NETWORKED EMBEDDED SOFTWARE LAB

POLITECNICO MILANO 1863

RI.SE

Contiki-NG
Protothreads

Luca Mottola
luca.mottola@polimi.it

Contiki-NG

Contiki NG
www.contiki-ng.org

- Why?
  - A **unique design point** when it comes to concurrency and networking
- Fork of original Contiki
- Focus on
  - RFC-compliant **IPv6 networking**
  - Modern **32-bit platforms**
- Mature: almost 15 years of development up to now (including original Contiki)
- Key features:
  - Configurability, ease of porting to new platforms
  - Efficient, configurable IPv6-based networking
  - Event-driven kernel, **protothread** concurrency model
  - Small memory footprint
- Used both in academia and industry

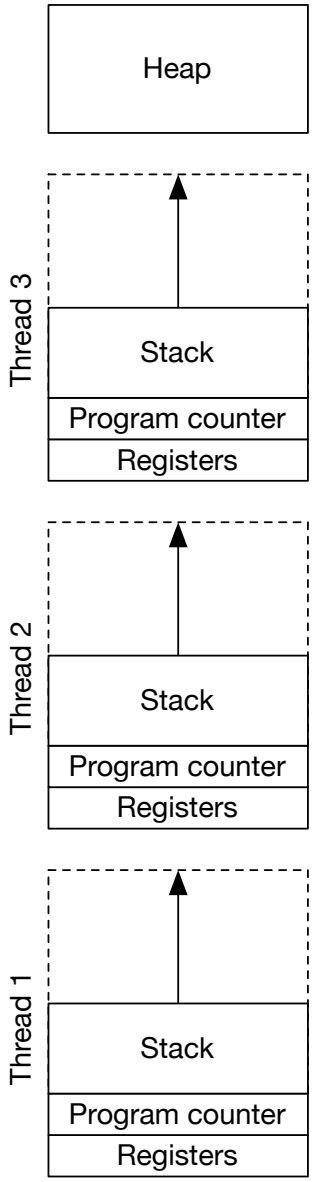
Contiki-NG Programming

- Use a dialect of the C language
  - Development environments and toolchains are essentially those of C
  - Debugging is a different ballgame though..
- Programs run on real hardware but also
  - On the **`native`** platform, that is, your host
  - In the COOJA simulator and MSPSim emulator

Protothreads

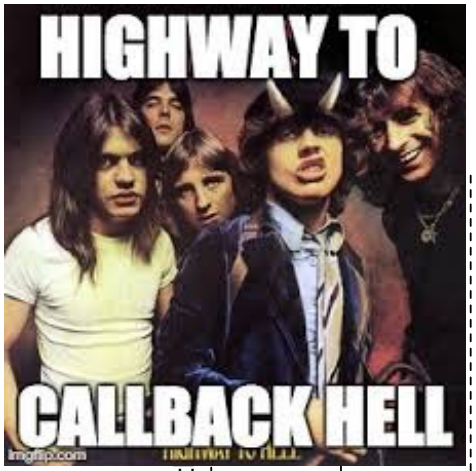
Existing Models: Multi-thread

- Similar to **multi-threading** in C/C++
  - A subset of traditional synchronization primitives available
- Not many embedded OSes provide it
  - RIoT, …
- Pros:
  - **Familiar to programmers**
  - Facilitates porting non-embedded code
- Cons:
  - Heavy on memory: each thread requires its own stack
  - Difficult to tune: hard to anticipate every thread's max stack size



Existing Models: Event-driven

- **Non-blocking calls** and (possibly asynchronous) **callbacks**
- Provided by many
  - ARM mBed, Contiki, Zephyr, Apache MyNewt…
- Pros:
  - Facilitates implementing reactive code
  - **Memory-efficient**: only one stack
- Cons:
  - Difficult to program: code as state machines, no sequential semantics



Event-driven: Example

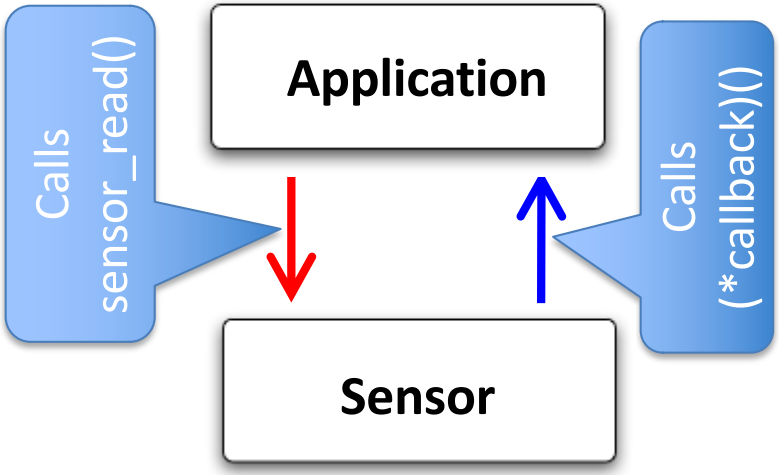
```
// Sensor interface: sensor.h
void sensor_read(void (*callback)(error_t, val_t));
```

The corresponding **sensor.c** is provided by sensor-specific implementations

```
#include "sensor.h"

void my_sensor_read_done(error_t err,
                         val_t value) {
  // …
  if  (err == OK) {
    do_something(value);
  }
}

void foo() {
  //   do something
  sensor_read(&my_sensor_read_done);
  // … do something more
}
```



If callbacks are asynchronous, the execution is preempted unless in an **atomic** block

Protothreads

- ”Best of both worlds”
- Maintain sequential semantics…
- …with a single stack
- Provided in Contiki and Contiki-NG

```
#include "contiki.h"
#include <stdio.h>


PROCESS(test_proc, "Test process");
AUTOSTART_PROCESSES(&test_proc);


PROCESS_THREAD(test_proc, ev, data)
{
  PROCESS_BEGIN();


  printf("Hello, world!\n");


  PROCESS_END();
}
```

We will see later where displayed…

Protothreads: Handling Events

- Protothreads process incoming events
  - A timer expiring, a packet received…
- They are scheduled cooperatively
  - Individual protothreads decide when to release the MCU

```
PROCESS_THREAD(test, ev, data)
{
  static struct etimer et;
  PROCESS_BEGIN();

  etimer_set(&et, CLOCK_SECOND); /* Trigger a 1s timer */
  for(;;) {
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
    printf("Hello, world!\n");=
    etimer_set(&et, CLOCK_SECOND);
  }

  PROCESS_END();
}
```

Set a new timer, triggering the next event for this protothread

Every time the protothread resumes, this condition is checked

Protothreads: Implementation

- Uses **local continuations** as result of macro expansion
  - When **set**, allow one to tag specific places in the code
  - When **resumed**, allow the execution to restart from within the code
- Use of macros spares the need for a dedicated preprocessor

Protothreads: Implementation

```
PROCESS_THREAD(test, ev, data)
{
  static struct etimer et;
  PROCESS_BEGIN();

  etimer_set(&et, CLOCK_SECOND);
  for(;;) {
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
    printf("Hello, world!\n");=
    etimer_set(&et, CLOCK_SECOND);
  }

  PROCESS_END();
}
```

```
static
char test (struct pt* pt,
           event_t ev, void* data)
{
  static struct etimer et;
  switch(pt->lc) { case 0:

  etimer_set(&et, CLOCK_SECOND);
  for(;;) {
    pt->lc = 15; case 15:
    if (!(etimer_expired(&et)) {
    return 0;
    }
    printf("Hello, world!\n");
    etimer_set(&et, CLOCK_SECOND)
  }

  pt->lc = 0; return 2;
}
```

- What really happens when resuming:
  - The protothread function is called again, and starts over
  - A **`switch`** statement reads the value of **`pt->lc`**
  - The execution jumps to the **`case:`** corresponding to where the execution yielded
- Therefore, local variables need to be **`static`**, or they get re-initialized every time the protothread resumes
  - **No free lunches**: they consume memory also when the protothread is not executing!
- Do not use **`switch`** inside protothreads, or you will mess things up

Protothreads: Events

- Events used to
  - Exchange data among protothreads
  - Achieve cooperative scheduling

```
PROCESS_THREAD(ping_process, ev, data)
{
  static struct etimer timer;


  PROCESS_BEGIN();

  /* Start the ping pong... */
  ping_event=process_alloc_event();
  printf("Sent the first ping!\n");
  process_post(&pong_process, ping_event, &count);

  // …
```

```
PROCESS_THREAD(pong_process, ev, data)
{
    // …
    PROCESS_WAIT_EVENT();
    if (ev == ping_event) {
      printf("Got a ping: %d!\n",*(int*)data);
      process_post(&ping_process, pong_event, &count);
      printf("Sent a pong event to ping!\n");
    } else {
      printf("Got an unknonw event!\n");
    }
    // …
  }
}
```

Find code under **`examples/events`**!

Protothreads: APIs

- **`PROCESS_PAUSE()`**: releases control, rescheduled immediately
- **`PROCESS_YIELD()`**: releases control, wait for next event
- **`process_poll(&process_name)`**: polls another protothread
- **`PROCESS_EXIT()`**: stops a protothread
- **`process_exit(&process_name)`**: kills another protothread