RTEMS SMP Status Report[1]

 Authors Alexander Krutwig (AK), Sebastian Huber (SH). Date 2016-12-16. Revision 3.

Contents

1 Overview **2**
- 1.1 Features . . . 2
- 1.2 Platforms . . . 3

2 Application Issues **3**
- 2.1 Task variables . . . 3
- 2.2 Disabling of thread pre-emption . . . 3
- 2.3 Highest priority thread never walks alone . . . 3
- 2.4 Disabling of interrupts . . . 3
- 2.5 Interrupt service routines execute in parallel with threads . . . 4
- 2.6 Timers do not stop immediately . . . 5
- 2.7 False sharing of cache lines due to objects table . . . 5

3 Self-Contained Objects **5**

4 Lock-Free Timestamps **6**

5 Scalable Timer and Timeout Support **6**

6 Clustered Scheduling **7**

7 Locking Protocols **8**
- 7.1 Inter-cluster priority queues . . . 8
- 7.2 Dependency tracking . . . 9
- 7.3 $O(m)$ Independence-Preserving Protocol (OMIP) . . . 9
- 7.4 Multiprocessor Resource-Sharing Protocol (MrsP) . . . 11

8 Parallel Languages **11**
- 8.1 Open Multi-Processing (OpenMP) . . . 12
- 8.2 Embedded Multicore Buidling Blocks (EMB^2) . . . 12
- 8.3 Google Go . . . 12

9 Linker-Set Based Initialization **13**

10 Profiling **13**

11 Low-Level Synchronization **14**

12 Thread Dispatching **14**

1 Overview

1.1 Features

The Real-Time Operating System for Multiprocessor Systems (RTEMS) is a multi-threaded, single address-space, real-time operating system with no kernel-space/user-space separation[2]. It is capable to operate in an Symmetric Multiprocessing (SMP) configuration providing a state of the art feature set.

- APIs
  - Classic
  - POSIX (pthreads)
  - C11 threads
  - C++11 threads
  - Newlib and GCC internal
  - Futex
- C11/C++11 Thread-Local Storage (TLS)[3]
- Lock-free timestamps (FreeBSD timecounters)
- Scalable timer and timeout support
- Operating system uses fine-grained locking
- Linker-set based initialization (similar to global C++ constructors)
- Clustered scheduling
  - Flexible link-time configuration
  - Fixed-priority scheduler
  - Job-level fixed-priority scheduler (EDF)
  - Proof-of-concept strong APA scheduler
- Locking protocols
  - Priority Inheritance
  - $O(m)$ Independence-Preserving Protocol (OMIP)
  - Priority Ceiling
  - Multiprocessor Resource-Sharing Protocol (MrsP)
- Programming languages
  - Ada
  - C/C++
  - Erlang
  - Fortran
- Parallel languages
  - Embedded Multicore Buidling Blocks (EMB^2)
  - Google Go[4]
  - OpenMP 4.5

[2] RTEMS uses a modified GPL license with an exception for static linking. It exposes no license requirements on application code.

[3] Thread-local storage requires some support by the tool chain and the RTEMS architecture support, e.g. context-switch code. It is supported at least on ARM, PowerPC, SPARC and m68k. Check the *RTEMS CPU Architecture Supplement* if it is supported.

[4] See https://devel.rtems.org/ticket/2832.

1.2 Platforms

Currently, the following architectures are supported

- ARM[5],
- PowerPC[6], and
- SPARC[7].

It is easily portable to other architectures.

2 Application Issues

Most operating system services provided by the uni-processor RTEMS are available in SMP configurations as well. However, applications designed for an uni-processor environment may need some changes to correctly run in an SMP configuration as listed below.

2.1 Task variables

Task variables are ordinary global variables with a dedicated value for each thread. During a context switch from the executing thread to the heir thread, the value of each task variable is saved to the thread control block of the executing thread and restored from the thread control block of the heir thread. This is inherently broken if more than one executing thread exists. Alternatives to task variables are POSIX keys and Thread-Local Storage (TLS)[8]. TLS is available in C11 and C++11. All use cases of task variables in the RTEMS code base were replaced with alternatives[9]. The task variable API is deprecated and is no longer available in RTEMS 4.12.

2.2 Disabling of thread pre-emption

A thread which disables pre-emption prevents that a higher priority thread gets hold of its processor involuntarily[10]. In uni-processor configurations, this can be used to ensure mutual exclusion at thread level. In SMP configurations, however, more than one executing thread may exist. Thus, it is impossible to ensure mutual exclusion using this mechanism. In order to prevent that applications using pre-emption for this purpose, would show inappropriate behaviour, this feature is disabled in SMP configurations and its use would case run-time errors.

2.3 Highest priority thread never walks alone

The highest priority thread runs in parallel with other threads or interrupt service routines in systems with more than one processor. It cannot assume to have exclusive access to resources like on an uni-processor machine.

2.4 Disabling of interrupts

A low overhead means that ensures mutual exclusion in uni-processor configurations is the disabling of interrupts around a critical section. This is commonly used in device driver code. In SMP configurations, however, disabling the interrupts on one processor has no effect on other processors. So, this is insufficient to ensure system-wide mutual exclusion. The macros

- `rtems_interrupt_disable()`,
- `rtems_interrupt_enable()`, and
- `rtems_interrupt_flush()`

[5] Altera Cyclone V, Xilinx Zynq, Raspberry Pi 2
[6] Freescale/NXP/Qualcom/Whatever QorIQ, e.g. P1020, T2080, T4240
[7] Cobham Gaisler GR712RC, GR740, ESA NGMP
[8] On the SPARC architecture access to TLS data is very efficient since this is done via the register `%g7` [8].
[9] Additionally to POSIX keys or TLS, the operating system can add things to the thread control block
[10] Actually, a thread with disabled pre-emption is still pre-emptible by a pseudo interrupt thread. So, this feature is basically broken even in uni-processor configurations. See https://devel.rtems.org/ticket/2365.

are disabled in SMP configurations and its use will cause compile-time warnings and linker-time errors. In the unlikely case that interrupts must be disabled on the current processor, the

- `rtems_interrupt_local_disable()`, and
- `rtems_interrupt_local_enable()`

macros are now available in all configurations.

Since disabling of interrupts is insufficient to ensure system-wide mutual exclusion on SMP a new low-level synchronization primitive was added – interrupt locks. The interrupt locks are a simple API layer on top of the SMP locks used for low-level synchronization in the operating system core. Currently, they are implemented as a ticket lock. In uni-processor configurations, they degenerate to simple interrupt disable/enable sequences by means of the C pre-processor. It is disallowed to acquire a single interrupt lock in a nested way. This will result in an infinite loop with interrupts disabled. While converting legacy code to interrupt locks, care must be taken to avoid this situation to happen.

```
#include <rtems.h>

void legacy_code_with_interrupt_disable_enable( void )
{
  rtems_interrupt_level level;

  rtems_interrupt_disable( level );
  /* Critical section */
  rtems_interrupt_enable( level );
}

RTEMS_INTERRUPT_LOCK_DEFINE( static, lock, "Name" )

void smp_ready_code_with_interrupt_lock( void )
{
  rtems_interrupt_lock_context lock_context;

  rtems_interrupt_lock_acquire( &lock, &lock_context );
  /* Critical section */
  rtems_interrupt_lock_release( &lock, &lock_context );
}
```

An alternative to the RTEMS-specific interrupt locks are POSIX spinlocks. The `pthread_spinlock_t` is defined as a self-contained object, e.g. the user must provide the storage for this synchronization object.

```
#include <assert.h>
#include <pthread.h>

pthread_spinlock_t lock;

void smp_ready_code_with_posix_spinlock( void )
{
  int error;

  error = pthread_spin_lock( &lock );
  assert( error == 0 );
  /* Critical section */
  error = pthread_spin_unlock( &lock );
  assert( error == 0 );
}
```

In contrast to POSIX spinlock implementation on Linux or FreeBSD, it is not allowed to call blocking operating system services inside the critical section. A recursive lock attempt is a severe usage error resulting in an infinite loop with interrupts disabled. Nesting of different locks is allowed. The user must ensure that no deadlock can occur. As a non-portable feature the locks are zero-initialized, e.g. statically initialized global locks reside in the `.bss` section and there is no need to call `pthread_spin_init()`.

2.5 Interrupt service routines execute in parallel with threads

On a machine with more than one processor, Interrupt Service Routines (ISRs) [11] and threads can execute in parallel. Interrupt service routines must take this into account and use proper locking mechanisms to protect critical sections from interference by threads (interrupt locks or POSIX spinlocks). This likely requires code modifications in legacy device drivers.

[11] For example, this includes timer service routines installed via `rtems_timer_fire_after()`.

2.6 Timers do not stop immediately

Timer service routines run in the context of the clock interrupt. On uni-processor configurations, it is sufficient to disable interrupts and remove a timer from the set of active timers to stop it. In SMP configurations, however, the timer service routine may already run and wait on an SMP lock owned by the thread which is about to stop the timer. This opens the door to subtle synchronization issues. During destruction of objects, special care must be taken to ensure that timer service routines cannot access (partly or fully) destroyed objects.

2.7 False sharing of cache lines due to objects table

The Classic API and most POSIX API objects are indirectly accessed via an object identifier. The user-level functions validate the object identifier and map it to the actual object structure which resides in a global objects table for each object class. So, unrelated objects are packed together in a table. This may result in false sharing of cache lines[12]. High-performance SMP applications need full control of the object storage [7]. Therefore, self-contained synchronization objects are now available for RTEMS.

3 Self-Contained Objects

The Classic API has some weaknesses:

- Dynamic memory (the workspace) is used to allocate object pools. This requires a complex configuration with heavy use of the C pre-processor.
- Objects are created via function calls which return an object identifier. The object operations use this identifier and map it internally to an object representation.
- The objects reside in a table, e.g. they are suspect to false sharing of cache lines.
- The object operations use a rich set of options and attributes. For each object operation these parameters must be evaluated and validated at run-time to figure out what to do exactly for this operation.

For applications that use fine grained locking the overhead to map the identifier to the object representation and the parameter evaluation are a significant overhead that may degrade the performance dramatically. An example is the new FreeBSD network stack (libbsd) which uses hundreds of locks in a basic setup. Here the performance can be easily measured in terms of throughput and processor utilization. The port of the FreeBSD network stack now uses its own priority inheritance mutex implementation which is not based on the Classic API. The blocking part however uses the standard thread queues. The overall implementation is quite simple since the difficult part (e.g. the blocking operations and locking protocol support) is provided by the thread queues.

New self-contained objects are available in RTEMS 4.12 via the Newlib supplied `<threads.h>`, `<pthread.h>` and `<sys/lock.h>` header files. The following synchronization objects are provided

- POSIX spinlocks,
- mutexes,
- recursive mutexes,
- condition variables,
- counting semaphores, and
- Futex synchronization[13].

They are used for the following parts

[12] The effect of false sharing of cache lines can be observed with the TMFINE 1 test program on a suitable platform, e.g. T4240.

[13] The Futex synchronization was originally created for the Linux operating system and is a building block for high performance synchronization objects [9]. However, only random fairness is provided, which is not enough for predictable real-time systems.

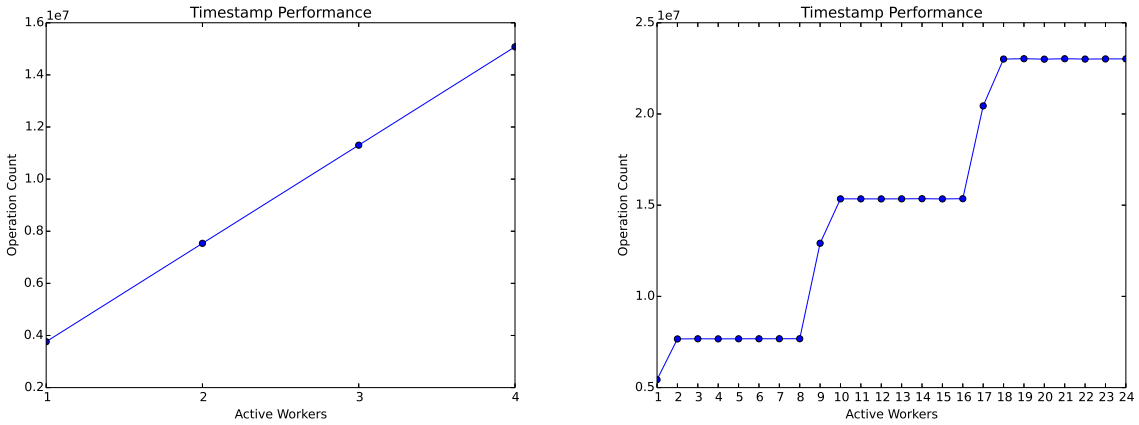


Figure 1: Timestamp performance measured on the four processor GR740 (left-side) and 24 processor QorIQ T4240 (right-side) with the SPTIMECOUNTER 2 test program. The QorIQ T4240 contains three L2 caches. Each L2 cache is shared by eight processors (cluster). It seems that a per-cluster bus limits the timestamp performance on this chip.

- Newlib internal locks[14],
- GCC run-time libraries,
- C11 threads support,
- C++11 threads support, and
- OpenMP support of GCC for RTEMS.

This allows much better performance on SMP. The application configuration is significantly simplified, since it is no longer necessary to account for lock objects used by Newlib and GCC. The Newlib defined self-contained objects can be a statically initialized and reside in the `.bss` section. Destruction is a no-operation.

4 Lock-Free Timestamps

A high performance timestamp implementation is vital for the overall system performance. During each thread dispatch, some timing information is updated using the current uptime timestamp. It is essential for low overhead run-time tracing where each processor has its own trace buffer and timestamps must be used to correlate events that occurred on different processors.

The nanoseconds extension used in previous RTEMS version for timestamps with higher resolution than the system tick is broken by design on SMP. In addition, the usage of an SMP lock to get the timestamps was a performance bottleneck. A different implementation had to be found. After an evaluation of existing implementations, the FreeBSD timecounters were selected due to the sound design and liberal license [11]. They were ported to RTEMS and show excellent results.

5 Scalable Timer and Timeout Support

Timer and timeout services are provided by the watchdog handler. The use cases for the watchdog handler fall roughly into two categories:

- Timeouts – used to detect if some operations need more time than expected. Since the unexpected happens hopefully rarely, timeout timers are usually removed before they expire. The critical operations are insert and removal. They are important for the performance of a network stack.
- Timers – used to carry out some work in the future. They usually expire and need a high resolution. An example user is a time driven scheduler, e.g. rate-monotonic or EDF.

[14] See https://devel.rtems.org/ticket/1247.

Previously, the watchdog handler was implemented by means of delta chains and global variables. The new watchdog handler uses a red-black tree with the expiration time as the key. This leads to $O(log(n))$ worst-case insert and removal operations. Each processor provides its own watchdog service point so that the watchdog handler scales well with the processor count of the system. For each operation it is sufficient to acquire and release a dedicated SMP lock only once. The drawback is that a 64-bit integer type must be used for the intervals to avoid a potential overflow of the key values[15].

An alternative to the red-black tree based implementation would be the use of a timer wheel based algorithm [14] which is used in Linux and FreeBSD [13] for example. A timer wheel based algorithm offers $O(1)$ worst-case time complexity for insert and removal operations. The drawback is that the run-time of the watchdog tick procedure is unpredictable due to the use of a hash table or cascading.

The red-black tree approach was selected, since it offers a more predictable run-time behaviour. However, this sacrifices the constant insert and removal operations offered by the timer wheel algorithms. See also [10]. The implementation can re-use the red-black tree support already used in RTEMS, e.g. for the thread priority queues. Less code is a good thing for size, testing and verification.

6 Clustered Scheduling

A scheduler manages a set of ready threads and assigns processors to some of them according to a specific policy, e.g. thread priority. We have clustered scheduling in case the set of processors of a system is partitioned into non-empty pairwise-disjoint subsets of processors. These subsets are called clusters. Clusters with a cardinality of one are partitions. Each cluster is owned by exactly one scheduler instance. In case the cluster size equals the processor count, it is called global scheduling.

Clustered scheduling helps to control the worst-case latencies in the system [3]. The goal is to reduce the amount of shared state in the system and consequently to prevent lock contention. Modern multiprocessor systems tend to have several layers of data and instruction caches. With clustered scheduling, it is possible to honour the cache topology of a system and to avoid expensive cache synchronization traffic. It is easy to implement. Providing adequate synchronization primitives for inter-cluster synchronization is a challenge, though. In RTEMS there are currently four means available

- events,
- message queues,
- mutexes using the $O(m)$ Independence-Preserving Protocol (OMIP), and
- mutexes using the Multiprocessor Resource-Sharing Protocol (MrsP).

The low-level SMP locks use FIFO ordering. So, the worst-case run-time of operations increases with each processor involved. The clustered scheduling approach enables separation of functions with low latency requirements and functions that profit from fairness and high throughput, provided that the scheduler instances are fully decoupled and adequate intra-cluster synchronization primitives are used.

The scheduler infrastructure is based on an object-oriented design. The scheduler operations for a thread are defined as virtual functions, for example to

- block a thread,
- unblock a thread,
- yield the processor,
- change the priority of a thread, and
- ask for help for a thread.

Each processor is assigned to at most one scheduler instance at link-time. It is possible to re-assign processors during run-time. A scheduler instance consists of a scheduler algorithm providing implementations for the scheduler operations and a set of data structures

- the *scheduler control*, a read-only structure defining the schedule name, operations and context,

[15] With a system tick interval of $1ns$ the system could run more than 500 years before an overflow happens. The Earliest Deadline First (EDF) scheduler profits from the 64-bit interval representation.

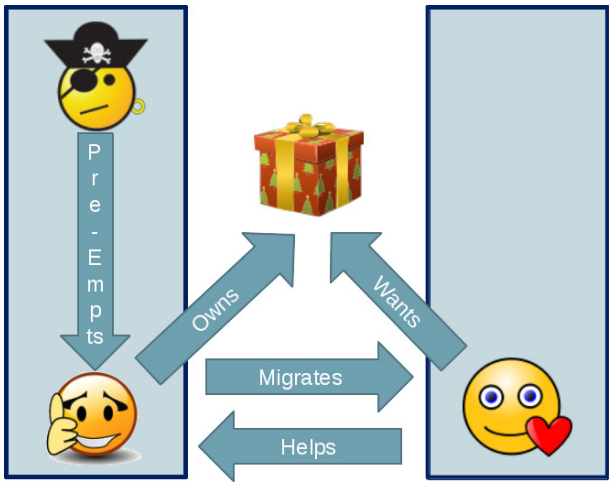


Figure 2: This image illustrates the helping mechanism used by the locking protocols with support for clustered scheduling. We have two scheduler instances. The shared resource protected by a mutex is depicted with a present. The scheduler instance on the left hand side has two threads – the mutex owner (thumbs-up) and a high priority thread (pirate) that pre-empts the mutex owner. The right hand side scheduler instance has only one thread – a thread that wants to get to the present (in love). Threads in other scheduler instances may help the resource owner and give it a temporary right to execute in their own scheduler instance with their own priority so that it can complete the critical section. Locking protocols that use this mechanism are OMIP and MrsP for example.

- the *scheduler context*, a read-write structure encapsulating all the data necessary to manage the set of threads assigned to this scheduler instance, and
- the *scheduler node*, a read-write structure attached to each thread which is used to register this thread in the corresponding scheduler context.

All currently available SMP-aware schedulers use a framework which is customized via inline functions. This eases the implementation of scheduler variants. Up to now, only priority-based schedulers are implemented. It is moderately easy to add new scheduler algorithms (e.g. proportional fair, ULE[16]).

7 Locking Protocols

The implementation of a basic clustered scheduler is quite easy and straight forward. What makes things a bit more difficult is the support for adequate locking protocols. There are two main issues:

1. Thread queues that contain blocked threads of different scheduler instances need an appropriate queuing discipline.
2. The locking protocols need some sort of dependency tracking to allow threads to temporarily migrate to foreign scheduler instances in case of pre-emption.

7.1 Inter-cluster priority queues

It makes no sense to compare the priority values of two different scheduler instances. Thus, it is impossible to simply use one plain priority queue for threads of different clusters. Two levels of queues can be used as one way to solve the problem. The top-level queue provides First-In First-Out (FIFO) ordering and contains priority queues. Each priority queue is associated with a scheduler instance and contains only threads of this scheduler instance. Threads are enqueued in the priority queue corresponding to their scheduler instance. To dequeue a thread, the highest priority thread of the first priority queue is selected. Once this is done, the first priority queue is appended to the FIFO queue. This guarantees fairness with respect to the scheduler instances.

[16] The FreeBSD ULE scheduler [12].

Such a two level queue needs a considerable amount of memory if fast enqueue and dequeue operations are desired. Previously, each synchronization object in RTEMS contained a set of queues providing different queuing disciplines. So, adding this new queue implementation would have increased the object size significantly. It was beneficial to use an approach used in the FreeBSD kernel. Here each thread has a queue attached which resides in a dedicated memory space independent of other memory used for the thread. In case a thread needs to block, there are two options

- the object already has a queue, then the thread enqueues itself to this already present queue and the queue of the thread is added to a list of free queues for this object, or
- otherwise, the queue of the thread is given to the object and the thread enqueues itself to this queue.

In case the thread is dequeued, there are two options

- the thread is the last thread in the queue, then it removes this queue from the object and reclaims it for its own purpose, or
- otherwise, the thread removes one queue from the free list of the object and reclaims it for its own purpose.

Since there are usually more objects than threads, this actually reduces the memory demands. In addition the objects only contain a pointer to the queue structure. This helps to hide implementation details and makes it possible to add self-contained one purpose objects to Newlib and GCC (C++ and OpenMP run-time support). Inter-cluster priority queues are implemented in RTEMS 4.12.

7.2 Dependency tracking

A thread that owns a shared resource needs to know if it can temporarily migrate to a foreign scheduler instance in case of pre-emption. The alternative scheduler instances and the corresponding thread priorities are determined by other threads that directly or indirectly depend on one of the resources owned by the thread. The waiting threads may be blocked with respect to the scheduler (e.g. in case of OMIP) or perform a busy wait loop (e.g. in case of MrsP).

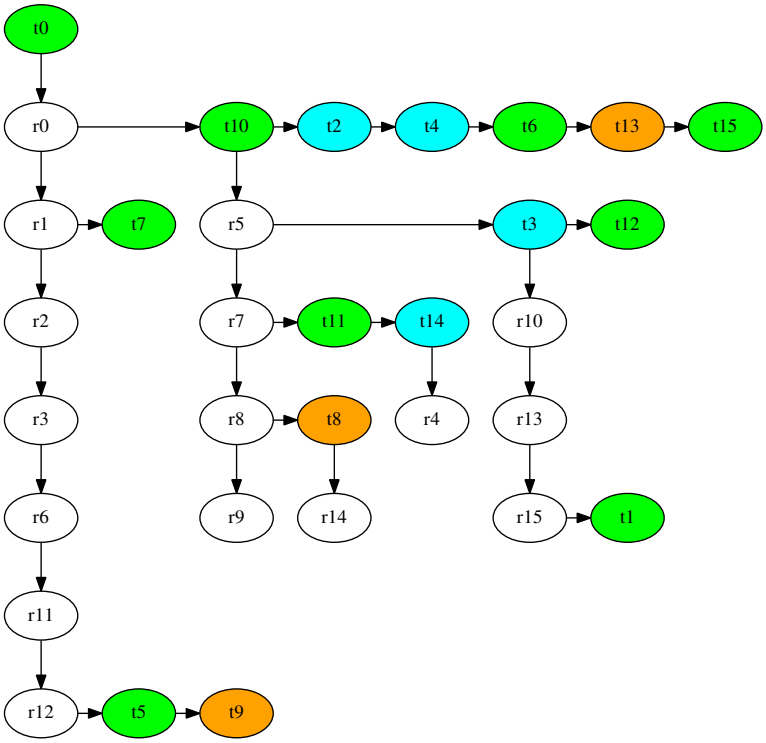
In order to simplify the low-level synchronization each thread is equipped with a scheduler node for each scheduler instance in the system. Normally, the thread can only use the scheduler node of its home scheduler instance. In case of resource conflicts other scheduler nodes of the thread are activated and give the thread the ability to use other scheduler instances. The scheduler nodes of a thread are updated in case of

- another thread wants to get a resource owned by the thread,
- a waiting thread times out and is no longer interested on a resource, or
- ownership is transfered from the thread to a new owner.

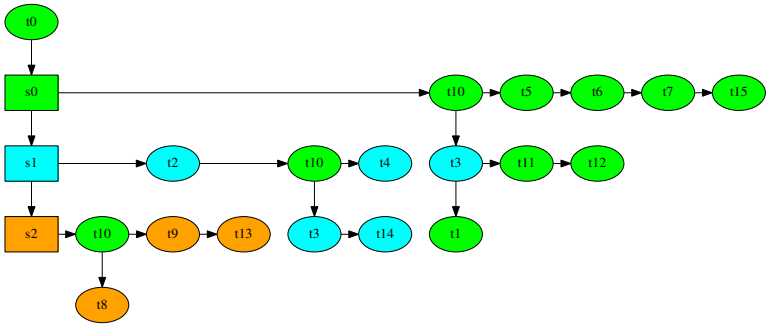
The run-time of these operations depend on the complexity of the dependency tree. This is application-specific and out of control of the operating system. On RTEMS, the dependency tracking is carried out by the thread queues using recursive red-black trees.

7.3 $O(m)$ Independence-Preserving Protocol (OMIP)

The $O(m)$ Independence-Preserving Protocol (OMIP) is a generalization of the priority inheritance protocol to clustered scheduling which avoids the non-preemptive sections present with priority boosting [4]. The m denotes the number of processors in the system. Similar to the uni-processor priority inheritance protocol, the OMIP mutexes don't need any external configuration data, e.g. a ceiling priority. This makes them a good choice for general purpose libraries that need internal locking. The complex part of the implementation is contained in the thread queues and shared with the MrsP support.



(a) Example resource dependency tree with sixteen threads $t0$ up to $t15$ and sixteen resources $r0$ up to $r15$. The root of this tree is $t0$. The thread $t0$ owns the resources $r0$, $r1$, $r2$, $r3$, $r6$, $r11$ and $r12$ and is in the ready state. The threads $t1$ up to $t15$ wait directly or indirectly via resources owned by $t0$ and are in a blocked state. The colour of the thread nodes indicate the scheduler instance.



(b) Example of a table of priority nodes with sixteen threads $t0$ up to $t15$ and three scheduler instances $s0$ up to $s2$ corresponding to figure 3a. The overall resource owner is $t0$. The colour of the nodes indicate the scheduler instance. Several threads of different scheduler instances depend on thread $t10$. So, the thread $t10$ contributes for example the highest priority node of scheduler instance $s2$ to thread $t0$ even though it uses scheduler instance $s0$.

Figure 3: Resource dependency tracking.

7.4 Multiprocessor Resource-Sharing Protocol (MrsP)

The Multiprocessor Resource-Sharing Protocol (MrsP) is a generalization of the priority ceiling protocol to clustered scheduling [5]. One of the design goals of MrsP is to enable an effective schedulability analysis using the sporadic task model. The MrsP implementation in RTEMS 4.12 was improved to overcome two problems present in the first generation of the protocol implementation [6]. Firstly, the run-time of some scheduler operations depended on the linear size of the resource dependency tree. Secondly, the scheduler operations of threads which didn't use shared resources must deal with the scheduler helping protocol in case an owner of a shared resource is somehow involved. These shortcomings have been fixed.

```
void example( void )
{
  rtems_task_priority new_prio;
  rtems_task_priority previous_prio;
  rtems_id scheduler_id;
  rtems_id mrsp_id;

  sc = rtems_semaphore_create(
    rtems_build_name( 'M', 'R', 'S', 'P' ),
    1,
    RTEMS_BINARY_SEMAPHORE
      | RTEMS_MULTIPROCESSOR_RESOURCE_SHARING,
    1,
    &mrsp_id
  );
  assert( sc == RTEMS_SUCCESSFUL );

  sc = rtems_scheduler_ident(
    rtems_build_name( 'W', 'O', 'R', 'K' ),
    &scheduler_id
  );
  assert( sc == RTEMS_SUCCESSFUL );

  new_prio = 123;
  sc = rtems_semaphore_set_priority(
    mrsp_id,
    scheduler_id,
    new_prio,
    &previous_prio
  );
  assert( sc == RTEMS_SUCCESSFUL );

  sc = rtems_semaphore_obtain(
    mrsp_id,
    RTEMS_WAIT,
    RTEMS_NO_TIMEOUT
  );
  assert( sc == RTEMS_SUCCESSFUL );

  /*
   * Some critical stuff.  This code block may execute on another
   * partition in case this thread gets pre-empted and a thread on
   * the other partition wants to obtain this semaphore during
   * that period.
   */

  sc = rtems_semaphore_release( mrsp_id );
  assert( sc == RTEMS_SUCCESSFUL );
}
```

8 Parallel Languages

The objective of parallel languages is to enable and simplify application development on parallel computers. They may be an own programing language (e.g. Google Go), a programing language extension (e.g. OpenMP, Cilk Plus) or available as a library (e.g. EMB^2). In some cases the user is responsible for the task scheduling (e.g. POSIX threads, MPI). Others take care of the task scheduling (e.g. OpenMP, Cilk Plus, EMB^2), for example using some variant of a work stealing scheduler [1]. In general they focus on high-performance computing and numerical simulations. What do they need from an operating system or platform?

- Atomic operations – this is not a problem for RTEMS, since this is provided by the hardware/compiler
- Worker threads (possibly pinned to a certain processor; threads, not processes) – no problem for RTEMS
- Memory management
  - Depends on the parallel language, maybe a non-issue, e.g. in case of OpenMP, EMB^2
  - Fork/join of worker tasks may lead to non-linear stacks (so called cactus stack problem)
  - General parallel languages may need difficult to understand dynamic memory
  - May need some kind of virtual memory, e.g. mmap()
  - Virtual memory is not in general supported by RTEMS
  - Virtual memory management is a problem for deterministic real-time systems, e.g. what happens in case of insufficient physical memory?
  - Open research problems

8.1 Open Multi-Processing (OpenMP)

OpenMP support for RTEMS is available via the GCC. In general the OpenMP support consists of two parts. One part is the compiler support which turns OpenMP pragmas into machine code. This part is completely independent of RTEMS. The other part is the libgomp run-time library. The initial OpenMP support used the POSIX configuration of libgomp and is available in GCC 4.9 and 5.1. However, the performance was quite poor. To overcome the performance issues we added a RTEMS configuration for libgomp which is available in GCC 6.1. It uses self-contained objects defined in the Newlib provided `<sys/lock.h>` header file. The barriers use a nearly one-to-one copy of the Futex based implementation of the Linux configuration of libgomp[17].

8.2 Embedded Multicore Buidling Blocks (EMB^2)

The Embedded Multicore Buidling Blocks (EMB^2) are a set of C/C++ libraries providing

- task management,
- dataflow,
- algorithms, and
- containers.

It was initially designed for embedded systems and is available under a 2-clause BSD license. It is developed and used by Siemens. As a component it ships the open source reference implementation of the Multicore Task Management API (MTAPI). It is fully supported by RTEMS. Integration of the RTEMS support into the main repository is currently in progress[18].

8.3 Google Go

Google Go is a programing language designed for parallel applications. RTEMS supports an early version of Google Go run-time library. For the latest version of Google Go RTEMS would need support for the `#include <ucontext.h>` provided services[19].

[17] See https://devel.rtems.org/ticket/2274.
[18] For example see pull request https://github.com/siemens/embb/pull/31.
[19] See https://devel.rtems.org/ticket/2832.

9 Linker-Set Based Initialization

Linker sets are used not only in RTEMS, but also for example in Linux, in FreeBSD, for the GNU C constructor extension and for global C++ constructors. They provide a space efficient and flexible means to initialize modules. A linker set consists of

- dedicated input sections for the linker (e.g. `.ctors` and `.ctors.*` in the case of global constructors),
- a begin marker (e.g. provided by `crtbegin.o`, and
- an end marker (e.g. provided by `ctrend.o`).

A module may place a specific data item into the dedicated input section. The linker will collect all such data items in this section and create a begin and end marker. The initialization code can then use the begin and end markers to find all the collected data items (e.g. pointers to initialization functions).

In the linker command file of the GNU linker we need the following output section descriptions.

```
/* To be placed in a read-only memory region */
.rtemsroset : {
    KEEP (*(SORT(.rtemsroset.*)))
}

/* To be placed in a read-write memory region */
.rtemsrwset : {
    KEEP (*(SORT(.rtemsrwset.*)))
}
```

The `KEEP()` ensures that a garbage collection by the linker will not discard the content of this section. This would normally be the case since the linker set items are not referenced directly. The `SORT()` directive sorts the input sections lexicographically. Please note the lexicographical order of the `.begin`, `.content` and `.end` section name parts in the RTEMS linker sets macros which ensures that the position of the begin and end markers are correct.

So, what is the benefit of using linker sets to initialize modules? It can be used to initialize and include only those RTEMS managers and other components which are used by the application. For example, in case an application uses message queues, it must call `rtems_message_queue_create()`. In the module implementing this function, we can place a linker set item and register the message queue handler constructor. In case the application does not use message queues, there will be no reference to the `rtems_message_queue_create()` function and the constructor is not registered, thus nothing of the message queue handler will be in the final executable.

For an example see test program SPLINKERSETS 1.

10 Profiling

To identify the bottlenecks in the system, support for profiling of low-level synchronization was added. The profiling support is a build time configuration option and is implemented with an acceptable overhead, even for production systems. A low-overhead counter for short time intervals must be provided by the hardware[20].

Profiling reports are generated in XML for most test programs of the RTEMS testsuite (more than 500 test programs). This gives a good sample set for statistics.

```
<ProfilingReport name="SMPMIGRATION_1">
  <PerCPUProfilingReport processorIndex="0">
    <MaxThreadDispatchDisabledTime unit="ns">36636</MaxThreadDispatchDisabledTime>
    <MeanThreadDispatchDisabledTime unit="ns">5065</MeanThreadDispatchDisabledTime>
    <TotalThreadDispatchDisabledTime unit="ns">3846635988
      </TotalThreadDispatchDisabledTime>
    <ThreadDispatchDisabledCount>759395</ThreadDispatchDisabledCount>
    <MaxInterruptDelay unit="ns">8772</MaxInterruptDelay>
    <MaxInterruptTime unit="ns">13668</MaxInterruptTime>
    <MeanInterruptTime unit="ns">6221</MeanInterruptTime>
    <TotalInterruptTime unit="ns">6757072</TotalInterruptTime>
    <InterruptCount>1086</InterruptCount>
  </PerCPUProfilingReport>
```

[20] On the GR712RC, there is a significant overhead if profiling is enabled, since this platform lacks support for a low-overhead hardware counter.


```
  <PerCPUProfilingReport processorIndex="1">
    <MaxThreadDispatchDisabledTime unit="ns">39408</MaxThreadDispatchDisabledTime>
    <MeanThreadDispatchDisabledTime unit="ns">5060</MeanThreadDispatchDisabledTime>
    <TotalThreadDispatchDisabledTime unit="ns">3842749508
      </TotalThreadDispatchDisabledTime>
    <ThreadDispatchDisabledCount>759391</ThreadDispatchDisabledCount>
    <MaxInterruptDelay unit="ns">8412</MaxInterruptDelay>
    <MaxInterruptTime unit="ns">15868</MaxInterruptTime>
    <MeanInterruptTime unit="ns">3525</MeanInterruptTime>
    <TotalInterruptTime unit="ns">3814476</TotalInterruptTime>
    <InterruptCount>1082</InterruptCount>
  </PerCPUProfilingReport>
  <!-- more reports omitted -->
  <SMPLockProfilingReport name="Scheduler">
    <MaxAcquireTime unit="ns">7092</MaxAcquireTime>
    <MaxSectionTime unit="ns">10984</MaxSectionTime>
    <MeanAcquireTime unit="ns">2320</MeanAcquireTime>
    <MeanSectionTime unit="ns">199</MeanSectionTime>
    <TotalAcquireTime unit="ns">3523939244</TotalAcquireTime>
    <TotalSectionTime unit="ns">302545596</TotalSectionTime>
    <UsageCount>1518758</UsageCount>
    <ContentionCount initialQueueLength="0">759399</ContentionCount>
    <ContentionCount initialQueueLength="1">759359</ContentionCount>
    <ContentionCount initialQueueLength="2">0</ContentionCount>
    <ContentionCount initialQueueLength="3">0</ContentionCount>
  </SMPLockProfilingReport>
</ProfilingReport>
```

11 Low-Level Synchronization

All low-level synchronization primitives are implemented using C11 or C++11 atomic operations, so no target specific hand-written assembler code is necessary. Four synchronization primitives are currently available

- ticket locks (mutual exclusion),
- Mellor-Crummey Scott (MCS) locks (mutual exclusion),
- barriers, implemented as a sense barrier, and
- sequence locks [2].

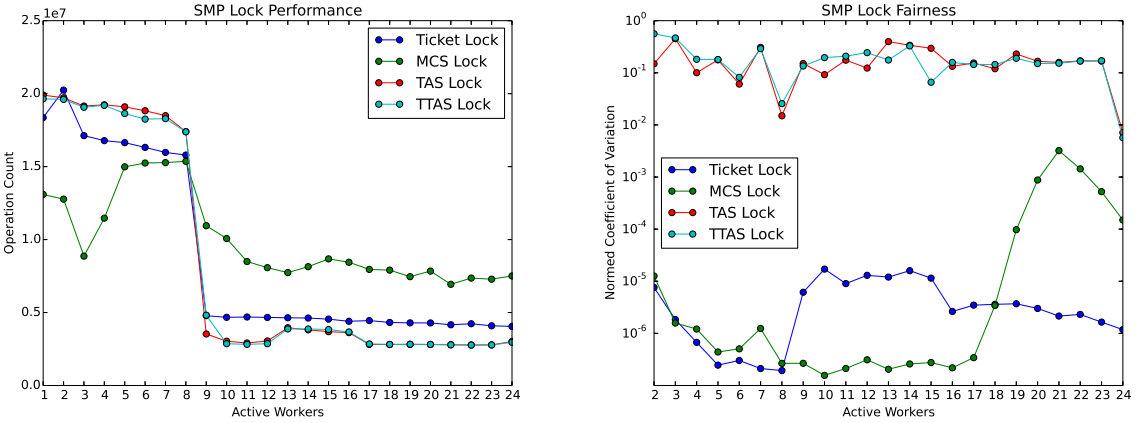
A vital requirement for low-level mutual exclusion is FIFO fairness since we are interested in a predictable system and not maximum throughput. With this requirement, the solution space is quite small. For reasons of simplicity, the ticket lock algorithm was chosen. However, the API is capable to support Mellor-Crummey Scott (MCS) locks, which may be interesting in the future for systems with a processor count in the range of 32 or more, e.g. NUMA, many-core systems.

The test program SMPLOCK 1 can be used to gather performance and fairness data for several scenarios.

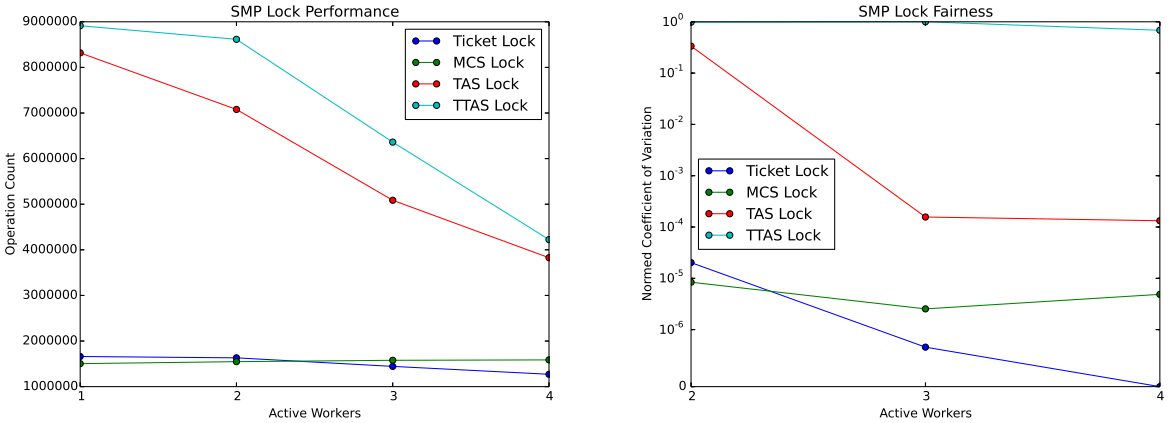
12 Thread Dispatching

In SMP systems, scheduling decisions on one processor must be propagated to other processors through inter-processor interrupts. A thread dispatch which must be carried out on another processor does not happen instantaneously. Thus, several thread dispatch requests might be in the air and it is possible that some of them may be out of date before the corresponding processor has time to deal with them. The thread dispatch mechanism uses three per-processor variables,

- the executing thread,
- the heir thread, and
- a boolean flag indicating if a thread dispatch is necessary or not.



(a) SMP lock performance and fairness measured on the QorIQ T4240. This chip contains three L2 caches. Each L2 cache is shared by eight processors.



(b) SMP lock performance and fairness measured on the GR740. The test-and-set locks deliver significantly more acquire/release operations per second, however, they are also extremely unfair.

Figure 4: SMP lock performance and fairness measured on two platforms.

Updates of the heir thread are done via a normal store operation. The thread dispatch necessary indicator of another processor is set as a side-effect of an inter-processor interrupt. So, this change notification works without the use of locks. The thread context is protected by a TTAS lock embedded in the context to ensure that it is used on at most one processor at a time. The thread post-switch actions use a per-processor lock. This implementation turned out to be quite efficient and no lock contention was observed in the testsuite. The heavy-weight thread dispatch sequence is only entered in case the thread dispatch indicator is set[21]. Thus, it is no longer relevant for the average-case performance. This enabled new techniques like a pre-emption intervention used by the SMP locking protocols.

The context-switch is performed with interrupts enabled. During the transition from the executing to the heir thread neither the stack of the executing nor the heir thread must be used during interrupt processing. For this purpose a temporary per-processor stack is set up which may be used by the interrupt prologue before the stack is switched to the interrupt stack.

[21] See _Thread_Do_dispatch().

Acronyms

APA Arbitrary Processor Affinity
API Application Programming Interface
ARM Advanced RISC Machine
C11 ISO/IEC 9899:2011
C++11 ISO/IEC 14882:2011
EDF Earliest Deadline First . . . 7
EMB² Embedded Multicore Buidling Blocks . . . 1
ESA European Space Agency
FIFO First-In First-Out . . . 8
Futex Fast User-Space Locking
GCC GNU Compiler Collection
GNU GNU's Not Unix
GPL GNU General Public License
ISR Interrupt Service Routine . . . 4
MCS Mellor-Crummey Scott . . . 14
MPI Message Passing Interface
MrsP Multiprocessor Resource-Sharing Protocol . . . 1
MTAPI Multicore Task Management API . . . 12
NGMP Next Generation Multiprocessor
NUMA Non-Uniform Memory Access
OMIP $O(m)$ Independence-Preserving Protocol . . . 1
OpenMP Open Multi-Processing . . . 1
RISC Reduced Instruction Set Computer
RTEMS Real-Time Operating System for Multiprocessor Systems . . . 2
POSIX Portable Operating System Interface
SMP Symmetric Multiprocessing . . . 2
SPARC Scalable Processor Architecture
TLS Thread-Local Storage . . . 2
TTAS Test and Test-and-set
XML Extensible Markup Language

References

[1] Blumofe, Robert D. and Charles E. Leiserson: *Scheduling multithreaded computations by work stealing.* Journal of the ACM, 46:720–748, 1999.

[2] Boehm, Hans J.: *Can Seqlocks Get Along With Programming Language Memory Models?* Technical report, HP Laboratories, June 2012. http://www.hpl.hp.com/techreports/2012/HPL-2012-68.pdf, HPL-2012-68.

[3] Brandenburg, Björn B.: *Scheduling and Locking in Multiprocessor Real-Time Operating Systems.* PhD thesis, 2011. http://www.cs.unc.edu/~bbb/diss/brandenburg-diss.pdf.

[4] Brandenburg, Björn B.: *A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications.* In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, pages 292–302, 2013. http://www.mpi-sws.org/~bbb/papers/pdf/ecrts13b.pdf.

[5] Burns, A. and A. J. Wellings: *A Schedulability Compatible Multiprocessor Resource Sharing Protocol - MrsP*. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, 2013. http://www-users.cs.york.ac.uk/~burns/MRSPpaper.pdf.

[6] Catellani, Sebastiano, Luca Bonato, Sebastian Huber, and Enrico Mezzetti: *Challenges in the Implementation of MrsP*. In *Reliable Software Technologies - Ada-Europe 2015*, pages 179–195, 2015.

[7] Drepper, Ulrich: *What Every Programmer Should Know About Memory*, 2007. http://www.akkadia.org/drepper/cpumemory.pdf.

[8] Drepper, Ulrich: *ELF Handling For Thread-Local Storage*, 2013. http://www.akkadia.org/drepper/tls.pdf.

[9] Franke, Hubertus, Rusty Russel, and Matthew Kirkwood: *Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux*. In *Proceedings of the Ottawa Linux Symposium 2002*, pages 479–495, 2002. https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf.

[10] Gleixner, Thomas and Douglas Niehaus: *Hrtimers and Beyond: Transforming the Linux Time Subsystems*. In *Proceedings of the Linux Symposium*, pages 333–346, 2006. https://www.kernel.org/doc/ols/2006/ols2006v1-pages-333-346.pdf.

[11] Kamp, Poul Henning: *Timecounters: Efficient and precise timekeeping in SMP kernels.*, 2002. http://www.freebsd.dk/pubs/timecounter.pdf.

[12] Robertson, Jeff: *ULE: A Modern Scheduler For FreeBSD*. In *Proceedings of BSDCon '03*, 2003. https://www.usenix.org/legacy/event/bsdcon03/tech/full_papers/roberson/roberson.pdf.

[13] Varghese, G. and A. Costello: *Redesigning the BSD callout and timer facilities*. Technical report, Washington University in St. Louis, November 1987. http://web.mit.edu/afs.new/sipb/user/daveg/ATHENA/Info/wucs-95-23.ps, WUCS-95-23.

[14] Varghese, G. and T. Lauck: *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, 1987. http://www.cs.columbia.edu/~nahum/w6998/papers/sosp87-timing-wheels.pdf.

Revision History

Revision	Date	Author(s)	Description
1	2015-06-30	SH	Initial release.
2	2015-10-29	AK, SH	Replace clustered/partitioned scheduling with clustered scheduling to be in line with the RTEMS C User's Guide. Mention that inter-cluster priority queues, priority boosting and self-contained objects are implemented. Mention the OpenMP support based on GCC. Mention ticket for implementation of linker sets. Editorial changes throughout.
3	2016-12-16	SH	Significant rework throughout to reflect one year of development.

