Protothreads –
Simplifying Programming of Memory-Constrained Embedded Systems

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

* Swedish Institute of Computer Science
** TU Delft

ACM SenSys 2006

What this talk is about

- Memory-constrained networked embedded systems
  - 2k RAM, 60k ROM; 10k RAM, 48K ROM
    - “Artificial” limitations – based on economy, not mother nature
    - More memory = higher per-unit cost
- Concurrent programming
  - Multithreading – requires “lots” of memory for stacks
    - 100 bytes is ~5% of 2k!
  - Event-driven – less memory

Why use the event-driven model?

"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."

J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. **System architecture directions for networked sensors**. [ASPLOS 2000]

Problems with the event-driven model?

"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."

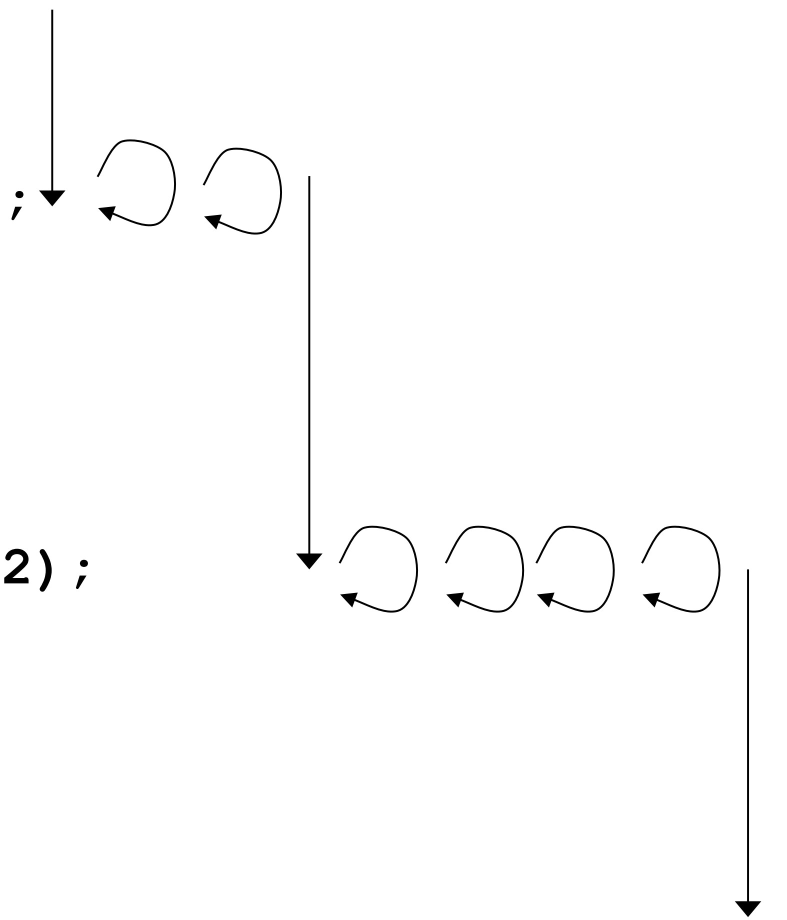
P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. **The Emergence of Networking Abstractions and Techniques in TinyOS**. [NSDI 2004]

Enter protothreads

- Protothreads – a new programming abstraction
  - For memory-constrained embedded systems
  - A design point between events and threads
  - Very simple, yet powerful idea
- Programming primitive: conditional blocking wait
  - PT_WAIT_UNTIL(*condition*)
  - Sequential flow of control
    - Programming language helps us: **if** and **while**
- Protothreads run on a single stack, like the event-driven model
  - Memory requirements (almost) same as for event-driven

An example protothread

```
int a_protothread(struct pt *pt) {
  PT_BEGIN(pt);

  /* … */

  PT_WAIT_UNTIL(pt, condition1);

  /* … */

  if(something) {

    /* … */

    PT_WAIT_UNTIL(pt, condition2);

    /* … */

  }

  PT_END(pt);
}
```



Implementation

- Proof-of-concept implementation in pure ANSI C
  - No changes to compiler
  - No special preprocessor
  - No assembly language
  - Two deviations: automatic variables not saved across a blocked wait, restrictions on switch() statements
- Six-line implementation will be shown on slide!
- Very low memory overhead
  - Two bytes of RAM per protothread
  - No per-thread stacks

SWEDISH INSTITUTE OF COMPUTER SCIENCE SICS

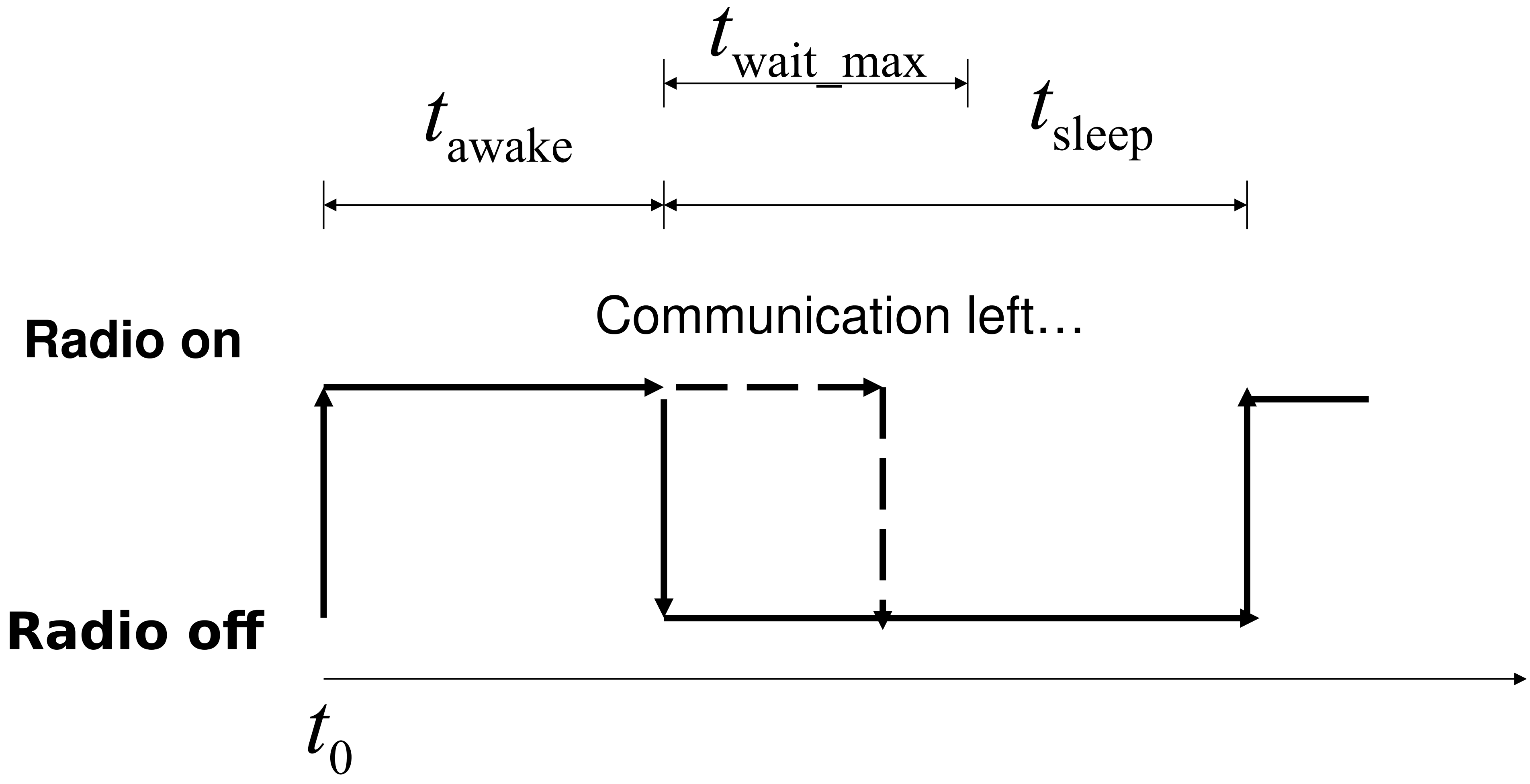
Evaluation, conclusions

- We can replace explicit state machines with protothreads
  - Protothreads let programs use if and while statements instead of state machines
  - 16% - 49% reduction in lines of code for rewritten programs
  - Most explicit state machines completely replaced
  - Code size increase/decrease depends on program
- Run-time overhead small (3-15 cycles)
  - Useful even in time-critical code; interrupt handlers
- Protothreads have been adopted, used by others

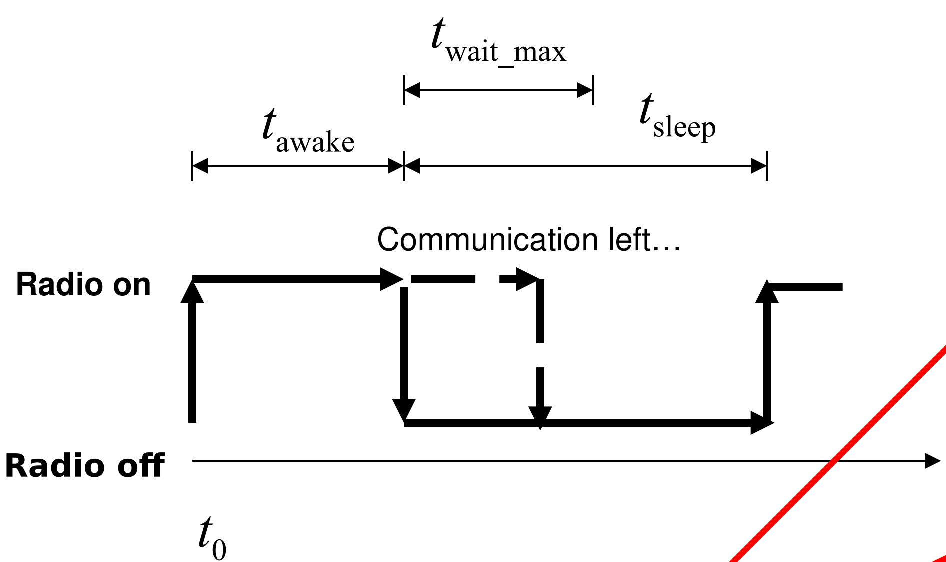
The details…

Example:
A hypothetical sensor network MAC protocol

Radio sleep cycle



Five-step specification



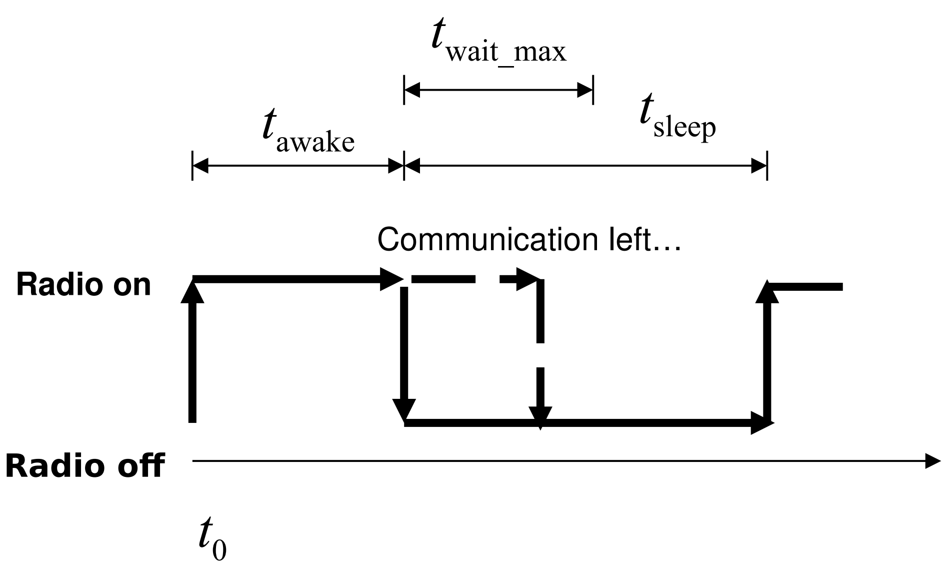
1. Turn radio on.
2. Wait until $t = t\_0 + t\_awake$.
3. If communication has not completed, wait until it has completed or $t = t\_0 + t\_awake + t\_wait\_max$.
4. Turn the radio off. Wait until $t = t\_0 + t\_awake + t\_sleep$.
5. Repeat from step 1.

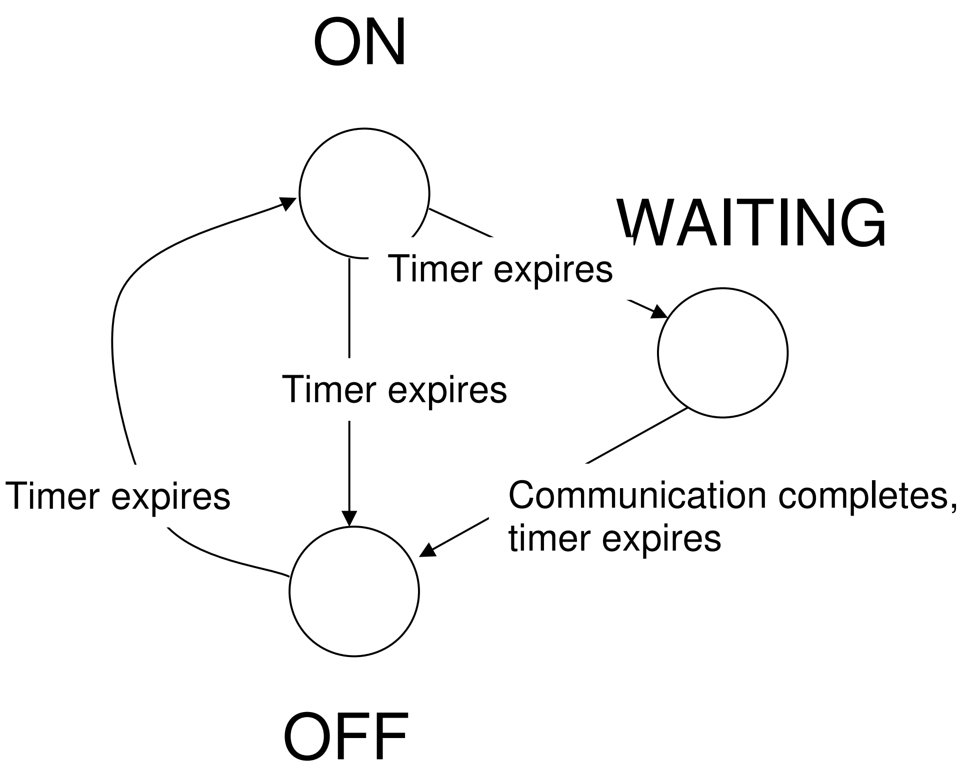
No blocking wait!

Problem: with events, we cannot implement this as a five-step program!

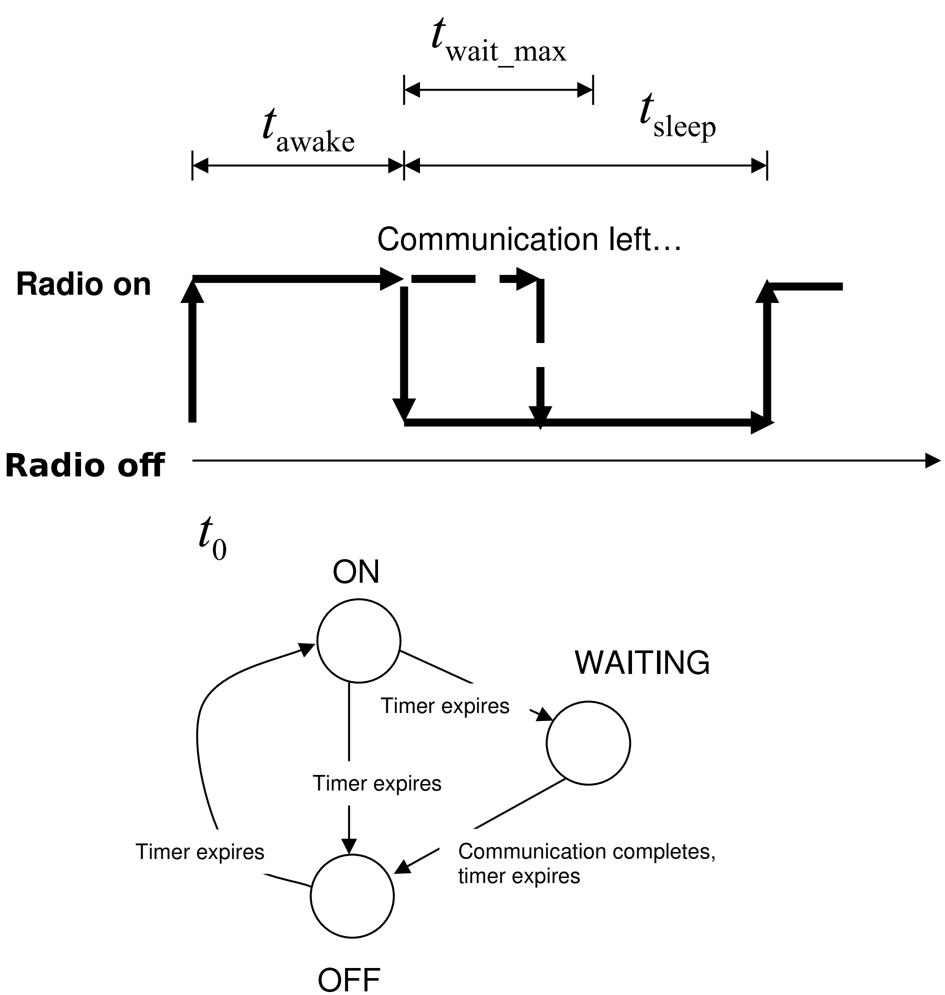
SWEDISH INSTITUTE OF COMPUTER SCIENCE

The event-based implementation: a state machine





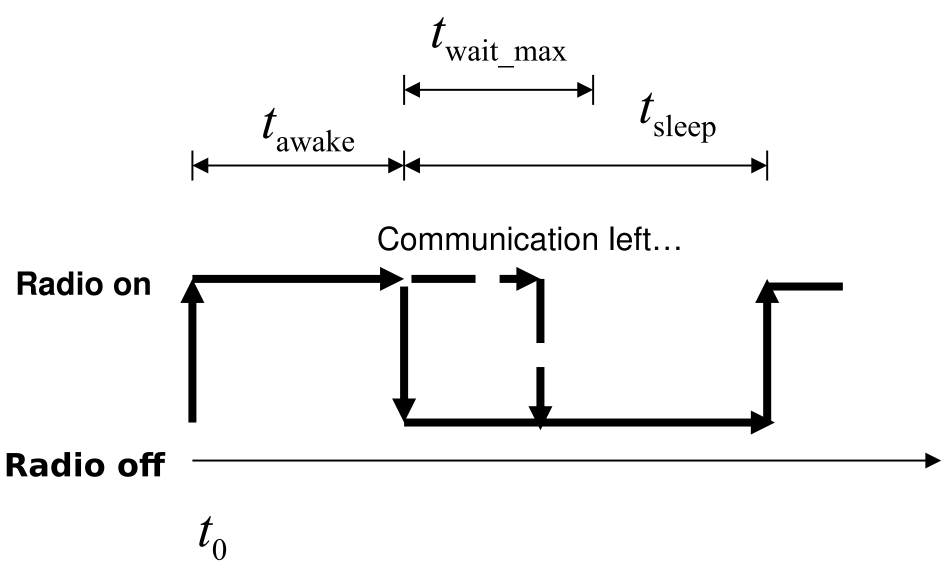
Event-driven state machine implementation: messy




```
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;
    }
  }
}
```

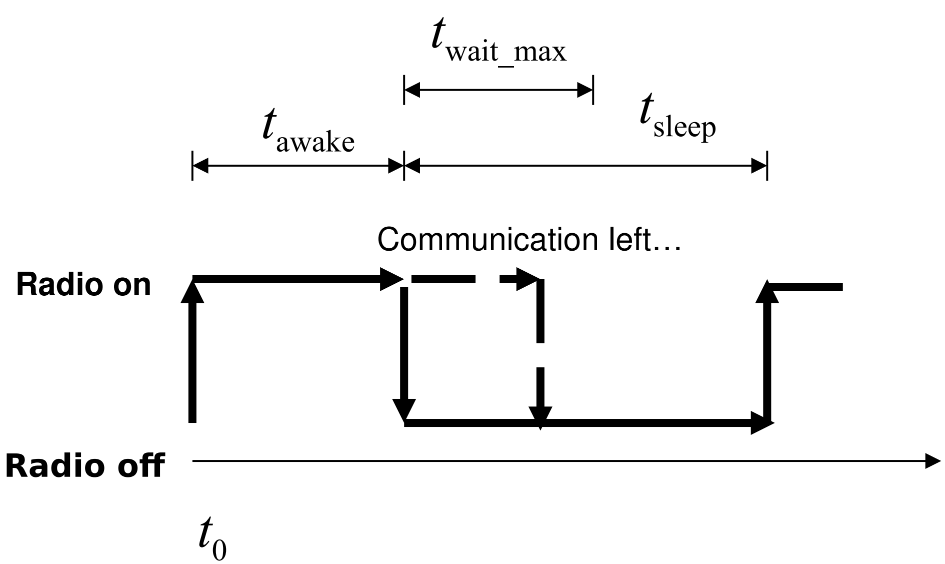

Adam Dunkels <adam@sics.se>

Protothreads makes implementation easier



- Protothreads – conditional blocking wait: **`PT_WAIT_UNTIL()`**
- No need for an explicit state machine
- Sequential code flow

Protothreads-based implementation is shorter




```
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
- Code uses structured programming (**if** and **while** statements)
- Mechanism evident from the code

Protothread scheduling

- A protothread runs in a C function
- We schedule a protothread by invoking its function
- We can invoke the protothread from an event handler
  - Protothreads as blocking event handlers
- We can let the operating system invoke our protothreads
  - Contiki
- Protothreads can invoke other protothreads
  - Can wait until a child protothread completes
  - Hierarchical protothreads

What’s wrong with using state machines?

- There is nothing wrong with state machines!
  - State machines are a powerful tool
  - Amenable to formal analysis, proofs
- But: state machines typically used to control the logical progam flow in many event-driven programs
  - Like using gotos instead of structured programming
  - The state machines not formally specified
  - Must be infered from reading the code
  - These state machines typically look like flow charts anyway
    - We’re not the first to see this
- Protothreads: use language constructs for flow control

Why not just use multithreading?

- Multithreading the basis of (almost) all embedded OS/RTOSes!
  - WSN community: Mantis, BTNut (based on multithreading); Contiki (multithreading on a per-application basis)
- Nothing wrong with multithreading
  - Multiple stacks require more memory
    - Networked = more concurrency than traditional embedded
    - Can lead to more expensive hardware
  - Preemption
  - Threads: explicit locking; Protothreads: implicit locking
- Protothreads are a new point in the design space
  - Between event-driven and multithreaded

How do we implement protothreads?

Implementing protothreads

- Modify the compiler?
  - There are many compilers to modify… (IAR, Keil, ICC, Microchip, GCC, …)
- Special preprocessor?
  - Requires us to maintain the preprocessor software on all development platforms
- Within the C language?
  - The best solution, if language is expressive enough
  - Possible?

Proof-of-concept implementation of protothreads in ANSI C

- Slightly limited version of protothreads in pure ANSI C
  - Uses the C preprocessor
  - Does not need a special preprocessor
  - No assembly language
- Very portable
  - Nothing is changed between platforms, C compilers
- Two approaches
  - Using GCCs C extension computed goto
    - Not ANSI C: works only with GCC
  - Using the C switch statement
    - ANSI C: works on every C compiler

Six-line implementation

Protothreads implemented using the C switch statement

```
struct pt { unsigned short lc; };

#define PT_INIT(pt)           pt->lc = 0
#define PT_BEGIN(pt)          switch(pt->lc) { case 0:
#define PT_EXIT(pt)           pt->lc = 0; return 2
#define PT_WAIT_UNTIL(pt, c)  pt->lc = __LINE__; case __LINE__: \
                              if(!(c)) return 0
#define PT_END(pt)            } pt->lc = 0; return 1
```

C-switch expansion

```
int a_protothread(struct pt *pt) {
  PT_BEGIN(pt);


  PT_WAIT_UNTIL(pt, condition1);


  if(something) {


    PT_WAIT_UNTIL(pt, condition2);


  }

  PT_END(pt);
}
```

```
int a_protothread(struct pt *pt) {
  switch(pt->lc) { case 0:


  pt->lc = 5; case 5:
  if(!condition1) return 0;

  if(something) {


    pt->lc = 10; case 10:
    if(!condition2) return 0;

  }

  } return 1;
}
```

Line numbers

Adam Dunkels <adam@sics.se>
SWEDISH INSTITUTE OF COMPUTER SCIENCE SICS

Limitations of the proof-of-concept implementation

- Automatic variables not stored across a blocking wait
  - Compiler does produce a warning
  - Workaround: use static local variables instead
  - Ericsson solution: enforce static locals through LINT scripts
- Constraints on the use of switch() constructs in programs
  - No warning produced by the compiler
  - Workaround: don’t use switches
- The limitations are due to the implementation, not protothreads as such

How well do protothreads work?

- Quantitative: reduction in code complexity over state machines
  - Rewritten seven state machine-based programs with protothreads
    - Four by applying a rewriting method, three by rewriting from scratch
  - Measure states, state transitions, lines of code
- Quantitative: code size
- Quantitative: execution time overhead
- Qualitative: useful in practice?

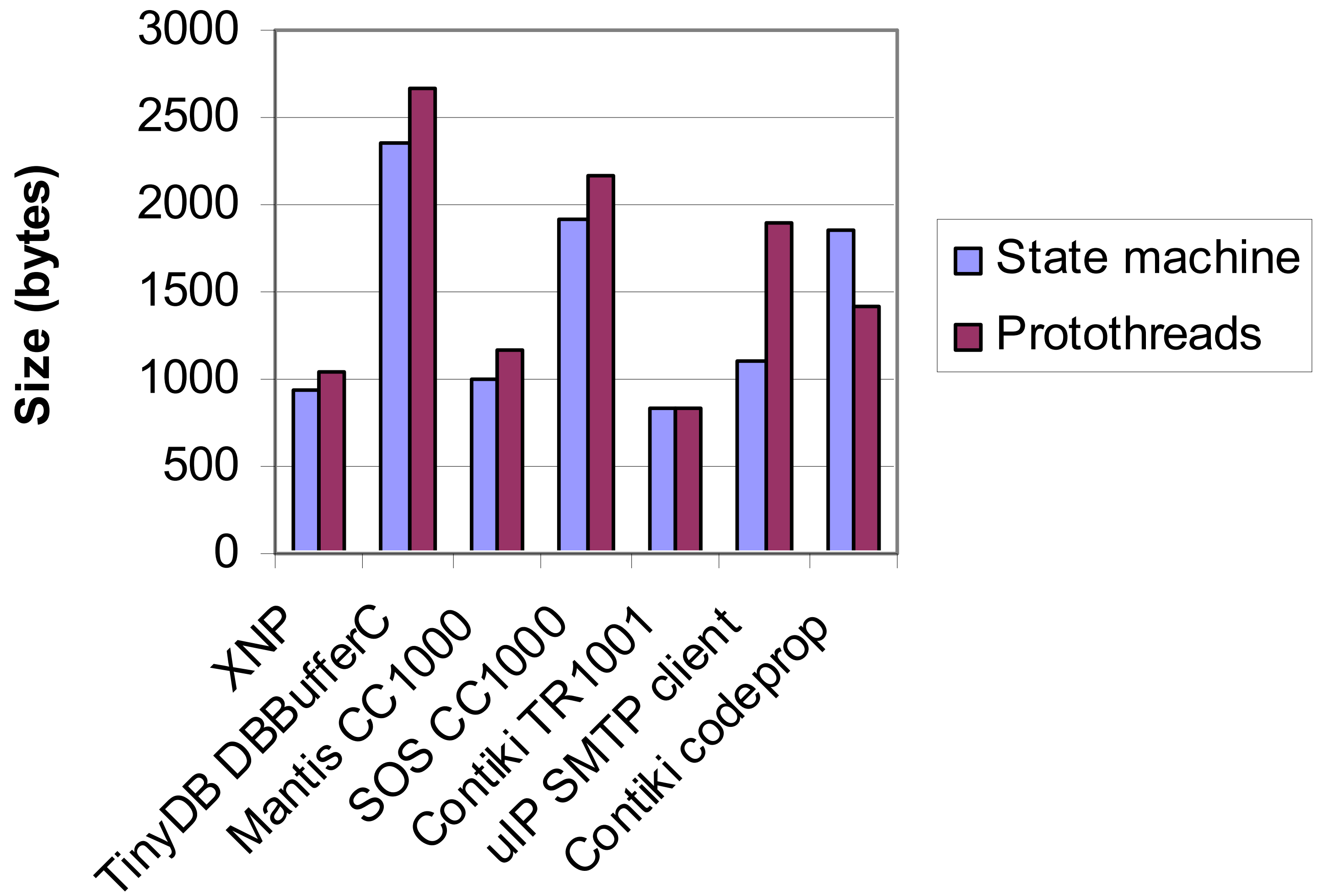
Reduction of complexity

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

Found state machine-related bugs in the Contiki TR1001 driver and the Contiki codeprop code when rewriting with protothreads

Code footprint

- Average increase ~200 bytes
  - Inconclusive



Execution time overhead is a few cycles

- Switch/computed goto jump incurs small fixed size preamble

	State machine	Protothreads, switch statement	Protothreads, computed gotos
gcc -Os	92	98	107
gcc –O1	91	94	103

Contiki TR1001 radio driver average execution time (CPU cycles)

Adam Dunkels <adam@sics.se>

Are protothreads useful in practice?

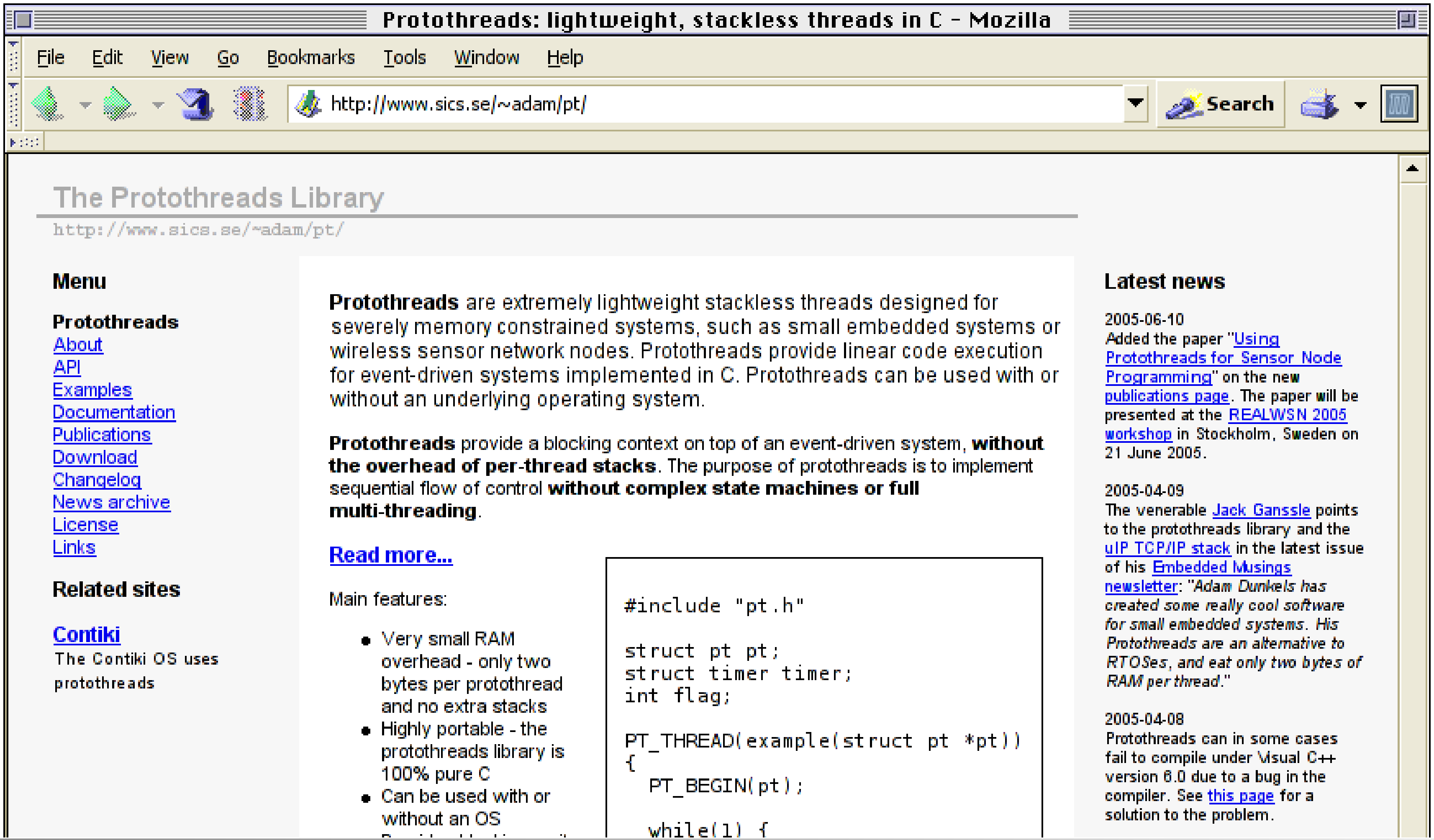
- We know that at least thirteen different embedded developers have adopted them
  - AVR, PIC, MSP430, ARM, x86
    - Portable: no changes when crossing platforms, compilers
  - MPEG decoding equipment, real-time systems
  - Others have ported protothreads to C++, Objective C
  - Probably many more
    - From mailing lists, forums, email questions
- Protothreads recommended twice in embedded "guru" Jack Ganssle's Embedded Muse newsletter

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)
- Protothreads have been adopted by and are recommended by others

Questions?

http://www.sics.se/~adam/pt/



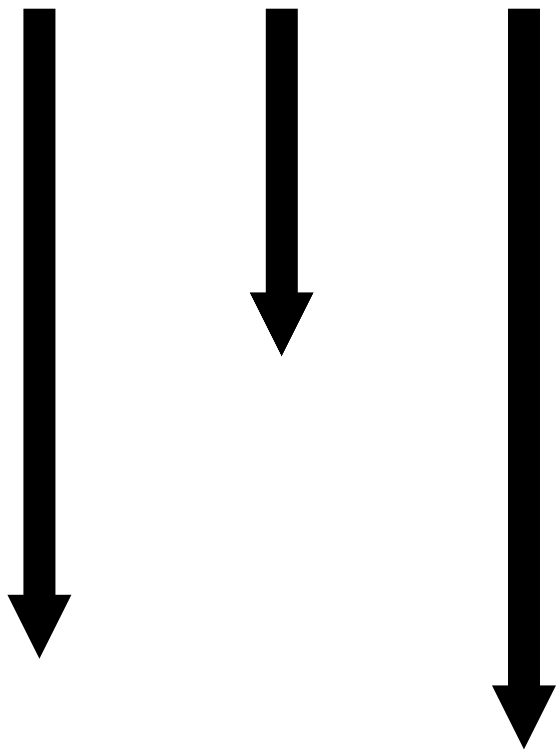
Hierarchical protothreads

```
int a_protothread(struct pt *pt) {
  static struct pt child_pt;

  PT_BEGIN(pt);

  PT_INIT(&child_pt);
  PT_WAIT_UNTIL(pt2(&child_pt) != 0);

  PT_END(pt);
}
```

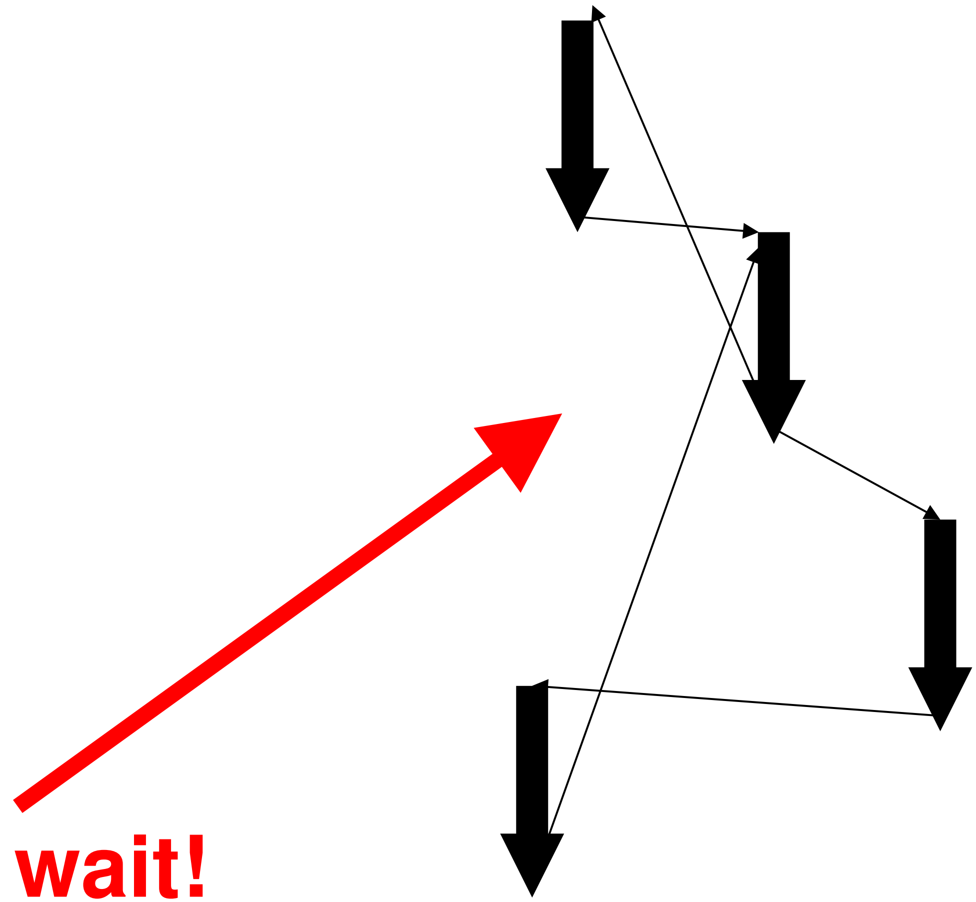
```
int pt2(struct pt *pt) {
  PT_BEGIN(pt);

  PT_WAIT_UNTIL(pt, condition);

  PT_END(pt);
}
```

Threads vs events…

Threads: sequential code flow



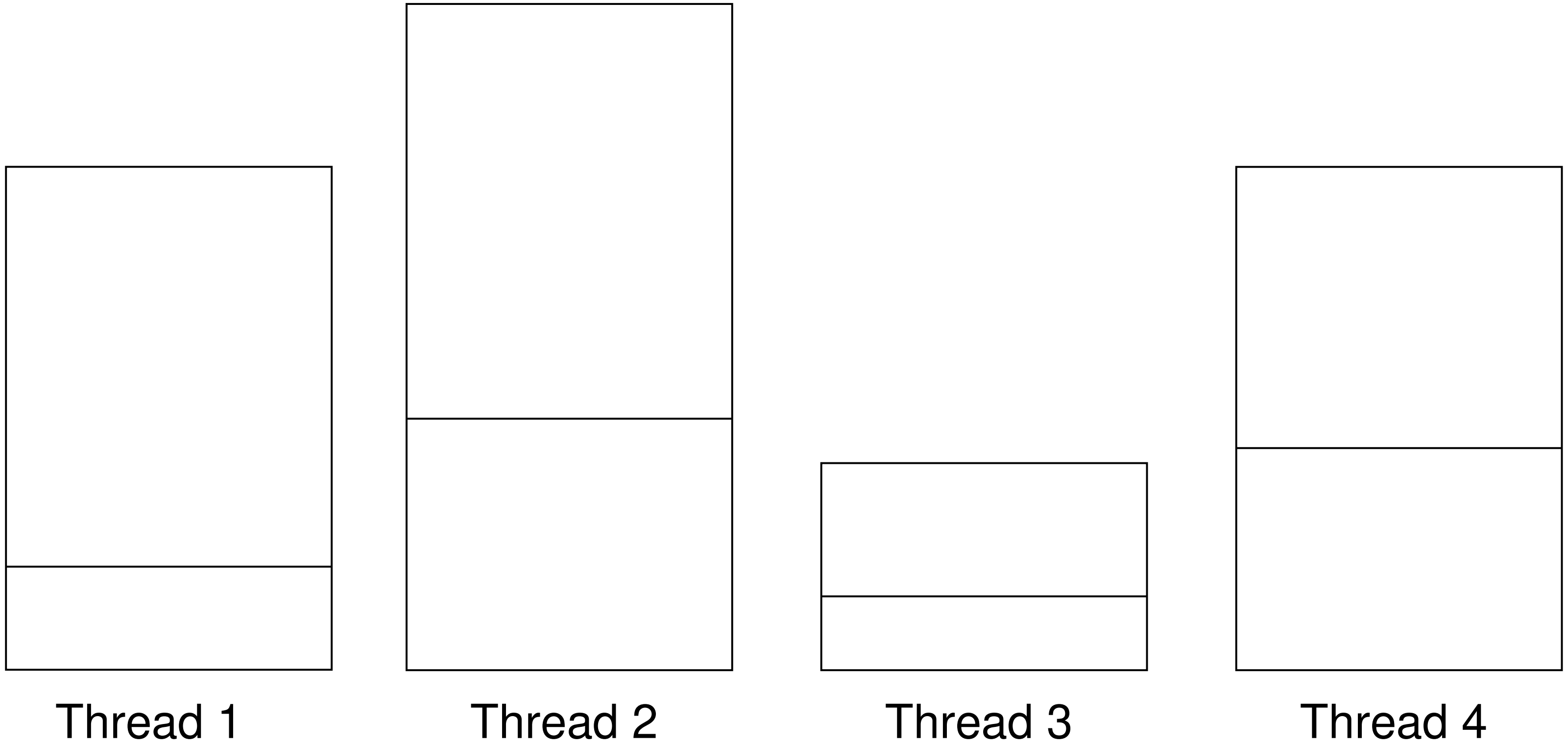
Events: unstructured code flow



No blocking wait!

Threads require per-thread stack memory

- Four threads, each with its own stack

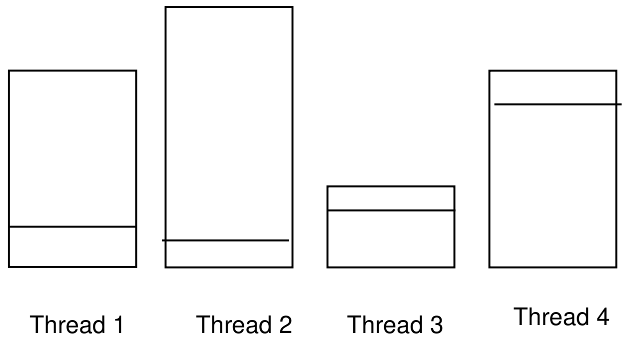


SWEDISH INSTITUTE OF COMPUTER SCIENCE SICS

Events require one stack

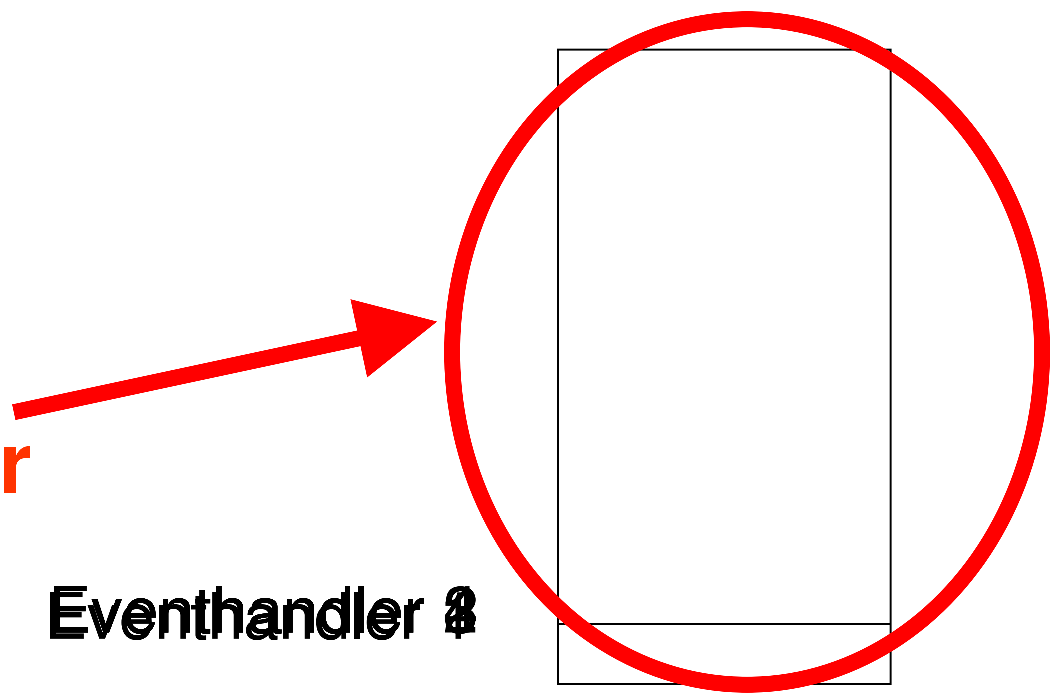
Threads require per-thread stack memory

- Four threads, each with its own stack



- Four event handlers, one stack

Stack is reused for every event handler

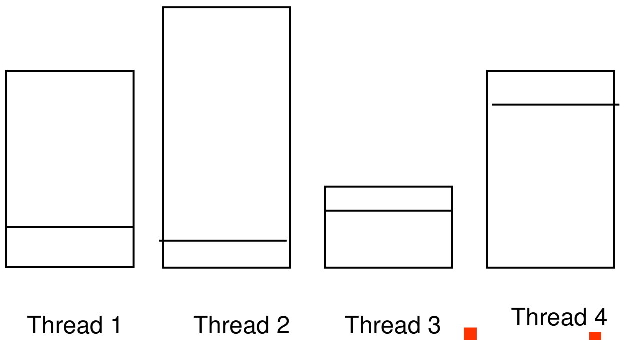


Protothreads require one stack

- Four protothreads, one stack

Threads require per-thread stack memory

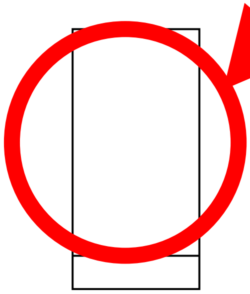
- Four threads, each with its own stack

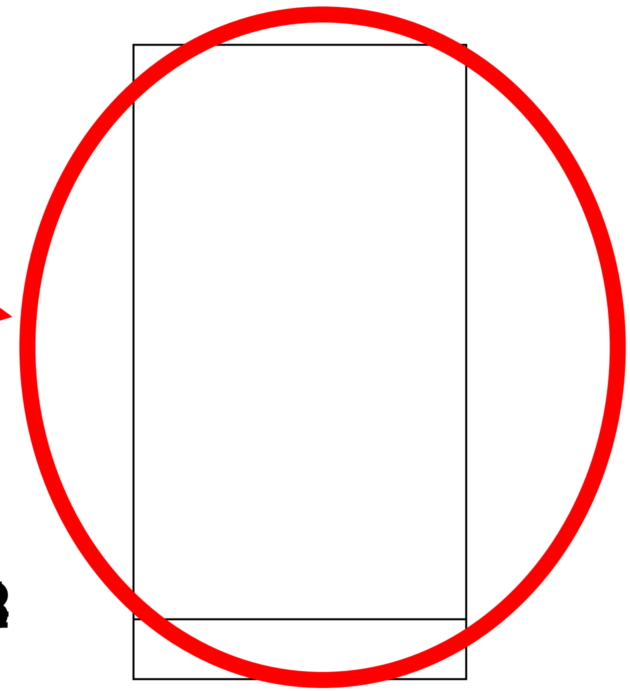


Just like events

Events require one stack

- Four event handlers, one stack





Protothread 4

Adam Dunkels <adam@sics.se>
SWEDISH INSTITUTE OF COMPUTER SCIENCE SICS

Trickle, T-MAC