A Systematic Framework for Evolving TinyOS

Eric Trumpler, Richard Han
University of Colorado, Boulder
Eric.Trumpler@colorado.edu, Richard.Han@colorado.edu

Abstract

TinyOS [1] is a key element of the software infrastructure for the research and development community involved in realizing wireless sensor networks (WSNs). In order to improve the long-term impact of the research developed by the WSN community, we contend that it is important to exploit Moore's Law to augment TinyOS on current and future sensor motes. The tenfold increase in available RAM since the earliest motes makes possible the introduction of important and useful operating system constructs such as priorities and true multithreading. The phased introduction of these capabilities should maintain compatability with the existing body of nesC source code. Our framework, called TinyMOS, provides an evolutionary pathway that ultimately allows nesC applications to execute in parallel with other system threads.

1. Introduction

TinyOS was developed for extremely resource-constrained sensor nodes. Since the early Rene mote included only 512 bytes of RAM, the combined system overhead and stack use of any developed operating system had to be considerably lower than today's standards. TinyOS obtained this goal by implementing an elegant event-driven system, denoted primarily by a scheduler operating within a single thread of execution. The TinyOS code written in nesC [8] has a 226 byte memory footprint adapted to early motes like the Rene. The emphasis on limited resources is apparent in the delegation of many system-critical tasks to application programmers.

In its current condition, the TinyOS scheduler manages a FIFO task queue to handle all processes not in an event context. Typical system execution is triggered by an event that posts one or more tasks to the queue and quickly leaves its event context. These posted tasks cannot be preemptively time sliced or system terminated. Therefore any long-running processes within the queue, such as compression and encryption, will effectively block the system. Current attempts at augmenting TinyOS to address these issues are largely ad hoc and inflexible [10, 11]. The modifications to TinyOS become application-specific and are not integrated into the developer source, which make them difficult to use by the wider community.

Our approach begins with treating the entirety of TinyOS as a single scheduled thread on top of a multithreaded scheduler. This "master" scheduler shown in Figure 1 provides two features we feel are critical additions to TinyOS: priority scheduling and a new capability to spawn *slave threads*. Slave threads create an opportunity to schedule long-running and blocking tasks, or tasks of high priority, independent of the task queue. In this way, we can provide a framework that cleanly supports both preemption and priorities in order to mitigate the TinyOS programmer's obligation to concurrent, task-safe code. Since the thread management of the master scheduler is invisible to TinyOS processes, developers are effectively able to create tasks decoupled from prior run-time restrictions. We have proved the feasibility of this approach with an implementation called *TinyMOS* described in this paper.

Within the context of Moore's Law, today's motes already offer sufficient memory to execute such advanced OS functions [3, 2]. The advent of the next generation has provided ten times more memory than in the first generation motes at reduced cost. Today's standard MICA2/z [7] and TELOSB [6] motes provide 4 and 10 KB of RAM respectively. While we don't necessarily expect that sensor nodes will follow these trends to become GHz/GB devices, primarily due to power constraints, we do expect the standard sensor mote to continue to evolve and show modest resource gains. We believe that it is important to the future of this community to take advantage of these additional resources and develop more sophisticated systems, and to do so in a manner that is evolutionary and systematic rather than ad hoc.

2. Related Work

SOS [3] is one of two WSN operating systems supporting dynamic modules. Although not the primary focus of

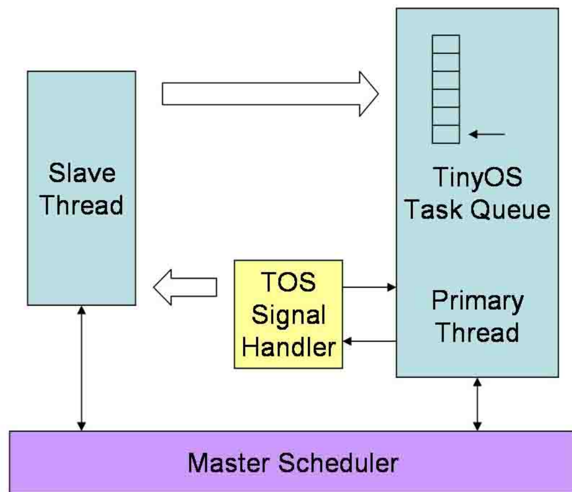

Figure 1. TinyMOS framework: slave threads are dynamically created and post results to the Primary thread as a task before terminating. The Primary thread and any slave threads are time sliced by the master scheduler.

SOS, the more relevant addition to SOS is a set of priority queues within the event-driven scheduler. The priority levels are designed to service messages - tasks between external modules - called from an interrupt context and deemed to be of a high priority. However, since the queued high priority task is unable to preempt the system, its execution is dependent on the current processes time to completion. We believe high priority tasks combined with preemptive thread control is a more effective long-term solution.

Contiki is another event-driven sensor OS themed toward dynamic modules. However, the other kernel features suggest a similar hybridized system with both simulated and preemptive multithreading. Protothreads [9] are thread-like methods that provide a blocking context to event-driven tasks without individually allocated thread stacks. Although this limits protothreads to only global variables, they are primarily designed to facilitate limited concurrency. When necessary, applications may link an external library for cooperative and preemptive multithreading functionality. However, whereas Contiki's library functions at a high level and is secondary to event-driven scheduling, we will show that enforcing a multithreaded scheduler at a low level is a generic, practical method to introduce preemption.

The MANTIS operating system (MOS) [2] uses a lightweight multithreaded scheduler with preemptive time slicing. We use this operating system as a template for the design and implementation of TinyOS above a multithreaded scheduler. Since the MOS code is written in ANSI C, it's backwards compatible with nesC, a language built on top of C. Applications in nesC are first compiled into C code which is then passed to a standard C compiler. By linking these applications with the MOS kernel they have access to a multithreaded scheduler and associated thread function calls.

A similar approach has been applied to real-time operating systems [12]. This example employs a hierarchical scheduler that adds a significant possibility for race errors. To address this problem they offer a concurrency analysis tool for detecting and avoiding race conditions at each hierarchy. The issue of concurrency is solved in TinyMOS by the dependent nature of slave threads, and locking access that is sufficient protection for persistent threads in our two level scheduling.

3. Augmenting TinyOS with Slave Threading

One of the key motivations of our TinyMOS framework is to offer the TinyOS developer the capability to spawn one or more slave threads from within TinyOS, and thereby bypass the problem noted earlier with cooperative scheduling, namely that a long-running task can block other urgent tasks from running in TinyOS. Rather than a construct, slave threading is the conceptual use of thread creation and termination to mirror the facilities of posted TinyOS tasks without leading them to become I/O driven. These slave threads will execute concurrently with the thread encapsulating the TinyOS scheduler and task queue, hereafter called the Primary thread. In this usage, the programmer will create one or more slave threads from within the Primary thread, and place within each slave a self-contained compute-bound long-running task, which will be time sliced in parallel with the main TinyOS thread, thereby minimizing blocking of the processor by long-running tasks. Our design allows TinyOS to be the dominant thread and new threads to execute as "slaves" to the core TinyOS scheduling mechanics. This approach also takes advantage of thread priorities and acts as a midway between an event-driven and a persistently multithreaded system.

3.1. TinyOS as a Thread - Implementation

With only minor modifications, we have implemented TinyOS to run as a thread within the multithreaded MANTIS operating system. The Unix-style scheduler gives each application a starting thread and allows users to create new threads at will with a specified stack size and priority. By setting the default thread to begin executing the nesC main function, we effectively start the TinyOS scheduler as an

event-driven thread. Without utilizing any multithreaded features, this is a self contained copy of TinyOS.

3.2. The Master Scheduler

Our implementation presents two levels of scheduling. While the TinyOS scheduler operates the Primary thread, we also enforce the underlying MANTIS scheduler, referred to as the master scheduler because it operates outside the TinyOS domain. When multiple threads are running, the master scheduler is essential for creating and preemptively time slicing new threads. However, if no additional threads are active, the master scheduler has only limited responsibilities concerning sleeping and power management.

3.3. Spawning Slave Threads

```
event result_t timer.fired()
  {
     post compute_bound_task();
     return SUCCESS;
  }

event_result_t timer.fired()
  {
     mos_thread_new(compute_bound_task,
                    stack_size,
                    PRIORITY_NORMAL);
     return SUCCESS;
  }
```

Figure 2. An interface transition for applying a long-running task to a slave thread.

The concept of slave threading enables a TinyOS developer to call *mos_thread_new(mytask, stack_size, priority)* to replace any computation that may excessively occupy the processor. The code in Figure 2 shows a practical use of a slave thread in place of a task. New threads are contextually equivalent to TinyOS tasks; they can be preempted by events, share global variables, and have no passed parameters. As the figure shows, modifying code to support the MOS thread interface is clean and minimal.

Figure 1 shows the TinyOS scheduling sequence with a single slave thread. When a TinyOS signal handler begins a new slave thread, the slave thread runs in parallel with the Primary thread. The essential benefit is whenever possible the programmer is not responsible for dividing an algorithm into concurrent-friendly segments. In addition to support long-running and blocking tasks, slave threads that wait for I/O or other signals will only take CPU cycles when not in the blocked state, and will never preempt the TinyOS scheduler while blocked. However, to protect access to shared resources at the application layer slave threads should not initiate signals of their own. Instead, a slave thread should post a standard TinyOS task to pass information to the Primary thread before it terminates. A more detailed discussion of how TinyMOS deals with locking access to shared resources to prevent race conditions can be found in Section 5.

Application	Blink	Surge
TinyOS	49 bytes	1929 bytes
Slave Threading	354 bytes	2234 bytes
Shared Resources	580 bytes	2263 bytes

Table 1. RAM usage for TinyOS applications Blink and Surge through our stages of modification.

The master scheduler framework requires additional memory, but is sufficiently lightweight that this approach is still practical on MICA2 and TELOS standard motes. Minimized without device drivers, the master scheduler and related code of MOS uses 305 bytes in RAM, including stack allocation for the TinyOS thread. This is a relatively small cost to achieve our proposed benefits in complex applications. The memory that a TinyOS application occupies is added to this value, though space is not typically consumed in the thread stack because data is globally defined. Table 1 shows the memory requirements for the Blink and Surge applications with the initial phases of our evolution implemented on the MICA2 (we will address the contention of shared resources and interfaces in section 5). Since the memory required for MOS is a static value, Blink gains an overhead six times greater than its original size. By comparison, a Surge application already uses over 1900 bytes in RAM so the features we add become more practical relative to our 300 byte cost. The total size of Surge is still significant but well under the MICA2's 4 KB maximum, leaving the remaining memory for additional thread stacks.

Overhead also exists for each concurrent thread, both in terms of stack size and context switching. Each new thread must allocate a minimum 30 byte thread stack from the heap, plus the size of local variables. As mentioned earlier, even the largest TinyOS applications still leave enough space for many new threads. CPU cycles required for context switching are inherent to preemptive multithreading. Within our MANTIS OS template each context switch requires about 60 microseconds. Although most event-driven systems argue that this is a substantial loss, only 3% of the processor is spent in context switches relative to the 20 millisecond time slice (since having been modified from the

published 10 ms) [2]. Slave threads are not static so they have no context switches until creation. However, constant thread turnover has a small cost. Each new thread takes approximately 50 microseconds to initialize. Since the relative CPU usage is proportional to the interval at which each task delegated to a slave thread, rapid thread creation becomes linearly more expensive. Thus slave threads are most efficiently used with long-running tasks that are relatively infrequent in being spawned.

4. Augmenting TinyOS with Priority Scheduling

A pivotal advantage to our two level scheduling TinyMOS framework is that it systematically provides a thread-based priority system set by the master scheduler. Evolving sensor applications have already encountered a need for such dynamics [10, 11], specifically in wireless security. TinySec, a security architecture integrated with TinyOS, requires a modified two-priority scheduler to support its encryption algorithms. Without priority scheduling, cryptographic operations in the task queue risk not meeting their real time deadlines. Although this case shows it is possible to replace the TinyOS scheduler, those changes are not developer supported, so in cases like TinySec application programmers are required to rewrite several scheduler functions themselves.

All MANTIS threads typically fall into HIGH, NORMAL or SLEEP priority. We default the priority of the Primary thread by setting the last argument in a *mos_thread_new(tosmain, stack_size, PRIORITY_NORMAL)* call. Although this may be changed, setting the TinyOS scheduler to a NORMAL priority permits new threads that are prioritized above or below it.

When a HIGH priority slave thread is created, it immediately preempts any threads at a lower priority level than itself, but will return control to those lower priority threads, including the Primary thread, only after it runs to completion. Although these threads are limited by their slave capabilities they can effectively preempt the TinyOS scheduler when the priority necessitates the action.

Slave threads spawned at a SLEEP priority have the unique ability to be entirely preempted by the TinyOS task queue. SLEEP priority threads are considered a low priority and will only run when the TinyOS scheduler is in an idle (sleep) state. Putting it another way, a low priority thread will only be given CPU control when the task queue is empty. Scheduling threads beneath the task queue keeps processor priority on important system tasks and events, while necessary but time-insensitive threads are permitted to run when no critical tasks are present. For example, a node aggregator may want to set slave-threaded data aggregation and compression algorithms at a SLEEP priority so they do not conflict with event-driven tasks used to transmit and receive incoming packets. However, since the radio interface could alternatively be set to a HIGH priority, this goes to show the flexibility of the master schedulers thread control.

5. The Path Towards Persistent Multithreading

As the next phase in the evolution of TinyOS, we describe how TinyOS can further exploit this TinyMOS master scheduler framework to not only spawn run-to-completion slave threads from the Primary thread, but also to spawn persistent parallel threads that handle other important tasks.

Our to-date implementation allows TinyOS to start one or more threads at the initialization sequence that then become permanent system components. Each of these threads are static, in contrast to dynamic slave threading, and are designed to decouple functionally different TinyOS interfaces from a single queue of tasks. Figure 3 illustrates a multithreaded TinyOS node running a standard routing protocol. As the figure emphasizes, the computations and signals for each interface are handled within that thread's context, and may be preempted. We keep these signals separate because they are no longer mutually exclusive; multiple threads may simultaneously be in a signal context without exclusive CPU access.

Since threads can now operate independently, promoting thread-safe operation within TinyOS requires resource contention tools within the nesC code. Event-driven systems have no locking mechanisms in place due to their linear process management. To solve this problem we redirected all TinyOS device accesses to a hardware contention layer (HCL) that interfaces to device drivers written in MOS. Using radio commands as an example, calls to the SendMsg interface are transparently translated by TinyMOS to *com_send(IFACE_RADIO, packet)* defined in the MOS radio driver. Upon entering the driver the resource is locked during execution by calling *mos_mutex_lock(radio_mutex)* and is unlocked on exit.

Unlike the real-time example [12] we only need two degrees of locking; in the layers above and below the HCL. Mutexes sufficiently protect against race errors below the HCL by locking device drivers. To protect resources at the layers above the HCL, access to global variables and thread-to-thread communication should only be facilitated by posting tasks to the FIFO queue.

Taking advantage of our shared resources optimizes the combined memory footprint. Referring to Table 1 again, every slave-threaded application has a 305 byte static overhead added by MOS. When resources are shared, the over-

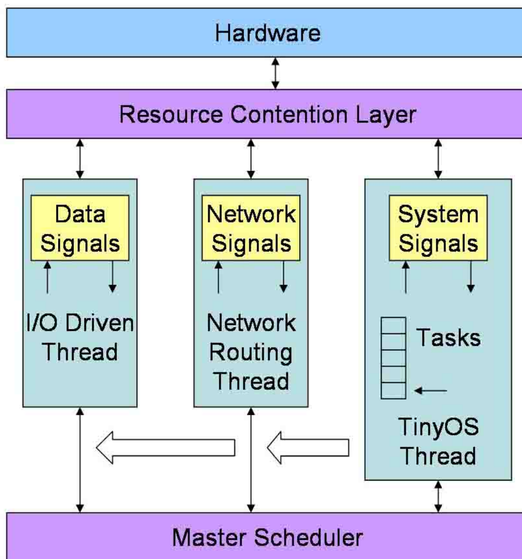

Figure 3. TinyMOS framework: A practical implementation of persistent multithreading within TinyOS. Multiple threads must access hardware through a resource contention layer for thread-safe device access.

head is initially high but subsequently normalizes when applied to complex applications with shared modules. In our implementation, the Blink application rises to 580 bytes, indicating a roughly twofold inflation due to MOS code. But, as elaborate applications like Surge utilize more of the optimized interfaces, we seea almost equivalently efficient memory use than a strictly slave-threaded implementation. In our example the optimized Surge footprint, including the resource contention layer, is only 29 bytes larger than its last code size in MOS.

6. Summary and Future Directions

We have offered a lightweight framework called TinyMOS for systematically evolving TinyOS to acquire more advanced yet highly useful OS constructs that minimize scheduling responsibilities given to the programmer. We have have shown through a succession of implementation examples the feasibility of using TinyOS as the ”primary” thread that spawns slave threads, higher priority threads, and persistent threads. In the future, we envision that this framework will be used to evolve beyond the master/slave relationship described here, and move towards a peer relationship where nesC application threads execute in parallel with threads that are their own masters.

Acknowledgments

We wish to thank Lakshman Krishnamurthy at Intel Research for suggesting this line of research.

References

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister. “System Architecture Directions for Networked Sensors”. Proceedings of Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), November 2000.

[2] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, Richard Han. “MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platform”. ACMKluwer Mobile Networks & Applications (MONET) Journal, Special Issue on Wireless Sensor Networks, August 2005.

[3] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler and Mani Srivastava, “SOS: A dynamic operating system for sensor networks”. Proceedings of the Third International Conference on Mobile Systems, Applications, And Services (Mobisys), 2005.

[4] Adam Dunkels, Bjrn Grnvall, and Thiemo Voigt. “Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors”. Proceedings of the First IEEE Workshop on Embedded Networked Sensors, 2004.

[5] Jonothan W. Hui, David Culler. “The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale”. Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys, 2004.

[6] Moteiv Tmote Sky motes, http://www.moteiv.com.

[7] Crossbow MICA motes, http://www.xbow.com.

[8] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. “The nesC language: A holistic approach to networked embedded systems”. In Proc. SIGPLAN’03, 2003.

[9] Adam Dunkels, Oliver Schmidt, and Thiemo Voigt. Using Protothreads for Sensor Node Programming. In Proceedings of the REALWSN 2005 Workshop on Real-World Wireless Sensor Networks, Stockholm, Sweden, June 2005.

[10] Chris Karlof, Naveen Sastry, and David Wagner. “TinySec: A Link Layer Security Architecture for Wireless Sensor Networks”. Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys 2004). November 2004.

[11] Joseph Polastre, Jonathan Hui, Philip Levis, Jerry Zhao, David Culler, Scott Shenker, Ion Stoica. “A Unifying Link Abstraction for Wireless Sensor Networks”. In Proceedings of the Third ACM Conference on Embedded Networked Sensor Systems (SenSys), November 2-4, 2005.

[12] John Regehr, Alastair Reid, Kirk Webb, Michael Parker, Jay Lepreau. “Evolving Real-Time Systems Using Heirarchical Scheduling and Concurrency Analysis”. Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003), December 2003.