TOSThreads Tutorial

From TinyOS Wiki

This lesson demonstrates how to use the TOSThreads library. You will learn how to do the following:

- Use the nesC API to create and manipulate both static and dynamic threads.
- Use the C API to create and manipulate threads.
- Add TOSThreads support to new system services.

Note: TOSThreads is part of TinyOS 2.1.

Contents

Introduction

TOSThreads is an attempt to combine the ease of a threaded programming model with the efficiency of a fully event-based OS. Unlike earlier threads packages designed for TinyOS, TOSThreads offers the following benefits:

1. It supports fully-preemptive application-level threads.
2. It does not need explicit continuation management, such as state variables between corresponding commands and events.
3. It does not violate TinyOS's concurrency model.
4. It requires minimal changes to the existing TinyOS code base. In addition, adding TOSThreads support to a new platform is a fairly easy process.
5. It offers both nesC and C APIs.

In TOSThreads, TinyOS runs inside a single high-priority kernel thread, while the application logic is implemented in user-level threads. User-level threads execute whenever the TinyOS core becomes idle. This approach is a natural extension to the existing TinyOS concurrency model: adding support for long-running computations while preserving the timing-sensitive nature of TinyOS itself.

In this model, application threads access underlying TinyOS services using a kernel API of blocking system calls. The kernel API defines the set of TinyOS services provided to applications, such as radio, collection (TEP119) (http://www.tinyos.net/tinyos-2.1.0/doc/html/tep119.html) , and so on. Each system call in the API is comprised of a thin blocking wrapper built on top of one of these services. The blocking wrapper is responsible for maintaining states across the non-blocking split-phase operations. TOSThreads allows system developers to re-define the kernel API by appropriately selecting an existing set or implementing their own blocking system call wrappers.

Please refer to TEP134 for more details on the TOSThreads implementation.

The TOSThreads library

At the time of writing, TOSThreads supports the following platforms: telosb, micaZ, and mica2. And, it supports various generic split-phase operations, such as the Read interface and the SplitControl interface, and system services, such as radio, serial, external flash (both Block and Log abstractions), CTP (Collection Tree Protocol), and so on. You can find the code in tos/lib/tosthreads/ (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/) as described below.

TOSThreads system files are located in several subdirectories under tos/lib/tosthreads/ (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/) .

1. **chips**: Some chip-specific files that shadow *tinyos-2.x/tos/chips* to add code such as the interrupt postamble.
2. **csystem**: Contain C API system files and the header file for different system services.
3. **interfaces**: Contain nesC API interfaces.
4. **lib**: Shadow some files in *tinyos-2.x/tos/lib*, and contain the blocking wrapper for CTP.
5. **platforms**: Shadow some files in *tinyos-2.x/tos/platforms*.
6. **sensorboards**: Contain blocking wrappers for telosb's onboard SHT11 sensors, and an universal sensor that generates a sine wave.
7. **system**: Contain nesC API system files and the blocking wrappers for different system services.
8. **types**: Define the structs used by TOSThreads system files.

You can find example TOSThreads applications are in apps/tosthreads/ (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/apps/tosthreads/) .

nesC API

Static threads

We will use *RadioStress* as an example to illustrate manipulating static threads with nesC API. The application creates three threads to stress the radio operations. The nesC version is located in apps/tosthreads/apps/RadioStress/ (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/apps/tosthreads/apps/RadioStress/) . Type **make telosb threads** to compile.

RadioStressAppC.nc:

```
.....
components new ThreadC(300) as RadioStressThread0;
components new BlockingAMSenderC(220) as BlockingAMSender0;
components new BlockingAMReceiverC(220) as BlockingAMReceiver0;

RadioStressC.RadioStressThread0 -> RadioStressThread0;
RadioStressC.BlockingAMSend0 -> BlockingAMSender0;
RadioStressC.BlockingReceive0 -> BlockingAMReceiver0;
.....
```

```
Statically create a thread that has 300-byte stack space

Blocking wrapper for AM Sender (AM ID is 220)

Blocking wrapper for AM Receiver (AM ID is 220)
```

RadioStressC.nc:

```
.....
```

```
event void Boot.booted() {
  call RadioStressThread0.start(NULL);
  call RadioStressThread1.start(NULL);
  call RadioStressThread2.start(NULL);
}
.....
event void RadioStressThread0.run(void* arg) {
  call BlockingAMControl.start();
  for(;;) {
    if(TOS_NODE_ID == 0) {
      call BlockingReceive0.receive(&m0, 5000);
      call Leds.led0Toggle();
    } else {
      call BlockingAMSend0.send(!TOS_NODE_ID, &m0, 0);
      call Leds.led0Toggle();
    }
  }
}
.....
```

Singal the thread scheduler to start executing thread's main function with NULL arguments

RadioStressThread0 thread's main function

Start the radio. The thread will be blocked until the operation completes

Try to listen for an incoming packet for 5000 ms. The thread is blocked until the operation completes

Send a packet *m0* of length 0 byte. The thread is blocked until the operation completes

The *ThreadC* component provides a Thread (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/interfaces/Thread.nc?view=markup) interface for creating and manipulating static threads:

Thread.nc:

```
interface Thread {
  command error_t start(void* arg);
  command error_t stop();
  command error_t pause();
  command error_t resume();
  command error_t sleep(uint32_t milli);
  event void run(void* arg);
  command error_t join();
}
```

Calling start() on a thread signals to the TOSThreads thread scheduler that the thread should begin executing (at some time later, the run() event will be signaled). The argument is a pointer to a data structure passed to the thread once it starts executing. Calls to start() return either SUCCESS or FAIL.

Calling stop() on a thread signals to the TOSThreads thread scheduler that the thread should stop executing. Once a thread is stopped it cannot be restarted. Calls to stop() return SUCCESS if a thread was successfully stopped, and FAIL otherwise. stop() MUST NOT be called from within the thread being stopped; it MUST be called from either the TinyOS thread or another application thread.

Calling pause() on a thread signals to the TOSThreads thread scheduler that the thread should be paused. Unlike a stopped thread, a paused thread can be restarted later by calling resume() on it. pause() MUST ONLY be called from within the thread itself that is being paused.

Calling sleep() puts a thread to sleep for the interval specified in its single 'milli' parameter. sleep() MUST ONLY be called from within the thread itself that is being put to sleep. SUCCESS is returned if the thread was successfully put to sleep, FAIL otherwise.

Dynamic threads

Other than running static threads as the example shows, nesC API can also create dynamic threads at run time. The nesC interface for creating and manipulating dynamic threads is DynamicThread (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/interfaces/DynamicThread.nc?view=markup) :

DynamicThread.nc:

```
interface DynamicThread {
  command error_t create(tosthread_t* t, void (*start_routine)(void*), void* arg, uint16_t stack_size);
  command error_t destroy(tosthread_t* t);
  command error_t pause(tosthread_t* t);
  command error_t resume(tosthread_t* t);
  command error_t sleep(uint32_t milli);
}
```

Blink_DynamicThreads (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/apps/tosthreads/apps/Blink_DynamicThreads/) is an example application that demostrates how to use dynamic threads.

C API

The C version of *RadioStress* is located in apps/tosthreads/capps/RadioStress (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/apps/tosthreads/capps/RadioStress/) . Type **make telosb cthreads** to compile.

RadioStress.c:

```
#include "tosthread.h"                  Header file that defines thread-related functions
#include "tosthread_amradio.h"          Header file that defines radio-related functions
#include "tosthread_leds.h"             Header file that defines LED-related functions
tosthread_t radioStress0;               Declare a thread object
.....
void tosthread_main(void* arg) {        Main thread's main function. This is run after the system
                                        successfully boots
  while( amRadioStart() != SUCCESS      Starts the radio. The main thread will be blocked until the
);                                      operation completes
  tosthread_create(&radioStress0,       Create a thread with 200-byte stack space.
radioStress0_thread, &msg0, 200);       radioStress0_thread is the main function.
.....
}
.....
void radioStress0_thread(void* arg) {   radioStress0 thread's main function
  message_t* m = (message_t*)arg;
  for(;;) {
    if(TOS_NODE_ID == 0) {
      amRadioReceive(m, 2000, 20);      Try to listen for an incoming packet for 2000 ms. The thread
                                        is blocked until the operation completes
```

```
      led0Toggle();
    }
    else {
    if(amRadioSend(!TOS_NODE_ID, Send a packet m0 of length 0 byte, and specify the AM ID to
m, 0, 20) == SUCCESS)            be 20. The thread is blocked until the operation completes
      led0Toggle();
    }
  }
}                                                                                    .....
```

Similarily, tosthread.h (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/csystem/tosthread.h?view=markup) provides commands to manipulate threads.

Support new system services

We will use the log abstraction as an example service to show how to add TOSThreads support to new system services. As mentioned before, TOSThreads overlays a blocking wrapper on top of the split-phase system service.

The first step is to define the interface file provided by the blocking wrapper. The user threaded application uses this interface to make the system call. The interface file for log abstraction is tos/lib/tosthreads/interfaces/BlockingLog.nc (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/interfaces/BlockingLog.nc?view=markup) .

With the interface file, you then can write the blocking wrapper. The blocking wrapper for log abstraction is implemented by tos/lib/tosthreads/system/BlockingLogStorageC.nc (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/system/BlockingLogStorageC.nc?view=markup) , tos/lib/tosthreads/system/BlockingLogStorageP.nc (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/system/BlockingLogStorageP.nc?view=markup) , and tos/lib/tosthreads/system/BlockingLogStorageImplP.nc (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/system/BlockingLogStorageImplP.nc?view=markup) .

BlockingLogStorageImplP.nc:

```
.....
typedef struct read_params {              System call arguments passed from the user thread to the
                                          kernel thread
  void *buf;
  storage_len_t* len;
  error_t error;
} read_params_t;
.....
void readTask(syscall_t *s) {             TinyOS kernel thread executes this function to carry out the
                                          system call
  read_params_t *p = s->params;
  p->error = call LogRead.read[s->id](p-  The split-phase system call
>buf, *(p->len));
  if(p->error != SUCCESS) {
    call SystemCall.finish(s);
  }
}
command error_t
BlockingLog.read[uint8_t volume_id]
```

```
  (void *buf, storage_len_t *len) {
    syscall_t s;                          // Contain a pointer to a structure used when making system calls into a TOSThreads kernel. This structure is readable by both a system call wrapper implementation and the TinyOS kernel thread.
    read_params_t p;
    atomic {
      if(call SystemCallQueue.find(&vol_queue, volume_id) != NULL)
        return EBUSY;
      call SystemCallQueue.enqueue(&vol_queue, &s);
    }
    p.buf = buf;                          // Save the system call argument, buf
    p.len = len;                          // Save the system call argument, len
    call SystemCall.start(&readTask, &s, volume_id, &p);   // Pause the user thread and pass the control to the TinyOS kernel thread
    atomic {
      call SystemCallQueue.remove(&vol_queue, &s);
      return p.error;                     // Return the error code to the user thread
    }
  }
  event void LogRead.readDone[uint8_t volume_id](void *buf, storage_len_t len, error_t error) {
    syscall_t *s = call SystemCallQueue.find(&vol_queue, volume_id);
    read_params_t *p = s->params;
    if (p->buf == buf) {
      p->error = error;                   // Save the error code returned by the system call
      *(p->len) = len;
      call SystemCall.finish(s);
    }
  }
  .....
```

With the nesC blocking wrapper, you can also add C API support. You will need a file that performs redirection from the C call to the nesC call, and the C header file. For the log abstraction, these files are in tinyos-2.x/tos/lib/tosthreads/csystem/ (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/csystem/) .

CLogStorageP.nc:

```
  .....
  error_t volumeLogRead(uint8_t volumeId, void *buf, storage_len_t *len) @C() @spontaneous() {
    return call BlockingLog.read[volumeId](buf, len);
  }
  .....
```

tosthread_logstorage.h:

```
#ifndef TOSTHREAD_LOGSTORAGE_H
#define TOSTHREAD_LOGSTORAGE_H
.....
extern error_t volumeLogRead(uint8_t volumeId, void *buf, storage_len_t *len);
.....
```

Based on what C API header files are included in the user application, TOSThreads includes the appropriate components. This step is done in tos/lib/tosthreads/csystem/TosThreadApiC.nc (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/csystem/TosThreadApiC.nc?view=markup) .

TosThreadApiC.nc:

```
#if defined(TOSTHREAD_BLOCKSTORAGE_H) || defined(TOSTHREAD_DYNAMIC_LOADER)
  components CLogStorageC;
#endif
```

CLogStorageC is included when *tosthread_logstorage.h* or dynamic threads are used.

Synchronization primitives

TOSThreads supports the following synchronization primitives:

1. **Mutex**: The interface file is tos/lib/tosthreads/interfaces/Mutex.nc (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/interfaces/Mutex.nc?view=markup) .
2. **Semaphore**: This is an implementation of counting semaphore. The interface file is tos/lib/tosthreads/interfaces/Semaphore.nc (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/interfaces/Semaphore.nc?view=markup) .
3. **Barrier**: All threads that call Barrier.block() are paused until *v* threads have made the block call. The interface file is tos/lib/tosthreads/interfaces/Barrier.nc (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/interfaces/Barrier.nc?view=markup) .
4. **Condition variable**: The interface file is The interface file is tos/lib/tosthreads/interfaces/ConditionVariable.nc (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/interfaces/ConditionVariable.nc?view=markup) .
5. **Blocking reference counter**: A thread can wait for the maintained counter to reach *count*. The interface provides a way for other threads to increment and decrement the maintained counter. The interface is tos/lib/tosthreads/interfaces/ReferenceCounter.nc (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/lib/tosthreads/interfaces/ReferenceCounter.nc?view=markup) .

Bounce (http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/apps/tosthreads/apps/Bounce) is an example application that demostrates how to use the barrier synchronization primitive.

Related documentation

- TEP 134: The TOSThreads Thread Library (http://www.tinyos.net/tinyos-2.1.0/doc/html/tep134.html)

< **Previous Lesson** | **Top** |

- This page has been accessed 40,366 times.