Adding Preemption to TinyOS

Cormac Duffy[1], Utz Roedig[2], John Herbert[1], Cormac J. Sreenan[1]
[1]Computer Science Department, University College Cork, Ireland
[2]InfoLab21, Lancaster University, Lancaster

***Abstract*— Event-driven operating systems such as TinyOS are the preferred choice for wireless sensor networks. Alternative designs such as MANTIS following a classical multi-threaded approach are also available. Event-based systems are generally more energy efficient than multi-threaded systems. However, multi-threaded systems are more capable than event-based systems of supporting time critical tasks as task preemption is supported. Timeliness can be traded for energy efficiency by choosing the appropriate operating system. In our recent work we have shown that the multi-threaded system MANTIS can be modified to be as energy efficient as TinyOS. As a result, the modified MANTIS can be used to fit both sensor network design goals of energy efficiency and timeliness. This solution is not considered optimal as most existing sensor network applications and software libraries are developed for TinyOS. Therefore, we present a TinyOS modification that adds preemption while retaining the existing TinyOS structure and features.**

I. Introduction

Sensor nodes must be designed to be energy efficient in order to allow long periods of unattended network operation. However, energy efficiency is not the only design goal in a sensor network. For example, timely processing and reporting of sensing information is often required as well. This might be needed to guarantee a maximum delivery time of sensing information from a sensor, through a multi-hop network, to a base-station. To be able to give such assurances, network components with a deterministic behavior will be required. The operating system running on sensor nodes is one such component.

Event-based operating systems are considered to be the best choice for building energy efficient sensor networks as they require little memory and processing resources. Hence, the event-based TinyOS [1] is currently the preferred operating system for sensor networks. Event-based operating systems are not very useful in situations where tasks have processing deadlines. As tasks are processed sequentially, prioritizing important tasks to meet processing deadlines is not possible. Multi-threaded operating systems are more suitable if such requirements must be fulfilled. Thread preemption and context switching enables such systems to prioritize tasks and meet deadlines. The MANTIS [2] operating system is a multi-threaded operating system designed specifically for wireless sensor networks. MANTIS has a relatively high processing overhead for thread management. This processing overhead is directly related to reduced energy efficiency because of the relative increase in CPU activity.

This creates the dilemma that both design goals - energy efficiency and timeliness - can only currently be optimized independently. One is forced to choose which goal is of higher importance in the considered application scenario. Therefore, it would be good if the dilemma could be resolved by either making MANTIS more energy efficient or TinyOS more responsive.

Our previous work [3] concentrated on the first option: A MANTIS kernel modification to increase power efficiency. As the results show, MANTIS can be modified to be as power-efficient as TinyOS without impacting vital kernel functionality. Thus, the modified MANTIS can be used to solve both important sensor network design goals. The result of this previous work also shows that the common belief that "multi-threaded operating systems are not suitable for resource constrained sensor networks" is not necessarily true.

The modified MANTIS provides a solution for our previously outlined dilemma but has other considerable limitations. The sensor network community selected TinyOS as the defacto standard with most existing applications, libraries and device drivers available for TinyOS. Therefore, to avoid re-coding existing software and allow re-usage of existing TinyOS infrastructures it is worth exploring the second option: A TinyOS modification to increase system responsiveness. This paper presents a modification that adds preemption to TinyOS which results in a responsive system that retains its existing structure and features.

The next Section of the paper presents related work. Section III describes briefly TinyOS and explains its limitations in terms of responsiveness. Section IV explains in detail our TinyOS modifications to add preemption. Section V presents an evaluation of the modified system. It is shown how existing applications can take advantage of the new preemptive scheduler. Section VI concludes the paper.

II. Related Work

In [4], the TinyOS operating system is executed within a multi-threaded AVRX kernel as part of a concurrency analysis study. Thus, any TinyOS task could be preempted by another AVRX thread. This solution has some drawbacks. The solution is bound to AVR based microprocessors. Furthermore, the AVRX kernel provides many threading features not necessarily needed for event-based programming. The system has memory requirements of both, TinyOS and the AVRX kernel.

A similar approach with comparable limitations can be seen in [5]. Here, the TinyOS operating system is executed as a thread within the multi-threaded MANTIS operating system. The resulting *TinyMOS* system has a large memory footprint (see Section V). Many context switches (for example, introduced by time-slicing) create a significant processing overhead

(see [3]). In addition, TinyOS and MANTIS programming semantics are mixed which makes TinyMOS usage difficult.

A different approach is described in [6]. Here, a multi-threading library for TinyOS called *TinyThread* is presented. The TinyThread library provides TinyOS programmers with a thread programming abstraction but does not enable task preemption. A thread scheduler in the form of a TinyOS task is periodically placed in the task queue. Threads are then scheduled and run to completion or until they block. This approach allows users to multiplex standard TinyOS tasks and threads, but does not facilitate preemption and cannot provide any degree of performance control. Furthermore, threads are programmed in a different fashion to normal TinyOS code which does not allow a seamless integration of TinyThreads with existing TinyOS applications.

The approach presented in this paper differs from the described existing works in major aspects. Thread preemption is added natively to TinyOS. Context switching is only used to facilitate task preemption and not to introduce a thread programming abstraction. Standard TinyOS programming conventions are used such that preemption features are seamlessly integrated.

III. The TinyOS Architecture

This section first describes the basic TinyOS functionality affected by our modifications. Thereafter, the limitations of TinyOS motivating our modifications are discussed.

A. Basic Functionality

The TinyOS system and specialized applications are written in a component based programming language called *nesC*. The components are self contained modules of code that interact with each other through strict interfaces. A component interface is characterised by a number of event handling functions. Event-based applications are implemented as series of event-handlers and tasks. TinyOS tasks are deferred function calls that are placed in a simple FIFO task-queue for execution. TinyOS tasks are taken sequentially from the queue and are run to completion. Once running, a TinyOS task can not be interrupted (preempted) by another TinyOS task. Event-handlers are triggered in response to a hardware interrupt and are able to preempt the execution of a currently running TinyOS task. Event-handlers perform the minimum amount of processing to serve the event. Further processing is performed within a TinyOS task that is normally created within the event handler. After all TinyOS tasks in the task queue are executed, the TinyOS system enters a sleep state to conserve energy. The sleep state is terminated when an interrupt occurs.

B. TinyOS Limitations

A new TinyOS task is normally posted to the scheduler from within an interrupt and usually processes data that was obtained during the interrupt routine. For example, the interrupt could signal sensor activity or the arrival of a network packet; the corresponding task will then process the sensor reading or handle the incoming data packet. A new task is inserted at the end of the FIFO task queue and it is executed as soon as all other tasks in the queue have been processed. Fig 1 A shows an example of four events creating four different tasks during interrupt handling to process the data. The problem is that some tasks might be of higher importance than others and it is desirable to schedule them before all others. For example, it might be desirable to handle a network packet before processing new sensor information in order to assure packet forwarding deadlines. This limitation of the FIFO task scheduling in TinyOS 1.0 was recognized and thus the new version TinyOS 2.x offers the option to alter the task scheduler which allows us to prioritize specific tasks. For example, as shown in Fig 1 B, task T_3 can be queued before T_2 to prioritize processing of the third event. Our TinyOS modification presented in the next section will make use of this feature introduced in TinyOS 2.x.

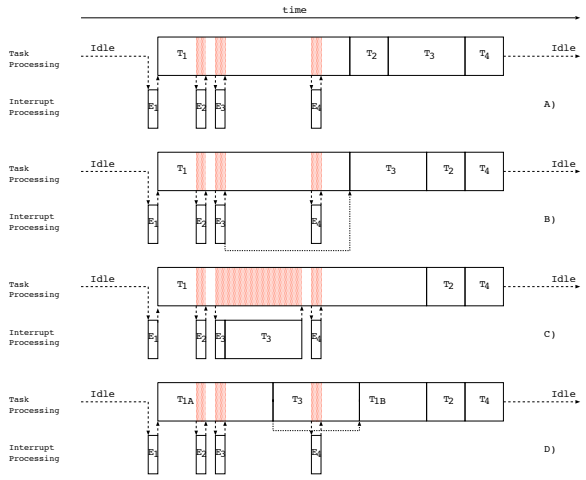

Fig. 1. TinyOS task processing options

The possibility of re-ordering tasks improves the event-handling capabilities of the operating system. However, a severe limitation of the system still exists. If a task is currently executing, a new task created during an interrupt will be executed after the current task finishes processing. The time at which this new task will be scheduled cannot be controlled in TinyOS 2.x as it is impossible to preempt the currently running task. In the example Fig 1 B, task T_3 is prioritized but still has to wait for T_1 to finish before it is executed. Some tasks can have a long processing duration which will defer the execution of an important task for an unacceptably long period of time[1].

Currently, this limitation can be addressed in two different ways. One option is to move task processing functionality in the interrupt processing routine (see Fig 1 C). The currently running task is preempted and high priority processing is performed in the interrupt context. This solution is not optimal as interrupts are disabled in TinyOS while executing an

[1]The complexity of a sensing operation depends on the phenomenon monitored, the sensor device used and the data pre-processing required. If, for example, an ATMEGA128 CPU with a processing speed of $4Mhz$ is considered (a currently popular choice for sensor nodes) in conjunction with a camera, image processing might take some time before a decision is made. Depending on camera resolution and image processing performed, a sensing task can easily take more than $100ms$ [7].

interrupt. For example, if in Fig 1 C E_4 would occur earlier during processing of T_3 in the context of E_3 the handling of E_4 would be deferred to the end of T_3. If E_4 has a higher priority than E_3, control over execution times will be lost at this point. Another option is to split longer tasks into smaller subtasks. For example, in Fig 1 D task T_1 is split in two smaller tasks T_{1A} and T_{1B}. T_{1A} is posted before task T_{1B} and therefore task T_3 can be scheduled before T_{1B}. This solution is not always optimal, as not all tasks can be split-up easily into several sub-tasks [7]. In addition, the programmer has to ensure that task-splitting is organized such that all processes can meet their deadlines, which is quite difficult to achieve in a practical scenario.

IV. The TinyOS Modifications

To mitigate the TinyOS limitations described in the previous section priority scheduling and task preemption is added to the TinyOS 2.x operating system.

A. Priority Scheduling

TinyOS 2.x facilitates component-based schedulers that can be included in the operating system if required. The first step in our TinyOS modification is the development of a new priority based scheduler component to replace the provided standard TinyOS 2.x FIFO scheduler. Depending on the performance requirements of the user application, this new Priority Level Scheduler (PL scheduler) can be wired into the application to facilitate greater control over which tasks are processed first. The PL scheduler provides five different priority levels:

- (P_1) High Priority Preemptive
- (P_2) High Priority Non-Preemptive
- (P_3) Basic Priority (Used for standard TinyOS tasks)
- (P_4) Low Priority Non-Preemptive
- (P_5) Low Priority Preemptive

In each level, tasks are scheduled in a FIFO manner. The basic priority level must always be supported as all standard TinyOS tasks are queued here by default. The adjacent priority levels provide a non preemptive higher and lower priority queue. Thus, tasks in either of these queues will be scheduled according to their priority but will not preempt any actively running task resulting in a behavior as shown in the example Fig 1 B. The high priority preemptive task and the low priority preemptive task queues can be used to schedule preemptive tasks. A high priority preemptive task will preempt any running task from the lower priority task levels and any task from these levels can preempt a running low priority preemptive task. Implementation details of the preemption mechanism are described later.

In practice not all levels of priority are necessary and as such allocating a task queue for five different priority levels can create a bloated scheduler. The component architecture of TinyOS facilitates counting up the number of tasks at compile time. The PL scheduler can determine exactly how many queues are required and the code elimination features of the nesC compiler remove redundant interfaces for task priorities not used. If more than five priority levels are required, the PL scheduler can be extended to provide these. However, we believe that five levels are sufficient to support common sensor network scenarios.

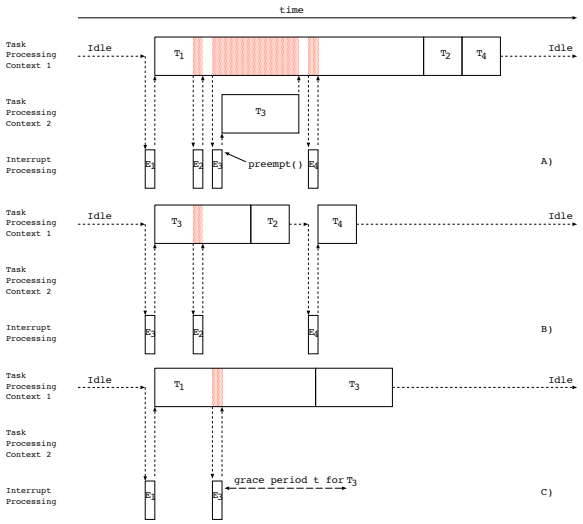

Fig. 2. Modified TinyOS task processing options

B. Preemption

Task preemption is facilitated by the PL scheduler for situations in which a cooperative task schedule will not meet the application's temporal requirements.

1) Conceptual Idea: Task preemption requires costly context switches that have to be supported by the operating system. These context switches must be implemented carefully to avoid a significant increase in system overhead and energy consumption. Our previous research [3] on optimizing preemptive scheduling for the multi-threaded MANTIS system highlighted that it is of paramount importance to reduce the number of context switches. With this design requirement in mind the PL scheduler avoids preemption where possible using two different principles.

As a first principle, a context switch is only performed if it is necessary to match processing deadlines. An example of this behavior is illustrated in Fig 2. Task T_1 is executing with basic priority P_3 and a task T_3 with priority P_1 (high priority preemptive) is scheduled at the end of the interrupt routine E_3. A context switch is now necessary to process the high priority task T_3. Thereafter, the context is switched back and the original task T_1 executes to completion (see Fig 2 A). If the same high priority task T_3 is scheduled in a scenario where the system is idle (see example Fig 2 B), no context switch is performed and the task executes immediately. In this case the high priority task T_3 will be executed in the standard context. In other words, a context switch is not associated with the priority level of a task, it is associated with the need for preemption. This mechanism reduces the number of context switches compared to existing preemption techniques in multi-threaded systems where the execution of a higher prioritized thread is normally bound to a context switch (for example, [5]).

Algorithm 1 Priority Task Structure

```
1: Module SomeComponentC{
2:   uses interface PriorityTask<HighPreempt>;
3: }
4: Implementation{
5:    event void someEvent(){
6:    call PriorityTask.postTask()
7: }
8:
9: event PriorityTask.runTask(){
10:    //task code
11: }
12: }

13: Configuration SomeComponent{
14: }
15: implementation{
16:   components new PriorityTask() as PremptingTask;
17:      components SomeComponentC, .....
18:      SomeComponetC.PriorityTask->PremptingTask;....
19: }
```

Algorithm 2 Priority Scheduler Structure

```
1: Module PLScheduler{
2:    provides interface TaskPriority<HighPreempt>[id];
3:    provides interface TaskPriority<HighNonPreempt>[id];
4:    provides interface TaskBasic[id];
5:    provides interface TaskPriority<LowNonPreempt>[id];
6:    provides interface TaskPriority<LowPreempt>[id];
7: }
```

As a second principle to reduce context switches, a grace period t for preemption is used. It is assumed that many high priority tasks need to be executed within a specific time frame but not necessarily immediately. A timer is used to mark the latest possible point in time when the task must be executed to match deadlines. If currently running lower prioritized tasks complete before the grace period t, all tasks can be executed without preemption. Such a scheduling situation is depicted in Fig 2 C. Task T_1 is executing with basic priority P_3 and a task T_3 with priority P_1 (high priority preemptive) is scheduled at the end of the interrupt routine E_3. Task T_3 has a grace period of t and thus, preemption is not necessary to schedule the high priority task in time.

The PL scheduler requires memory for three separate stacks to store the processing state of the three preemptive task priority levels (P_1, P_3, P_5). As there are only three preemptive priorities only three stacks need to be allocated. The number of required stacks is dependent on the number of preemptive priority levels and not on the number of tasks used. Due to the fixed number of stacks used, a calculation of required stack sizes is simplified. In practice, the bulk of all tasks will be run in the same stack as regular TinyOS tasks are defaulted to the basic priority level P_3.

2) Implementation Specifics: A component specifies a priority task by wiring a priority task interface to the PL scheduler component and by implementing the interface event *runTask()*. This procedure conforms to the TinyOS Enhancement Proposal (TEP) 106 on tasks and schedulers.

An example of an implemented priority task can be seen in Alg. 1. For a component to use a priority task it must implement the *PriorityTask* interface and specify the task priority as one of the interface parameters (Alg. 1, line 2). The interface provides a *postTask* command which is the same as the basic task syntax *post [task name]* (Alg. 1, line 6) and the *runTask* event handler which stores the task functionality (Alg. 1, line 9). The event handler is invoked by the scheduler when the task is scheduled to be processed.

Each task must then be wired up to one of the five parameterized *taskPriority* interfaces provided by the PL Scheduler (Alg. 2, link 2-6). The wiring process is somewhat simplified by the generic *PriorityTask* component (Alg. 1, line 16), which uses the interface parameter information to determine the task priority and uniquely wire each task to the appropriate scheduler interface.

The PL scheduler, is an extension of the TinyOS 2.x FIFO scheduler. Depending on the number of task priorities of the operating system, up to five different task queues are initialized. A bit field is initialized to keep track of which task priorities are either preempted or actively processing. On receiving a posted task the scheduler first ensures that the task has not already been posted (TinyOS TEP 106 requirement). Second, the scheduler checks if the task will be delayed by a lower priority task actively running. If preemption is required, the scheduler will perform a context switch or set a grace period timer to delay the context switch. The grace period time t is a fixed global value for all tasks in the current implementation.

The context switch requires that the current registers are saved to the current stack and the stack pointer register is then directed to the next stack context containing the preempted queue scheduler. The preempted queue scheduler executes all tasks sequentially starting from the highest priority task down as far as the priority of the preempted task. The scheduler can therefore process multiple high priority tasks waiting on the preempted task to finish requiring only 2 context switches to execute a set of high priority tasks. When there are no more tasks enqueued waiting on the preempted task, the context is switched back to the preempted task context and the preempted task can finish executing.

The stack size of the required stacks is currently specified and allocated at compile time. To obtain a good estimate of the required stack size, a tool as proposed in [6] can be used.

Unlike preemptive multi-threaded systems such as [5], [6] thread blocking procedures are not necessary. The PL scheduler schedules and executes all tasks according to the event-based architecture.

C. Race Conditions

The PL scheduler adheres to all the finalized TinyOS 2.x TEP specifications on TinyOS tasks and schedulers with one exception: tasks are not guaranteed to execute in a sequential manner. Only tasks within the same priority are guaranteed to execute sequentially. A higher priority task can preempt a lower priority task and modify shared memory creating a race condition. Currently, the TinyOS nesC compiler is not designed to detect such race conditions. Thus, the programmer must be aware of these additional programming complications introduced in using a preemptive scheduler.

In TinyOS, the defacto method to prevent race conditions is to enclose the race condition sensitive code in an *atomic*

statement. The *atomic* statement prevents race conditions by disabling hardware interrupts, which are the only events that can cause race conditions in the TinyOS concurrency model. However, in the modified TinyOS, race conditions can also occur when a task preemption occurs. To ensure that the atomicity of *atomic* sections is preserved, the PL Scheduler checks that the active task is not executing *atomic* code before preempting.

V. Evaluation of the Modified TinyOS

The usability of the TinyOS modifications to extend existing application code to provide preemption is evaluated. In addition, the modified TinyOS is compared with the existing solutions *TinyMOS* [5] described in the related work section.

A. Usability Evaluation

Tmote Sky nodes with a cc2420 zigbee radio transceiver are used for the evaluation. To test the TinyOS modification a slightly modified version of the well known TinyOS 2.x *RadioCountToLeds* application is used. The application periodically broadcasts a 3 bit message to other nodes every $t_{blink} = 250ms$ and displays any received messages by toggling the LEDs. The *RadioCountToLeds* application uses the standard TinyOS communications stack to send and receive messages. After receiving a message, the cc2420 radio stack posts a task *receiveDone_task()* to signal to higher level components that a message has been received. In addition, a timer is used to post a computationally expensive task every $t_{comp} = 1000ms$. This task requires $100ms$ to finish on the Tmote Sky node. This computational expensive task is not part of the standard *RadioCountToLeds* application and is used to visualize the advantage of preemption features.

Standard TinyOS: In the standard TinyOS system, the computational expensive task blocks the task posted by the cc2420 radio stack that finally processes incoming messages. Therefore LEDs are toggled with a delay of $100ms$ if the computational expensive task was just scheduled. Delayed toggling of LEDs is obviously not a serious problem but this demonstrates problems in TinyOS driven sensor networks if they are to be used in time critical application scenarios.

Modified TinyOS: In the modified TinyOS system a low priority P_5 is assigned to the computationally expensive task. This is done by re-wiring this task to the correct priority level of the PL scheduler. The rest of the application remains unaltered. Now the computationally expensive task is preempted by the cc2420 task posted after receiving a radio message as standard TinyOS tasks run in priority level P_3. The LEDs now toggle state as in the original *RadioCountToLeds* application; the computationally expensive task is executing in the background. Only a minimal application modification is necessary to achieve the desired application behavior.

B. Comparative Evaluation

The previously described *RadioCountToLeds* application version was implemented using the *TinyMOS* concept. In the TinyMOS variation the computationally expensive task is implemented as a MANTIS thread running at a lower priority than the thread carrying the TinyOS system. Thus, the resulting application has the same behavior as the system using the modified TinyOS with PL scheduler.

However, both solutions differ in important aspects. First, the computational expensive task has to be implemented using MANTIS semantics. A programmer has to be familiar with both TinyOS and MANTIS syntax to develop the application. A seamless integration of preemption features in TinyOS is not achieved. Second, the TinyMOS solution has a significantly higher processing overhead as more context switches than in the presented PL scheduler solution are required. This additional overhead translates to an increase in energy consumption as less idle-time is available for sleep periods. However, as shown in [3] this processing overhead can be reduced to acceptable levels. Third, the TinyMOS solution has a larger memory footprint. Adding the TinyMOS solution to the modified *RadioCountToLeds* application increases the code size by 10926 byte, the RAM size by 267 byte. Adding the PL scheduler solution to the modified *RadioCountToLeds* application, increases the code size by 864 byte and the RAM size by 30 byte. Space necessary for stacks is not included here for either solution. The, presented TinyOS modification has a significantly smaller memory requirement than TinyMOS (92% less code size increase, 89% less RAM size increase).

VI. Conclusion

As it is shown in the paper, it is possible to add preemption to the TinyOS system without introducing overheads used in multi-threaded systems. The established event driven processing concepts can be retained while adding preemption through context switching. Established TinyOS programming conventions can be used which ensures that existing application code can be re-used. Preemption features can be integrated seamlessly in existing TinyOS infrastructures.

References

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *ACM SIGOPS Operating Systems Review*, vol. 34, pp. 93–104, December 2000.

[2] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, and R. Han, "MANTIS: System support for multimodal networks of in-situ sensors," in 2^{nd} *ACM International Workshop on Wireless Sensor Networks and Applications*, pp. 50–59, September 2003.

[3] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan, "Improving the Energy Efficiency of the MANTIS Kernel," in *Proceedings of the 4th IEEE European Workshop on Wireless Sensor Networks (EWSN2007), Delft, Netherlands*, Jan. 2007.

[4] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau, "Evolving real-time systems using hierarchical scheduling and concurrency analysis," in 24^{th} *IEEE Internation Real-Time Systems Symposium*, pp. 25–36, December 2003.

[5] E. Trumpler and R. Han., "A systematic framework for evolving TinyOS," in *IEEE Workshop on Embedded Networked Sensors*, pp. 61–65, May 2006.

[6] W. P. McCartney and N. Sridhar, "Abstractions for safe concurrent programming in networked embedded systems," in *Proceedings of the 4th international conference on Embedded networked sensor system*, pp. 167 – 180, October 2006.

[7] M. Rahimi, R. Baer, O. I. Iroezi, J. C. Garcia, J. Warrior, D. Estrin, and M. Srivastava., "Cyclops: In situ image sensing and interpretation in wireless sensor networks," in *In proc. 3^{rd} international conference on Embedded Networked Sensor Systems,*, pp. 192–204, November 2005.