The TinyOS-2.x Priority Level Scheduler

Cormac Duffy

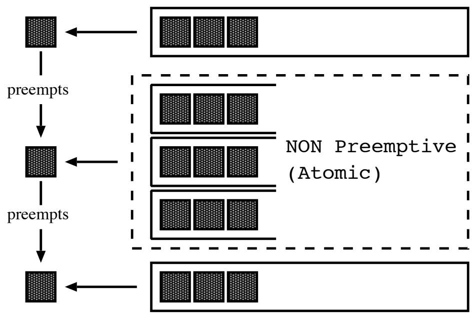

Fig. 1. Queue Structure

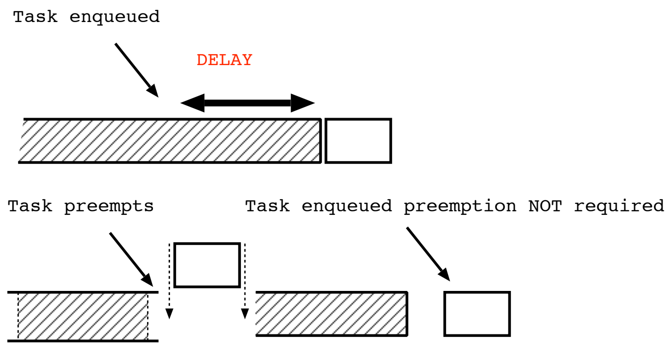

Fig. 2. Task Preemption

I. Overview

The Preempting Priority Based Scheduler, is a preemptive scheduler for TInyOS-2.x. Applications can achieve greater performance control by designating each task to a specific priority. Based on the original TinyOS-2.x scheduler, this scheduler uses up to five basic FIFO task queues. Each Queue stores only the number of tasks required by the TinyOS application. Further more the Priority Scheduler will only allocate the number of task Queues required by your application. The middle three tasks queues, LOW, BASIC *(default TinyOS task),* and HIGH task queues are designed to execute atomically with respect to one another. Thus a task in either of these queues will not preempt another in any one of these queues. The VERYHIGH priority queue is designed to preempt all other queues. If a task of lower priority is processing, than its execution cycle will be interrupted to process the VERYHIGH priority task. The lowest priority, VERYLOW allows programmers to designate unimportant tasks which can be preempted by all other task priorities. Particularly suitable for non critical tasks with long execution cycles.

II. Dynamic Queue Size

The PL scheduler extends from the design of the default FIFO scheduler in TinyOS-2.x. The PL scheduler provides 5 possible priority FIFO tasks queues. Like the default FIFO scheduler, both schedulers rely on the NESC compiler to predetermine the number of application tasks at compiler time. This information is used to determine the queue size. In addition the PL scheduler also uses the nesc compiler to determine the number of Priority queues that are necessary at compile time.

```
NUM_BASIC_TASKS = uniqueCount("TinySchedulerC.TaskBasic"),
```

Using the uniqueCount instruction the nesc compiler counts the number of TaskBasic (generic TinyOS tasks) that are wired to the scheduler. The PL Scheduler also uses this instruction to determine the number of:

- TaskPriority<TASKVERYHIGH>
- TaskPriority<TASKHIGH>
- TaskPriority<TASKLOW>
- TaskPriority<TASKVERYLOW>

Further more the PL Scheduler also uses this static analysis functionality to determine the number of task queues required, and the relevant index of each queue.

```
enum{
VERYLOW=(NUM_VERYLOW_TASKS>0)?0:NO_TASK,
LOW=(NUM_LOW_TASKS>0)?(1-
(VERYLOW==NO_TASK)?1:0):NO_TASK,
BASIC=(NUM_BASIC_TASKS>0)?(2-(((VERYLOW==NO_TASK)?1:0)
+ ((LOW==NO_TASK)?1:0))):NO_TASK,
HIGH=(NUM_HIGH_TASKS>0)?(3- (((VERYLOW==NO_TASK)?1:0) +
((LOW==NO_TASK)?1:0) + ((BASIC==NO_TASK)?1:0))):NO_TASK,
VERYHIGH=(NUM_VERYHIGH_TASKS>0)?(4-
(((VERYLOW==NO_TASK)?1:0)    +    ((HIGH==NO_TASK)?1:0)    +
((LOW==NO_TASK)?1:0) + ((BASIC==NO_TASK)?1:0))):NO_TASK,
NUM_PRIORITIES=(5-(((VERYLOW==NO_TASK)?1:0)                +
((HIGH==NO_TASK)?1:0)       +       ((LOW==NO_TASK)?1:0)       +
((BASIC==NO_TASK)?1:0) + ((VERYHIGH==NO_TASK)?1:0))),
};
```

By knowing at compile time the number of tasks and queues required, the scheduler will require less memory and will also require less instruction cycles, as less can efficiently skip unused priority queues.

III. Preemption

Task preemption play an important role in the responsiveness of the PL Scheduler. Task preemption allows critical tasks of higher priority to interrupt currently processing tasks and immediately begin their execution cycle.

While task preemption facilitates a responsive execution of tasks it does incur a processing overhead. To preempt a running process, the processing state or context must be stored somewhere in memory and a new state must be initialized or restored. [2], [1] provide more detailed analysis of the overheads of preemption for multh-threaded operating systems. For multi-threaded systems such as MANTIS, eCos, or the threaded library in Contiki, each thread is initialised with an empty stack, all registers set to 0. A preemptive thread

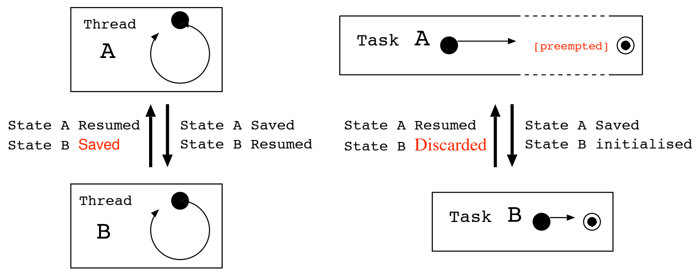

Fig. 3. Task Preemption

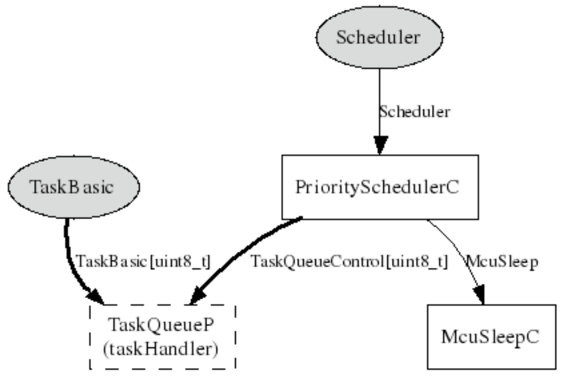

Fig. 4. PL Scheduler components

scheduler will use a *push* function to save all registers to memory, and a *pop* function to load a new set of register values directing the processor program counter to a new thread.

A fundamental difference between multi-threaded systems and event-based systems is that threads are designed to run continuously in parallel, where as tasks are designed to run to completion. As such event-based system can facilitate preemption with a reduced overhead as once a preempting task has finished execution its state can be discarded. Thus preemption in a multi-threaded operating system requires a push pop operation to switch to preempting state and a push pop operation to switch back, this operation is reduced to a push operation pop new context, to switch to the preempting context, followed by a pop operation to restore the original context.

IV. Scheduling algorithm

All task scheduling is handled in two components: The PrioritySchedulerC component and the TaskQueueP component. The TaskQueueP is a generic component that provides FIFO queuing functionality very similar to BasicSchedulerC component in TinyOS-2.x. The TasksQueueP interfaces with each application through the TaskBasic interface, which provides methods for post task. Since there can be up to 5 Priority queues, the TaskQueueP component is instantiated multiple times for each required priority.

The TaskQueueP component interfaces with the PrioritySchedulerC component via a TaskQueueControl interface, which allows the TaskQueueP component to signal to the PrioritySchedulerC component that a new task has been posted. The PrioritySchedulerC component also uses this interface to call the *runNextTask* method in the TaskQueueP component.

The PrioritySchedulerC component controls the

- priority execution of each queue,
- task preemption
- partially responsible for power-management.

Two important variables tasksWaiting & tasksProcessing provide bit masks for the task priorities waiting to be processed and the task priorities currently executing.

```
TaskQueueControl.queueNotify[uint8_t id](){
tasksWaiting|=(1 << PRIORITY); .....
```

It is necessary to know the number of tasks processing at all times to decide if task preemption is necessary, or if simple posting the task will be enough to process the task efficiently. It is more desirable to simple call the task rather than switch contexts as the operation requires less execution overhead.

The handleTask method in the PriorityShedulerC component provides a complex but important switch statement to determining which task to run next. It provides 32 different case statements for each of the different possible combinations of bits in the tasksWaiting bit mask. Essentially this will be compiled to an efficient jump table to quickly execute the appropriate task, without numerous conditions or other processing overheads.

References

[1] Cormac Duffy, Utz Roedig, John Herbert, and Cormac J. Sreenan. An Experimental Comparison of Event Driven and Multi-Threaded Sensor Node Operating Systems. In *Proceedings of the Third IEEE International Workshop on Sensor Networks and Systems for Pervasive Computing (PERSENS2007), White Plains, USA*, March 2007.

[2] Cormac Duffy, Utz Roedig, John Herbert, and Cormac 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*, January 2007.