```c
// This is the lock variable used by all threads. Interface functions
// for it are:
//      void lock_init(unsigned *threadlockptr);        // You write this
//      unsigned lock_acquire(unsigned *threadlockptr); // Shown in class
//      void lock_release(unsigned *threadlockptr);     // You write this
unsigned threadlock;

// A "free" lock is non-zero
void lock_init(unsigned *lock)
{
  *lock = 1;
}

unsigned lock_acquire(unsigned *lock) __attribute__((naked));
unsigned lock_acquire(unsigned *lock)
{
  asm volatile ("\
  MOV     r1, #0       \n\
  LDREX   r2, [r0]     \n\
  CMP     r2, r1       \n\
  ITT     NE           \n\
  STREXNE r2, r1, [r0] \n\
  CMPNE   r2, #1       \n\
  BEQ     1f           \n\
  MOV     R0, #1       \n\
  BX      LR           \n\
1:                     \n\
  CLREX                \n\
  MOV     R0, #0       \n\
  BX      LR           \n\
  ");
}

// Simple implementation....more robust implementation ensures that only the
// thread that is holding the lock is allowed to release it.
void lock_release(unsigned *lock)
{
  lock_init(lock);
}
```

```
Review
======

E. Interruptible threads, with interrupts and global variables used to
   communicate from ISR's to each thread, and between threads

   Advantages:
     * No need to explicitly yield to a scheduler...the scheduler interrupts
       a thread when its "time slice" is up.
     * The code is minimally intertwined and there are only well-defined
       interfaces between threads<-->scheduler and thread<-->thread.
     * If time slices are small enough, the system provides the "appearance
       of concurrency" and has very good latency.

   Disadvantages:
     * Complex: interrupt-driven system is easy to get wrong, debugging is
       difficult.
     * Each thread needs its own stack.
     * The "concurrency problem".

Inter-Thread Communication
==========================

 * Shared memory (AKA global variables) as in model E above

   Advantages:
    - simple (well...needs kernel support on MMU systems), and FAST!

   Disadvantages:
    - The "concurrency problem": consistency issue with separate X,Y,Z data -->
      must use locking which adds complexity. Now you need to share a global
      lock object!
    - for continuous-update communication you have inter-thread dependency
      in having to poll for the data often enough else data is lost
    - each thread author must remember and respect the agreement to use a
      lock object (unless mutually-exclusive shared memory support is
      provided by the kernel)
    - does not scale to multi-processor distributed-memory systems

 * Message buffers: use shared memory to implement a circular buffer

   Advantages:
    - asynchronous delivery (e.g., producer/consumer) --> "send and forget"
    - a single message buffer "class" (or pseudo-class in C) can be
      instantiated multiple times for all necessary message queues to make
      programming easier since locking/unlocking is implemented in the class
      definition

   Disadvantages:
    - leads to "ugly" poll()-yield() loops --> reintroduces FLP programming
      if multiple message buffers are involved for reception. Why? Writing
      to a circular buffer doesn't generate interrupts.
    - does not scale to multi-processor distributed-memory systems

 * Message passing: circular buffers implemented in the kernel

   Advantages:
    - can implement both synchronization and message delivery
    - very-low-interdependency implementation --> only need to know thread
```

```
      identifier of recipient, no need for shared memory
    - even then, can implement "registry thread" to keep track of threads
      by attributes such as their name
    - can easily scale to a multi-processor implementation --> kernel takes
      responsibility for using networking fabric to route messages to
      off-processor threads (e.g., Java RMI, CORBA, SOAP, MPICH)
    - can receive any message from multiple transmitters --> no need for
      poll()-yield() FLP loops. This is the event-loop model of windowing
      systems.
    - kernel can put processes in "suspended" state waiting for messages,
      less overhead than poll()-yield() loops. Thread state needs to have
      support for this.

   Disadvantages:
    - can involve lots of copying of memory from user space to kernel
      space and back

   NOTES:
    - Modern OS'es provide message-passing-type services such as pipes and
      sockets, mostly for inter-process communication though they can also be
      used for inter-thread communication.

Finally
=======

F. Interruptible threads, forced pre-emption, kernel implements message
   passing, kernel wakes up threads waiting on ISR events or waiting
   on messages

   Advantages:
     - There is no need to run a thread AT ALL if all it's going to do
       is poll a global variable then call yield() once it realizes there
       is nothing to do (i.e., no more poll()-yield() loops).
     - Threads "go to sleep" waiting for either an ISR event or thread message,
       thus processor utilization is maximized.
     - User-written code need not worry about concurrency issues in
       inter-thread communication or kernel API.

   Disadvantages:
     - The kernel has to be 100% correct or everything falls apart.
     - Structuring the software system into threads that make sense together
       can take some work. Doing it wrong means a lot of "impedance
       mismatch" in the code and lower efficiency.
     - Context-switch time (interrupt-->save thread 1 state-->restore
       thread 2 state-->return from interrupt) is "wasted" time and happens
       Fc times per second where Fc is the time slice timer frequency.
     - Still very hard to debug (non-deterministic behavior)
```

Why Threads Are A Bad Idea

(for most purposes)

John Ousterhout
Sun Microsystems Laboratories
john.ousterhout@eng.sun.com
http://www.sunlabs.com/~ouster
1996 USENIX Technical Conference
(January 25, 1996)

Introduction

- Threads:
  - Grew up in OS world (processes).
  - Evolved into user-level tool.
  - Proposed as solution for a variety of problems.
  - Every programmer should be a threads programmer?
- Problem: threads are very hard to program.
- Alternative: events.
- Claims:
  - For most purposes proposed for threads, events are better.
  - Threads should be used only when true CPU concurrency is needed.

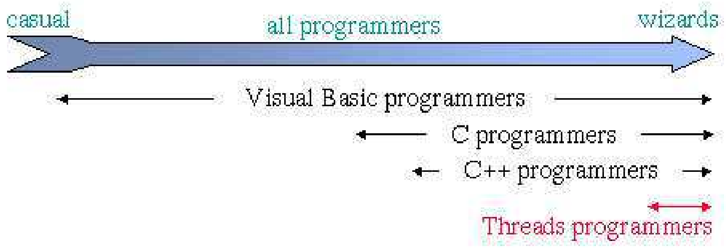
What Are Threads?

- General-purpose solution for managing concurrency.
- Multiple independent execution streams.
- Shared state.
- Pre-emptive scheduling.
- Synchronization (e.g. locks, conditions).

What Are Threads Used For?

- Operating systems: one kernel thread for each user process.
- Scientific applications: one thread per CPU (solve problems more quickly).
- Distributed systems: process requests concurrently (overlap I/Os).
- GUIs:
  - Threads correspond to user actions;
    can service display during long-running computations.
  - Multimedia, animations.

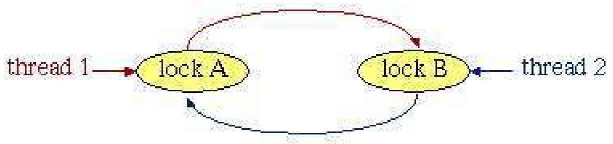
What's Wrong With Threads?



- Too hard for most programmers to use.
- Even for experts, development is painful.

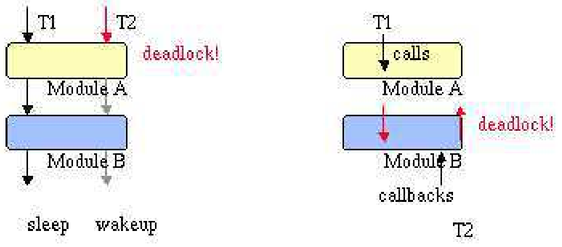
Why Threads Are Hard

- Synchronization:
  - Must coordinate access to shared data with locks.
  - Forget a lock? Corrupted data.
- Deadlock:
  - Circular dependencies among locks.
  - Each process waits for some other process: system hangs.



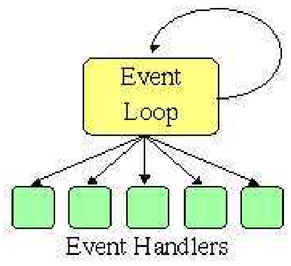
Why Threads Are Hard, cont’d

- Hard to debug: data dependencies, timing dependencies.
- Threads break abstraction: can’t design modules independently.
- Callbacks don’t work with locks.



Why Threads Are Hard, cont’d

- Achieving good performance is hard:
  - Simple locking (e.g. monitors) yields low concurrency.
  - Fine-grain locking increases complexity,
    reduces performance in normal case.
  - OSes limit performance (scheduling, context switches).
- Threads not well supported:
  - Hard to port threaded code (PCs? Macs?).
  - Standard libraries not thread-safe.
  - Kernel calls, window systems not multi-threaded.
  - Few debugging tools (LockLint, debuggers?).
- Often don’t want concurrency anyway (e.g. window events).



Event-Driven Programming

- One execution stream: no CPU concurrency.
- Register interest in events (callbacks).
- Event loop waits for events, invokes handlers.
- No preemption of event handlers.
- Handlers generally short-lived.

What Are Events Used For?

- Mostly GUIs:
  - One handler for each event (press button, invoke menu entry, etc.).
  - Handler implements behavior (undo, delete file, etc.).
- Distributed systems:
  - One handler for each source of input (socket, etc.).
  - Handler processes incoming request, sends response.
  - Event-driven I/O for I/O overlap.

Problems With Events

- Long-running handlers make application non-responsive.
  - Fork off subprocesses for long-running things (e.g. multimedia), use events to find out when done.
  - Break up handlers (e.g. event-driven I/O).
  - Periodically call event loop in handler (reentrancy adds complexity).
- Can’t maintain local state across events (handler must return).
- No CPU concurrency (not suitable for scientific apps).
- Event-driven I/O not always well supported (e.g. poor write buffering).

Events vs. Threads

- Events avoid concurrency as much as possible, threads embrace:
  - Easy to get started with events: no concurrency, no preemption, no synchronization, no deadlock.
  - Use complicated techniques only for unusual cases.
  - With threads, even the simplest application faces the full complexity.
- Debugging easier with events:
  - Timing dependencies only related to events, not to internal scheduling.
  - Problems easier to track down: slow response to button vs. corrupted memory.

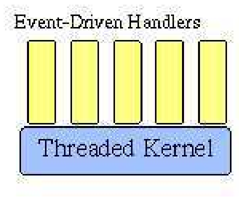
Events vs. Threads, cont’d

- Events faster than threads on single CPU:
  - No locking overheads.
  - No context switching.
- Events more portable than threads.
- Threads provide true concurrency:

    - Can have long-running stateful handlers without freezes.
    - Scalable performance on multiple CPUs.

Should You Abandon Threads?

- No: important for high-end servers (e.g. databases).
- But, avoid threads wherever possible:
    - Use events, not threads, for GUIs, distributed systems, low-end servers.
    - Only use threads where true CPU concurrency is needed.
    - Where threads needed,
      isolate usage in threaded application kernel:
      keep most of code single-threaded.



Conclusions

- Concurrency is fundamentally hard; avoid whenever possible.
- Threads more powerful than events, but power is rarely needed.
- Threads much harder to program than events; for experts only.
- Use events as primary development tool
  (both GUIs and distributed systems).
- Use threads only for performance-critical kernels.

Using Protothreads for Sensor Node Programming

Adam Dunkels
Swedish Institute of Computer Science
adam@sics.se

Oliver Schmidt
oliver@jantzer-schmidt.de

Thiemo Voigt
Swedish Institute of Computer Science
thiemo@sics.se

ABSTRACT

Wireless sensor networks consist of tiny devices that usually have severe resource constraints in terms of energy, processing power and memory. In order to work efficiently within the constrained memory, many operating systems for such devices are based on an event-driven model rather than on multi-threading. While event-driven systems allow for reduced memory usage, they require programs to be developed as explicit state machines. Since implementing programs as explicit state machines is hard, developing, maintaining, and debugging programs for event-driven systems is difficult.

In this paper, we introduce protothreads, a programming abstraction for event-driven sensor network systems. Protothreads simplify implementation of high-level functionality on top of event-driven systems, without significantly increasing the memory requirements. The memory requirement of a protothread is that of an unsigned integer.

1. INTRODUCTION

Wireless sensor networks consist of tiny devices that usually have severe resource constraints in terms of energy, processing power and memory. Most programming environments for wireless sensor network nodes today are based on an event-triggered programming model rather than traditional multi-threading. In TinyOS [7], the event-triggered model was chosen over a multi-threaded model because of the memory overhead of threads. According to Hill et al. [7]:

> "In TinyOS, we have chosen an event model so that high levels of concurrency can be handled in a very small amount of space. A stack-based threaded approach would require that stack space be reserved for each execution context."

While the event-driven model and the threaded model can be shown to be equivalent [9], programs written in the two models typically display differing characteristics [1]. The advantages and disadvantages of the two models are a debated topic [11, 14].

In event-triggered systems, programs are implemented as *event handlers.* Event handlers are invoked in response to external or internal events, and run to completion. An event handler typically is a programming language procedure or function that performs an action, and makes an explicit return to the caller. Because of the run-to-completion semantics, an event-handler cannot execute a *blocking wait.* With run-to-completion semantics, the system can utilize a single, shared stack. This reduces the memory overhead over a multi-threaded system, where memory must be allocated for a stack for each running program.

The run-to-completion semantics of event-triggered systems makes implementing certain high-level operations a complex task. When an operation cannot complete immediately, the operation must be split across multiple invocations of the event handler. Levis et al. [10] refer to this as a split-phase operation. In the words of Levis et al.:

> "This approach is natural for reactive processing and for interfacing with hardware, but complicates sequencing high-level operations, as a logically blocking sequence must be written in a state-machine style."

In this paper, we introduce the notion of using *protothreads* [3, 6] as a method to reduce the complexity of high-level programs in event-triggered sensor node systems. We argue that protothreads can reduce the number of explicit state machines required to implement typical high-level sensor node programs. We believe this reduction leads to programs that are easier to develop, debug, and maintain, based on extensive experience with developing software for the event-driven uIP TCP/IP stack [4] and Contiki operating system [5].

The main contribution of this paper is the protothread programming abstraction. We show that protothreads reduce the complexity of programming sensor nodes. Further, we demonstrate that protothreads can be implemented in the C programming language, using only standard C language constructs and without any architecture-specific machine code.

The rest of this paper is structured as follows. Section 2 presents a motivating example and Section 3 introduces the notion of protothreads. Section 4 discusses related work, and the paper is concluded in Section 5.

2. MOTIVATION

To illustrate how high-level functionality is implemented using state machines, we consider a hypothetical energy-conservation mechanism for wireless sensor nodes. The mechanism switches the radio on and off at regular intervals. The mechanism works as follows:

```
enum {
  ON,
  WAITING,
  OFF
} state;

void radio_wake_eventhandler() {
  switch(state) {

  case OFF:
    if(timer_expired(&timer)) {
      radio_on();
      state = ON;
      timer_set(&timer, T_AWAKE);
    }
    break;

  case ON:
    if(timer_expired(&timer)) {
      timer_set(&timer, T_SLEEP);
      if(!communication_complete()) {
        state = WAITING;
      } else {
        radio_off();
        state = OFF;
      }
    }
    break;

  case WAITING:
    if(communication_complete()
       || timer_expired(&timer)) {
      state = ON;
      timer_set(&timer, T_AWAKE);
    } else {
      radio_off();
      state = OFF;
    }
    break;
  }
}
```

Figure 1: The radio sleep cycle implemented with events.

1. Turn radio on.
2. Wait for t_{awake} milliseconds.
3. Turn radio off, but only if all communication has completed.
4. If communication has not completed, wait until it has completed. Then turn off the radio.
5. Wait for t_{sleep} milliseconds. If the radio could not be turned off before t_{sleep} milliseconds because of remaining communication, do not turn the radio off at all.
6. Repeat from step 1.

To implement this protocol in an event-driven model, we first need to identify a set of states around which the state machine can be designed. For this protocol, we can see three states: *on* – the radio is turned on, *waiting* – waiting for any remaining communication to complete, and *off* – the radio is off. Figure 3 shows the resulting state machine, including the state transitions.

```
PT_THREAD(radio_wake_thread(struct pt *pt)) {
  PT_BEGIN(pt);

  while(1) {
    radio_on();
    timer_set(&timer, T_AWAKE);
    PT_WAIT_UNTIL(pt, timer_expired(&timer));

    timer_set(&timer, T_SLEEP);
    if(!communication_complete()) {
      PT_WAIT_UNTIL(pt, communication_complete()
                        || timer_expired(&timer));
    }

    if(!timer_expired(&timer)) {
      radio_off();
      PT_WAIT_UNTIL(pt, timer_expired(&timer));
    }
  }

  PT_END(pt);
}
```

Figure 2: The radio sleep cycle implemented with protothreads.

To implement this state machine in C, we use an explicit state variable, `state`, that can take on the values OFF, ON, and WAITING. We use a C switch statement to perform different actions depending on the `state` variable. The code is placed in an event handler function that is called whenever an event occurs. Possible events in this case are that a timer expires and that communication completes. The resulting C code is shown in Figure 1.

We note that this simple mechanism results in a fairly large amount of C code. The structure of the mechanism, as it is described by the six steps above, is not immediately evident from the C code.

3. PROTOTHREADS

Protothreads [6] are an extremely lightweight stackless type of threads, designed for severely memory constrained systems. Protothreads provide *conditional blocking* on top of an event-driven system, without the overhead of per-thread stacks.

We developed protothreads in order to deal with the com-

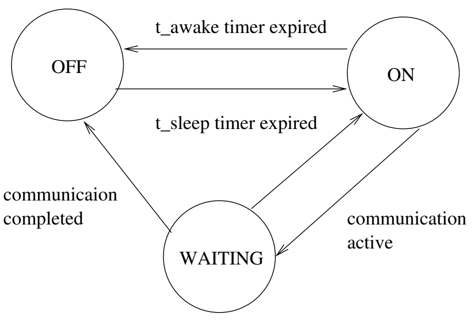


Figure 3: State machine realization of the radio sleep cycle protocol.

plexity of explicit state machines in the event-driven uIP TCP/IP stack [4]. For uIP, we were able to substantially reduce the number of state machines and explicit states used in the implementations of a number of application level communication protocols. For example, the uIP FTP client could be simplified by completely removing the explicit state machine, and thereby reducing the number of explicit states from 20 to one.

3.1 Protothreads versus events

Programs written for an event-driven model typically have to be implemented as explicit state machines. In contrast, with protothreads programs can be written in a sequential fashion without having to design explicit state machines. To illustrate this, we return to the radio sleep cycle example from the previous section.

Figure 2 shows how the radio sleep cycle mechanism is implemented with protothreads. Comparing Figure 2 and Figure 1, we see that the protothreads-based implementation not only is shorter, but also more closely follows the specification of the radio sleep mechanism. Due to the linear code flow of this implementation, the overall logic of the sleep cycle mechanism is visible in the C code. Also, in the protothreads-based implementation we are able to make use of regular C control flow mechanisms such as while loops and if statements.

3.2 Protothreads versus threads

The main advantage of protothreads over traditional threads is that protothreads are very lightweight: a protothread does not require its own stack. Rather, all protothreads run on the same stack and context switching is done by stack rewinding. This is advantageous in memory constrained systems, where a stack for a thread might use a large part of the available memory. In comparison, the memory requirements of a protothread that of an unsigned integer. No additional stack is needed for the protothread.

Unlike a thread, a protothread runs only within a single C function and cannot span over other functions. A protothread may call normal C functions, but cannot block inside a called function. Blocking inside nested function calls is instead implemented by spawning a separate protothread for each potentially blocking function. Unlike threads, protothreads makes blocking explicit: the programmer knows exactly which functions that potentially may yield.

3.3 Comparison

Feature	Events	Threads	**Proto-threads**
Control structures	No	Yes	**Yes**
Debug stack retained	No	Yes	**Yes**
Implicit locking	Yes	No	**Yes**
Preemption	No	Yes	**No**
Automatic variables	No	Yes	**No**

Table 1: Qualitative comparison between events, threads and protothreads

Table 1 summarizes the features of protothreads and compares them with the features of events and threads.

```
void radio_wake_thread(struct pt *pt) {
  switch(pt->lc) {
  case 0:

  while(1) {
    radio_on();
    timer_set(&timer, T_AWAKE);

    pt->lc = 8;
  case 8:
    if(!timer_expired(&timer)) {
      return;
    }

    timer_set(&timer, T_SLEEP);
    if(!communication_complete()) {

      pt->lc = 13;
  case 13:
      if(!(communication_complete() ||
           timer_expired(&timer))) {
        return;
      }

    }

    if(!timer_expired(&timer)) {
      radio_off();

      pt->lc = 18;
  case 18:
      if(!timer_expired(&timer)) {
        return;
      }

    }
  }
  }
}
```

Figure 4: C switch statement expansion of the protothreads code in Figure 2

Control structures. One of the advantages of threads over events is that threads allow programs to make full use of the control structures (e.g., *if* conditionals and *while* loops) provided by the programming language. In the event-driven model, control structures must be broken down into two or more pieces in order to implement continuations [1]. In contrast, both threads and protothreads allow blocking statements to be used together with control structures.

Debug stack retained. Because the manual stack management and the free flow of control in the event-driven model, debugging is difficult as the sequence of calls is not saved on the stack [1]. With both threads and protothreads, the full call stack is available for debugging.

Implicit locking. With manual stack management, as in the event-driven model, all yield points are immediately visible in the code. This makes it evident to the programmer whether or not a structure needs to be locked. In the threaded model, it is not as evident that a particular function call yields. Using protothreads, however, potentially blocking statements are explicitly implemented with a `PT_WAIT` statement. Program code between such statements never yields.

Preemption. The semantics of the threaded model allows for preemption of a running thread: the thread's stack is saved, and execution of another thread can be continued. Because both the event-driven model and protothreads use a single stack, preemption is not possible within either of these models.

Automatic variables. Since the threaded model allocates a stack for each thread, automatic variables—variables with function local scope automatically allocated on the stack—are retained even when the thread blocks. Both the event-driven model and protothreads use a single shared stack for all active programs, and rewind the stack every time a program blocks. Therefore, with protothreads, automatic variables are not saved across a blocking wait. This is discussed in more detail below.

3.4 Limitations

While protothreads allow programs to take advantage of a number of benefits of the threaded programming model, protothreads also impose some of the limitations from the event-driven model. The most evident limitation from the event-driven model is that automatic variables—variables with function-local scope that are automatically allocated on the stack—are not saved across a blocking wait. While automatic variables can still be used inside a protothread, the contents of the variables must be explicitly saved before executing a wait statement. The reason for this is that protothreads rewind the stack at every blocking statement, and therefore potentially destroy the contents of variables on the stack.

Many optimizing C compilers, including gcc, are able to detect if an automatic variable is unsafely used after a blocking statement. Typically a warning is produced, stating that the variable in question "might be used uninitialized in this function". While it may not be immediately apparent for the programmer that this warning is related to the use of automatic variables across a blocking protothreads statement, it does provide an indication that there is a problem with the program. Also, the warning indicates the line number of the problem which assists the programmer in identifying the problem.

The limitation on the use of automatic variables can be handled by using an explicit *state object*, much in the same way as is done in the event-driven model. The state object is a chunk of memory that holds the contents of all automatic variables that need to be saved across a blocking statement. It is, however, the responsibility of the programmer to allocate and maintain such a state object.

It should also be noted that protothreads do not limit the use of *static local* variables. Static local variables are variables that are local in scope but allocated in the data section. Since these are not placed on the stack, they are not affected by the use of blocking protothreads statements. For functions that do not need to be re-entrant, using static local variables instead of automatic variables can be an acceptable solution to the problem.

3.5 Implementation

Protothreads are based on a low-level mechanism that we call *local continuations* [6]. A local continuation is similar to ordinary continuations [12], but does not capture the program stack. Local continuations can be implemented in a variety of ways, including using architecture specific machine code, C-compiler extensions, and a non-obvious use of the C *switch* statement. In this paper, we concentrate on the method based on the C switch statement.

A local continuation supports two operations; it can be either *set* or *resumed*. When a local continuation is set, the state of the function—all CPU registers including the program counter but excluding the stack—is captured. When the same local continuation is resumed, the state of the function is reset to what it was when the local continuation was set.

A protothread consists of a C function and a single local continuation. The protothread's local continuation is *set* before each conditional blocking wait. If the condition is true and the wait is to be performed, the protothread executes an explicit return statement, thus returning to the caller. The next time the protothread is invoked, the protothread *resumes* the local continuation that was previously set. This will effectively cause the program to jump to the conditional blocking wait statement. The condition is reevaluated and, once the condition is false, the protothread continues to execute the function.

```
#define RESUME(lc) switch(lc) { case 0:

#define SET(lc)    lc = __LINE__; case __LINE__:
```

Figure 5: The local continuation *resume* and *set* operations implemented using the C switch statement.

Local continuations can be implemented using standard C language constructs and a non-obvious use of the C switch statement. With this technique, the local continuation is represented by an unsigned integer. The resume operation is implemented as an open switch statement, and the set operation is implemented as an assignment of the local continuation and a case statement, as shown in Figure 5. Each set operation sets the local continuation to a value that is unique within each function, and the resume operation's switch statement jumps to the corresponding case statement. The case 0: statement in the implementation of the resume operation ensures that the resume statement does nothing if is the local continuation is zero.

Figure 4 shows the example radio sleep cycle mechanism from Section 2 with the protothreads statements expanded using the C switch implementation of local continuations. We see how each `PT_WAIT_UNTIL` statement has been replaced with a case statement, and how the `PT_BEGIN` statement has been replaced with a switch statement. Finally, the `PT_END` statement has been replaced with a single right curly bracket, which closes the switch block that was opened by the `PT_BEGIN` statement. We also note the similarity between Figure 4 and the event-based implementation in Figure 1. While the resulting C code is very similar in the two cases, the process of arriving at the code is different. With the event-driven model, the programmer must explicitly design and implement a state machine. With protothreads, the state machine is automatically generated.

The non-obviousness of the C switch implementation of local continuations is that the technique appears to cause problems when a conditional blocking statement is used inside a nested C control statement. For example, the case 13: statement in Figure 4 appears inside an if block, while the corresponding switch statement is located at a higher block. However, this is a valid use of the C switch statement: case statements may be located anywhere inside a switch block. They do not need to be in the same level of nesting, but can be located anywhere, even inside nested if or for blocks. This use of the switch statement is likely to first have been publicly described by Duff [2]. The same technique has later been used by Tatham to implement coroutines in C [13].

The implementation of protothreads using the C switch statements imposes a restriction on programs using protothreads: programs cannot utilize switch statements together with protothreads. If a switch statement is used by the program using protothreads, the C compiler will is some cases emit an error, but in most cases the error is not detected by the compiler. This is troublesome as it may lead to unexpected run-time behavior which is hard to trace back to an erroneous mixture of one particular implementation of protothreads and switch statements. We have not yet found a suitable solution for this problem.

4. RELATED WORK

Kasten and Römer [8] have also identified the need for new abstractions for managing the complexity of event-triggered programming. They introduce OSM, a state machine programming model based on Harel's StateCharts. The model reduces both the complexity of the implementations and the memory usage. Their work is different from protothreads in that OSM requires support from an external OSM compiler to produce the resulting C code, whereas protothreads only make use of the regular C preprocessor.

5. CONCLUSIONS

Many operating systems for wireless sensor network nodes are based on an event-triggered programming model. In order to implement high-level operations under this model, programs have to be written as explicit state machines. Software implemented using explicit state machines is often hard to understand, debug, and maintain.

We have presented *protothreads* as a programming abstraction that reduces the complexity of implementations of high-level functionality for event-triggered systems. With protothreads, programs can perform *conditional blocking* on top of event-triggered systems with run-to-completion semantics, without the overhead of full multi-threading.

Acknowledgments

This work was partly financed by VINNOVA, the Swedish Agency for Innovation Systems, and the European Commission under contract IST-004536-RUNES.

6. REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the USENIX Annual Technical Conference*, pages 289–302, 2002.

[2] T. Duff. Re: Explanation please! Usenet news article, Message-ID: <8144@alice.UUCP>, August 1988.

[3] A. Dunkels. Protothreads web site. Web page. Visited 2005-03-18. http://www.sics.se/~adam/pt/

[4] A. Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, May 2003.

[5] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004.

[6] A. Dunkels and O. Schmidt. Protothreads – Lightweight Stackless Threads in C. Technical Report T2005:05, Swedish Institute of Computer Science.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[8] O. Kasten and K. Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *The Fourth International Conference on Information Processing in Sensor Networks (IPSN)*, Los Angeles, USA, April 2005.

[9] H. C. Lauer and R. M. Needham. On the duality of operating systems structures. In *Proc. Second International Symposium on Operating Systems*, October 1978.

[10] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proc. NSDI'04*, March 2004.

[11] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Invited Talk at the 1996 USENIX Technical Conference, 1996.

[12] J. C. Reynolds. The discoveries of continuations. *Lisp Symbol. Comput.*, 6(3):233–247, 1993.

[13] S. Tatham. Coroutines in C. Web page, 2000. http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html

[14] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, May 2003.