TOSThreads: Thread-Safe and Non-Invasive Preemption in TinyOS

Kevin Klues, Chieh-Jan Liang, Jeongyeup Paek, Razvan Musaloiu-E,
Philip Levis, Andreas Terzis, and Ramesh Govindan

Published in SenSys'09

Presented by
Jinkyu Koo

PURDUE UNIVERSITY

Event-driven vs. Multi-threaded

- Thread-based execution
  - A process runs concurrently two or more tasks typically by time-division multiplexing
  - One stack per thread
- Event-based execution
  - Flow of the program is determined by events
  - Single stack

PURDUE UNIVERSITY

Inconvenience with event-driven model

- Event-driven programming model of TinyOS provides greater concurrency for motes
  - Memory constraint → single stack
  - Each event typically perform short computation
- Long-running computation may cause a problem in event-driven model
  - We cannot do anything else while processing an event
  - e.g., data compression

PURDUE UNIVERSITY

Goals of TOSThreads

- Combine the ease of a threaded programming model with the efficiency of an event-based kernel
  - Provide application threads
  - However, the application threads do not affect the event-based kernel
    - Thread-safety: Thread preemption does not cause the kernel to fail
    - Non-invasive preemption: Thread priorities are always preserved; kernel operations are top priority.

PURDUE UNIVERSITY

The Challenge of Preemption

- Concurrently running threads need the ability to invoke kernel functions
- Concurrency of kernel invocations must be managed in some way
- Three basic techniques
  - Cooperative threading
  - Kernel Locking
  - Message Passing

Cooperative threading

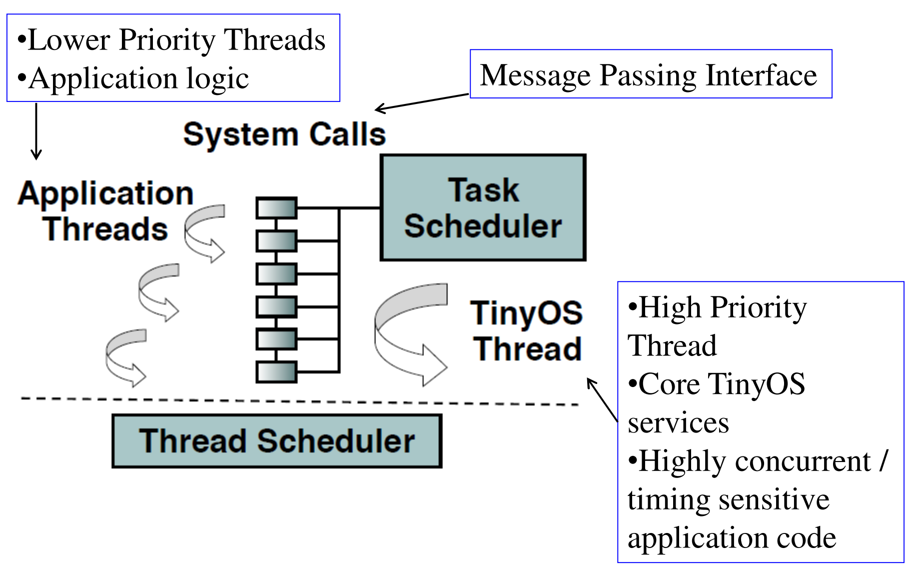
- One thread yields a processor to other threads
  - Avoid challenge of kernel reentrancy → simple kernel
- The correctness of the entire system depends upon application code voluntarily yielding the processor at specific intervals
  - If a thread does not relinquish the processor for a long time, then other threads may be unable to service requests
- Determining the right strategy for inserting yield points in a long running computation is a non-trivial exercise
  - Computations are data dependent, so the commonly-used strategy of placing fixed yield points in the code can result in highly-variable inter-yield intervals

Kernel locking

- Entire kernel has a single lock around it such that only one thread can be executing a system call at any time
- More fine-grained locking is also possible
  - Each subsystem (radio, sensors, flash) can have its own lock
- The kernel is more complex
- The locks reduce performance due to the absence of parallelism

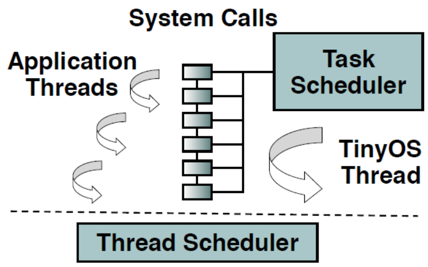
PURDUE UNIVERSITY

TOSThreads architecture



PURDUE UNIVERSITY

TOSThreads architecture



- TinyOS concurrency model: synchronous (tasks) and asynchronous (interrupts)
- Asynchronous code can preempt synchronous code but synchronous code is run-to-completion.
- Application threads exist at the lowest level of the hierarchy and are prohibited from preempting either synchronous code or asynchronous code

- TinyOS runs inside the kernel thread dedicated to running the standard TinyOS task scheduler
- A system call from an application thread posts a task onto the kernel thread.
- Only the kernel thread ever directly executes TinyOS code
  - allow core TinyOS code to execute unchanged

Modifications to TinyOS

- Limited to three small changes
  - Pre-amble in the boot sequence
    - Encapsulates TinyOS inside high priority kernel thread
  - Small change in the TinyOS task scheduler
    - Invokes the thread scheduler when TinyOS thread falls idle
  - Post-ambles in each interrupt handler
    - Ensures TinyOS thread woken up if interrupt handler posts tasks

Boot Sequence

Standard TinyOS Boot

```
command void TinyOS.boot()
{
   /* Initialize the hardware */
   call Hardware_init();

   /* Initialize the software */
   call Software_init();

   /* Signal boot to the application */
   signal Boot.booted();

   /* Spin in the Scheduler */
   call Scheduler.taskLoop();
}
```

TOSThreads TinyOS Boot

```
int main()
{
   /* Encapsulate TinyOS inside a thread */
   call setup_TinyOS_in_kernel_thread();

   /* Boot up TinyOS*/
   call TinyOS.boot();
}
```


Task Scheduler

Standard TinyOS Task Scheduler

```
command void Scheduler.taskLoop() {
  for (;;) {
    uint8_t nextTask;
    atomic {
        while ((nextTask = popTask()) == NO_TASK))
        call McuSleep.sleep();
    }
    signal TaskBasic.runTask[nextTask]();
  }
}
```

TOSThreads TinyOS Task Scheduler

```
command void Scheduler.taskLoop() {
  for (;;) {
    uint8_t nextTask;
    atomic {
        while ((nextTask = popTask()) == NO_TASK)
        call ThreadScheduler.suspendThread(TOS_THREAD_ID);
    }
    signal TaskBasic.runTask[nextTask]();
  }
}
```

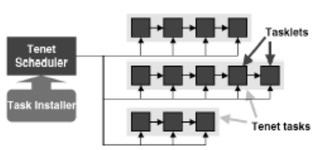
Interrupt Handlers

```
TOSH_SIGNAL(ADC_VECTOR) {
   signal SIGNAL_ADC_VECTOR.fired();
   atomic interruptCurrentThread();
 }
TOSH_SIGNAL(DACDMA_VECTOR) {
   signal SIGNAL_DACDMA_VECTOR.fired();
   atomic interruptCurrentThread();
 }
....
....

void interruptCurrentThread() {
  if (call TaskScheduler.hasTasks() ) {
    call ThreadScheduler.wakeupThread(TOS_THREAD_ID);
    call ThreadScheduler.interruptCurrentThread();
  }
}
```

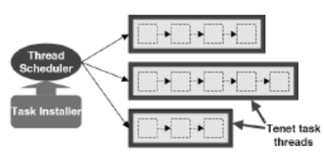

Evaluation (1/3)

- Original



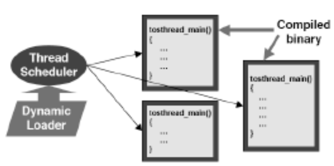
- Tenet Tasks composed of series of static run-to-completion TinyOS tasks

- Tenet-T



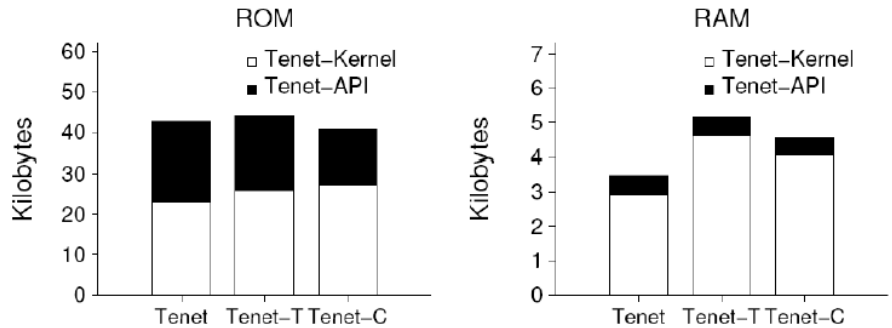
- Tenet Tasks implemented as preemptive threads, composed of static code blocks.

- Tenet-C



- Tenet Tasks implemented as dynamically loadable preemptive threads with arbitrary code blocks

Evaluation (2/3)

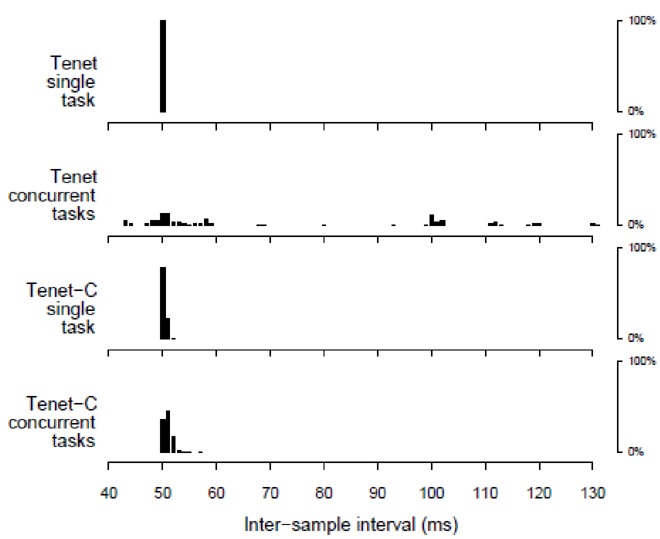


PURDUE UNIVERSITY

Evaluation (3/3)

Two Tenet tasks concurrent

TakeLong: 10ms
TakeSample: every 50ms



PURDUE UNIVERSITY

Q & A