TOSTthreads
Thread-Safe and Non-Invasive Preemption in TinyOS

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

November 5, 2009

SenSys 2009
Events vs. Threads

- **Event-Based Execution**
  - Single thread of control
  - No context switch overheads
  - Less RAM usage (no per thread stacks)
  - Manually managed continuations
  - Good model for highly event driven code

- **Thread-Based Execution**
  - Multiple threads of control
  - Context switch overheads
  - More RAM usage (one stack per thread)
  - System manages continuations automatically
  - Good model for code with many sequential operations
Events vs. Threads

- Event-Based Model

```c
int i = 0;
uint8_t val[3*NUM_ITERS];

void ReadSensors() {
    readTemp();
}
void readTempDone(uint8_t v) {
    val[ i++ ] = v;
    readHumidity();
}
void readHumidityDone(uint8_t v) {
    val[ i++ ] = v;
    readLight();
}
void readLightDone(uint8_t v) {
    val[ i++ ] = v;
    readLight();
    if ( i < NUM_ITERS)
        readTemp();
}
```

- Thread-Based Model
Events vs. Threads

- Event-Based Model

```c
int i = 0;
uint8_t val[3*NUM_ITERS];

void ReadSensors() {
    readTemp();
}

void readTempDone(uint8_t v) {
    val[ i++ ] = v;
    readHumidity();
}

void readHumidityDone(uint8_t v) {
    val[ i++ ] = v;
    readLight();
}

void readLightDone(uint8_t v) {
    val[ i++ ] = v;
    readLight();
    if ( i < NUM_ITERS)
        readTemp();
}
```

- Thread-Based Model
Events vs. Threads

- Event-Based Model

```c
int i = 0;
uint8_t val[3*NUM_ITERS];

void ReadSensors() {
    readTemp();
}
void readTempDone(uint8_t v) {
    val[ i++ ] = v;
    readHumidity();
}
void readHumidityDone(uint8_t v) {
    val[ i++ ] = v;
    readLight();
}
void readLightDone(uint8_t v) {
    val[ i++ ] = v;
    readLight();
    if ( i < NUM_ITERS)
        readTemp();
}
```

- Thread-Based Model
Events vs. Threads

- Event-Based Model

```c
int i = 0;
uint8_t val[3*NUM_ITERS];

void ReadSensors() {
    readTemp();
}
void readTempDone(uint8_t v) {
    val[ i++ ] = v;
    readHumidity();
}
void readHumidityDone(uint8_t v) {
    val[ i++ ] = v;
    readLight();
}
void readLightDone(uint8_t v) {
    val[ i++ ] = v;
    readLight();
    if (i < NUM_ITERS)
        readTemp();
}
```

- Thread-Based Model
Events vs. Threads

- Event-Based Model

```c
int i = 0;
uint8_t val[3*NUM_ITERS];

void ReadSensors() {
    readTemp();
}

void readTempDone(uint8_t v) {
    val[ i++ ] = v;
    readHumidity();
}

void readHumidityDone(uint8_t v) {
    val[ i++ ] = v;
    readLight();
}

void readLightDone(uint8_t v) {
    val[ i++ ] = v;
    readLight();
    if ( i < NUM_ITERS)
        readTemp();
}
```

- Thread-Based Model
Events vs. Threads

- Event-Based Model

```c
int i = 0;
uint8_t val[3*NUM_ITERS];

void ReadSensors() {
    readTemp();
}
void readTempDone(uint8_t v) {
    val[ i++ ] = v;
    readHumidity();
}
void readHumidityDone(uint8_t v) {
    val[ i++ ] = v;
    readLight();
}
void readLightDone(uint8_t v) {
    val[ i++ ] = v;
    readLight();
    if ( i < NUM_ITERS)
        readTemp();
}
```

- Thread-Based Model

```c
uint8_t val[3*NUM_ITERS];

void ReadSensors() {
    for (int i=0; i<NUM_ITERS; i+=3) {
        val[i] = readTemp();
        val[i+1] = readHumidity();
        val[i+2] = readLight();
    }
}
```
Events vs. Threads

- Event-Based Model

```c
int i = 0;
uint8_t val[3*NUM_ITERS];

void ReadSensors() {
    readTemp();
}
void readTempDone(uint8_t v) {
    val[i++] = v;
    readHumidity();
}
void readHumidityDone(uint8_t v) {
    val[i++] = v;
    readLight();
}
void readLightDone(uint8_t v) {
    val[i++] = v;
    readLight();
    if (i < NUM_ITERS)
        readTemp();
}
```

- Thread-Based Model

```c
uint8_t val[3*NUM_ITERS];

void ReadSensors() {
    for (int i=0; i<NUM_ITERS; i+=3) {
        val[i] = readTemp();
        val[i+1] = readHumidity();
        val[i+2] = readLight();
    }
}
```

TOSThreads aims to resolve this tension for TinyOS-based applications.
TOSTThreads Goals

● Thread Safety
  ● Building a thread library is easy – ensuring thread safety is not
  ● Introduces thread-safe preemption through message passing

● Non-Invasiveness
  ● Requires minimal changes to existing TinyOS code
  ● 100% backwards compatible with TinyOS
  ● Minimal overheads (energy, memory footprint, performance)
TOSTThreads Goals

- **Ease of Extensibility**
  - Ability to leverage future innovations in TinyOS
  - TinyOS service wrappers for system calls
- **Flexible Application Development**
  - Easily customizable system call API
  - Mixed use of events and threads
  - Dynamic linking and loading
  - C and nesC based APIs
Outline

- The Challenge of Preemption
- TOSTthreads Architecture
- Interesting Results
- Conclusion
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
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

Contiki (EmNets ’04)
Cooperative Threading

- Advantages:
  - Simple Kernel

- Disadvantages:
  - Complex applications
  - No Preemption

- Avoids challenge of kernel reentrancy
- Kernel only context switches on pre-defined functions (blocking I/O, yields)
- TinyThreads (Sensys ’06)
Kernel Locking

Advantages:
- Simple applications

Disadvantages:
- Limits concurrency
- Complex kernel

- All kernel accesses explicitly locked enabling re-entrancy
- Coarse vs. Fine grained locks
- TinyMOS (EmNets ’06)
Message Passing

Advantages:
- Simple kernel
- Simple applications

Disadvantages:
- Context Switch on every kernel operation

- Applications never invoke kernel code directly
- All kernel accesses through single thin messaging interface
- LiteOS (IPSN ’08)
Outline

- The Challenge of Preemption
- TOSThreads Architecture
- Interesting Results
- Conclusion
Architecture Overview

- **Thread-Based Applications**
  - Lower Priority Threads
  - Application logic

- **System Calls**
  - Message Passing Interface

- **Event-Based Kernel**
  - Single High Priority Thread
  - Core TinyOS services
  - Highly concurrent / timing sensitive application code
Architecture Overview
Architecture Overview

Task Scheduler

TinyOS Thread
Architecture Overview

Task Scheduler

TinyOS Thread

Thread Scheduler
Architecture Overview

Application Threads

Task Scheduler

TinyOS Thread

Thread Scheduler
Architecture Overview

System Calls

Application Threads

Task Scheduler

TinyOS Thread

Thread Scheduler
Blink Example (nesC)

configuration BlinkAppC {
}

implementation {
    components MainC, BlinkC, LedsC;
    components new ThreadC(STACK_SIZE);

    MainC.Boot <- BlinkC;
    BlinkC.Thread -> ThreadC;
    BlinkC.Leds -> LedsC;
}

module BlinkC {
    uses {
        interface Boot;
        interface Thread;
        interface Leds;
    }

    implementation {
        event void Boot.booted() {
            call Thread.start(NULL);
        }

        event void Thread.run(void* arg) {
            for(;;) {
                call Leds.led0Toggle();
                call Thread.sleep(BLINK_PERIOD);
            }
        }
    }
}
Blink Example (nesC)

configuration BlinkAppC {
}
implementation {
    components MainC, BlinkC, LedsC;
    components new ThreadC(STACK_SIZE);
    MainC.Boot <- BlinkC;
    BlinkC.Thread -> ThreadC;
    BlinkC.Leds -> LedsC;
}

module BlinkC {
    uses {
        interface Boot;
        interface Thread;
        interface Leds;
    }
}
implementation {
    event void Boot.booted() {
        call Thread.start(NULL);
    }

    event void Thread.run(void* arg) {
        for(;;) {
            call Leds.led0Toggle();
            call Thread.sleep(BLINK_PERIOD);
        }
    }
}
configuration BlinkAppC {
}
implementation {
    components MainC, BlinkC, LedsC;
    components new ThreadC(STACK_SIZE);
    MainC.Boot <- BlinkC;
    BlinkC.Thread -> ThreadC;
    BlinkC.Leds -> LedsC;
}

module BlinkC {
    uses {
        interface Boot;
        interface Thread;
        interface Leds;
    }
}
implementation {
    event void Boot.booted() {
        call Thread.start(NULL);
    }
    event void Thread.run(void* arg) {
        for(;;) {
            call Leds.led0Toggle();
            call Thread.sleep(BLINK_PERIOD);
        }
    }
}
configuration BlinkAppC {
}

implementation {
components MainC, BlinkC, LedsC;

components new ThreadC(STACK_SIZE);

MainC.Boot <- BlinkC;
BlinkC.Thread -> ThreadC;
BlinkC.Leds -> LedsC;
}

module BlinkC {
uses {
  interface Boot;
  interface Thread;
  interface Leds;
}

implementation {
  event void Boot.booted() {
    call Thread.start(NULL);
  }

  event void Thread.run(void* arg) {
    for(;;) {
      call Leds.led0Toggle();
      call Thread.sleep(BLINK_PERIOD);
    }
  }
}
Blink Example (nesC)

configuration BlinkAppC {
}
implementation {
    components MainC, BlinkC, LedsC;
    components new ThreadC(STACK_SIZE);
    MainC.Boot <- BlinkC;
    BlinkC.Thread -> ThreadC;
    BlinkC.Leds -> LedsC;
}

module BlinkC {
    uses {
    interface Boot;
    interface Thread;
    interface Leds;
    }
    implementation {
    event void Boot.booted() {
        call Thread.start(NULL);
    }
    event void Thread.run(void* arg) {
        for(;;) {
            call Leds.led0Toggle();
            call Thread.sleep(BLINK_PERIOD);
        }
    }
}

Mixed Event / Thread Application Logic
Blink Example (standard C)

```c
#include "tosthread.h"
#include "tosthread_leds.h"

// Initialize variables associated with a thread
tosthread_t blink;
void blink_thread(void* arg);

void tothread_main(void* arg) {
    tothread_create(&blink, blink_thread, NULL, STACK_SIZE);
}
void blink_thread(void* arg) {
    for(;;) {
        led0Toggle();
        tothread_sleep(BLINK_PERIOD);
    }
}
```
Blink Example (standard C)

```c
#include "tosthread.h"
#include "tosthread_leds.h"

//Initialize variables associated with a thread

tosthread_t blink;
void blink_thread(void* arg);

void tothread_main(void* arg) {
    tothread_create(&blink, blink_thread, NULL, STACK_SIZE);
}

void blink_thread(void* arg) {
    for(;;) {
        led0Toggle();
        tothread_sleep(BLINK_PERIOD);
    }
}
```
Blink Example (standard C)

```c
#include "tosthread.h"
#include "tosthread_leds.h"

//Initialize variables associated with a thread
tosthread_t blink;
void blink_thread(void* arg);

void tosthread_main(void* arg) {
    tosthread_create(&blink, blink_thread, NULL, STACK_SIZE);
}

void blink_thread(void* arg) {
    for(;;) {
        led0Toggle();
        tosthread_sleep(BLINK_PERIOD);
    }
}
```
Blink Example (standard C)

```c
#include "tosthread.h"
#include "tosthread_leds.h"

// Initialize variables associated with a thread
tosthread_t blink;
void blink_thread(void* arg);

void tothread_main(void* arg) {
    tothread_create(&blink, blink_thread, NULL, STACK_SIZE);
}
void blink_thread(void* arg) {
    for(;;) {
        led0Toggle();
        tothread_sleep(BLINK_PERIOD);
    }
}
```
Blink Example (standard C)

```c
#include "tosthread.h"
#include "tosthread_leds.h"

//Initialize variables associated with a thread
tosthread_t blink;
void blink_thread(void* arg);

void tothread_main(void* arg) {
    tothread_create(&blink, blink_thread, NULL, STACK_SIZE);
}

void blink_thread(void* arg) {
    for(;;) {
        led0Toggle();
        tothread_sleep(BLINK_PERIOD);
    }
}
```
### Message Passing System Calls

<table>
<thead>
<tr>
<th>Application Threads</th>
<th>Kernel Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Message Passing System Calls

<table>
<thead>
<tr>
<th>Application Threads</th>
<th>Kernel Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send</td>
<td></td>
</tr>
<tr>
<td>Receive</td>
<td></td>
</tr>
<tr>
<td>Sample Sensor</td>
<td></td>
</tr>
</tbody>
</table>
Message Passing System Calls

Application Threads

Send

Receive

Sample Sensor

Kernel Thread

Task Scheduler

Scheduler
Message Passing System Calls

Application Threads

- Send
- Receive
- Sample Sensor

Kernel Thread

Task Scheduler

Thread Scheduler
Message Passing System Calls

Application Threads

- Send
- Receive
- Sample Sensor

Kernel Thread

- Task Scheduler

Thread Scheduler
Message Passing System Calls

Application Threads

- Send
- Receive
- Sample Sensor

Kernel Thread

- Task Scheduler
- Thread Scheduler
Message Passing System Calls

Application Threads
- Send
- Receive
- Sample Sensor

Kernel Thread
- Task Scheduler
- Send

Thread Scheduler
Message Passing System Calls

Application Threads

- Send
- Receive
- Sample Sensor

Kernel Thread

- Task Scheduler
- Send
- Thread Scheduler
Message Passing System Calls

Application Threads

- Send
- Receive
- Sample Sensor

Kernel Thread

- Task Scheduler

Thread Scheduler
Message Passing System Calls

Application Threads

- Send
- Receive
- Sample Sensor

Kernel Thread

- Task Scheduler

Thread Scheduler
Message Passing System Calls

Application Threads

- Send
- Receive
- Sample Sensor

Kernel Thread

- Receive

Thread Scheduler

Task Scheduler
Message Passing System Calls

Application Threads

- Send
- Receive
- Sample Sensor

Kernel Thread

- Task Scheduler
- Receive

Thread Scheduler
Message Passing System Calls

Application Threads

Send

Receive

Sample Sensor

Kernel Thread

Task Scheduler

Thread Scheduler
Message Passing System Calls

Application Threads

Send

Receive

Sample Sensor

Kernel Thread

Task Scheduler

Thread Scheduler
Message Passing System Calls

Application Threads | Kernel Thread
---|---
Send
Receive
Sample Sensor

Thread Scheduler

Task Scheduler
Message Passing System Calls

Application Threads

Send

Receive

Sample Sensor

Kernel Thread

Task Scheduler

Sample Sensor

Thread Scheduler
Message Passing System Calls

Application Threads

Send

Receive

Sample Sensor

Kernel Thread

Task Scheduler

Sample Sensor

Thread Scheduler
Message Passing System Calls

Application Threads

- Send
- Receive
- Sample Sensor

Kernel Thread

- Task Scheduler

Thread Scheduler
Message Passing System Calls

Application Threads

- Send
- Receive
- Sample Sensor

Kernel Thread

- Task Scheduler

Thread Scheduler
Message Passing System Calls

Application Threads
- Send
- Receive
- Sample Sensor

Kernel Thread
- Task Scheduler
- SendDone

Thread Scheduler
Message Passing System Calls

Application Threads

Send
Receive
Sample Sensor

Kernel Thread

Task Scheduler

Thread Scheduler

Receive
Message Passing System Calls

Application Threads

- Send
- Receive
- Sample Sensor

Kernel Thread

Task Scheduler

Thread Scheduler
Message Passing System Calls

Application Threads

- Send
- Receive
- Sample Sensor

Kernel Thread

- Task Scheduler

Thread Scheduler
Message Passing System Calls

Application Threads

Send
Receive
Sample Sensor

Kernel Thread

Task Scheduler
Thread Scheduler
Message Passing System Calls

Application Threads: Receive, Sample Sensor

Kernel Thread: Task Scheduler

Thread Scheduler
Message Passing System Calls

Application Threads

- ... 
- Receive
- Sample Sensor

Kernel Thread

- Task Scheduler

Thread Scheduler
Message Passing System Calls

Application Threads

- ...
- ...
- Sample Sensor

Kernel Thread

- Task Scheduler
- Thread Scheduler
Message Passing System Calls

Application Threads

Kernel Thread

- Task Scheduler
- Sample
- SensorDone

Sample Sensor

Thread Scheduler
Message Passing System Calls

Application Threads

Kernel Thread

Task Scheduler

Thread Scheduler

Sample Sensor

…

…
Message Passing System Calls

Application Threads

Kernel Thread

Thread Scheduler

Task Scheduler

Sample Sensor

...
Message Passing System Calls

Application Threads
- Sample Sensor
- ...
- ...

Kernel Thread
- Task Scheduler
- Thread Scheduler
Message Passing System Calls

Application Threads

| ... |
| ... |
| Sample Sensor |

Kernel Thread

| Task Scheduler |
|               |

Thread Scheduler
Message Passing System Calls

Application Threads

Kernel Thread

Task Scheduler

Thread Scheduler
Message Passing System Calls

Application Threads

...  

Ensures Primary Goal of Thread Safety

Kernel Thread

Task Scheduler

Thread Scheduler
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
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

Ensures Primary Goal of Non-Invasiveness
Boot Sequence

Standard TinyOS Boot

TOSThreads TinyOS Boot

```c
int main()
{
    /* 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();
}
```
Boot Sequence

Standard TinyOS Boot

```c
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();
}
```

TOSTThreads TinyOS Boot

```c
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

```c
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

```c
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();
}

....
....
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();
    }
}

Outline

- The Challenge of Preemption
- TOSThreads Architecture
- Interesting Results
- Conclusion
Microbenchmarks

- Overhead of thread operations
  - Less than 1% on Sense-Store-Forward application

- Linking and loading relatively cheap
  - TinyLD: RAM 100 bytes, ROM 800 bytes
  - 100 ms loading time for sense-store-forward

- Major costs include
  - Extra RAM needed for per thread stacks
  - ROM usage of thread scheduler and API wrappers
Application Comparison

![Bytes Comparison Chart]

- **TinyOS**
- **TOSTThreads using C API**

![Lines of Code Comparison Chart]
Application Comparison

![Graphs showing bytes and lines of code for different applications using TinyOS and TOSTThreads using C API.](image)
Application Comparison

Bytes

<table>
<thead>
<tr>
<th></th>
<th>TinyOS</th>
<th>TOSThreads using C API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Blink</td>
<td>104</td>
<td>110</td>
</tr>
<tr>
<td>Radio Stress</td>
<td>350</td>
<td>298</td>
</tr>
<tr>
<td>Sense SF</td>
<td>974</td>
<td>672</td>
</tr>
<tr>
<td>Base Station</td>
<td>1022</td>
<td>708</td>
</tr>
</tbody>
</table>

Lines of Code

<table>
<thead>
<tr>
<th></th>
<th>TinyOS</th>
<th>TOSThreads using C API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>Blink</td>
<td>48</td>
<td>32</td>
</tr>
<tr>
<td>Radio Stress</td>
<td>114</td>
<td>69</td>
</tr>
<tr>
<td>Sense SF</td>
<td>356</td>
<td>130</td>
</tr>
<tr>
<td>Base Station</td>
<td>229</td>
<td>122</td>
</tr>
</tbody>
</table>
Reimplementation of Tenet

- Reimplementation of Tenet using TOSThreads
  - Original
  - Tenet Tasks composed of series of static run-to-completion TinyOS tasks
Reimplementation of Tenet

- Reimplementation of Tenet using TOSThreads
  - Original
  - Tenet-T

- Tenet Tasks composed of series of static run-to-completion TinyOS tasks
- Tenet Tasks implemented as preemptive threads, composed of static code blocks.
Reimplementation of Tenet

- Reimplementation of Tenet using TOSThreads
  - Original
  - Tenet-T
    - Tenet Tasks composed of series of static run-to-completion TinyOS tasks
  - Tenet-C
    - Tenet Tasks implemented as preemptive threads, composed of static code blocks.
    - Tenet Tasks implemented as dynamically loadable preemptive threads with arbitrary code blocks.
Reimplementation of Tenet

**ROM**
- Tenet-Kernel
- Tenet-API

**RAM**
- Tenet-Kernel
- Tenet-API
Reimplementation of Tenet

**ROM**
- Tenet-Kernel
- Tenet-API

**RAM**
- Tenet-Kernel
- Tenet-API
Reimplementation of Tenet

**ROM**
- Tenet-Kernel
- Tenet-API

**RAM**
- Tenet-Kernel
- Tenet-API
Reimplementation of Tenet

Slight RAM overhead using TOSThreads, but much less constrained programming model
Conclusion

- TOSThreads Goals
  - Thread Safety
  - Non-Invasiveness
  - Ease of Extensibility
  - Flexible Application Development
Questions & Resources

- Details of Dynamic Linking (slightly outdated)
- The Latte Programming Language
- TOSThreads TEP
- Source Code
  Library Code - tinyos-2.x/tos/lib/tosthreads
  Apps - tinyos-2.x/apps/tosthreads