relates to the RTEMS classic API.
3. POSIX: relates to the POSIX API.

`Node`: identifies the node where the object resides. Used only in case of MP systems (not in SMP), each node is a different binary file.

`index`: identifies the offset of the object inside the pool of memory created by the manager (see Section 3.1.3).

3.1.3 Object_Informations

A *Manager* is a set of services that works on the same set of kernel structures: such services share the same prefix.

> As an example, `rtems_semaphore_create` and `rtems_semaphore_obtain` share the same prefix: they belongs to the *Semaphore Manager*. Indeed, all services belonging to the *Semaphore Manager* share the same prefix in the format `rtems_semaphore_<operation>`.

However, for these services to be available, the kernel must first initialize the manager. As a matter of fact, inside the kernel the *Manager* is a structure: *Objects_Information*. This structure is populated by the parameters that the user specifies for the system (generally through the file `confdefs.h`). It coordinates the creation, deletion and identification of *Objects*. Moreover, it also allocates the pool of memory needed to actually store the objects it must manage.

> As an example, if the user specifies the macro `CONFIGURE_MAXIMUM_TASKS 5`, then the *Objects_Information* related to the task manager will have its `Objects_Information.max` set to 5 and it will allocate enough space in memory in order for at most 5 threads to be active simultaneously. Trying to create a 6th thread when there are already 5 present, it will result in an error because there won't be enough space to create it.

Figure 3.3 depicts a *Objects_Information* and the relationship of some of its fields with the memory it allocates/manages.

`the_api`: the API that the manager manages. All the *Objects* it creates will have this value in their *rtems_id*s.

`the_class`: the type of *Objects* it manages. All the *Objects* it creates will have this value in their *rtems_id*s.

`minimum_id`
`maximum_id`: minimum and maximum *rtems_id* that an *Object* of this manager can have. All valid *rtems_id* of this manager will be comprised in this range.

`maximum`: the maximum number of *Objects* that this manager can have.

`autoextend`: enables the manager to dynamically allocate memory for a growing population of *Objects*. This value is set if and only if the user specifies that this manager will have unlimited/unbounded *Objects*.
If this value is `false`, then the number of *Objects* will be capped by the user and the manager will allocate (only once, during initialization) enough memory for all those *Objects*. Therefore `maximum==allocation_size`.
If this value is `true`, then every time that the manager runs out of memory for its *Objects*, a new block of memory is allocated. This new block of memory can store `allocation_size` *Objects*, therefore both `maximum_id` and `maximum` are increased by `allocation_size` in order to account for these new objects that the manager can handle.

`allocation_size`: the number of objects that can be stored in a block of memory. The allocation of a block of memory is regulated by `autoextend`.

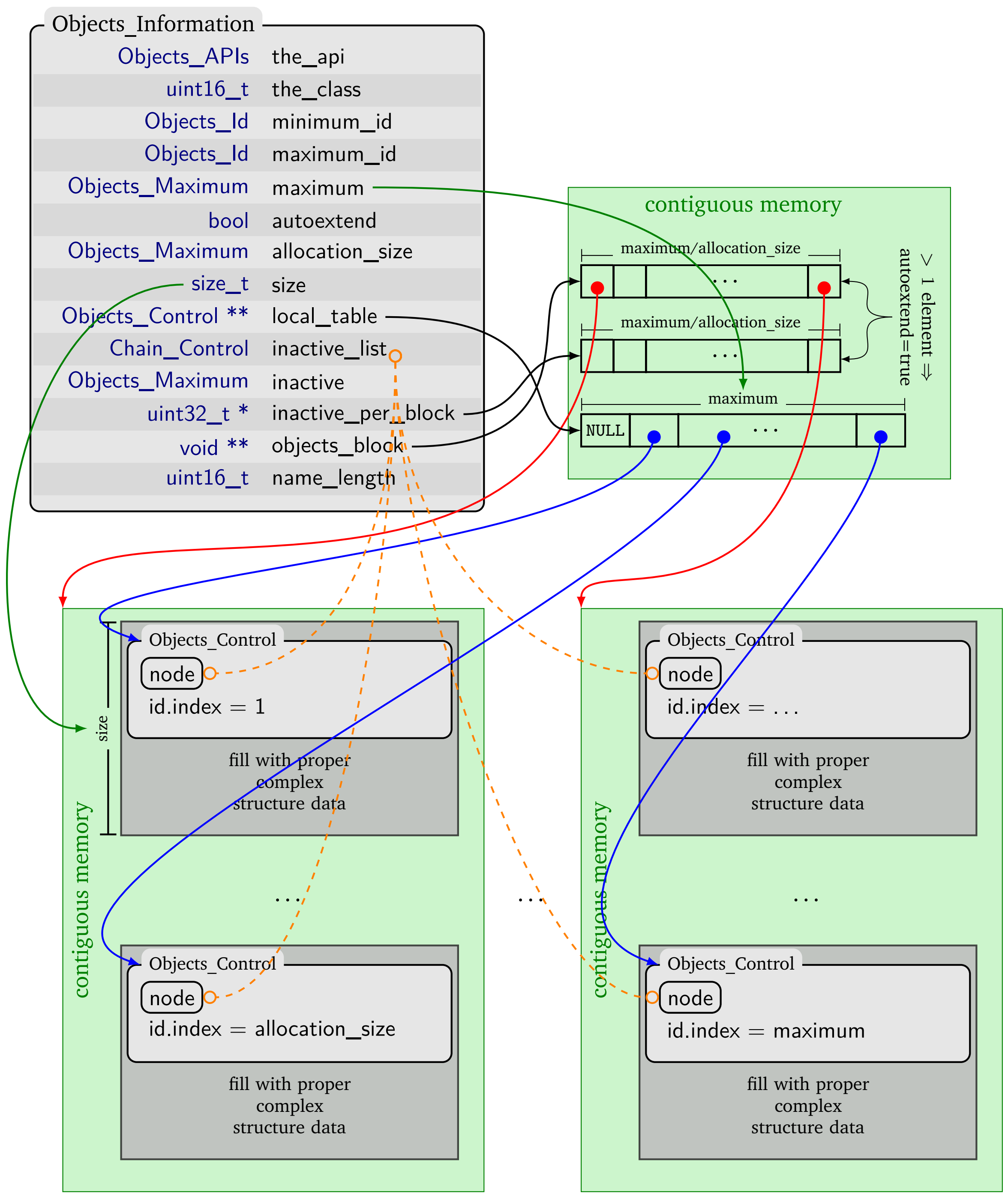

Figure 3.3: *Objects_Information*

`local_table`: a list of pointers. Each entry points to the memory location of an active *Object*, otherwise it is NULL. In this context a *Object* is active when the memory allocated to it has been claimed through an `<manager>_create` directive. The address of an active *Object* whose `rtems_id.index` is i can be stored only in the i-th entry of this array. The first entry of this table is always NULL: the value 0 for an `rtems_id.index` is not valid, as it is used as the special value `RTEMS_SELF` (meaningful for the *Task Manager*). See Section 3.2.1.3 to understand how the memory for each object is allocated.

`inactive_list`: the anchor point for the list of inactive/unused *Objects*. In this context a *Object* is inactive when the memory allocated to it has not been claimed through an `<manager>_create` service. This anchor point leverages the `Objects_Control.node` to build the chain.

`inactive`: the length of `inactive_list`, aka the total number of inactive *Objects*.

`inactive_per_block`: the number of inactive *Objects* in each block of memory. Shortcut to understand when a block of memory is fully used.

`objects_block`: a list of pointers. Each pointer points to the memory location of a memory block. A memory block is a contiguous memory region in which `allocation_size` *Objects* can be created. The number of memory blocks is regulated by `autoextend`.

`name_length`: the upper bound of the length of the name of *Objects_Control*.

3.2 Usage

A kernel *Object* is created by the manager when the user calls the service `<manager-prefix>_create`. However, this "creation" is relative to the initialization of the *Object*'s fields. The memory needed by these fields is reserved by the manager during the system start-up phase: the manager preallocates the memory that will become a kernel *Object*. While preallocating this memory, the manager also creates and initializes the *Objects_Controls* of its *Objects*. This means that the *Objects_Id* (aka *rtems_id*) is created when the memory is preallocated, and it will relates to any *Object* that will occupy that specific chunk of memory. This is because the `rtems_id.index` field is in fact the displacement inside `Objects_Information.local_table` of the pointer pointing to the chunk of preallocated memory that will become the *Object*.

3.2.1 Manager initialization

A *Objects_Information* is initialized during the start-up phase of RTEMS. The memory preallocation happens during the initialization of the *Objects_Information* or when the *Manager* runs out of inactive preallocated *Objects* and the user declared the *Manager* to have an unbounded number of *Objects*.

3.2.1.1 Manager location

The *Objects_Information* relative to a specific *Manager* is instantiated in a specific file with a specific name. These structures are then collected as pointers in a 2-dimensional array: `_Objects_Information_table[<api>][<class>] = &<the Objects_Information>`.

3.2.1.2 Manager initialization

The initialization of a *Manager* has two main steps:

1. Initialize the relative *Objects_Information*. This step is common to all *Manager*s and the inputs are values supplied by the user (generally through `confdefs.h`). Performed by `_Object_Initialize_information`.

2. Perform any *Manager* specific initialization. Each manager will supply additional information if needed.

`_Objects_Initialize_information`

1. It sets the constant values of *Objects_Information* (e.g., `the_api`, `allocation_size`).
2. It registers the *Objects_Information* to the `_Objects_Information_table`.
3. It calls `_Objects_Extend_information` to allocate the first chunk of memory.

3.2.1.3 Memory preallocation

Memory is effectively reserved with `_Workspace_allocate`, which is the RTEMS version of the C/C++ `malloc`.

The function requesting the memory allocation is `_Objects_Extend_information`. If there is already memory allocated to the specific *Manager* (i.e., `autoextend` is `true`), it will allocates a new block of memory and it will deep copy the tables. The tables are copied (and not extended) because they are arrays and must reside on contiguous memory.

`_Objects_Extend_information`

1. It allocates a new block of memory through `_Workspace_allocate`. A block of memory has size `allocation_size`×`size`.
2. It allocates a new chunk of memory where the arrays `objects_block`, `inactive_per_block` and `local_table` are stored. In case it is actually extending the available memory (i.e., it is not initializing the *Objects_Information*) it will deep copy the content of the previous arrays in the newly allocated ones.
3. It initializes the content of the new arrays.
   - `objects_block` points to the block memory allocated in point (1).
   - `inactive_per_block` is set to `allocation_size` (all newly allocated objects in the memory block are unused).
   - Each entry in `local_table` is set to `NULL` since all the *Objects* are inactive.
4. It initializes the content of the new memory block.
   - It creates `Objects_Control.id` for each *Objects_Control* depending on their position inside the memory block.
   - It initializes `Objects_Control.node` and append it to the chain `inactive_list`.

Each *Objects_Control* inside the memory block is `size` bytes apart from the others: it contains enough padding to actually store the content of the specific *Object*.

Note that `size` will also account for the memory needed to store the *Objects_Control* since it will be part of the specific *Object* structure.

3.2.2 Object creation

The creation of an object is performed by a specific manager through the service `<manager>_create`. Each one of these services follows a common template.

1. It performs some checks on the input (e.g., are the parameters within a specific range?).
2. It requests the (already preallocated) memory through `_Objects_Allocate`.
3. It initializes the *Object* specific fields.
4. It "install" the newly created object with `_Objects_Open`.

`_Objects_Allocate`

1. It gets and return the first inactive *Objects_Control* from the specific `Objects_Information.inactive_list`. In case there are no more inactive objects and `Objects_Information.auto_extend` is `true` then it calls `_Objects_Extend_information` to allocate the memory for the new *Object*.

`_Objects_Open`

1. It sets the `Objects_Control.name` with the name supplied by the user.
2. It installs the address of the *Objects_Control* inside the `Objects_Information.local_table`. The entry used is the one specified by the `rtems_id.index`:
   `local_table[<the_Objects_Control>.id.index] = &<the_Objects_Control>`

3.2.3 Object retrieval

The retrieval of an *Object* is done inside the kernel of RTEMS and never exposed to the user. This operation is performed in order to translate the *rtems_id* supplied by the user into the real *Object*. Each manager has its own specific procedure, but they all follow the same template:

1. Call `_Objects_Get` in order to get the *Objects_Control* with the supplied *rtems_id*.
2. Cast the generic *Objects_Control* to the specific data structure.

`_Objects_Get`

1. It checks that the supplied *rtems_id* is a valid id for the current *Manager* (i.e., same API, class and node).
2. It checks that the supplied *rtems_id* relates to an active *Object*.
3. It returns the *Objects_Control* registered in the `local_table` of the specific *Manager* at the entry corresponding to `rtems_id.index`.

4 | Scheduler Manager

There is not really a manager for the scheduler, at least not a manager like the one for the other services. In this chapter it is shown how the scheduler is configured and how it performs its scheduling decisions.

The focus in on the SMP Deterministic Priority Scheduler.

4.1 Threads and Scheduler Nodes

The main goal of a scheduler is to decide which one among the available threads should execute.

It is important to note that the scheduler does not directly manage *Thread_Controls*, but instead it deals with *Scheduler_Nodes* (or one of its specializations): indeed the queues inside the *Scheduler_Context* contain *Scheduler_Nodes* and not *Thread_Controls*. The *Thread_Control* remains a fundamental piece inside the scheduler, however the *Scheduler_Nodes* are the items used to perform the scheduling decisions and can be viewed as placeholders for threads. For this reason, *Thread_Controls* and *Scheduler_Nodes* have separate scheduling state. The state of a node represents the scheduling state of the placeholder with respect to a specific scheduler. The state of a thread represents the "union" of the scheduling states of the nodes that it can use to execute (possibly more than 1 if it is using MrsP resources).

This distinction between threads and nodes is forced by the use of the helping mechanism (a piece of the MrsP protocol): when a thread uses the MrsP protocol, it must be able to be scheduled (execute) in place of a thread that is waiting for the same MrsP resource. When such scenario occurs, the thread needing help starts using the *Scheduler_Node* of the thread offering help, while its own *Scheduler_Node* remains in the scheduler. In this way the migrating thread inherits with ease the scheduling state of the the thread offering help.

As an example, we can imagine *Scheduler_Nodes* as boxes that are moved and ordered inside the queues of the scheduler, while *Thread_Controls* as tokens that can be placed inside such boxes. The boxes are normally closed with their own token inside. When a thread uses the MrsP protocol, its box opens and the token can be exchanged conveniently with the tokens of other open boxes. In this way any open box that is occupying a place in the `Scheduled` queue can exchange its token with the token of the ready box whose token is the thread that holds the MrsP resource. The fact that such boxes are used in the queues of the schedulers enforces the correct behavior of MrsP: even if a thread is being helped by another scheduler, in its own

> scheduler there is a placeholder that prevents lower priority threads to execute (an important property for MrsP).

*Scheduler_Node*s are tightly coupled with the threads they belong to. They are better described in Section 5.1.4.

4.2 Data Structures

The scheduler (any scheduler) has two main goals that are embodied in the two main structures that compose it.

1. *Scheduler context*. It remembers the scheduling state of the system (e.g., which tasks are ready).
2. *Scheduler operations*. It enumerates the basic scheduling operations and link them to their specific implementation (i.e., the operations that can correctly mange the data structures of the scheduler).

4.2.1 Scheduler Context

Figure 4.1 depicts the main stateful scheduling structure used when the user chooses the deterministic SMP priority scheduler. This structure, the *Scheduler_priority_SMP_Context*, contains (recursively) as its first member its base class: the *Scheduler_Context*.

`processor_count`: the number of processors that are managed by the processors. When using a partitioned approach, this number is always 1.

`Scheduled`: the list of scheduled (i.e., running, being executed) threads. The length of this queue equals `processor_count`. This list is ordered by the priority of the thread. This list contains *Scheduler_Node*s or one of its specialization[1].

`Idle_threads`: the list of idle threads available to this scheduler. There is one idle thread for each processor that this scheduler must manage. This list contains *Thread_Control*s (not *Scheduler_Node*s). This list is used to have easy access to the idle threads when they must be used as placeholders for the MrsP protocol. See Section 4.3.1.4.

`Bit_Map`: the structure that makes the scheduler deterministic. In constant time it is able to determine which is the (index of the) highest priority ready queue inside `Ready` that is non-empty (i.e., which is the priority of the thread that has the highest priority among the ready threads). The internals of this structure is further visualized in Figure 5.3.

`Ready`: the array of ready threads. This array is in fact a 2 dimensional structure: the array `Ready` contains a chain for each one of the possible priorities that a thread can have (value specified by the user through the macro `CONFIGURE_MAXIMUM_PRIORITY`). A ready thread is appended only in the chain corresponding to its priority (the index of the chain mirrors the priority of the thread). This list contains *Scheduler_Node*s or one

[1] See note at page 22

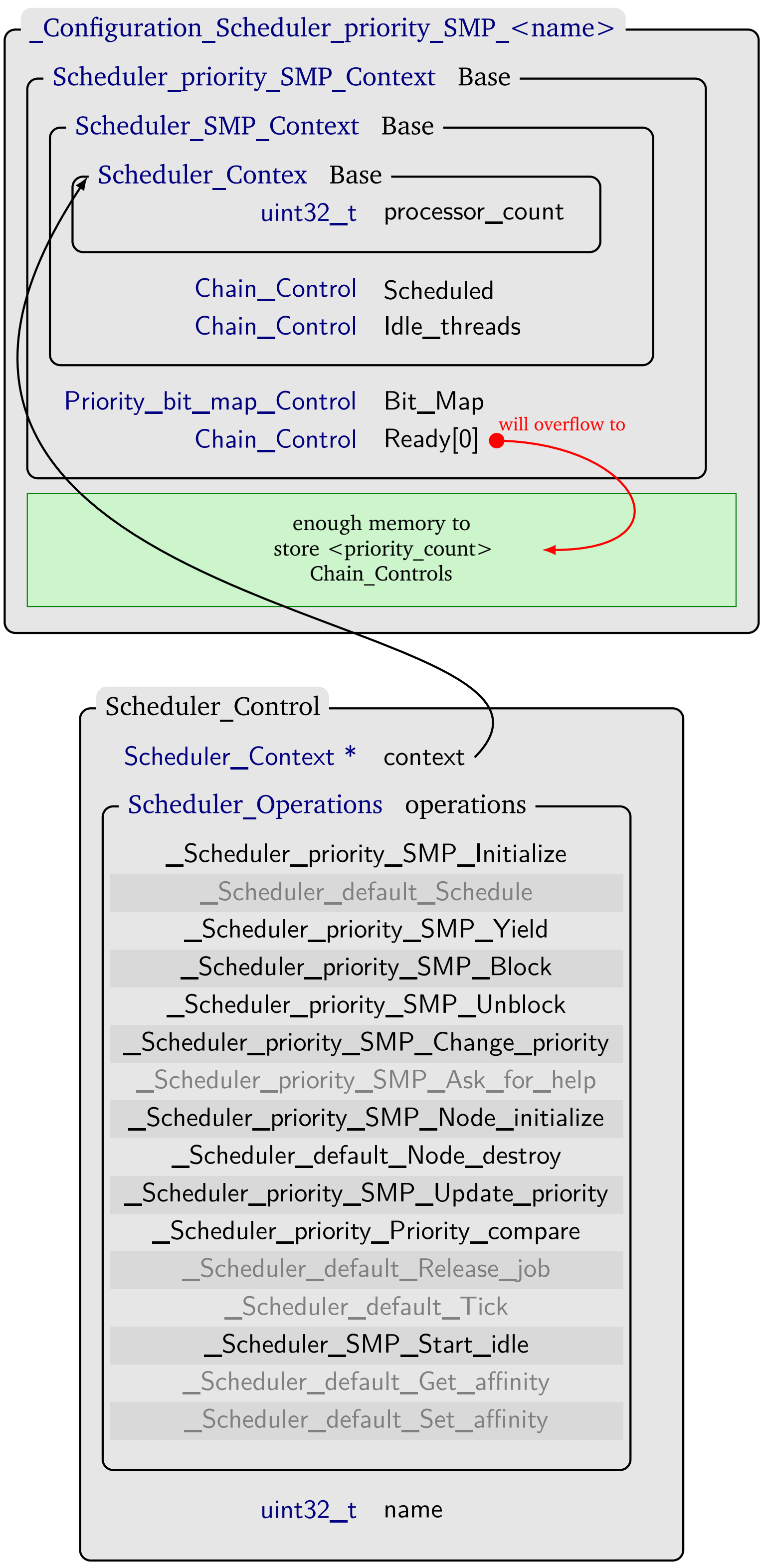
_Configuration_Scheduler_priority_SMP_<name>
Scheduler_priority_SMP_Context Base
Scheduler_SMP_Context Base
Scheduler_Contex Base
uint32_t processor_count
Chain_Control Scheduled
Chain_Control Idle_threads
Priority_bit_map_Control Bit_Map
Chain_Control Ready[0]
will overflow to
enough memory to store <priority_count> Chain_Controls
Scheduler_Control
Scheduler_Context * context
Scheduler_Operations operations
_Scheduler_priority_SMP_Initialize
_Scheduler_default_Schedule
_Scheduler_priority_SMP_Yield
_Scheduler_priority_SMP_Block
_Scheduler_priority_SMP_Unblock
_Scheduler_priority_SMP_Change_priority
_Scheduler_priority_SMP_Ask_for_help
_Scheduler_priority_SMP_Node_initialize
_Scheduler_default_Node_destroy
_Scheduler_priority_SMP_Update_priority
_Scheduler_priority_Priority_compare
_Scheduler_default_Release_job
_Scheduler_default_Tick
_Scheduler_SMP_Start_idle
_Scheduler_default_Get_affinity
_Scheduler_default_Set_affinity
uint32_t name

Figure 4.1: *Scheduler_priority_SMP_Control*

of its specialization[2]. This array is defined to have 0 elements: the correct number of elements is per-scheduler specific (defined by the user, can differ between schedulers), and these elements will be allocated immediately after `Ready` (enough memory will be reserved to allocate the whole array). Every access to this array will overflow to the specifically allocated memory that contains all the elements of the array.

4.2.2 Scheduler Operations

Scheduler_Operations is a stateless structure that gathers the base scheduling operations that the scheduler must be able to perform. These operations must be able to correctly manage the stateful part of the scheduler. Each scheduler has its own set of scheduling operation: for the deterministic SMP priority scheduler the set is defined by the macro `SCHEDULER_PRIORITY_SMP_ENTRY_POINTS`. These core operations are then used by higher level scheduling primitives.

In Figure 4.1 are shown the specific core operations used by the deterministic SMP priority scheduler. The suffix of their name represents the name of the field they belong to. Faded-out operations are not used in the deterministic SMP priority scheduler.

`initialize`: initialize the scheduler's context.

`schedule`: performs a scheduling operation. In the SMP scheduler it is unused: each operation that modify the scheduler's context triggers a re-schedule.

`yield`: yield the specified thread. A context-switch can follow. The scheduler's context is updated.

`block`: block the specified thread. A context-switch can follow. The *Scheduler_Node* of a blocked thread is not enqueued in the scheduler's context.

`unblock`: unblock the specified thread. A context-switch can follow. The *Scheduler_Node* of the thread is enqueued in the scheduler's context.

`change_priority`: change the priority of the node of the specified thread. A context-switch can follow. The *Scheduler_Node* of the thread is enqueued again based on the new priority.

`ask_for_help`: not used in the new implementation of the MrsP protocol. It was used to trigger the helping mechanism.

`node_initialize`: initialize the *Scheduler_Node* of the specified thread.

`node_destroy`: destroy the *Scheduler_Node* of the specified thread.

`update_priority`: update the priority of the node of the specified thread. It does not perform scheduling operations and it does not queue the node on the correct queue.

`priority_compare`: compare the supplied priorities.

`release_job`: unused.

[2] See note at page 22

`tick`: called by the periodic tick interrupt. It does not perform any work for P-FP scheduler.

`start_idle`: create, make ready and start the idle threads of the scheduler. Used only during the initialization of the scheduler. *It is not used to let a system idle thread execute when there are no other ready threads in the scheduler*: an idle thread will be scheduled as any other thread, and having the lowest possible priority it will execute only when there are no other ready thread.

`get_affinity`
`set_affinity`: specific for the priority affinity SMP scheduler. Unused.

4.3 Usage

4.3.1 Scheduler initialization and installation

The schedulers must be defined and initialized by the user. There are some macros that can be used in conjunction with `confdefs.h` in order ease the initialization of the schedulers.

> The specialization of *Scheduler_Node* used inside the scheduler depends on the scheduler selected by the user. The user can select a scheduler through the macro `CONFIGURE_SCHEDULER_<scheduler>`. *In this document we consider only the* deterministic SMP priority scheduler. This scheduler is selected by `CONFIGURE_SCHEDULER_PRIORITY_SMP`. The corresponding specialization of *Scheduler_Node* is *Scheduler_priority_SMP_Node*.

4.3.1.1 Scheduler installation

RTEMS expects to find informations about the schedulers in two global arrays.

`const Scheduler_Control _Scheduler_Table[]` It gathers all the schedulers structures. It is filled with the content of the user-defined macro `CONFIGURE_SCHEDULER_CONTROLS`.

`const Scheduler_Assignment _Scheduler_Assignments[]` It gathers the relationships between schedulers and processors. It is filled with the content of the user-defined macro `CONFIGURE_SMP_SCHEDULER_ASSIGNMENTS`.

> It is important to note that a scheduler cannot be created dynamically. They must be statically created at the start of the system. Indeed `_Scheduler_Table` and `_Scheduler_Assignments` arrays are defined to be `const`.

4.3.1.2 Scheduler to CPU assignment

The start-up phase of RTEMS checks `_Scheduler_Assignments` in order to understand whether to start or not a core. Table 4.1 shows the outcome of this assignment phase depending on the `attr` value specified by the user inside the *Scheduler_Assignment* and the

	CPU available	CPU not available
SCHEDULER_ASSIGN_MANDATORY	CPU wake up	fatal error
SCHEDULER_ASSIGN_OPTIONAL	CPU wake up	ignored assignment
no assignment	CPU keep sleeping	nothing happens

Table 4.1: Scheduler_Assignment vs CPUs availability

availability of CPUs. After a CPU has been awakened, the scheduler reference inside the *Scheduler_Assignment* is set to the *_Per_CPU_Control* of the specific processors: when the processors will be released (after the initialization phase has ended), each CPU will perform its scheduling decision based on the scheduler set inside its own *_Per_CPU_Control*.

4.3.1.3 Scheduler creation

The creation of a scheduler is a 3 steps operation, and for each step there is a macro (defined in `confdefs.h`) that can be used to ease the process.

1. *Create the context of the scheduler.*
   The macro `RTEMS_SCHEDULER_CONTEXT_PRIORITY_SMP(name, prio)` creates the one-time-use structures *_Configuration_Scheduler_priority_SMP_<name>*. As depicted in Figure 4.1, this structure is a wrapper for *Scheduler_priority_SMP_Context*: it contains some padding memory that will become the content of array `Ready`.

2. *Create the whole scheduler.*
   The macro `RTEMS_SCHEDULER_CONTROL_PRIORITY_SMP(name, objname)` links the previously created context (named after `name`) with a specific set of scheduler operations (specific to the deterministic priority SMP scheduler). The schedulers created in this way should be listed inside `CONFIGURE_SCHEDULER_CONTROLS` such to let `confdefs.h` to correctly create `_Scheduler_Table`. The `objname` is the name of the scheduler and is used to identify it.

3. *Create a relationship between scheduler and CPU.*
   The macro
   `RTEMS_SCHEDULER_ASSIGN(index, attr)` tells the start-up phase of RTEMS that the scheduler `_Scheduler_Table[index]` must or must not be assigned to a CPU (based on the value of `attr`).

4.3.1.4 Idle threads

Idle threads have two main uses inside the scheduler.

1. Being the idle thread when there are no other threads available to execute.

2. Being a placeholder for the MrsP protocol (see Section 6.2.1.1).

The idle threads are somehow special.

- Idle threads are created during the start-up phase by operations.start_idle. They are the default threads that should execute after the boot process ends (unless the user specifies a main thread for a specific processor/scheduler).
- The thread control block (TCB) of idle threads are stored in a different place than the TCB of user-defined threads.
- The *Scheduler_Node* of a idle thread behaves like the node of any other thread: it can be queued either in the Scheduled or Ready queue of the scheduler. On the other hand, the queue Idle_threads lists all the idle threads of the scheduler, but this queue lists directly the threads (each element of the queue is Thread_Control.Node) and not their *Scheduler_Node*s. Indeed, this latter queue is used to quickly reference the idle threads when they must be used as placeholders inside the MrsP protocol (see Section 6.2.1.1). In fact, but for for the queue Idle_threads, the scheduler has no other mean to recognize a idle thread except performing a linear search in its own queues and double checking where the TCBs of the threads reside.

4.3.2 Scheduler retrieval

A scheduler does not have a real
rtems_id. Every time the *rtems_id* of a scheduler is needed (e.g., it is necessary to let a thread migrate to a specific CPU/scheduler) it is computed on the fly. The only real information contained in this *rtems_id* is the index of the scheduler inside _Scheduler_Table.

A scheduler can be dynamically identified by
rtems_scheduler_ident. It is necessary to provide the name of the scheduler: the name of the scheduler is stored in Scheduler_Control.name (and corresponds to objname of the macro RTEMS_SCHEDULER_CONTROL_PRIORITY_SMP). *It is important that each scheduler has a unique name* otherwise it will never be addressable by the user.

rtems_scheduler_ident

1. It searches for the first scheduler inside _Scheduler_Table with the specified name.
2. It returns a *rtems_id* created on the fly based on the index of the scheduler inside the _Scheduler_Table.

4.3.3 Scheduler operations

The structure *Scheduler_Control* stores the basic operations inside operations, and those operations are used as plug-in (or a jump table). Every time a scheduling operation must be performed over a thread, the scheduler that is currently managing that specific thread is considered and its own operations are applied to the thread.

The basic operations are scheduler-specific, since they must be able to correctly manage the context of the scheduler. However, the logical steps that must be performed in order to achieve a specific side-effect over the scheduler are similar to almost all the schedulers.

> As an example, the steps needed to make the highest-priority ready thread the running thread are the same for every scheduler: remove the highest-priority thread from the ready queue, remove the running thread from the scheduled queue, put the highest-priority ready thread in the scheduled queue, put the old running thread in the ready queue, perform a context switch. What changes from scheduler to scheduler is how these small step are implemented in order to correctly manage the context of the scheduler.

In order to have a modular and reusable kernel, these small logical steps are implemented as stand-alone auxiliary procedures and are scheduler-specific. Each basic operation of the scheduler is just the instance of a general/high-level template where these auxiliary scheduler-specific procedures are used (generally, the auxiliary scheduler-specific procedures are passed as arguments to more general functions).

> As an example, the scheduler operation `operations.yield` for the deterministic SMP priority scheduler is `_Scheduler_priority_SMP_Yield`. This function is just an instance of the more general function `_Scheduler_SMP_Yield` which expects 3 functions as input: a function capable to extract a thread from the ready queue, a function capable to queue a thread to the ready queue, and a function capable to queue a thread in the scheduled queue. Having those 3 functions, the template function `_Scheduler_SMP_Yield` is able to perform a yield operation. In case of `_Scheduler_priority_SMP_Yield`, `_Scheduler_SMP_Yield` is called with 3 scheduler-specific auxiliary functions that are able to manage the context of the deterministic SMP priority scheduler (i.e., they know how to manage the queues of the scheduler).

It is useful to note that not all procedures are strictly scheduler-specific: if two or more schedulers have common (i.e., identical) internal traits, then they use the same procedures in order to manage those traits. This happens when a scheduler is a specialization of another scheduler: this sort of hierarchy between schedulers can also be seen from the prefixes of the procedures used inside the kernel. Indeed, the prefix of a procedure identifies the scheduler it addresses, and if a procedure addresses more schedulers then the prefix will reflect the first ancestor that they have in common.

> As an example, both the deterministic and the simple SMP priority scheduler have the same way to account for the scheduled queue: a single
> *Chain_Control*. Therefore, the operations that manage the scheduled queue can be shared among these two scheduler. The prefix for the deterministic SMP scheduler is `_Scheduler_priority_SMP` while the prefix for the simple SMP scheduler is `_Scheduler_simple_SMP`. Their common ancestor is `_Scheduler_SMP` and this prefix is used for those procedures that manage the scheduler queue.

In this section we try to:

1. Enumerate the auxiliary scheduler-specific procedures. We focus only on the procedures used for the deterministic SMP priority scheduler. However, by doing so we also *try to highlight the side effects and invariants that each procedure must have*. Indeed, different procedures must manage different *Scheduler_Context*s, but the generic templates expect the same behavior and same side effects.

2. Enumerate the main entries of the scheduler and show how they are redirected to the small scheduler-specific procedures.

4.3.3.1 Auxiliary scheduler-specific procedures

`_Scheduler_priority_SMP_Get_highest_ready`

It belongs to the function type *Scheduler_SMP_Get_highest_ready*.

No side effect. It returns the reference to the node of the highest-priority ready thread. Node is not moved from the queue.

`_Scheduler_SMP_Get_lowest_scheduled`

It belongs to the function type *Scheduler_SMP_Get_lowest_scheduled*.

No side effect. It returns the node of the running thread (if the scheduler manages more than 1 CPU then it picks up the lowest-priority thread that is running in any CPU). Node is not moved from the queue.

`_Scheduler_SMP_Extract_from_scheduled`

It belongs to the function type *Scheduler_SMP_Extract*.

It removes the supplied node from the scheduled queue. It does not update the internal state of the thread or node. The node must be on the scheduled queue.

`_Scheduler_priority_SMP_Extract_from_ready`

It belongs to the function type *Scheduler_SMP_Extract*.

It removes the supplied node from the ready queue. It does not update the internal state of the thread or node. It does update the bitmaps. The node must be on the ready queue.

`_Scheduler_priority_SMP_Insert_ready_fifo/lifo`

It belongs to the function type *Scheduler_SMP_Insert*.

It appends (FIFO) or prepend (LIFO) the supplied node to the correct ready queue (depending on the priority of the node). It updates the state of the bitmap.

`_Scheduler_SMP_Insert_scheduled_fifo/lifo`

It belongs to the function type *Scheduler_SMP_Insert*.

It inserts the node in the scheduled queue. The position inside the queue is decided by the priority of the node (the scheduled queue is an ordered list). FIFO or LIFO behavior applies only on the sublist of node with the same priority. This procedure should be used in conjunction with `_Scheduler_SMP_Extract_from_scheduled` in order to maintain the correct number of nodes (i.e., having as much nodes as the number of CPUs that the scheduler must manage).

`_Scheduler_priority_SMP_Move_from_scheduled_to_ready`

It belongs to the function type *Scheduler_SMP_Move*.

Same behavior as the sequence:

1. `_Scheduler_SMP_Extract_from_scheduled`
2. `_Scheduler_priority_SMP_Insert_ready_fifo`

It moves one node from one queue to the other. Bitmaps are updated. It does not update the internal state of the node or the thread.

_Scheduler_priority_SMP_Move_from_ready_to_scheduled

It belongs to the function type *Scheduler_SMP_Move*.

Same behavior as the sequence:

1. `_Scheduler_priority_SMP_Extract_from_ready`
2. `_Scheduler_SMP_Insert_scheduled_fifo`

It moves one node from one queue to the other. Bitmaps are updated. It does not update the internal state of the node or the thread.

_Scheduler_priority_SMP_Do_update

It belongs to the function type *Scheduler_SMP_Update*.

It changes the priority of the node. It updates the bitmap information inside the node. It does not manage any queue (it does not move the node in the correct queue) and it does not change the priority of the thread.

_Scheduler_priority_SMP_Enqueue_fifo/lifo

It belongs to the function type *Scheduler_SMP_Enqueue*.

It ends up calling `_Scheduler_SMP_Enqueue_ordered` with the following parameters:

1. `order` ← `_Scheduler_SMP_Insert_priority_fifo/lifo_order`
2. `insert_ready` ← `_Scheduler_priority_SMP_Insert_ready_fifo/lifo`
3. `insert_scheduled` ← `_Scheduler_SMP_Insert_scheduled_fifo/lifo`
4. `move_from_scheduled_to_ready` ←
   `_Scheduler_priority_SMP_Move_from_scheduled_to_ready`
5. `get_lowest_scheduled` ← `_Scheduler_SMP_Get_lowest_scheduled`
6. `allocate_processor` ← `_Scheduler_SMP_Allocate_processor_lazy`

It inserts a node into the correct queue: the ready queue or the scheduled queue. The input node (the node that must be queued) must have its internal state set as a ready node (e.g., the node could have just been extracted from the ready queue). In case the input node must replace a scheduled node, the internal state of both nodes and threads are updated and a context switch will be pending. In this same case, if the input node belongs to a thread using a MrsP resource, the MrsP invariant is updated to account for the new available node, and the internal state of node is updated to match the global state of the MrsP protocol.

It returns a thread that could be in need of help: the thread that was evicted from the scheduled queue (in case the input node is set to be scheduled) or the thread of the input node (in case the the input note is queued in the ready queue).

_Scheduler_priority_SMP_Enqueue_scheduled_fifo/lifo

It belongs to the function type *Scheduler_SMP_Enqueue_scheduled*.

It ends up calling
`_Scheduler_SMP_Enqueue_scheduled_ordered` with the following parameters:

1. `order` ← `_Scheduler_SMP_Insert_priority_fifo/lifo_order`
2. `extract_from_ready` ← `_Scheduler_priority_SMP_Extract_from_ready`

3. `get_highest_ready` ← `_Scheduler_priority_SMP_Get_highest_ready`
4. `insert_ready` ← `_Scheduler_priority_SMP_Insert_ready_fifo/lifo`
5. `insert_scheduled` ← `_Scheduler_SMP_Insert_scheduled_fifo/lifo`
6. `move_from_ready_to_scheduled` ← `_Scheduler_priority_SMP_Move_from_ready_to_scheduled`
7. `allocate_processor` ← `_Scheduler_SMP_Allocate_processor_lazy`

It inserts a node into the correct queue: the ready queue or the scheduled queue. The input node (the node that must be queued) must be a scheduled node that is no more queued in the scheduled queue (its internal state must be that of a scheduled node and thread). In case there are no higher priority ready nodes, the input node is reinserted in the scheduled queue. If there is a ready node with higher priority then the two nodes (and threads) are swapped and their internal state is updated to match the new position in the queues. In this latter case, if the ready node belongs to a thread using a MrsP resource, the MrsP invariant is updated to account for the new available node, and the internal state of node is updated to match the global state of the MrsP protocol .

It returns a thread that could be in need of help: the thread that was evicted from the scheduled queue (in case the input node is transitioning to the ready state).

`_Scheduler_SMP_Allocate_processor_lazy`

It belongs to the function type *Scheduler_SMP_Allocate_processor*.

It prepares the internal state of the CPU for a context switch:

- It updates the CPU reference of the thread that is going to execute.
- It updates the heir of the CPU.
- It signals the CPU that a dispatch is necessary (with a inter processor interrupt if necessary).

The "lazy" part is because it avoids (when it is possible) to migrate an already running thread: in this case[a] the running thread keeps executing in its CPU and instead the thread that was going to displace it will migrate.

[a] The scheduler must be in charge of both CPUs.

`_Scheduler_SMP_Allocate_processor`

- It updates the the internal state of the node (and thread) that is going to execute: to SCHEDULED state.
- It prepares for the upcoming context-switch (by `_Scheduler_SMP_Allocate_processor_lazy` or a type equivalent procedure).

`_Scheduler_SMP_Enqueue_to_scheduled`

This is the core of `_Scheduler_SMP_Enqueue_ordered`.

- It updates the internal state of the lowest scheduled node (and thread) that is going to be evicted: to READY state.
- It insert the next-to-run node in the scheduled queue (by

_Scheduler_SMP_Insert_scheduled_fifo/lifo).

- It moves the lowest scheduled node from the scheduled queue to the ready queue (by _Scheduler_priority_SMP_Move_from_scheduled_to_ready).
- It prompts the update to the internal state of the next-to-run node and it prompts the preparation of the context-switch (by _Scheduler_SMP_Allocate_processor).

_Scheduler_SMP_Schedule_highest_ready

It elects a ready node to become running.

- It gets the highest-priority ready node (by _Scheduler_priority_SMP_Get_highest_ready: no side effect to the ready queue).
- It updates the internal state of the highest-priority ready node in case it participates in the helping protocol: it makes sure that the MrsP invariant hold.
- It updates the internal state of the highest-priority ready node such to prepare it for its execution and context-switch (by _Scheduler_SMP_Allocate_processor).
- It moves the node from the ready queue to the scheduled queue (by _Scheduler_priority_SMP_Move_from_ready_to_scheduled).

There must be an empty slot in the scheduled queue (this procedure does not remove any scheduled node/thread). There will always be a ready node available (the idle threads).

_MRSP_check_invariant_for_resume_execution

It makes sure that the MrsP invariant holds when a *Scheduler_Node* resumes its execution.

1. If the thread of the node is waiting for a MrsP resource, it makes sure that the owner of the resource is already executing, otherwise it let the owner execute in place of the waking thread.
2. If the node belongs to the thread owning the resource, it makes sure that if the owner is already executing (because it is being helped) it will not execute in the waking node. This last case avoids to perform a migration of a running thread and uses the idle thread as a placeholder for the owner of the resource.

It updates the global state of the MrsP resources to reflect the fact that now a new processor can be used for the helping mechanism (only in the case that the thread of the node is using the MrsP protocol).

It is called by the following procedures

1. _Scheduler_SMP_Enqueue_ordered
2. _Scheduler_SMP_Enqueue_scheduled_ordered
3. _Scheduler_SMP_Schedule_highest_ready

when a node transitions from the ready to the scheduled queue.

_MRSP_check_invariant_for_stop_execution

It makes sure that the MrsP invariant holds when a thread is going to be switched-out.

1. If the thread is owning a MrsP resource, it makes sure to migrate it to a CPU where there is a thread spinning for the same resource tree.

2. If the thread is a idle thread used as a placeholder, it restores the idle thread to its rightful place. In this case the owner of the resource must be executing (it is helped) somewhere else.

It updates the global state of the MrsP resources to reflect the fact that now there is one less processor that can be used for the helping mechanism (only in the case that the thread is using the MrsP protocol).

This function is embedded inside `_Scheduler_Ask_for_help_if_necessary`.

4.3.3.2 Template scheduling procedures

`_Scheduler_SMP_Yield`

Calling tree:

`_Thread_Yield` →

`_Scheduler_Yield` →

`operations.yield` (`_Scheduler_priority_SMP_Yield`) →

`_Scheduler_SMP_Yield` using the auxiliaries:

(1) [input] `_Scheduler_priority_SMP_Extract_from_ready`

(2) [input] `_Scheduler_priority_SMP_Enqueue_fifo`

(3) [input] `_Scheduler_priority_SMP_Enqueue_scheduled_fifo`

(4) [embedded] `_Scheduler_SMP_Extract_from_scheduled`

In case the thread to yield is running, its node is extracted from the queue `Scheduled` with (4) and is then enqueued again using (3).
In case the thread to yield is not running, its node is extracted from the queue `Ready` with (1) and is then enqueued again using (2).
Cannot be used on a thread that is using the MrsP protocol.

`_Scheduler_SMP_Block`

Calling tree:

`_Thread_Set_state` →

`_Scheduler_Block` →

`operations.block` (`_Scheduler_priority_SMP_Block`) →

`_Scheduler_SMP_Block` using the auxiliaries:

(1) [input] `_Scheduler_priority_SMP_Extract_from_ready`

(2) [input] `_Scheduler_priority_SMP_Get_highest_ready`

(3) [input] `_Scheduler_priority_SMP_Move_from_ready_to_scheduled`

(4) [input] `_Scheduler_SMP_Allocate_processor_lazy`

(5) [embedded] `_Scheduler_SMP_Schedule_highest_ready`

(6) [embedded] `_Scheduler_SMP_Extract_from_scheduled`

It updates the state of the input thread and the node to BLOCKED.
In case the thread to block is running, it removes the node from the queue `Scheduled` with (6) and it elects a new running thread with (5) (and supplying it with (1-4)).
In case the thread to block is not running, its node is removed from the queue `Ready` with (1).
Cannot be used on a thread that is using the MrsP protocol.

`_Scheduler_SMP_Unblock`

Calling tree:

`_Thread_Set_state`, `_Thread_Ready`, `_Scheduler_default_Start_idle` →

`_Scheduler_Unblock` →

`operations.unblock` (`_Scheduler_priority_SMP_Unblock`) →

`_Scheduler_SMP_Unblock` using the auxiliaries:

(1) [input] `_Scheduler_priority_SMP_Enqueue_fifo`

It updates the state of the input thread and the node to BLOCKED.
In case the thread to block is running, it removes the node from the queue `Scheduled` with (6) and it elects a new running thread with (5) (and supplying it with (1-4)).
In case the thread to block is not running, its node is removed from the queue `Ready` with (1).
Cannot be used on a thread that is using the MrsP protocol.

`_Scheduler_SMP_Change_priority`

Calling tree:

`_Thread_Change_priority` →

`_Scheduler_Change_priority` →

`operations.change_priority` (`_Scheduler_priority_SMP_Change_priority`) →

`_Scheduler_SMP_Change_priority` using the auxiliaries:

(1) [input] `_Scheduler_priority_SMP_Extract_from_ready`

(2) [input] `_Scheduler_priority_SMP_Do_update`

(3) [input] `_Scheduler_priority_SMP_Enqueue_fifo`

(4) [input] `_Scheduler_priority_SMP_Enqueue_lifo`

(5) [input] `_Scheduler_priority_SMP_Enqueue_scheduled_fifo`

(6) [input] `_Scheduler_priority_SMP_Enqueue_scheduled_lifo`

(7) [input] `_Scheduler_priority_SMP_Enqueue_fifo`

(8) [embedded] `_Scheduler_SMP_Extract_from_scheduled`

It removes the node of the input thread from its queue (with (1) if in the queue `Ready` or with (8) if in the queue `Scheduled`). It updates the priority of the node with (2).
If the node was previously queued inside the scheduler (i.e., it is not in the state BLOCKED) it is queued again in the same queue. It is queued with (3) or (5) if the policy is to append the node after nodes with the same priority, or with (4) or (6) if the node must be prepended (thus maintaining a higher urgency with respect to nodes that are already queued). The node

is queued by taking into account the new priority.
This procedure returns a thread that is possibly in need of help. This candidate is (if any) a thread that is being evicted from execution: or the thread that is changing priority (by setting a lower priority) or a thread that is being replaced by the input thread (now with a higher priority). No checks are performed on the returned thread (i.e., it is possible that the thread is not using the helping protocol).
The priority of the thread (not of the node) is changed by the calling procedures (e.g., by `_Thread_Set_priority`).

`_Scheduler_SMP_Update_priority`

Calling tree:

`_Thread_Set_priority`, `_Thread_Change_priority`, `_Scheduler_Set` →

`_Scheduler_Update_priority` →

`operations.update_priority` (`_Scheduler_priority_SMP_Update_priority`):

(1) [embedded] `_Scheduler_priority_SMP_Do_update`

It updates the priority of the node with (1).
The priority of the thread (not of the node) is changed by the calling procedures (e.g., by `_Thread_Set_priority`).

`_Scheduler_priority_SMP_Ask_for_help_if_necessary`

Calling tree:

`_Scheduler_Yield`, `_Scheduler_Unblock`, `_Scheduler_Change_priority` →

`_Scheduler_priority_SMP_Ask_for_help_if_necessary`:

(1) [embedded] `_MRSP_check_invariant_for_stop_execution`

It checks whether the input thread must trigger the helping mechanism: if the thread is using the MrsP protocol then the procedure (1) is called.
The input thread comes from the scheduling operations performed by `_Scheduler_Yield`, `_Scheduler_Unblock` or `_Scheduler_Change_priority`. This thread must be a running thread that is being evicted from these operations. *It is not possible that a non-running thread can be in need of help: a thread using MrsP cannot block and therefore cannot be unblocked, it is only possible that a higher priority thread evicts an already running thread participating in the MrsP protocol.*

5 | Task Manager

The task manager offers the user a set of operations with which it is possible to control the task population of the system. This means being able to create, start, suspend, resume and delete tasks. Moreover, with the advent of the SMP part of the scheduler, it means being able to assign a task to a specific scheduler (or processor).

In this chapter it is shown how the life cycle of threads is managed and stored inside the kernel.

5.1 Data Structures

The kernel representation of a task is a *Thread_Control*, also known as thread control block (TCB). This structure is used for many different things.

1. Monitoring the thread's life cycle.
2. Keeping track of shared resources acquired or requested by the thread.
3. Managing part of the incoming and outgoing messages.
4. Storing events.
5. Other things.

Since this structure is quite overloaded by other bits coming from other managers (indeed, the thread is the actor that actually uses the other managers), in this chapter we focus only on the pieces used to monitor a thread's life cycle (also with respect to the scheduler). Other managers will describe the data structures inside the TCB that are mainly used for other purposes.

5.1.1 Thread Control

The *Thread_Control* is the base structure for a thread. Its members are known to be of constant size: it does not contain data whose size depends on some configuration option. Indeed, those fields which can vary depending on a user's configuration are defined as pointers and the actual memory is reserved inside *Configuration_Thread_control*. In this way, the kernel of RTEMS can be compiled on its own, and can be later linked with any application. Figure 5.1 shows part of this structure (it shows the fields that are useful to this manual).

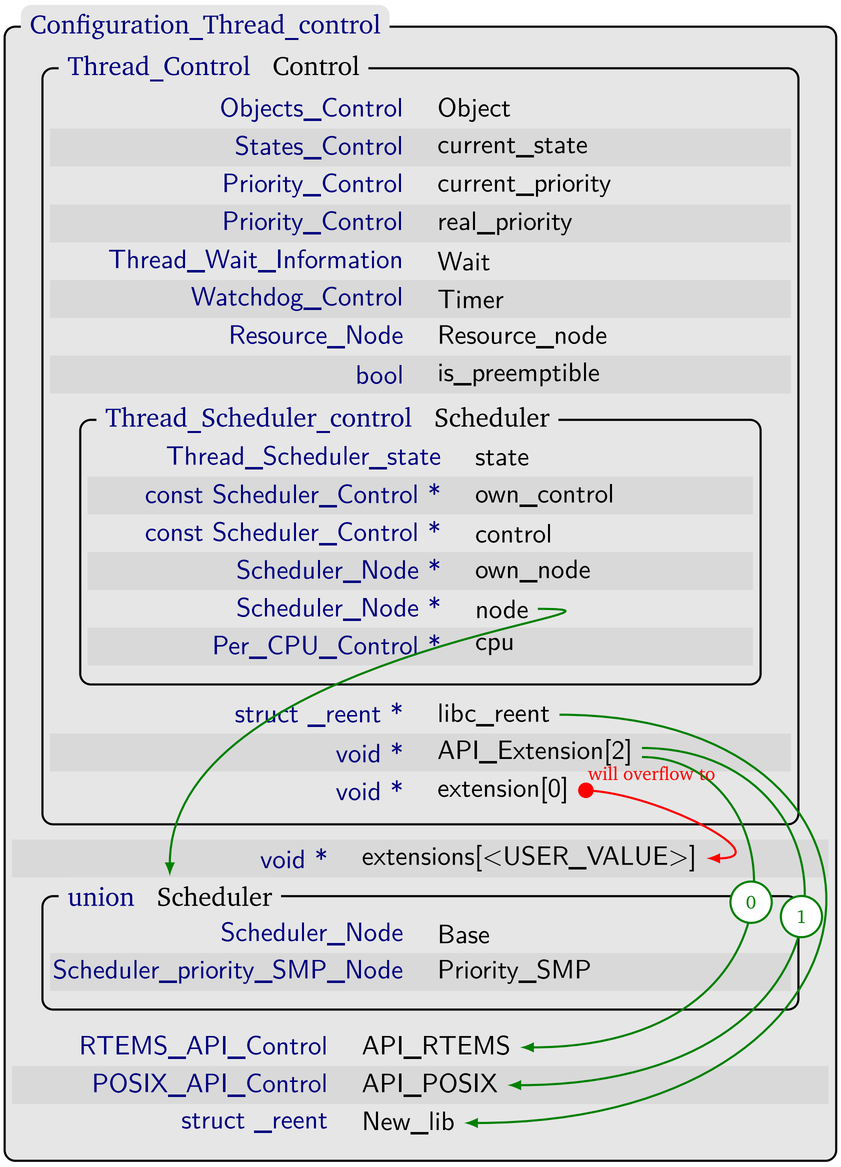
Configuration_Thread_control
Thread_Control Control
Objects_Control Object
States_Control current_state
Priority_Control current_priority
Priority_Control real_priority
Thread_Wait_Information Wait
Watchdog_Control Timer
Resource_Node Resource_node
bool is_preemptible
Thread_Scheduler_control Scheduler
Thread_Scheduler_state state
const Scheduler_Control * own_control
const Scheduler_Control * control
Scheduler_Node * own_node
Scheduler_Node * node
Per_CPU_Control * cpu
struct _reent * libc_reent
void * API_Extension[2]
void * extension[0]
will overflow to
void * extensions[<USER_VALUE>]
union Scheduler
Scheduler_Node Base
Scheduler_priority_SMP_Node Priority_SMP
0
1
RTEMS_API_Control API_RTEMS
POSIX_API_Control API_POSIX
struct _reent New_lib

Figure 5.1: *Thread_Control*. Some fields are omitted. Green lines represent the relationships established by `_Thread_Control_add_ons`.

`Object`: the *Control_Object* of the thread. It contains the data need by the manager to perform the basic operation and it contains the *Chain_Node* that can be used to append the thread to any chain/list.

`current_state`: bitmask used to remember in which state of its life cycle the thread is. It is mainly used to understand if it is possible to perform some operation on the thread (e.g., a blocked thread is not queued in any scheduler queue, if another blocking operation is performed while the thread is still blocked, it must avoid to remove its node from the scheduler). This field can assume several states (see file `score/statesimpl.h`). Most of these states can be grouped in some categories (e.g., `STATES_LOCALLY_BLOCKED`, `STATES_BLOCKED`). Such categories are in fact masks that can be used against current_state to determine if the thread is in a state of such categories. To note that the state STATES_READY represents the state of a thread that has been created and can execute if there is an available processor: in this case "ready" does not refer only to the ready queue, but also to the scheduled queue (or conversely it means that the thread is not blocked).

current_priority: the current priority of the thread. It is the dynamic priority of the thread, and can change when the thread uses priority ceiling/priority inheritance protocols.

real_priority: the original priority of the thread. This value is set when the thread is created and it changes only when the user calls rtems_task_change_priority.

Wait: this structure is used when the thread performs a blocking operation on some object. It mainly serves as a callback storage. This structure is further explained in Chapter 7 and Chapter 8.

Timer: each thread has its own timer to satisfy time-related operations (e.g., rtems_task_wake_after). This structure is not explained further.

Resource_node: this structure remembers which MrsP resources are owned by the thread and which MrsP resource it is waiting for (if any).This structure is further explained in Chapter 6.

is_preemptible: thread attribute used to make a thread non-preemptive.

Scheduler: this structure remembers which scheduler the thread belongs to and other scheduler/processor related information.

libc_reent: used for libc.

API_Extenxion: array used to reference specific API data.

```
API_RTEMS API_Extension[0]
API_POSIX API_Extension[1]
```

These extensions are defined by the kernel. The size of the array is always 2 (unless more APIs are added to the kernel).

extension: should behave in the same way as API_Extension. The possible extension(s) are user-defined. Since the size of this array is defined by the configuration of the application, it is defined to have 0 elements: it will overflow on the following memory area. This memory area is reserved by *Configuration_Thread_control* and its size is computed in order to correctly store the whole array.

5.1.2 Configuration Thread control

The *Configuration_Thread_control* is a wrapper for the thread. It is a structure that is defined in confdefs.h and therefore it exists only when the application is compiled (it is not known by the kernel). This structure varies depending on the system's configuration: most of its fields are enclosed inside #ifdef and are therefore defined only if they are used by the system. In Figure 5.1 are shown the most notable variable structure.

`Control`: this is the *Thread_Control*, the invariant part of *Configuration_Thread_control*. It must be the first member because it is in fact the superclass of *Configuration_Thread_control*.

`extensions`: this is not a structure. This is padding, memory reserved for the array `Control.extension`. `Control.extension` will overflow in this memory area.

`Schedule`: this is a union containing the possible types *Scheduler_Node*. Each type of node is enclosed inside `#ifdef` and are defined only if their relative scheduler is being used. The node `Base` that is always defined independently of the scheduler used, is in fact overlapped by any other *Scheduler_Node* (since any specialization of nodes starts with a *Scheduler_Node* as its first member). This structure is referenced by the pointer `Control.Scheduler.node`.

`API_RTEMS`: this structure contains the data used by the classic RTEMS API. This structure is referenced by the pointer `Control.API_Extension[0]`.

`API_POSIX`: this structure contains the data used by the POSIX API. This structure is referenced by the pointer `Control.API_Extension[1]`.

5.1.3 Thread Scheduler Control

The *Thread_Scheduler_Control* holds the state of the thread with respect to the scheduler. Figure 5.1 shows this structure inserted inside the *Thread_Control*.

`state`: this flag mirrors the real runtime state of the thread. This state can be different from the one of the thread's node in case the thread is using the MrsP protocol.

- `THREAD_SCHEDULER_SCHEDULED`: the thread is currently executing on a processor. The thread is using a node that is in the `Scheduled` queue.
- `THREAD_SCHEDULER_READY`: the thread is not executing because some other thread is using the processor. The thread is using a node that is in the `Ready` queue.
- `THREAD_SCHEDULER_BLOCKED`: the thread can not execute because it is blocked (e.g., waiting on an already occupied mutex, sleeping for some ticks). The thread is using a node that is not queued in any scheduler's queues. Most of the times, the thread (not node) is queued on the object that will wake it (e.g., a thread blocked while waiting for a message queues itself on the message queue so that when a message arrives the thread can be woken up).

`own_control`: a pointer to the *Scheduler_Control* to which the thread belong. This field is set when a thread is created and is updated only when the user explicitly forces the thread to change scheduler with `rtems_task_set_scheduler`.

`control`: a pointer to the *Scheduler_Control* in which the thread is. This field differs from `own_control` only when a thread uses the helping mechanism: when a thread migrates to another node because of MrsP it updates this field to point the scheduler to which it migrates.

`own_node`: a pointer to the *Scheduler_Node* that belongs to the thread. This field is set when a thread is created and is updated only when the user explicitly forces the thread to change scheduler with `rtems_task_set_scheduler`. In this case the *Scheduler_Node* must be created again since the *Scheduler_Node* is tightly coupled to the scheduler (each scheduler uses a different specialization of *Scheduler_Node*) and this pointer must then be updated. The *Scheduler_Node* it points to is located in the *Configuration_Thread_control* of the thread.

`node`: a pointer to the *Scheduler_Node* that is currently using. This field differs from `own_node` only when a thread uses the helping mechanism: when a thread migrates to another node this field is updated.

`cpu`: a pointer to the structure holding per-cpu specific information (e.g., which thread is executing? is a dispatch pending on this processor?). This field is updated every time the thread execute on a different processor. It is useful to note that this can happen also without changing the scheduler since a scheduler can manage more that 1 CPU. While using a partitioned system, this field can change because the user is forcing a migration on the thread or because the thread is using the helping mechanism of MrsP.

5.1.4 Scheduler Node

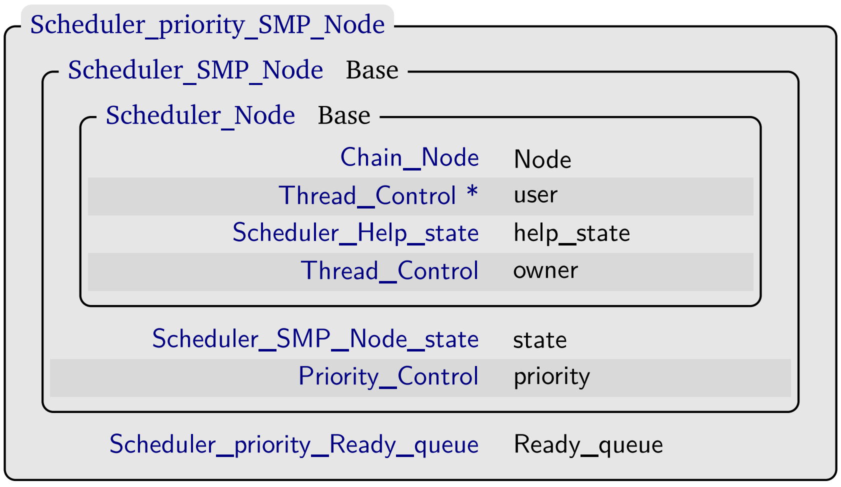
Scheduler_priority_SMP_Node
Scheduler_SMP_Node Base
Scheduler_Node Base
Chain_Node Node
Thread_Control * user
Scheduler_Help_state help_state
Thread_Control owner
Scheduler_SMP_Node_state state
Priority_Control priority
Scheduler_priority_Ready_queue Ready_queue

Figure 5.2: *Scheduler_Node*

The *Scheduler_Node* is the item used by the scheduler to organize its queues. It acts as a placeholder for the thread (see Section 4.1) and each thread has its own. It is instantiated when a thread is created or it is overwritten when the user manually sets a new scheduler for the thread by calling `rtems_task_set_scheduler`: each scheduler have a specialization of *Scheduler_Node* they are able to manage, and in fact the specific type of node is created by the scheduler with `Scheduler->Operations.node_initialize`. The area that the node occupies resides inside *Configuration_Thread_control*. Figure 5.2 shows the specific node used by the *Deterministic SMP priority scheduler*:

Scheduler_priority_SMP_Node. This structure contains its “base classes” *Scheduler_SMP_Node* and *Scheduler_Node*.

`Node`: the anchor point for the chains/queue.

`owner`: a pointer to the thread for which this node was created. Its value never changes.

`user`: a pointer to the thread that is currently using the node. It is the thread that will execute when the node becomes scheduled. This field differs from `owner` only in case the thread owning the node makes use of the helping mechanism of MrsP. In this case this field is either the thread that is owning a MrsP resource (the owner is helping the resource owner) or it is the idle thread (acting as placeholder while the owner of the node is being helped by someone).

`help_state`: a flag denoting the helping state of the thread *owning* the node. Useful only if when the thread uses MrsP.

- `SCHEDULER_HELP_YOURSELF`: the thread is not using any MrsP resource and does not participate in the helping mechanism. This means that the state of the node and the state of the thread matches, and that the thread can use only its own node[1].
- `SCHEDULER_HELP_ACTIVE_OWNER`: the thread is using the MrsP protocol and it has acquired/locked the last resource it requested. This flag means that the thread owning this node can execute using other *Scheduler_Node*s. Such nodes must participate in the helping mechanism (their `help_state` must be `SCHEDULER_-HELP_ACTIVE_RIVAL`) and must be related (directly or indirectly) to the same MrsP resource.
- `SCHEDULER_HELP_ACTIVE_RIVAL`: the thread is using the MrsP protocol and its last request has not been satisfied (i.e., the last requested MrsP resource is locked by some other thread). This flag means that the node is available to host and to let execute the resource holder.
- `SCHEDULER_HELP_DUMMY_IDLE`: the thread is the idle thread and it is executing as placeholder in another node. A node in this state is never going to be scheduled (its owner, the idle thread, is executing as placeholder on a node that has -by definition- a higher priority). This flag is used to understand if an evicted thread is in fact just the idle thread playing as placeholder.

`state`: the runtime state of the node. This flag represent the state of the node with respect to the scheduler (i.e., in which queue it is staying and therefore what the thread using this node should do).

- `SCHEDULER_SMP_NODE_READY`: the node is queued in the `Ready` queue. The thread `user` cannot execute.

[1] Except for the idle thread. The idle thread is always participating in the helping mechanism as a placeholder

- SCHEDULER_SMP_NODE_SCHEDULED: the node is queued in the Scheduled queue. The thread user is executing on a processor managed by the scheduler.
- SCHEDULER_SMP_NODE_BLOCKED: the node is not queued in any queue and the thread user must not execute. The operation Scheduler->Operations.unblock will enqueue the node in the correct queue. A node can be in this state only if its help_state is SCHEDULER_HELP_YOURSELF, indeed a thread using a MrsP resource is forbidden to perform blocking operations and therefore its node will never block: this is to avoid to break MrsP.

priority: the priority of the node. It mirrors the priority of its owner thread.

Ready_queue: a structure to ease the insertion of the node in the Ready queue and to ease the updating of the Bit_map of the scheduler. See Section 5.1.5.

5.1.5 Priority Masks

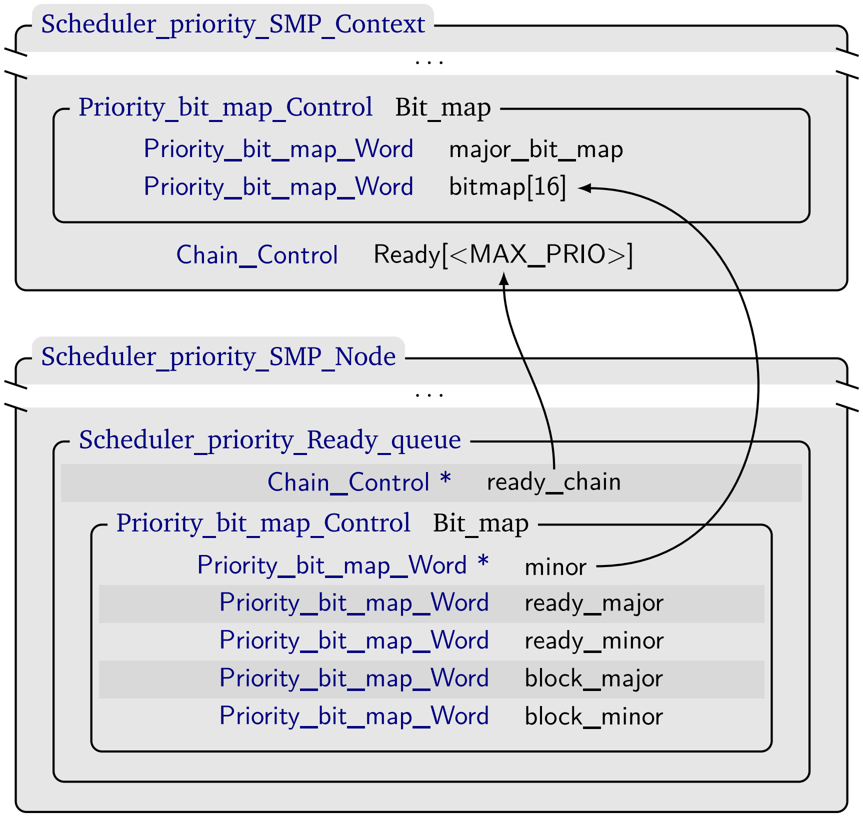
Scheduler_priority_SMP_Context
...
Priority_bit_map_Control Bit_map
Priority_bit_map_Word major_bit_map
Priority_bit_map_Word bitmap[16]
Chain_Control Ready[<MAX_PRIO>]
Scheduler_priority_SMP_Node
...
Scheduler_priority_Ready_queue
Chain_Control * ready_chain
Priority_bit_map_Control Bit_map
Priority_bit_map_Word * minor
Priority_bit_map_Word ready_major
Priority_bit_map_Word ready_minor
Priority_bit_map_Word block_major
Priority_bit_map_Word block_minor

Figure 5.3: *Priority_bit_map_Control* and *Scheduler_priority_Ready_queue*

There are 3 structures involved in easing the insertion and selection of a *Scheduler_Node* inside the Ready queue of the *Deterministic SMP priority Scheduler*. Figure 5.3 shows them and how they relate to the *Scheduler_Context* and *Scheduler_Node*.

5.1.5.1 Priority bit map Control

The structure *Priority_bit_map_Control* represents a bitset of size 256. To achieve this it is implemented as a 2-level hierarchical bitset. This bitset is used to index the highest priority ready queue that is non empty: in fact RTEMS support at most 256 thread priorities. Figure 5.3 shows this structure. It is inserted inside the scheduler, as shown in Figure 4.1, and it is used by the scheduler in order to understand (without a linear search) what is the priority of the "highest ready" thread.

`major_bit_map`: the first level of the bitset. Contains 16 bits. Each bit represents if in the relative 2nd level bitset has non-zero bits. If the i-th bit in `major_bit_map` is set, then `bit_map[i]` has at least one bit set.

`bit_map`: the second level of the bitset. This array contains 16 bitsets, and each bitset contains 16 bits. If a bitset has only zeroes, then the relative bit on `major_bit_map` is zero.

The actual priority of the "highest ready" thread is computed by combining the values from both the 1st and the 2nd level of bitsets.

> As an example, let us assume that `major_bit_map` has the highest set bit at position i, and that `bit_map[i]` has the highest set bit at position j. Then, the final priority p is $i \times 16 + j$. This priority p is then used to index the queue `Ready[p]` from which the scheduler will extract a *Scheduler_Node* (the queue will be non-empty because *Priority_bit_map_Control* is kept updated).

5.1.5.2 Scheduler priority Ready queue

The structure *Scheduler_priority_Ready_queue* represents the shortcut that a thread can use when it must manage its scheduler node inside the `Ready` queue of the scheduler. The structure *Priority_bit_map_Information* inserted inside
Scheduler_priority_Ready_queue holds additional informations that are used to quickly update the bitsets of
Priority_bit_map_Control of the scheduler when the thread's node transitions to or from the READY state. Figure 5.3 shows these structures.

`ready_chain`: a pointer to the `Ready` queue of the scheduler where the thread's node must be inserted when transitioning to the READY state.

`minor`: a pointer to the 2nd level bitset of *Priority_bit_map_Control* of the scheduler. It points to the bitset that contains the information of the `Ready` queue corresponding to the priority of the thread's node.

`ready_minor`
`block_minor`: bitmasks used to set or unset the correct bit in the bitset pointed by `minor`. The bit is set when the thread's node transitions to the READY state and there are no other nodes already queued on the same `Ready` queue. The bit is unset when the thread's node transitions out from the READY state and it is the last node queued in the `Ready` queue.

`ready_major`

block_major: bitmasks used to set or unset the correct bit in the major_bit_map of *Priority_bit_map_Control*. The bit is set when the bitset pointed by minor transitions from 0 (meaning that no nodes within the specific range of priorities are in the READY state) to ready_minor (meaning that the thread's node has transitioned to the READY state and that therefore there is now 1 ready node in the specific range of priorities). The bit is unset when the bitset pointed by minor transitions to 0 (meaning that the thread's node is no more queued in the Ready queue and that there are no other ready nodes in the specific range of priorities).

5.2 Usage

5.2.1 Thread Initialization

As explained in Section 5.1.2 the final thread structure is not known to the kernel because it depends on the configuration selected by the user. The kernel knows how to manage each sub-structure of the thread but it does not know how the memory is laid out. And it should not even care except for two points.

1. The kernel must know how big is *Configuration_Thread_control* since its manager must allocate enough space for each thread object (see Section 3.2.1.3).
2. The *Task Manager* must initialize some pointers inside *Thread_Control* to some memory addresses that depends on the layout of *Configuration_Thread_control* (see red arrows in Figure 5.1).

Thread size. To solve this issue, the size reserved for each object from the *Task Manager* depends on an extern value defined in the file confdefs.h: this value is just a sizeof() of the final thread structure, but it is defined in this file since it is in fact a configuration-dependent value.

Thread pointers. To solve this issue, the initialization sequence for a thread (i.e., whenever a thread is created) refers to the global array _Thread_Control_add_ons[]. This array, made of
Thread_Control_add_on, knows how to connect the fields inside the *Thread_Control* to the data structures laid out inside *Configuration_Thread_control*. Each *Thread_Control_add_on* has two fields.

destination: it remembers the offset of the pointer that must be initialized. This means that destination bytes after the starting address of the thread that is being created, is the address of a pointer that must point to a structure inside *Configuration_Thread_control*.

source: it remembers the offset of the starting address of a data structure inside *Configuration_Thread_control*. Since the starting addresses of *Thread_Control* and *Configuration_Thread_control* are the same[2], this offset is in fact the displacement in bytes (inside the thread that is being created) of the data structure that must be pointed from destination.

[2] This is because *Thread_Control* is the first field inside *Configuration_Thread_Control*.

> As an example, let's assume that every pointer inside this paragraph is a `char*`, such to be able to address any byte at any alignment. Given that `thread` is the pointer of the thread that is being created, for every *Thread_Control_add_on*: *(`thread`+`destination`) = `thread`+`source`.

The array `_Thread_Control_add_ons` is defined inside `confdefs.h` since it must know the final layout of *Configuration_Thread_control*. This array is then exported so that the kernel can use it at runtime.

6 | Semaphore Manager

The semaphore manager offers the user a way to synchronize tasks while using shared resources. Such manager is able to create different type of semaphores (locks) depending on the parameters that the user selects when creating one. However, while considering the SMP personality of the kernel, there is just one type of semaphore that can be used by the user when the shared data is shared by tasks that resides in different cores. This protocol, the MrsP protocol, was first presented in the paper *"A Schedulability Compatible Multiprocessor Resource Sharing Protocol - MrsP"* by A. Burns and A. J. Wellings during ECRTS 2013. Such protocol is the main focus of this chapter (we do not consider the uniprocessor resource sharing protocols supported by RTEMS).

The internals of this protocol reported in this chapter is quite different from its implementation inside the master branch of RTEMS.

It is advisable to understand how the protocol works before reading this chapter. Refer to Section 6.2.1

6.1 Data Structures

Being a ceiling priority protocol, MrsP must have hooks on several places inside the kernel.

- On the thread that is using or waiting for the resource, in order to update the internal state of the resource.
- On the resource itself, to keep track of the pending requests and the internal state of the resource.
- On the scheduler, in order to manage the ceiling priority and the eventual scheduling decisions that enforce the change in the priority of threads. Moreover, also the helping mechanism must interact with the whole scheduler to enforce the helping mechanism invariant.

6.1.1 Resource Node

The *Resource_Node* is the structure that remembers which MrsP resources are owned by a specific thread. This is the hook of the protocol inside the thread structure and each

Thread_Control has its own, as can be seen in Figure 5.1. Note that the structure itself is embedded inside *Thread_Control* as it is not a pointer: the memory it requires is accounted for in *Thread_Control*. Figure 6.1 shows this structure.

Resource_Node	
Chain_Control	resources_owned
MRSP_Control *	pending_on
uint32_t	index_of_pending_request

Figure 6.1: *Resource_Node*

`resources_owned`: the LIFO-ordered list of resources currently owned by the thread.

`pending_on`: a pointer to the MrsP resource that the thread requested but whose request could not be granted since the resource is already owned by another thread. If this pointer is not `NULL` then the thread participates in the helping mechanism whenever it uses its processor.

`index_of_pending_request`: the index of the request inside the FIFO queue of the MrsP resources. This is a useful shortcut since the FIFO queue of the MrsP resources is implemented as a circular buffer.

6.1.2 Semaphore Control

The *Semaphore_Control* is the "base class" for any semaphore that RTEMS can use. Figure 6.2 shows this structure.

Semaphore_Control		
Objects_Control	Object	
rtems_attribute	attribute_set	
union	Core_control	
	CORE_mutex_Control	mutex
	CORE_semaphore_Control	semaphore
	MRSP_Control	mrsp

Figure 6.2: *Semaphore_Control*

`Object`: the *Object_Control* of the semaphore.

`attribute_set`: the attributes specified by the user when the semaphore is created. These are used to discriminate between the 3 structures inside the `Core_control` and they are also used to discriminate the runtime behavior of the semaphore if it is not a MrsP semaphore.

`Core_control`: this is a union containing the actual semaphore-specific data of the type of semaphore selected by the user. In this chapter we focus only on *MRS_Control*.

6.1.3 MrsP Control

The *MRSP_Control* is the structure representing a MrsP resource, and as such it holds all the the MrsP-specific data. Figure 6.3 shows this structure.

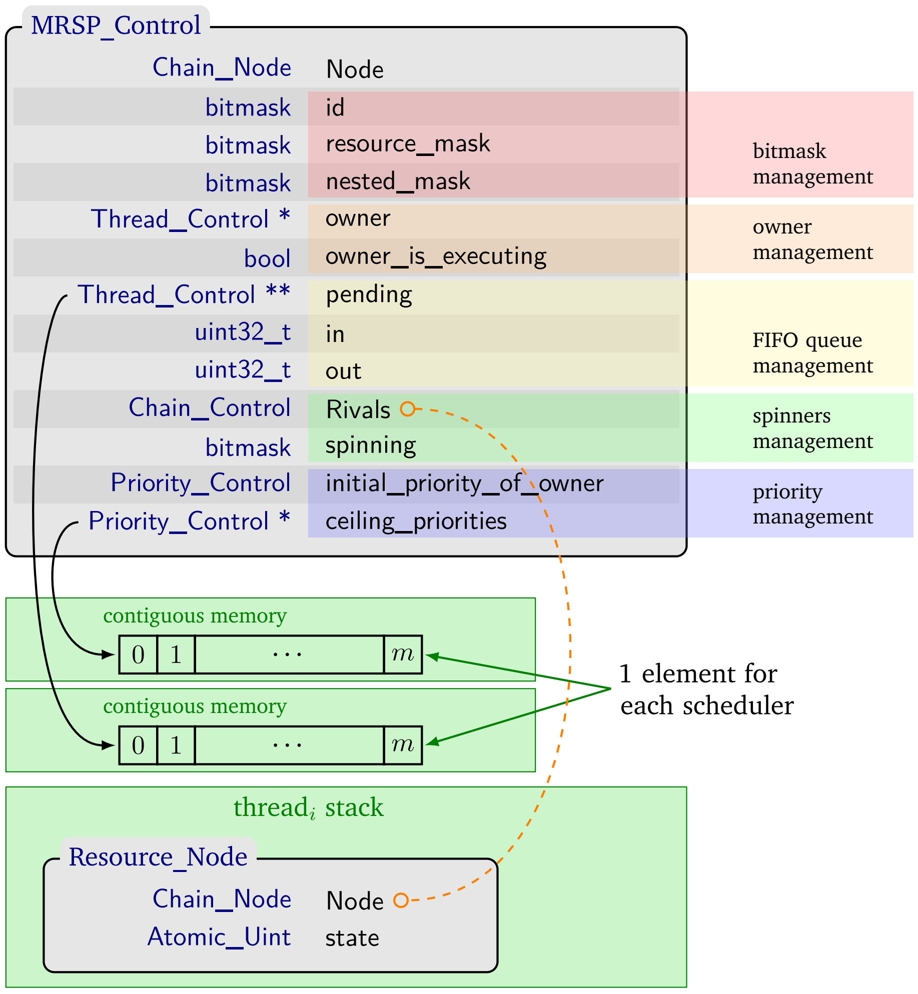

Figure 6.3: *MRSP_Control* and *MRSP_Rival*

`Node`: the chain node of the resource. It is the element that is inserted in `Resource_Node.resources_owned`.

`id`: the numeric id of the resource. This value is unique and is produced from `MRSP_global_Control.available_ids`. It is used to index the resource inside *MRSP_global_Control*: since lots of operations are managed through bitmask operations, this numeric value identifies which bit inside the bitmasks (the `id`-th bit) represents the specific resource.

`resource_mask`: a shortcut. This is the bitmask where only the `id`-th bit is set. It is used to perform bitmask operations in *MRSP_global_Control*.

`nested_mask`: a bitmask that remembers all the resources that are directly or indirectly related by nesting to this resource. *The "nesting relationship" is transitive*. A i-th bit set in this mask means that the resource whose `id` is i is related to this resource through nesting. Refer to Section 6.2.4 for the specific use of this field.

`owner`: a pointer to the thread that is currently owning the resource. This field is `NULL` if the resource is free.

`owner_is_executing`: flag to determine whether `owner` is currently executing (possibly using the *Scheduler_Node* of a rival). This field is kept updated by the scheduler through `_MRSP_check_invariant_for_XXX_execution`.

`pending`: array used to implement the FIFO queue of each MrsP resource. It is implemented as a circular buffer. It is not implemented as a list because it is related to `spinning`: `spinning` is treated as a bitmask and it produces "absolute" indexes that correspond to the elements inside this array. If if the i-th bit of `spinning` is set, it means that the thread `pending[i]` is currently spinning (or offering help to `owner`).

`in`
`out`: indexes used to manage the circular buffer `pending`. `in` remembers the first free position inside `pending`, while `out` remembers the index of the oldest (valid) element inside `pending`.

`Rivals`: the chain containing the ticket locks used to implement the spinning mechanism. The chain is a FIFO queue: it has the "same" elements and the same orders that `pending`.

`spinning`: the bitmask used to remember which of the pending thread are currently executing (i.e., which pending thread are spinning or are helping `owner`). This bitmask is used to quickly index an element inside `pending`. Its content is kept updated from the scheduler through `_MRSP_check_invariant_for_XXX_executing`.

`initial_priority_of_owner`: the priority of `owner` before it raised its priority to the ceiling of the resource. This priority is restored to the thread's node once it releases the resource. To note: this priority could be the ceiling priority of a previously nested resources, in which case that resource has the original priority of the thread's node.

`ceiling_priorities`: array used to remember the ceiling priority of the resource for each scheduler. In partitioned systems, each scheduler corresponds to a specific processor. Each value should be updated by the user through `rtems_semaphore_set_priority`.

6.1.4 MrsP Rival

MRSP_Rival is a temporary structure used to implement a MCS (Mellor-Crummey and Scott) queue-based locks. This low-level lock is used to implement the spinning mechanism. This kind of lock has been selected in the hope of exploiting the memory hierarchy of the platform: spinning is performed on a local flag, easily fitting inside L1 cache (possibly no bus accesses), whose value is updated only once by the remote task that is releasing the resource (just one bus access). Figure 6.3 shows this structure.

`Node`: the chain node inserted inside `MRSP_Control.Rivals`.

`state`: the state of the spinlock. There are only two states.

1. `MRSP_RIVAL_STATE_WAITING`: the thread that created this *MRSP_Rival* should be spinning: it is a rival.
2. `MRSP_RIVAL_STATE_NEW_OWNER`: the thread that created this *MRSP_Rival* should stop spinning since it became the new owner of the resource.

6.1.5 MrsP Global Control

The *MRSP_global_Control* is a structure summarizing the state of the protocol throughout the whole system. Its main use is to ease the enforcement of the helping mechanism by the scheduler. Its internal state is updated by both the scheduler and the semaphore manager. Figure 6.4 shows this structure. Section 6.2.4 and 6.2.3.5 further explain how this structure is used.

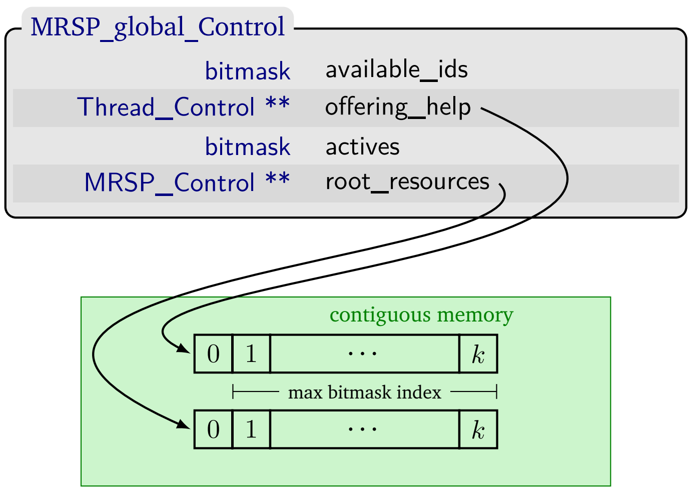

Figure 6.4: *MRSP_global_Control*

`available_ids`: a bitmask remembering which ids are still unused. Every time a MrsP resource is created a bit inside this bitmask is cleared, and every time a MrsP resource is deleted a bit inside this bitmask is set. This bitmask is an easy way to generate unique numbers[1] within a specific range. The ids produced from this bitmask are used for `MRSP_Control.id`. The ids produced from this bitmask range from 1 to the mask index inside the bitmask: the produced id values are used to extract bits in other bitmasks.

`offering_help`: an array summarizing the state of the helping mechanism. This array is in fact a dictionary: the index of the array is the `id` of a MrsP resource, while the value stored represents one of the threads that can be used for the helping mechanism in case the owner of the MrsP resource identified by `MRSP_Control.id` is not executing. If there

[1] Unique in this context means that the same number is not used simultaneously by different objects. Objects whose life times do not overlap may have/use the same unique number.

are several threads that can help the resource owner, just one of them is remembered. This array is kept updated by both the lock/unlock procedures and the scheduler: the array stored inside this array are thread that are currently executing. This array is used in conjunction with `active`.

`actives`: a bitmask remembering which resources have threads that are currently spinning. This bitmask is used to quickly indexing `offering_help`. Its value is kept updated by both the lock/unlock procedures and the scheduler. The bit i is 0 iff the resource whose `id` is i has no set bits inside `spinning`.

`root_resources`: an array summarizing how resources are nested. This array is in fact a dictionary: the index of the array is the `id` of a MrsP resource, while the value stored represents outermost resource that prevents the use of the MrsP resource identified by `MRSP_Control.id`. Such outermost resource is in fact the root of the **resource tree**. The resource tree is the hierarchical representation of the relationships formed by threads owning resources and thread waiting for resources (see Section 6.2.1.2).

6.2 Usage

6.2.1 Overview of the MrsP Protocol

MrsP is a generalization of the uniprocessor Stack Resource Protocol (SRP). It was developed to be Response Time Analysis (RTA) friendly and in such a way to minimize the blocking time of tasks pending on the same resources. In this section we assume that the protocol is defined to work in a partitioned fixed-priority system, but the protocol can be generalized.

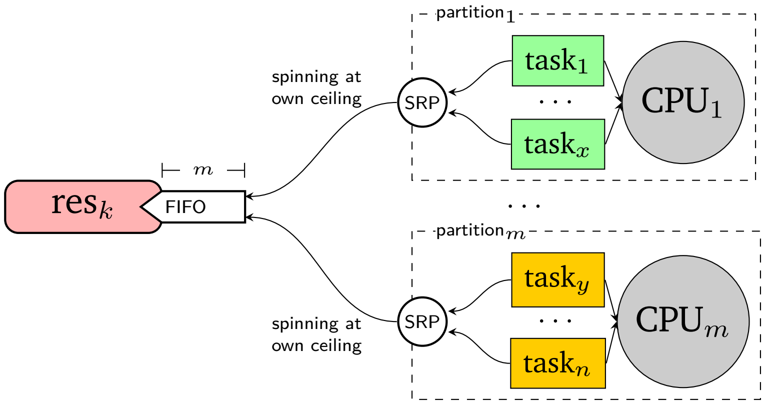

Figure 6.5: Visual representation of the MrsP protocol.

The protocol is build upon some core concepts. Figure 6.5 shows the general structure and base concepts of MrsP.

- It is a ceiling priority protocols. Every time a task acquires a MrsP lock it raises its priority to the highest priority of those tasks that reside in the same partition and that can request the same lock.

- If a task requests a lock that is already locked by a task that resides in another partition, then its request is inserted in a per-lock FIFO queue and the task start spinning at the ceiling priority of the lock. Note that every time a task requests a lock, it is assured that no other task inside the same partition has requested the same lock (because of the ceiling priority). The FIFO queue guarantees that the task will eventually become the lock holder after a finite (and accountable) amount of time. The task performs busy wait (spinning) such to ensure the ceiling priority of the protocol: no other lower priority tasks can start using the processor and possibly start using the same lock.

- It is leverages a helping mechanism in order to speed up the use of the resource (and consequently the release of the lock) by remote tasks. This helping mechanism states an invariant that must hold for every time instant: every spinning cycle performed by tasks waiting on a resource can be converted in the execution of the task holding the resource. This means that if the resource holder is preempted in its own core, then this task should progress in its execution (until it releases the resource) by using the busy wait of those tasks (if any) waiting on the same resource.

6.2.1.1 Implementation-specific issues

The protocol itself is quite general and does not bother with implementation issues, which must however be addressed in any RTOS.

Helping mechanism The protocol itself does not forces a specific implementation on the helping mechanism. The helping mechanism was implemented with a temporary migration of the task holding the resource: such a thread can temporary use the *Scheduler_Node* of a thread pending on the same resource and thus make progress on a remote processor.

Original processor becomes available If the resource holder is being helped in a remote processor and in the meanwhile its own original processor becomes available again, it was decided that the resource holder should not migrate back. In its stead, a placeholder is used to keep the processor occupied while preventing lower priority tasks to execute. Such a placeholder is in fact the idle thread using the *Scheduler_Node* of the resource holder (and thus inheriting its ceiling priority).

Blocking operations The whole protocol leverages the fact that the resource holder makes progress whenever there is at least one task busy waiting for the same resource: letting a task be blocked while holding or waiting for a MrsP resource invalidates this property. It is enforced inside the scheduler that no blocking operations can be performed on threads using the MrsP protocol

6.2.1.2 Resource Tree and Nested Resources

It is useful to note that the helping mechanism invariant can be easily generalized when tasks are permitted to nest resources, as this invariant is transitive. Indeed, a task that is nesting resources is enabled to use the spinning cycles of any of those tasks waiting for any of the resources it is nesting. Moreover, such a task can leverage all those tasks that are indirectly blocked by the resource it is nesting. Figure 6.6 shows graphically a scenario where the helping mechanism must be applied transitively. The hierarchical structure depicted in

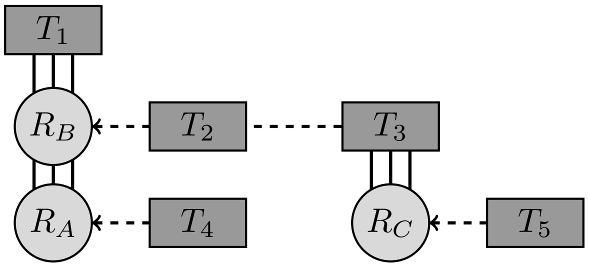

Figure 6.6: Scenario with nested resources: the property of being a "task offering help" must be transitive. In this example, task T_5 which is waiting for resource R_C, can help both task T_3, which is directly preventing its execution, *and* task T_1, which is instead doing it indirectly, through R_B. Symmetrically, a helper for T_1 is to be searched within the set of tasks it is directly or indirectly blocking.

the same Figure 6.6 is a *resource tree*: the transitivity of the MrsP invariant translates in a set of resources and threads that must all cooperate in order to make progress. In the simplest case, where threads do not nest resources, the resource tree equals to the FIFO queue of pending threads and the thread owning the resource.

6.2.2 Semaphore Initialization

The semaphore manager must preallocate enough memory in order for its objects to hold any type of semaphore: in fact the user can create at runtime any kind of semaphore, and such semaphore-object are initialized only at runtime. This is partly done by the union `Core_control` inside *Semaphore_Control*: it will hold enough memory for the largest data structure among the ones listed. `confdefs.h` also account for the memory through several macros: indeed a MrsP resource must be able to allocate several arrays and such memory overhead is accounted for inside `confdefs.h` through the macro
`CONFIGURE_MEMORY_FOR_MRSP_SEMAPHORES`.

6.2.2.1 MrsP Global Control Initialization

The two arrays used by *MRSP_global_Control* must be allocated and initialized during the start-up phase. The current implementation creates these two arrays inside `confdefs.h`. Their size depends on the number of MrsP resources needed by the user, and such value is upperbounded by the maximum index that the type *bitmask* can produce.

6.2.2.2 MrsP Control Initialization

As shown in Figure 6.3, the two arrays
`pending` and `ceiling_priorities` are not embedded inside *MRSP_Control* because their size is user-defined (upperbounded by the total number of schedulers in the system). Each

of these arrays are created by rtems_workspace_allocate, and their memory overhead is accounted for inside confdefs.h. Such arrays are initialized with default values.

- pending is filled with NULL pointers.
- ceiling_priorities is filled with the value (a single value) supplied by the user. The user should then update this array with rtems_semaphore_set_priority in order to set the specific ceiling priority for each scheduler.

The id of a resource is extracted from MRSP_global_Control.available_ids. Every time a resource is created, a bit is cleared inside available_ids and the index of such bit becomes the id of the resource. When the resource is deleted, its id is used to set the corresponding bit inside available_ids such that other resources can be created. This is an important step since most of the bitmask operations rely on the fact that each MrsP resource has a unique id and that such id can be used as index inside several arrays (such arrays are in fact dictionaries whose key is the numeric id of the resource).

6.2.3 Runtime MrsP Resource Management

As shown in Figure 6.3, the *MRSP_Control* contains several fields that can be grouped together since they are used for a specific purpose.

6.2.3.1 Owner Management

The management of the owner is straightforward. Once a thread becomes the owner of a resource the field MRSP_Control.owner is updated.

The field MRSP_Control.owner_is_executing is updated by the lock/unlock procedures or by the scheduler itself.

- The lock procedure updates this field in case a thread requests a resource while the owner of the resource is not executing. The helping mechanism is applied and as a side effect also this field.
- The unlock procedure updated this field in case the next thread that must use the resource is not executing (and there are no other spinning threads).
- The scheduler updates this field every time the helping mechanism must be applied or when the thread owning the resource is preempted while there are no spinning threads available.

6.2.3.2 FIFO Queue Management

The FIFO queue of the resource (MRSP_Control.pending) is implemented as a circular buffer in order to avoid to perform a linear search in a list looking for a non-preempted spinning thread whenever the helping mechanism must be applied, and this can be avoided thanks to the bitmask MRSP_Control.spinning (see Section 6.2.3.4). The parameters in and out are used as indexes pointing to the head and tail of the circular buffer, and they are updated every time a thread is inserted or removed from the FIFO queue. Every time a thread is inserted in the FIFO queue, the position of the circular buffer where it is inserted is stored in `Resource_Node.index_of_pending_request` (since each thread can be pending on at most one resource at a time, a single field suffices). This field is used by the scheduler to update the bitmask `spinning` whenever a spinning thread is preempted or resumes its execution.

6.2.3.3 Ceiling Priority Management

Once a thread requests a MrsP resource it must inherit the ceiling priority of the resource. As a first thing, the priority is changed only on the *Scheduler_Node* of the thread, while the thread maintains its own priority: indeed this change relates to the runtime state of the scheduler, which is kept within the nodes and not the threads. As a second thing, it is important to remember which was the priority of the *Scheduler_Node* before its thread attempted to request the resource: such priority must be restored once the thread releases the resource. Moreover, since threads can nest resources and as such they can inherit a cascading set of ceiling priorities, there must be a way to remember the priority of the node at each step of its nesting.

Ceiling priority. The ceiling priority is extracted from `MRSP_Control.ceiling_priorities`, using as index the scheduler to which the node belongs to. It is important to note that if a thread nests resources while being helped (while using the *Scheduler_Node* of a spinning thread) then the ceiling priority must be updated on the original node of the thread, and *the helping node must not be updated*. This is because the helping node has no relation with the newly nested resources: it is offering help for the base resource and as such there will probably be higher priority tasks in its scheduler that should not be delayed because the helped (alien) thread is nesting other resources. The thread can however keep using the *Scheduler_Node* of the spinning thread as long as it is not preempted.

Original priority. It is possible to implement the history of the several ceiling priorities without using lists. This is possible thanks to the thread stack and to the field `MRSP_Control.initial_priority_of_owner`. Every time a thread requests a resource it must change the priority of its node, however while doing so it has access to its previous priority.

1. If the resource it requests is free, then such previous priority is saved inside `initial_priority_of_owner`. Since nesting resources is managed as a LIFO queue, when the thread releases the resource it will restore its priority to the value saved inside `initial_priority_of_owner`, thus obtaining the priority it had before the request (whether it was the ceiling priority of a previously obtained resource or its own original priority).

2. If the resource it requests is not free, then the thread will start spinning. This operation is “blocking”, meaning that the thread will not return from the `rtems_semaphore_obtain` procedure before it stops spinning. Therefore, the previous

priority can be stored as a temporal variable inside the stack of the thread. Once the thread stops spinning (and will therefore be the owner of the resource), it can save such value in the field `initial_priority_of_owner`.

In this way, all spinning threads can remember (step by step) the history of their ceiling priorities.

6.2.3.4 Spinning Management

The spinning management is strictly related to the helping mechanism: the spinning threads are the ones that can offer help to the resource owner.

Threads spin on a local variable (inside *MRSP_Rival*) that they create while inside the `rtems_semaphore_obtain` procedure. Being a temporary local variable on the stack of the thread, its lifetime equals the lifetime of the stack of the `rtems_semaphore_obtain` procedure. Such structure is inserted inside the chain `MRSP_Control.Rival` in a FIFO order (the same order of the FIFO queue `MRSP_Control.pending`). Since threads will keep spinning until they become the owners of the resource, there will not be inconsistencies inside the chain `MRSP_Control.Rival`: even if *MRSP_Rival*s are stack variables used in the "global chain" `Rivals`, they will not be deleted out of order, as they will be deleted at specific times (when the thread transitions from being a spinning thread to being the owner of the resource).

The field `MRSP_Control.spinning` is a bitmask remembering which of the spinning threads are currently executing. This bitmask is used to quickly select a spinning thread that can be used by the helping mechanism. This value is kept updated by the scheduler itself since every time a spinning thread is preempted or resumes its execution this field must be updated. It is important to note that this field is strictly related to the circular buffer implementing the FIFO queue of the resource: each bit inside `spinning` relates to the thread of the same index inside `pending`. The scheduler can correctly update this bitmask thanks to `Resource_Node.index_of_pending_request`: being a circular buffer, the position inside `pending` will not change during the spinning cycles of the thread, thus making `index_of_pending_request` always coherent.

When it is necessary to select a spinning thread for the helping mechanism:

1. From the bitmask `spinning` is selected the first set bit;
2. The position of this bit is translated to a index;
3. This index is used to extract an element from the array `pending`;
4. This extracted value represents a currently spinning thread that can therefore be used for the helping mechanism.

6.2.3.5 Global State Management

The global state mainly manages two pieces of information.

Resource tree summary. MRSP_global_Control.root_resources is a dictionary that can be used to understand which is the root resource for a specific resource. How this field is updated is explained in Section 6.2.3.6.

Threads offering help. There can be several threads spinning on the same resource, but only one at a time can be used for the helping mechanism. The fields
MRSP_global_Control.offering_help and
MRSP_global_Control.actives are used to leverage this observation: they are a shortcut for the helping mechanism to select which thread can offer help when necessary. These two fields are updated by both the lock/unlock procedures and the scheduler. Their updates are driven by very specific invariants (and whichever operation modifies the premises of these invariants, must then update these fields).

actives. When there is at least one thread spinning on a resource whose id is i, then <resource$_i$>.spinning must have at least one set bit. As long as <resource$_i$>.spinning is not zero, then the i-th bit of actives is set.

offering_help. This array collapses the content of array MRSP_Control.pending to a single value: one value for each available resource id. Each element can be easily computed (i.e., updated) thanks to the bitmasks inside *MRSP_Control*. Refer to Section 6.2.3.4 to understand how a spinning thread can be elected from MRSP_Control.pending in constant time.

active and offering_help must be coherent: offering_help[i]==NULL iff the i-th bit in active is not set.

6.2.3.6 Nested Resources Management

Managing nested resources requires a little bit of iterations over the resource tree: indeed nesting resources means that two different resource trees must be joined together, and updating a hierarchical structure requires some iterations. The building block of the resource tree is the bitmask
MRSP_Control.nested_mask: it remembers all the resources[2] of the resource tree rooted on the specified *MRSP_Control*. Looking at those nested_mask in a cascading fashion shows how resources are nested together. Moreover, in order to avoid to incrementally build the resource tree every time it is needed, the field MRSP_global_Control.root_resources acts as a cache for the most important and required information about the resource tree: which resource is the root. By knowing the root, it is possible to address every part of the whole resource tree, and more importantly, by accessing the nested_mask of the root it is possible to know all the resources that are part of the tree (important piece of information for the helping mechanism).

Acquiring a resource. When a thread nests resources, it combines two different resource trees: the one of the already obtained resources (tree Old), and the one of the resource R that is going to be acquired (tree New). In case this resource R is free, then the

[2] Each bit in the bitmask corresponds to a specific *MRSP_Control*: index of the bit equals the id of the resource.

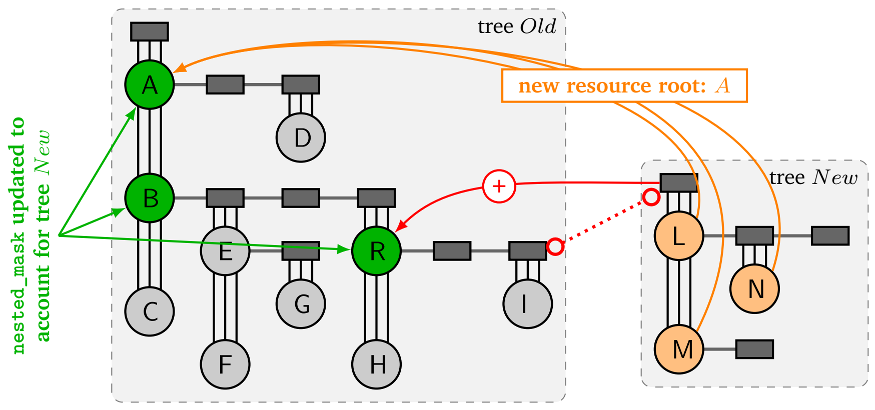

Figure 6.7: Joining two resource trees: tree New joins tree Old because the thread owning the root of tree New tries to obtain resource R. Gray rectangles represent threads, circles represent resources. Resources of tree New change their resource root from L to A (in `MRSP_global_Control.root_resources`). The resource sub-trees rooted on resources A, B and R have changed (tree New is a new branch): they must update their `MRSP_Control.nested_mask`. The non-green resources of tree Old remain untouched.

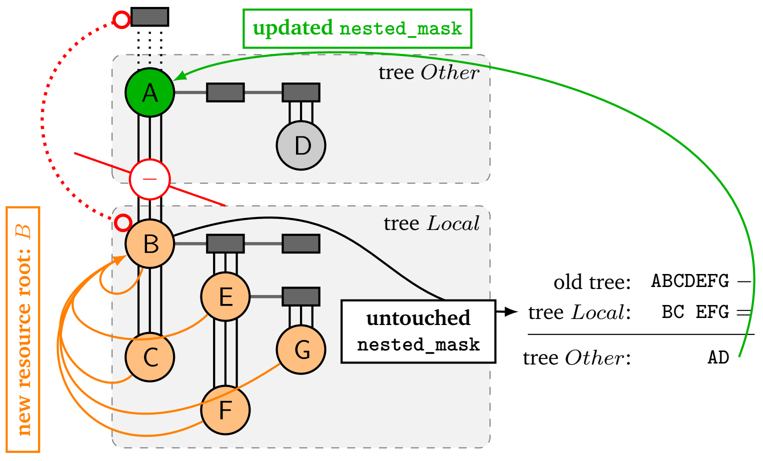

Figure 6.8: Dividing a resource tree: the thread owning the root of the old resource tree releases A, thus creating two subtrees, tree $Local$ and tree $Other$. Gray rectangles represent threads, circles represent resources. Resources of tree $Local$ change their resource root from A to B (in `MRSP_global_Control.root_resources`). The root resource in tree $Other$ changes its `nest_mask` to forget the resources belonging to the old resource tree that are now part of tree $Local$.

combination is easy since tree New comprises only resource R. On the other side, if resource R is already occupied, tree New can be a big and complex tree. In order for the bitmask `nested_mask` to hold its meaning, *part* of the tree New must be updated: some of its resources now reckon an additional branch (the new branch is tree Old). The part of the tree that must be updated, as shown in Figure 6.7, is the chain of resources that leads from resource R up to the root of the tree New. Moreover, *every* resource in tree Old has now a new resource root: therefore the array

`MRSP_global_Control.root_resources` must be updated.

Releasing a resource. When a thread releases a nested resource, it divides the resource tree into two new trees: the tree *Local* is the one that the thread still uses (whose resource root is the outermost resource locked by the thread), and the tree *Other* that comprises all the resources and threads that were still pending on the released resource. Since a thread can release only one resource at a time, updating the `MRSP_Control.nested_mask` is easy. As shown in Figure 6.8, the new root of tree *Local* knows exactly all the resources of its own resource tree: `nested_mask` remember exactly this information. This means that the tree *Other* has all the resources of the non-divided resource tree minus all the resources of tree *Local*. This subtraction can be easily performed by the bitmask. However, the tree *Local* has now a new resource root, and therefore the array `MRSP_global_Control.root_resources` must be updated.

6.2.4 Helping Mechanism

The helping mechanism revolves around enforcing that the thread owning a resource makes progress as long as there are at least one thread spinning waiting for one of the resources it blocks to be freed. This translates in making sure to pick up a spinning thread and preempt it in favor of the resource owner. Once a thread available to help is selected, it is just a matter of performing a context-switch (and possibly a migration). The selection of a helping thread can be decomposed in several steps.

1. Understanding which thread must be helped. This means understanding which is the root resource of the resource tree. The resource tree is identified by the resource(s) owned by the thread or by the resource on which a thread is pending on. If the thread is not pending on any resource (and is using the MrsP protocol), it means that it is the root of the resource tree. Otherwise, if a thread is pending on a resource, the root of the resource tree can be obtained by looking at the `MRSP_global_Control.root_resources`: the root resource is
   `<root_resource> = MRSP_global_Control.root_resources[<thread>.Resource_Node.pending_on.id]`.

2. Understanding which resources in the resource tree have spinning threads that are wasting CPU cycles. This means understanding which resources are part of the resource tree and selecting those which currently have at least one pending thread spinning. This can be done with a simple bitmask operation:
   `<active_resources> = <root_resource>.nested_mask AND MRSP_global_Control.actives`.

3. Selecting one of the spinning threads of the resource tree. This means selecting one of the “active resources” and picking one of its currently spinning threads.
   `<helping_task> = MRSP_global_Control.offering_help[FIRST_BIT_TO_INDEX(<active_resources>)]`.

This can be done in constant time and the result is correct because all the involved data structures are kept updated by both the lock/unlock procedures and by the scheduling operations.

The selection of the helping thread is performed in 4 distinct scenarios.

1. Locking a resource. When a thread requests a resource, and the owner of the resource is not executing[3].
2. Unlocking a resource. When the next-in-line pending thread is not executing: the thread releasing the resource must make sure that the MrsP invariant holds for the released resource tree.
3. Preempting the resource owner. The scheduler must make sure that the MrsP invariant holds: if there are spinning threads, then one of them must be selected as the helping thread. This is done by the `MRSP_check_invariant_for_stop_execution` procedure.
4. Resuming a spinning thread. The scheduler must make sure that the root of the resource tree is already executing, and if it is not the case then the just resumed thread is eligible as the helping thread. This is done by the `MRSP_check_invariant_for_resume_execution` procedure.

[3] In this case the selection of the helper is redundant, since the thread requesting the resource is the logical candidate.

7 | Message Manager

The message manager offers the user a set of operations which can be used to transmit one-time-use data among the task population. Section 7.2.1 illustrates its expected runtime behavior.

7.1 Data Structures

The message manager makes use of two kind of data structures.

- Some data structures are used only by the message manager itself. Such structures manages the specific aspects of the manager (e.g., how the buffer storing messages is implemented). Such structures are: *Message_queue_Control*, *CORE_message_queue_Control*, *CORE_message_queue_Buffer_control*, *CORE_message_queue_Buffer*.

- Some data structures are shared among other managers: these structures mainly relate to the suspension of tasks and their management. One of these structures can be used by at most one manager at any time (i.e., a task can not suspend itself in order to wait simultaneously for a message and an event) and is therefore placed in the TCB: it is the *Thread_Wait_information*. Another structure is instead just a template that is instantiated also by other manager, however each manager uses its field in a specific way: it is *Thread_queue_Control*. In this chapter it is shown and explained how such structures and their fields are used by the *Message Manager*.

7.1.1 Message queue Control

The *Message_queue_Control* is the overlay *Object* manipulated by the manager. It remembers how the message queue must be managed and it encapsulates the runtime state of the message queue. As shown in Figure 7.1 it is mainly a wrapper for the kernel-level message queue.

`Object`: the *Objects_Control* of the message queue. It represents the *Object* that can be used by user as a channel of communication.

`attribute_set`: the attributes specified by the user that the message queue must have (see the user manual for the list of attributes and their meaning).

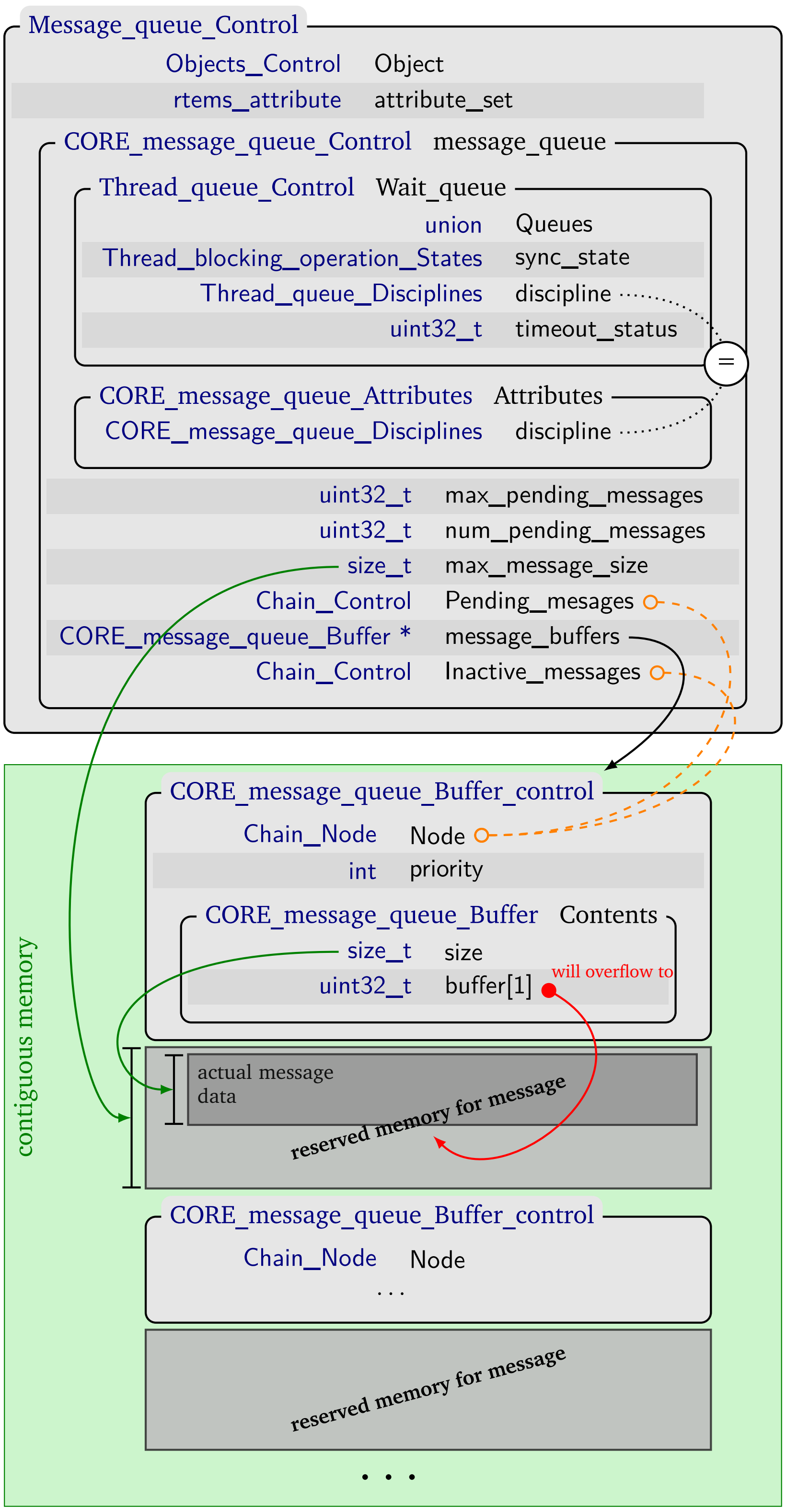
Message_queue_Control
Objects_Control Object
rtems_attribute attribute_set
CORE_message_queue_Control message_queue
Thread_queue_Control Wait_queue
union Queues
Thread_blocking_operation_States sync_state
Thread_queue_Disciplines discipline
uint32_t timeout_status
=
CORE_message_queue_Attributes Attributes
CORE_message_queue_Disciplines discipline
uint32_t max_pending_messages
uint32_t num_pending_messages
size_t max_message_size
Chain_Control Pending_mesages
CORE_message_queue_Buffer * message_buffers
Chain_Control Inactive_messages
contiguous memory
CORE_message_queue_Buffer_control
Chain_Node Node
int priority
CORE_message_queue_Buffer Contents
size_t size
uint32_t buffer[1]
will overflow to
actual message data
reserved memory for message
CORE_message_queue_Buffer_control
Chain_Node Node
...
reserved memory for message
...

Figure 7.1: Data structures for the *Message Queue Manager*.

7.1.2 CORE message queue Control

The
CORE_message_queue_Control represents the kernel-level data structure for the message queue. It contains both the pointer to buffer where the messages can be stored and the list of thread pending on the buffer (waiting of a message or waiting to write into the buffer). Figure 7.1 shows this structure and how it relates to *CORE_message_queue_Buffer_control*.

`Wait_queue`: the queue used to store the threads that are blocked on the message queue. Threads are blocked either because they are waiting for a message while the message queue is empty, or because they want to store a message while the message queue is full.

`Attributes`: a flag used to understand in which order blocked tasks should wait.

`max_pending_messages`: the maximum number of messages that can be stored in the buffer. Indeed, the memory reserved to store buffered messages depends on this value.

`num_pending_messages`: the number of messages stored on the buffer. It is the length of the queue `Pending_messages`.

`max_message_size`: the maximum size that a message sent using this message queue can have. This value is used to compute the amount of memory reserved for the buffer.

`Pending_messages`: the anchor point for the list of pending messages. The nodes of this list are `CORE_message_queue_Buffer_control.Node`. The order of pending messages depends on the primitive used by the user: using RTEMS API messages can be queued either in FIFO order (by using `rtems_message_queue_send`) or in LIFO order (by using `rtems_message_queue_urgent`). Messages can be queued based on their priority only through the POSIX APIs.

`message_buffers`: the pointer to the allocated memory that contains buffered messages. This field remembers the starting address of the memory allocated through `rtems_workspace_allocate` and it is used only to free the same memory.

`Inactive_messages`: the anchor point for the list of free entries of the message buffer. A message must be queued either in this list or in the `Pending_messages` list.

7.1.2.1 Thread queue Control

The *Thread_queue_Control* is used to manage sets of tasks blocked on a object. In this section it is shown how this structure is used by the *Message Manager*. Refer to Section 7.2.4 for further information.

`Queues`: it is actually a *union*. It is the anchor point for a list or a tree: the flag `discipline` signals which of the two structure must be used. This is the queue used to collect the set of tasks blocked on the message queue.

`Chain_Control Fifo`: the list used when tasks must be queued in FIFO order.

`RBTree_Control Priority`: the tree used when tasks must be queued in priority order.

`sync_state`: a flag (*enum*) used to make sure that the thread that is going to block on the message queue must really be blocked. It is to avoid to block a thread when an ISR or timeout that is going to unblock the same thread is fired during the blocking operation (in order to reduce the code executed while having interrupt disabled, only some pieces of the blocking procedure is executed with disabled interrupt).

`discipline`: a flag specifying what kind of `Queues` is used. It is an *enum*. It has a direct relation to *CORE_message_queue_Attributes*.

THREAD_QUEUE_DISCIPLINES_FIFO: tasks will queue in FIFO order. `Queue` is implemented using the *Chain_Control*.

THREAD_QUEUE_DISCIPLINES_PRIORITY: tasks will queue in priority order. The priority is the priority of the task (`Thread_Control.current_priority`). `Queue` is implemented using the *RBTree_Control*.

`timeout_status`: it holds the value that will be returned to threads still pending on `Queues` when their timeout fires.

7.1.2.2 CORE message queue Attributes

The *CORE_message_queue_Attributes* just encapsulates a single flag. This flag has a direct relation to *Thread_queue_Control.discipline*: it is in fact used only to understand which kind of `Thread_queue_Control.Queue` must be created.

`discipline`: it specifies in which order blocked tasks will be ordered. It is an *enum*.

CORE_MESSAGE_QUEUE_DISCIPLINES_FIFO: tasks will queue in FIFO order. Relates to `THREAD_QUEUE_DISCIPLINES_FIFO`.

CORE_MESSAGE_QUEUE_DISCIPLINES_PRIORITY: tasks will queue in priority order. Relates to `THREAD_QUEUE_DISCIPLINES_PRIORITY`.

7.1.3 CORE message queue Buffer control

The *CORE_message_queue_Buffer_control* is the wrapper for the actual message. This wrapper contains the information that are used by the *Message Manager* to manage the actual message. Figure 7.1 shows this wrapper and where it is allocated.

`Node`: the element that is used to queue the message in the list `CORE_message_queue_Control.Pending_message` (meaning that the message is buffered and contains valid data) or in the list `CORE_message_queue_Control.Inactive_messages` (meaning that the memory of this structure contains non-valid data and that it can then be used for buffering a new message).

`priority`: the priority of the message. This value is used only with the POSIX API.

7.1.3.1 CORE message queue Buffer

The *CORE_message_queue_Buffer* represents the actual message. Since the size of the message is unknown to the kernel and is specified by the user, the actual data of the message will overflow to the memory that the manager allocated exactly for this purpose. Even if the size of the messages is unknown, in order to allocate enough memory for them the manager leverage the fact that each message must have a size of at most `CORE_message_queue_Control.max_message_size` (value provided by the user during the initialization of the message queue).

`size`: the actual size of the message. This value is always smaller or equal than `CORE_message_queue_Control.max_message_size`. This field is used in order to copy only the actual content of the message. From the point of view of the kernel, a message is just a sequence of bytes with no special meaning (i.e., it is not structured data since the content of the message differs from queue to queue): in order to copy its content correctly it must know only how big (i.e., how much size) a message uses.

`buffer`: the actual content of the message. This array marks only the start of the message's content: the whole message will overflow from this array and will use the memory that follows it (such memory is preallocated when the message queue is created).

7.1.4 Thread Wait information

The *Thread_Wait_information* remembers all the information needed by a thread when it is suspended waiting for something. In this case, a thread can be waiting on a message queue in order to receive a message or waiting for the message queue to have some place in the buffer to store a message. It's main purpose is to decouple the thread to the "event" it is waiting for, and therefore storing the possible callbacks and results in an asynchronous way. In case the thread is waiting because the message queue is empty, then this structure is used to acquire the information about the message when it will be received. In case the thread is waiting because the message queue is full, then this structure is used to store the information about the message that should be sent. Since a thread can be waiting for at most one "event" at time, this structure is placed inside the TCB (as seen in Figure 5.1) and is shared among several manager. Figure 7.2 shows this structure, and in this section it is shown how its field are used by the *Message Manager*.

`id`: the id of the message queue on which the thread is waiting.

`count`: the "priority" with which the message was sent. Considering the RTEMS API, this field is used to store whether the message is sent with `rtems_message_queue_send` or `rtems_message_queue_urgent`.

`return_argument`: the address of the local variable that is going to store the size of the message. This field is used only for thread suspending while waiting for a message.

`return_argument_second`: it stores the pointer for the content of the message. It is in fact a *union* of either a *void ** or a *const void **: the difference is determined on the fact that the message must

Thread_Wait_information	
Objects_Id	id
uint32_t	count
void *	return_argument
Thread_Wait_information_Object_argument_type	return_argument_second
uint32_t	option
uint32_t	return_code
Thread_queue_Control *	queue
Thread_Wait_flags	flags

Figure 7.2: *Thread_Wait_information*

be sent (in which case it is used as *const void ** because the message is read-only) or on the fact that the message must be received (in which case it is used as *void ** since it is the pointer where the message's content must be written inside the receiving thread).

`option`: the size of the message. This field is used only when a thread suspends waiting for a free entry inside the buffer of the message queue.

`return_code`: the flag signaling whether the operation was successful or not. Since threads can be blocked on a message queue, the actual acquisition or buffering of the message can be performed by someone else, and the blocked thread is not assured to resume immediately its execution (i.e., while being blocked a higher priority thread was released preventing the blocked thread to resume its execution after it is no more waiting on the message queue). Therefore this field is used as an intermediate place to store the actual outcome of the operation. This value is then converted into *rtems_status_code* and returned to the user when the blocked thread resumes its execution.

`queue`: the queue where the thread is suspended.

`flags`: unused.

7.2 Usage

7.2.1 Overview of the Message Protocol

The message protocol offers the threads a way to communicate and synchronize. A message is a user-defined amount of data, and how this data is structured is transparent to the protocol (as long as the size of the data is known). The whole protocol is centered on the concept of message queue. A message queue is a channel over which a thread sends and receives messages. The message queue also act as a buffer for sent messages. Moreover, threads can wait (i.e., suspend with or without timer) on a message queue while waiting for a message (if the message queue is empty) or while waiting for a free entry into the buffer (when the message queue is full). Whether threads are suspends, and which policy they follow while waiting (FIFO or priority) is specified by the user.

In the time composable version of RTEMS, the message protocol acts as a sampling port. This greatly simplifies its implementation (e.g., no need for timers) and provides a more analysable runtime behavior.

7.2.2 Message Queue Initialization

The *Message Manager* is tasked to create a *Message_queue_Control* every time the user calls `rtems_message_queue_create`. It is then its duty to reserve enough memory that will be used as the buffer for the messages. The layout of the memory is shown in Figure 7.1. This (single) chunk of memory is then initialized: at regular intervals are created and instantiated the *Chain_Node* which are then chained inside `CORE_message_queue_Control.Inactive_messages`. Each of these *Chain_Node*s is placed in such a way as to have enough memory for both the structure *CORE_message_queue_Buffer_control* and the content of the message. Indeed, the size of the message's content is accounted for (because the user specifies the maximum size that a message of the queue can have) only during the initialization phase of the message queue: it is the *Message Manager* that makes sure that between any two *CORE_message_queue_Buffer_control* there is enough memory for the content of the messages. Such memory (even if it does not belong to any data structure) is used because `CORE_message_queue_Buffer.buffer` will overflow in it.

7.2.3 Sending and Receiving Messages

The message queue has a buffer where messages can be stored. This buffer is used only in case there are no threads already waiting to receive a message: in this case there is at least one waiting thread, the thread sending the message bypasses the buffer and write the content of the message directly to one pending thread (using the address provided by the pending thread's `Wait_information.return_argument_second`).

Messages can be any kind of data: the kernel is just concerned on their size. Indeed, the act of sending, buffering and receiving messages is performed with a deep copy of the content of the message. How much data needs to be copied is specified by `CORE_message_queue_buffer.size`. It is the receiving thread that must know which kind of data it receives. It is useful to note that the messages normally live on the thread stack but for the period when they are buffered on the message queue (and this is why the messages are deep copied).

7.2.4 Waiting Threads

Threads can wait on a message queue either because there are no messages pending on the queue to be received or either because the message queue is full and there is no place to store the message to send. Since these two scenarios are mutually exclusive, just a single `Wait_queue` can be used for both situations and therefore the memory footprint of the kernel can be reduced: understanding why the thread are blocked on the queue (and therefore understanding what to do with them) is just a matter of checking whether the message queue has pending messages.

8 | Event Manager

The event manager offers the user a way to synchronize the task population. Section 8.2.1 illustrates its expected runtime behavior.

8.1 Data Structures

Similar to the message manager, the event manager uses two kind of data structures.

1. One data structure is used only for the event manager: it is the one that embodies the concept of event set. This data structure is minimal: since an event is just a flag, the event set is simply a bitfield inside *_Configuration_Thread_control*. Such data structure is *Event_Control*. It is useful to note that the an “event” is not properly an *Object*, indeed it does not have an *rtems_id*: the event manager is an overlay upon the task population, and as such it uses the *rtems_id* of the thread in order to perform its operations.

2. Some data structures are shared among other managers: these structures mainly relate to the suspension of tasks and their management. These structures can be used by at most one manager at a time therefore are placed inside the TCB: they are the *Thread_Wait_information* and *Watchdog_Control*. The *Watchdog_Control* is not discussed in this document.

8.1.1 Event Control

The *Event_Control* is the structures that gathers the events. This structure is placed inside *RTEMS_API_Control* which in turn is defined inside the *_Configuration_Thread_control*: since the events are per-task specific they are placed inside the thread. It is possible to retrieve the *RTEMS_API_Control* (and therefore the
Event_Control) from the TCB: it is `Thread_Control.API_Extension[0]` (see Figure 5.1 to see its placement inside the TCB and refer to Section 5.2.1 on how this field is initialized). Figure 8.1 shows this structure.

`Event`: the structure containing the event set. The event set is just a bit field, and each bit corresponds to an event. In fact, *rtems_event_set* is just a *uint32_t* and therefore each single event is a power of two. Since each event uses exactly one bit, by combining them with the `or` operator it is possible to form a set of events. The field

RTEMS_API_Control	
Event_Control	Event
rtems_event_set	pending_events
Event_Control	System_event
ASR_Information	Signal
Thread_Action	Signal_action
uint32_t	Notepads[0]

Figure 8.1: *Event_Control* inside *RTEMS_API_Control*

`pending_events` remembers all the events sent to the task that are still not used. An event is used when a task resumes its execution because that event is necessary to satisfy its waiting condition. A set bit at position i inside `pending_events` means that the task received `RTEMS_EVENT_<i>`.

`System_event`: events used by the kernel (e.g., for the networking layer). This exists in order to not occupy “events slots” from the user.

`Signal`: structure used for the *Asynchronous Signal Routine (ASR) Manager*.

`Signal_action`: structure used for the *ASR Manager*.

`Notepads`: deprecated. Array size is 0 because it will overflow in the following memory area.

8.1.2 Thread Wait information

The *Thread_Wait_information* remembers all the information needed by a thread when it is suspended waiting for something. Figure 8.2 shows this structure, and in this section it is shown how its field are used by the *Event Manager*. This structure is placed inside the TCB, as shown in Figure 5.1.

Thread_Wait_information	
Objects_Id	id
uint32_t	count
void *	return_argument
Thread_Wait_information_Object_argument_type	return_argument_second
uint32_t	option
uint32_t	return_code
Thread_queue_Control *	queue
Thread_Wait_flags	flags

Figure 8.2: *Thread_Wait_information*

`id`: unused.

`count`: it stores the set of events upon which the task is waiting. It is used as a read-only value: each task sending events will update the `Event_Control.pending_events` field and it will then check such updated value against `count` to determine whether the waiting condition of the task is satisfied or not. The set of events upon which the task is waiting is maintained separated to the set of events received by the task because the set of events received by the task must be updated (i.e., cleared) only once, exactly when the task resumes its execution: if during the suspension of the task the same events are sent several times they should still count as events sent only once.

`return_argument`: it stores the pointer to the variable that will contain the set of satisfied events. The variable is supplied by the user.

`return_argument_second`: unused.

`option`: it stores the policy used by the suspended task. It can be either `RTEMS_EVENT_ANY` or `RTEMS_EVENT_ALL`. This information is used by a task sending an event in order to understand whether the waiting condition of the suspended task is satisfied or not.

`return_code`: it is the *rtems_status_code* that will be returned to the user when the task will resume its execution. This field is set by tasks sending events or by the timeout routine. This field exists because a suspended task does not know why it resumes its execution: it can be both because its waiting condition is satisfied or because the timeout fired.

`queue`: unused.

`flags`: unused.

8.2 Usage

8.2.1 Overview of the Event Protocol

With respect to the RTEMS API, an event is a flag upon which a task can synchronize its execution. Being a flag, an event does not carry any additional information: its meaning is completely determined by how the user decides to use the events. Normally this means that the set of events upon which a task is waiting translates in a synchronization barrier for the waiting task.

A task can suspend itself waiting for one or more events to be satisfied: an event is satisfied when the corresponding flag is set. Moreover, a task has several ways to wait for an event: it can decide to wait for at least one or for all the events to be satisfied (in case the task is waiting for more than one event) and it can decide to wait for at most a specific amount of time before resuming its execution even if the even(s) is not satisfied (the task arms a timer that forces the task to wake up).

A task can send one or more events to any other task: it is not required that the task that receives the event must be suspended. Indeed, sending an event translates in setting a specific flag inside the targeted task: whether the targeted task uses or not the event is application specific. Moreover, an event can be sent several times, but the receiving task has no way to determine the number of times its events have been set: the flag is a binary flag and is reset only when the task consumes the event (i.e., when the task waits for the specified event and then resumes its execution).