Protothreads: Simplifying Event Driven Programming of Memory-Constrained Embedded Systems

Adam Dunkels *, Oliver Schimidt, Thiemo Voigt *, Muneeb Ali **

* Swedish Institute of Computer Science
** TU Delft

SenSys 2006

Presented by:
Rajesh Krishna Panta
Dependable Computing Systems Lab (DCSL), Purdue University
**Introduction: Event-driven vs Multithread?**

- Many operating systems for sensor networks (TinyOS, Contiki, SOS) use event-driven programming model instead of traditional multithreaded approach
- Event-driven systems support high level of concurrency with little RAM
  - Single stack
  - Sensor nodes have limited RAM (few kilobytes)
- **Multithreading is difficult due to limited RAM**
  - Per-thread stack (Default stack size for each stack in MANTIS is 128 bytes)
  - Expensive context switching, thread-scheduling, synchronization, reentrancy etc.
  - Can limit concurrency
Protothreads: Motivation

• An event-driven model does not support blocking wait abstraction

• Programming is difficult – a logically blocking sequence must be written in a state machine style
  – Split phase operation in TinyOS

• Thus many practical event-based programs are difficult to understand

• Protothreads was originally developed for managing the complexity of state machines in the event-driven uIP embedded TCP/IP stack
Protothreads: Goals

• Simplify programming by reducing or eliminating state machine management
• Negligible RAM overhead
• Not intended to replace event-driven systems
  – rather be able to use protothreads on top of event-driven system if state management becomes difficult
Protothreads

- Protothreads provide conditional blocking abstractions to simplify programming for memory-constrained embedded devices
  - The blocking wait semantics allow linear sequencing of statements in event-driven systems
- Protothreads are stackless—all protothreads in a system run on the same stack, which is rewound every time a protothread blocks.
- A protothread is invoked by repeated calls to the function in which it runs
Example: Hypothetical MAC

1. Turn radio on.
2. Wait until \( t = t_0 + t_{awake} \).
3. If communication has not completed, wait until it has completed or \( t = t_0 + t_{awake} + t_{wait\_max} \).
4. Turn the radio off. Wait until \( t = t_0 + t_{awake} + t_{sleep} \).
5. Repeat from step 1.

Problem: With events, we cannot implement this as a five-step program!
Event-driven state machine implementation is messy

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

void eventhandler() {
    if(state == ON) {
        if(expired(timer)) {
            timer = t_sleep;
            if(!comm_complete()) {
                state = WAITING;
                wait_timer = t_wait_max;
            } else {
                radio_off();
                state = OFF;
            }
        } else if(state == WAITING) {
            if(comm_complete() || expired(wait_timer)) {
                state = OFF;
                radio_off();
            }
        } else if(state == OFF) {
            if(expired(timer)) {
                radio_on();
                state = ON;
                timer = t_awake;
            }
        }
    }
}
```

Diagram showing the state transitions:

- Radio on
- Radio off
- Communication left...

Key states:
- ON
- WAITING
- OFF

Key events:
- Timer expires
- Communication completes, timer expires
- Remaining communication
- Radio off
Protothreads-based implementation is easier

```
int protothread(struct pt *pt) {
    PT_BEGIN(pt);
    while(1) {
        radio_on();
        timer = t_awake;
        PT_WAIT_UNTIL(pt, expired(timer));
        timer = t_sleep;
        if(!comm_complete()) {
            wait_timer = t_wait_max;
            PT_WAIT_UNTIL(pt, comm_complete()
                          || expired(wait_timer));
        }
        radio_off();
        PT_WAIT_UNTIL(pt, expired(timer));
    }
    PT_END(pt);
}
```

- Code shorter than the event-driven version
- Mechanism evident from the code
Protothread statements

```c
int a_protothread(struct pt *pt) {
    PT_BEGIN(pt);
    /* ... */
    PT_WAIT_UNTIL(pt, condition1);
    /* ... */
    if(something) {
        /* ... */
        PT_WAIT_UNTIL(pt, condition2);
        /* ... */
    }
    /* ... */
    PT_END(pt);
}
```
Protothread scheduling

• The protothreads mechanism does not specify any specific method to invoke or schedule a protothread

• If a protothread is run on top of an underlying event-driven system, the protothread is scheduled whenever the event handler containing the protothread is invoked by the event scheduler
Prototype Implementation

- Proof-of-concept implementation in pure ANSI C
  - No changes to compiler
  - No architecture specific machine code
- Very simple implementation
- Very low memory overhead
  - Two bytes of RAM per protothread
  - No per-thread stacks
Example Implementation

Local continuations implemented with GCC labels-as-values C extension

```
typedef void * lc_t;
#define LC_INIT(c) c = NULL
#define LC_RESUME(c) if(c) goto *c
#define LC_SET(c) { __label__ r; r: c = &r; }
#define LC_END()

struct pt { lc_t lc; }
#define PT_WAITING 0
#define PT_EXITED 1
#define PT_ENDED 2
#define PT_INIT(pt) LC_INIT(pt->lc)
#define PT_BEGIN(pt) LC_RESUME(pt->lc)
#define PT_END(pt) LC_END(pt->lc); return PT_ENDED
#define PT_WAIT_UNTIL(pt, c) LC_SET(pt->lc); if(!((c)) return PT_WAITING
#define PT_EXIT(pt)
return PT_EXITED
```

Local continuations implemented with C switch statement

```
typedef unsigned short lc_t;
#define LC_INIT(c) c = 0
#define LC_RESUME(c) switch(c) { case 0:
#define LC_SET(c) c = __LINE__; case __LINE__:
#define LC_END(c) }

int sender(pt) {
PT_BEGIN(pt);
/* ... */
do {
PT_WAIT_UNTIL(pt, cond1);
} while(cond);
/* ... */
PT_END(pt);
}
```

Expanded C code with local continuations implemented with the C switch statement

```
int sender(pt) {
switch(pt->lc) {
case 0:
/* ... */
do {
 pt->lc = 8;
case 8:
 if(!cond1)
 return PT_WAITING;
} while(cond);
/* ... */
}
return PT_ENDED;
```
Memory overhead

- The prototype implementation needs to store local continuation for each protothread
  - 2 bytes in MSP430 and 3 bytes in AVR
- No per-thread stack
Limitations of the prototype implementation

• Automatic variables are not saved across a blocking wait
  – User needs to save them explicitly before executing a wait statement
  – For functions that do not need to be reentrant, static local variables can be used instead of automatic variables

• C switch based implementation limits the use of the C switch statement together with protothreads statements

• A protothread cannot span across functions
Evaluation

• Authors rewrote seven event-driven state machine-based applications using protothreads

• Evaluation metrics
  – Reduction in code complexity
    • Number of explicit states
    • Number of explicit state transitions
    • LOC
  – Code footprint
  – Execution time
## Reduction of complexity

<table>
<thead>
<tr>
<th></th>
<th>States before</th>
<th>States after</th>
<th>Transitions before</th>
<th>Transitions after</th>
<th>Reduction in lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>XNP</td>
<td>25</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>32%</td>
</tr>
<tr>
<td>TinyDB</td>
<td>23</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>24%</td>
</tr>
<tr>
<td>Mantis CC1000 driver</td>
<td>15</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>23%</td>
</tr>
<tr>
<td>SOS CC1000 driver</td>
<td>26</td>
<td>9</td>
<td>32</td>
<td>14</td>
<td>16%</td>
</tr>
<tr>
<td>Contiki TR1001 driver</td>
<td>12</td>
<td>3</td>
<td>22</td>
<td>3</td>
<td>49%</td>
</tr>
<tr>
<td>uIP SMTP client</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>45%</td>
</tr>
<tr>
<td>Contiki codeprop</td>
<td>6</td>
<td>4</td>
<td>11</td>
<td>3</td>
<td>29%</td>
</tr>
</tbody>
</table>

Protothreads completely eliminate or significantly reduce the state machine management problem. The source code is also significantly shortened.

Found state machine-related bugs in the Contiki TR1001 driver and the Contiki codeprop code when rewriting with protothreads.
For these applications, code footprint increases by 200 bytes on average. The increase/decrease is dependent on the nature of the application. No conclusion can be drawn about the code footprint based on these examples.
Protothreads incur very small machine code instruction overhead

<table>
<thead>
<tr>
<th></th>
<th>State machine</th>
<th>Proto-thread</th>
<th>Yielding protothread</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSP430</td>
<td>9</td>
<td>12</td>
<td>17</td>
</tr>
<tr>
<td>AVR</td>
<td>23</td>
<td>34</td>
<td>45</td>
</tr>
</tbody>
</table>
### Execution time overhead

<table>
<thead>
<tr>
<th></th>
<th>State machine</th>
<th>Protothreads, switch statement</th>
<th>Protothreads, computed gotos</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc -Os</td>
<td>92</td>
<td>107</td>
<td>97</td>
</tr>
<tr>
<td>gcc -O1</td>
<td>91</td>
<td>103</td>
<td>94</td>
</tr>
</tbody>
</table>

Contiki TR1001 radio driver average execution time (CPU cycles)

Execution time overhead of protothreads is very low
Conclusions

• Protothreads can reduce the complexity of event-driven programs by removing flow-control state machines
  – ~33% reduction in lines of code
• Memory requirements very low
  – Two bytes of RAM per protothread, no stacks
• Seems to be a slight code footprint increase (~ 200 bytes)
• Performance hit is small (~ 10 cycles)