Scheduling and Thread Management with RTEMS

Gedare Bloom
Dept. of Computer Science
George Washington University
Washington, DC
gedare@gwu.edu

Joel Sherrill
OAR Corporation
Huntsville, AL
joel.sherrill@oarcorp.com

ABSTRACT

The goal of a real-time operating system (RTOS) is to support real-time and embedded system (RT/ES) application development, which differ from general-purpose applications because of the size, weight, and power (SWaP) and timing constraints imposed by embedded applications. Useful RTOS features include real-time thread scheduling, thread communication, synchronization, interrupt handling, memory management, file systems, device drivers, networking, and debugging support. The Real-Time Executive for Multiprocessor Systems (RTEMS) is a free and open-source RTOS that supports over a dozen processor architecture families and over 150 embedded system boards. RTEMS is designed to support embedded applications with stringent real-time requirements while being compatible with open standards such as POSIX. RTEMS includes optional services such as TCP/IP networking and file systems while still offering minimum executable sizes under 20 KB in useful configurations.

One of the primary functions of an RTOS is to select threads that can obtain access to resources such as shared memory and processor time. RTEMS uses multiple algorithms to manage both waiting threads and those ready to execute. The thread execution schedulers include the traditional RTOS round robin and deterministic priority schedulers, rate monotonic, earliest deadline first (EDF), constant bandwidth server (CBS), and simple SMP scheduling algorithms. The RTEMS scheduling framework allows the application developer to select the thread scheduling algorithm that best meets the application's space and time requirements. ÂăWe will present how this framework can be used by researchers to integrate their own scheduling algorithm into RTEMS and test it using a scheduling simulator before deploying it on target hardware.

Categories and Subject Descriptors

Computer Systems Organization [**Real-time systems**]: Real-time operating systems

Keywords

Scheduling, RTEMS

EWiLi '13 August 26–27, 2013, Toulouse, FRANCE

1. INTRODUCTION

An operating system is a collection of software services and abstraction layers that enable applications to be concerned with business logic rather than the specifics of device control and protocol stacks. With this abstraction of lower level concerns, the application can be portable across hardware platforms. Operating system standards such as IEEE POSIX define operating systems interfaces which enable applications to be portable from one operating system implementation to another.

A real-time operating system (RTOS) is an operating system designed to provide applications services which assist in meeting application requirements which including timing. These timing requirements are typically related to interaction with external devices, other computers, or humans. For example, a sensor may need to be sampled every fifty (50) milliseconds or the data is corrupt. Airline reservation systems have been classified as real-time systems because their requirements specify that a user request must be responded to within a specific amount of time. Even though this response time is specified in seconds rather than milliseconds this still meets the definition of real-time. These timing requirements impose deadlines on the behavior of the application.

Real-time applications may further be divided into soft real-time, hard real-time, and safety critical based upon the negative impact of missing a deadline. If missing a single deadline can be compensated for and has little impact, the deadline is considered soft. Missing multiple soft deadlines may or may not have severe consequences. In contrast, a hard deadline is one which must be met or negative consequences occur. Many hard deadlines may be further classified as safety critical in case missing the deadline may result in harm to persons or property. An example of a common safety critical real-time system is an anti-lock brake system.

Use of an operating system decouples the application from the hardware platform and provides a core set of resource management services. Operating systems provide services such as concurrency, memory management, file systems, and networking. Relying on operating systems allows applications to be developed quicker and at lower costs. This provides faster time to market and reduces the likelihood of reengineering required if the application must be rehosted on a different hardware platform.

An RTOS provides all of the advantages of a non-real-time operating system while adding determinism. Determinism ensures that the operating system is designed to perform actions with bounded response times for arbitrary inputs. For example, the time required to block or unblock a single thread is independent of the number of

threads created by the application.

Although there are many RTOSes available and there are great differences between them, there are a common set of features found in most of them. First, they are designed to handle real-time constraints correctly and provide services to the application to enable this. They provide a multi-threaded environment usually including preemptive, priority-based scheduling often with timeslicing capability. This is sufficient to implement applications based on the rate monotonic scheduling algorithm [7], and some RTOSes provide extra services for periodic threads and statistics gathering to further assist such applications.

Since RTOSes provide multi-threaded environments, they must also provide mechanisms for those threads to communicate and synchronize. Although the application programming interface (API) and specific details may differ, it is common for RTOSes to provide some combination of message queues, semaphores, mutexes, events, and condition variables to coordinate threads. Other thread communication and synchronization mechanisms include barriers, asynchronous signals, and read/write locks. RTOSes with mutexes typically offer algorithms to avoid priority inversion, such as the priority ceiling or priority inheritance protocols [9].

RTOSes may provide various types of memory management services. It is common to have the RTOS provide support for the C program heap used by the `malloc` family as well as for special purpose memory pools. These special purpose memory pools may be managed using variable or fixed allocation schemes. Some RTOSes provide memory management unit interfaces which can be used to denote that certain areas of memory are read-only, executable, or not to be cached.

Although modern personal computers tend to have x86-compatible processors and a common hardware architecture, target hardware for embedded real-time applications can vary enormously. There are dozens of microprocessors available for this market and thousands of boards. Many real-time applications also include custom designed boards. RTOSes need to provide a design framework to support such wide variety—the board support package (BSP) framework. A BSP will include hardware initialization, device drivers, bus adapters, and interrupt controller management. Given the wide variety of embedded hardware, it is important for individual device drivers to be shared across boards. Some RTOSes include many BSPs as part of their standard deliverable, while other RTOS vendors license individual BSPs.

RTOSes are often very modular so that unused features can easily be left out of a system. This has the technical benefit of allowing real-time applications to be fielded in lower end target hardware and the business benefit of allowing some RTOS vendors to sell those capabilities as extra-cost items. Some additional features that are often conditionally available include file systems, networking, and graphics.

2. RTEMS OVERVIEW

RTEMS [4] is a free, open-source real-time operating system designed to provide deterministic performance on a wide variety of target hardware with portability as a primary design goal. Currently RTEMS supports over fifteen (15) processor architectures and one-hundred sixty (160) BSPs, which includes well-known architectures such as x86, PowerPC, ARM, MIPS, and SPARC as well as those only known within the embedded systems community

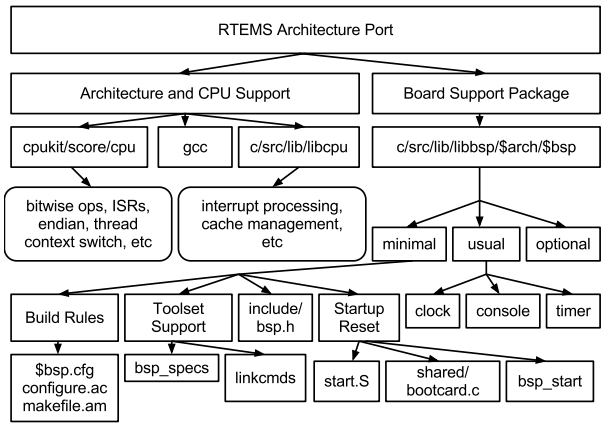

Figure 1: Dependency tree for an architectural port in RTEMS.

such as the LM32 and NIOS2. Figure 1 shows the dependencies of a typical architectural port for RTEMS.

Being a free, open-source project means that all RTEMS source code is available to the user for inspection, analysis, tailoring, and redistribution. RTEMS does not include software for the target hardware with any licenses that impose obligations on the licensing of the user's application. In the following, we summarize the primary components of RTEMS before we discuss the RTEMS scheduling framework in the next section. Extra details are available in the RTEMS manuals that are included with the RTEMS source code [5] and are provided online for convenience [3].

2.1 API Layers

From a technical perspective, RTEMS provides a single-address space, multi-threaded environment for applications. Historically applications have been statically linked, but now RTEMS supports dynamic loading of code with the RTEMS Dynamic Loader project. Still, the user's application, BSP, and RTEMS all reside in the same address space.

To limit the complexity of applications, RTEMS uses a layered software architecture with the Classic, POSIX, and SAPI interfaces; the ITRON interface has been dropped during the 4.11 release series. The Classic API was based upon the RTEID and ORKID RTOS API standards, which were proposed but never standardized by the VMEBus Industry Trade Association. RTEMS supports a large subset of POSIX 1003.1b through the POSIX API, which is POSIX-compliant for the features of POSIX that are necessary for single-user, single process applications. SAPI ("sappy") exposes features such as initialization, configuration, user extensions, and device driver management—these features are not standardized across operating systems. In addition to these three interfaces, RTEMS also supports the C library using newlib [1].

2.2 SuperCore: the RTEMS Kernel

Traditional RTOS kernel functionality is implemented in the SuperCore, located at `cpukit/score`. The SuperCore provides a common foundation for RTEMS' API layers with services such as task scheduling, interrupt handling, synchronization, inter-thread communication, and memory management implemented with *multilib* C source code. A multilib is a collection of libraries that are compiled with different CPU-specific flags. Thus, the SuperCore

can be compiled and used for a given architecture regardless of the CPU variant of the application. Another aspect of the SuperCore that propagates to other RTEMS components are the implementations of data structures, in particular the chains (doubly-linked lists) and red-black tree.

2.3 CPU and BSP Libraries

CPU-specific and board-specific support is implemented under the `c/src/lib/libcpu` and `c/src/lib/libbsp` directories, respectively. This support includes clocks and timers, console I/O, interrupt handlers or ISRs, cache and memory management unit handling, and bootstrapping into a C environment from which RTEMS can initialize. A minimal libcpu for a particular CPU will contain the cache management support; additional support may be included that is shared across all BSPs. A minimal BSP, called the "basic BSP", includes a clock driver, console driver, bootstrap code, and a linker script, which is usually called *linkcmds*.

2.4 Application Configuration

RTEMS is configured statically using `configure` (autoconf) options, and dynamically using configuration tables that are interpreted by the confdefs.h file in the SAPI.

`configure` is useful for coarse-grained configuration options that can provide conditional compilation, which can reduce the size of the compiled binary image by removing large subsystems that a user does not need. Static configuration can reduce runtime overheads because configuration options are not checked during the runtime. However, configure options are hard to maintain, pollute source code with `#ifdef` statements, and cannot be changed without recompiling RTEMS.

Configuration tables in confdefs.h are the main way in which users are able to configure certain aspects of RTEMS at runtime. These tables are flexibile because a user can change them and reset the application (board) in order to change a system's configuration without having to upload a new binary image. The main disadvantages of configuration tables are the lack of conditional compiilation and runtime overhead for configuration points, though the latter can often be accounted for during application initialization. Most of the overhead from configuration tables comes from stubs that are inserted for unused services.

2.5 Filesystems, Networking, and Languages

In addition to the core services, RTEMS implements a number of optional features that may be useful for real-time and embedded system applications. Some of these features include file I/O, networking, and language runtime support. For file I/O, RTEMS has a block device interface and buffer cache that is optimized for performance, along with implementations of commonly used filesystems such as NFS, and FAT, and filesystems that are especially useful for real-time systems such as the in-memory filesystem (IMFS) and RTEMS filesystem (RFS). RTEMS networking support uses a port of the FreeBSD networking stack that is designed to diverge minimally from the upstream so that updating can be painless. Some of the languages that RTEMS support for user applications are C, C++, Ada, Java, Go, and Lua; also, RTEMS can run Parrot [2], which provides support for many dynamic languages including Ruby, PHP, Python, Perl, and .NET.

3. RTEMS MODULAR SCHEDULER

Due to the variety of application scheduling needs, and especially with the rise of multicore platforms in embedded systems, an RTOS needs a task scheduler that permits users to choose the scheduling algorithm that best fits their application. Up to and including RTEMS version 4.10, the scheduler only had limited flexibility with support for round-robin and fixed priority scheduling. Starting with the 4.11 release series, RTEMS has a new, modular scheduler that uses a virtual function table of function pointers to hook scheduler-specific code throughout the existing thread management. A specific scheduling algorithm is implemented by instantiating a table of functions. Each application development team now can select a scheduling algorithm that best fits their application's needs, and RTEMS ships with a set of useful scheduler algorithms including a refactored version of the pre-existing scheduler and a handful of new scheduling algorithms. In the next section, we discuss the scheduler framework and supported scheduling algorithms.

3.1 Design

The scheduler is located in the SuperCore (cpukit/score) and defines the virtual base class from which all scheduling algorithms are derived. The primary definitions of the scheduler are in the files

- cpukit/score/include/rtems/score/scheduler.h
- cpukit/score/inline/rtems/score/scheduler.inl
- cpukit/score/src/scheduler.c

The scheduler is responsible for managing the set of threads in the ready state and determining which thread will execute next. When a thread exits or enters the ready state, scheduler hooks are invoked. The scheduler implementation determines if this thread state change necessitates changing the running thread by a thread dispatch.

Scheduler decisions are communicated with other parts of RTEMS via fields in the _Per_CPU_Information structure, a set of variables that are defined for each core. _Per_CPU_Information in a single core system is a single instance of the structure Per_CPU_Control, which is defined in the cpukit/score/include/rtems/score/percpu.h file. In a multicore system, _Per_CPU_Information is an array of Per_CPU_Control structures indexed by the CPU core number. The following fields in this structure are managed by a scheduler implementation:

- executing: pointer to the running thread (on this core)
- heir: pointer to the next runnable thread (on this core)
- dispatch_necessary: boolean variable indicate a thread dispatch is needed (on this core)

In a fully-preemptive application, the executing and heir point to the same thread because the most important thread is the one that is executing. If non-preemptible code runs, then the heir may point to a different thread than executing, for example if the running thread unblocks a higher priority task but remains non-preemptible.

3.2 Plugin Data Structure

The scheduler hooks are defined by a single data structure and one pointer in each thread control block (TCB, defined by struct Thread_Control). The data structure is named Scheduler_Control and is defined in cpukit/score/include/rtems/score/scheduler.h as

```
typedef struct {
  void                  *information;
  Scheduler_Operations    Operations;
} Scheduler_Control;
```

Scheduler_Control contains a pointer named information that points to the data used by the scheduler to manage the entire set of ready threads, e.g. the ready queue. The Operations field is the virtual function table provided by the scheduler, defined as:

```
typedef struct {
  void (*initialize)(void);
  void (*schedule)(void);
  void (*yield)(void);
  void (*block)(Thread_Control *);
  void (*unblock)(Thread_Control *);
  void *(*allocate)(Thread_Control *);
  void (*free)(Thread_Control *);
  void (*update)(Thread_Control *);
  void (*enqueue)(Thread_Control *);
  void (*enqueue_first)(Thread_Control *);
  void (*extract)(Thread_Control *);
  void (*tick)(void);
} Scheduler_Operations;
```

A scheduler implementation provides pointers to functions for the Scheduler_Operations. The Deterministic Priority Scheduler, for example, defines in score/include/rtems/score/schedulerpriority.h an initializer for the virtual function table as

```
#define SCHEDULER_PRIORITY_ENTRY_POINTS \
  { \
    _Scheduler_priority_Initialize,    \
    _Scheduler_priority_Schedule,      \
    _Scheduler_priority_Yield,         \
    _Scheduler_priority_Block,         \
    _Scheduler_priority_Unblock,       \
    _Scheduler_priority_Allocate,      \
    _Scheduler_priority_Free,          \
    _Scheduler_priority_Update,        \
    _Scheduler_priority_Enqueue,       \
    _Scheduler_priority_Enqueue_first, \
    _Scheduler_priority_Extract,       \
    _Scheduler_priority_Tick           \
  }
```

This initializer macro is used during application configuration to instantiate a scheduling algorithm. In the following, we describe the purpose each function in Scheduler_Operations.

3.2.1 Initialize

The initialize method allocates and sets default values for scheduler-specific global variables. This scheduler specific method is invoked during RTEMS initialization by rtems_initialize_data_structures before any idle threads or other tasks are started. This method can allocate memory from the RTEMS Workspace using the method _Workspace_Allocate_or_fatal_error. RTEMS considers failure to allocate memory during system initialization to be a fatal error.

3.2.2 Schedule

The schedule method updates the heir pointer when the priority of a thread changes if it was the executing or heir thread. This method is invoked from _Thread_Change_priority, which is invoked by rtems_task_set_priority, pthread_setschedparam, and when a priority is altered as part of acquiring or releasing a mutex with the priority inheritance and ceiling protocols.

3.2.3 Yield

The yield method determines if another thread should execute when the running thread voluntarily yields the processor. In addition to operations that explicitly yield the processor, methods that sleep or delay for 0 time call yield. This method is invoked in multiple places in RTEMS:

- cpukit/posix/src/nanosleep.c: nanosleep method when the requested delay is zero
- cpukit/posix/src/sched_yield.c: sched_yield method
- cpukit/rtems/src/taskwakeafter.c: if rtems_task_wake_after is passed zero ticks
- cpukit/score/src/threadtickletimeslice.c: when the executing thread's timeslice has expired in _Thread_Tickle_timeslice

3.2.4 Block

The block method removes the specified thread from the set of ready threads and, if necessary, updates the heir thread or dispatches a new executing thread. _Thread_Set_state invokes block when the specified thread is in the ready state.

3.2.5 Unblock

The unblock method adds the specified thread to the set of ready threads that it manages and, if necessary, updates the heir thread or dispatches a new executing thread. This method is invoked in _Thread_Clear_state when a change in a thread's current state results in all blocking states being removed so that the thread must be added to the set of ready threads.

3.2.6 Allocate

A scheduler implementation has two classes of data. The data it uses to manage the collection of threads, e.g. the ready queue, and the data it maintains on a per-thread basis. The allocate method is responsible for allocating any memory required for the per-thread data by the scheduler, returning a pointer to the allocated information structure that the scheduler stores in the TCB. This method is invoked as a side-effect of the _Thread_Initialize method, which is itself invoked by services that create and allocate threads, such as rtems_task_create and pthread_create.

3.2.7 Free

The free method is responsible for de-allocating any memory required on a per-thread basis by the scheduler. This method is invoked as a side-effect of _Thread_Close, which is called by services that delete threads like rtems_task_delete and pthread_exit.

3.2.8 Update

The update method modifies per-thread information when a thread's priority changes, which can occur during dynamic priority changes or as part of thread initialization. This method performs no scheduling actions and just updates the per-thread information structure to reflect the new priority.

3.2.9 Enqueue

The enqueue method is responsible for adding a thread to the set of ready tasks, and is invoked by _Thread_Change_priority, which is invoked by rtems_task_set_priority, pthread_setschedparam, and when a thread's priority is altered as part of acquiring or releasing a mutex with the priority inheritance and ceiling protocols. Enqueue performs no scheduling operations except to ensure the thread is placed in a scheduler-specific location in the set of ready threads.

3.2.10 Enqueue First

The enqueue first method places the specified thread into the set of ready tasks with a preference to break ties (e.g., when multiple ready threads have the same priority) in favor of the thread. This method is invoked when changing a thread's priority due to priority inheritance or ceiling protocols. Enqueue first does not call other scheduling operations, it just ensures the thread is placed in the appropriate, scheduler-specific location in the set of ready threads.

3.2.11 Extract

The extract method removes the specified thread from the set of ready tasks as a side-effect of setting a transient state for deletion or changing priority. This method updates the data structures for the management of the set of ready threads, but does not update the heir or executing threads.

3.2.12 Tick

The tick method implements the functionality a scheduler needs during a clock tick, for example tick-driven scheduling and timeslice management. This method is invoked as part of processing a clock tick by rtems_clock_tick.

3.3 Scheduling Algorithms

At the time of this writing, RTEMS ships with 5 scheduling algorithms: the Deterministic Priority Scheduler, EDF, CBS, Simple Priority, and Simple SMP Priority. All of these algorithms support preemptive scheduling with round-robin scheduling and timeslices for aperiodic tasks. The Deterministic Priority and Simple algorithms offer fixed priority scheduling, while EDF and CBS offer dynamic task priority, fixed job priority scheduling for periodic tasks. EDF schedules aperiodic tasks in the background, and CBS schedules them with a server algorithm.

RTEMS does not provide periodic tasks within the core scheduling subsystem. Instead, periodic tasks are implemented using a timer to track each periodic task's period. A regular task becomes periodic by creating a periodic timer and executing a loop that starts by settings its timer to the current tick plus its period. If the task finishes the loop body, it will sleep until its period elapses, at which point the timer will fire and wake the task. If the timer fires before the task finishes the loop body, a deadline overrun may have occurred and can be detected.

The Deterministic Priority Scheduler is highly optimized with one FIFO per priority and a two-level bitmap to ensure that both insertions and heir determination occur in constant execution time. This is the default scheduling algorithm in uniprocessor systems. Rate monotonic scheduling is supported on top of the Deterministic Priority Scheduler.

The EDF scheduler is an implementation of the well-known algorithm [8]. Since task priorities change each time a new job releases, the EDF scheduler needs to use a ready queue that can handle dynamic task priorities. The ready queue for the EDF scheduler uses a red-black tree that is capable of storing duplicate priorities (deadlines) in FIFO order.

The CBS scheduler is a resource-reservation scheduler built on top of the EDF scheduler with the addition of timer servers [6].

The Simple Priority Scheduler is a simple implementation of a fixed priority scheduling algorithm. It uses a single list to manage all ready threads and performs a linear search to perform insertion when a thread is unblocked (e.g. O(number of ready threads). This scheduling algorithm is inefficient from an execution perspective but uses little memory and may be appropriate for use in systems with few threads.

The Simple SMP Scheduler is the first SMP-aware scheduler implemented for RTEMS. It was designed to implement the same behavior as the Deterministic and Simple Priority Schedulers but extended to multiple processors.

3.4 Configuring a Scheduler

RTEMS can be configured to use either a built-in scheduler or a user provided scheduler. The Deterministic Priority Scheduler is the default scheduler and can be explicitly configured with the parameter CONFIGURE_SCHEDULER_PRIORITY. Alternate schedulers can be configured by defining one of the following parameters:

- CONFIGURE_SCHEDULER_EDF
- CONFIGURE_SCHEDULER_CBS
- CONFIGURE_SCHEDULER_SIMPLE
- CONFIGURE_SCHEDULER_SIMPLE_SMP

The scheduler also allows users to provide their own scheduling algorithm by defining configuration macros that will instantiate the scheduler.

- CONFIGURE_SCHEDULER_USER to indicate the application provides its own scheduling algorithm
- CONFIGURE_SCHEDULER_USER_ENTRY_POINTS must be defined with the set of methods which implement this scheduler, i.e. an initializer for the Scheduler_Operations virtual function table.
- CONFIGURE_MEMORY_FOR_SCHEDULER must be defined with the maximum amount of the RTEMS Workspace the scheduler allocates not including per-thread data, which is accounted for separately. Usually, this workspace memory is the storage required for the management of ready tasks, i.e. the ready queue.

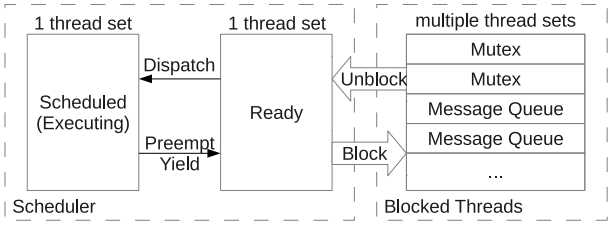

Figure 2: Management of thread sets in RTEMS.

- CONFIGURE_MEMORY_PER_TASK_FOR_SCHEDULER must be defined as the amount of memory required for all of the scheduler's per-task data, which is usually a formula multiplying the number of tasks (_tasks) configured by the maximum amount for each task.

3.5 Scheduler Simulator

In parallel to the refactoring of the scheduler, the Scheduler Simulator was designed and implemented to enable testing new scheduling algorithm implementations in a simple environment on the host. This testing can be done before writing test cases in C and attempting to execute them in the more complex environment of the actual RTEMS run-time on real hardware or a CPU simulator. The Scheduler Simulator script language enables creation of reproducible sequences which may not be easy to do with the scheduler on real hardware. The Scheduler Simulator consists of a subset of the RTEMS SuperCore, Classic API, shell, and a set of custom commands to access a critical subset of threading, semaphore, mutex, and time services. These are provided in the form of a library which can be utilized to instance scheduler simulator variants for custom algorithms.

The following is a simple scenario that initializes RTEMS, creates one user thread, has that thread sleep for three clock ticks, and then advances time by 5 clock ticks. This scenario includes the user thread blocking and the preemption of the IDLE thread from the clock tick ISR.

```
rtems_init
task_create USER
task_wake_after 3
clock_tick 5
exit
```

More complex scenarios can be constructed for example to create priority inversions and deterministic parallelism for testing SMP schedulers.

4. CONCLUSION AND FUTURE WORK

An RTOS enables developers to separate the effort of individual application development from the design, implementation, and maintenance of general-purpose services that spans multiple applications. In this paper, we have described some of these services and how RTEMS is designed to support their use. We have also discussed the details of RTEMS' modular scheduling framework, which supports the use of different schedulers across applications. We anticipate that such modularity will be important for future applications as the prominence of multicore systems increases in the real-time and embedded communities.

A modular scheduler has enabled new directions for RTEMS to grow. One direction is to return to the concept that the core of a real-time executive is the management of a set of threads, as shown in Figure 2. RTEMS uses thread sets to manage threads that are ready to execute, threads that are executing, and threads blocked waiting on a resource. The management of theses thread sets and associated data structures can be refactored into a set of helper classes that are, in an object-oriented sense, derived from the same virtual class. Such refactoring will allow the same source code to be used across schedulers and other thread-related services such as synchronization primitives. Another direction that RTEMS is rapidly moving toward is more and better support for SMP systems. The addition of more SMP-aware scheduling algorithms is viewed as an important evolution for RTEMS. Such algorithms can include features like processor affinity, better efficiency, and the ability to support powering down unused cores. The Scheduler Simulator was designed to support rapid design space explorations for such new scheduling algorithms.

Acknowledgments

Gedare Bloom is supported by NSF Grant CNS-0934725.

5. REFERENCES

[1] The newlib homepage. http://sourceware.org/newlib/, 2013.

[2] Parrot VM. http://www.parrot.org/, 2013.

[3] RTEMS on-line library. http://rtems.org/onlinedocs/doc-current/share/rtems/html/, 2013.

[4] RTEMS: Real-Time executive for multiprocessor systems. http://www.rtems.com/, 2013.

[5] rtems.git. http://git.rtems.org/rtems/, 2013.

[6] P. Benes. *Porting of resource reservation framework to RTEMS executive*. Master's thesis, Czech Technical University, Prague, Czech Republic, 2011.

[7] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171, 1989.

[8] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[9] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.