Distributed Systems

Inga Rüb

21 November 2018

Contiki

an open source OS for tiny low-cost, low-power microcontrollers

enables standardized low-power wireless communication

http://www.contiki-os.org/index.html

Adam Dunkels, PhD, is the CEO and co-founder of Thingsquare, the creator of the Contiki open source OS, and an Internet of Things pioneer.

The MIT Technology Review named him **one of the top 35 innovators in the world** for having created the minimal wireless networking protocols that allow almost any device to communicate over the Internet.

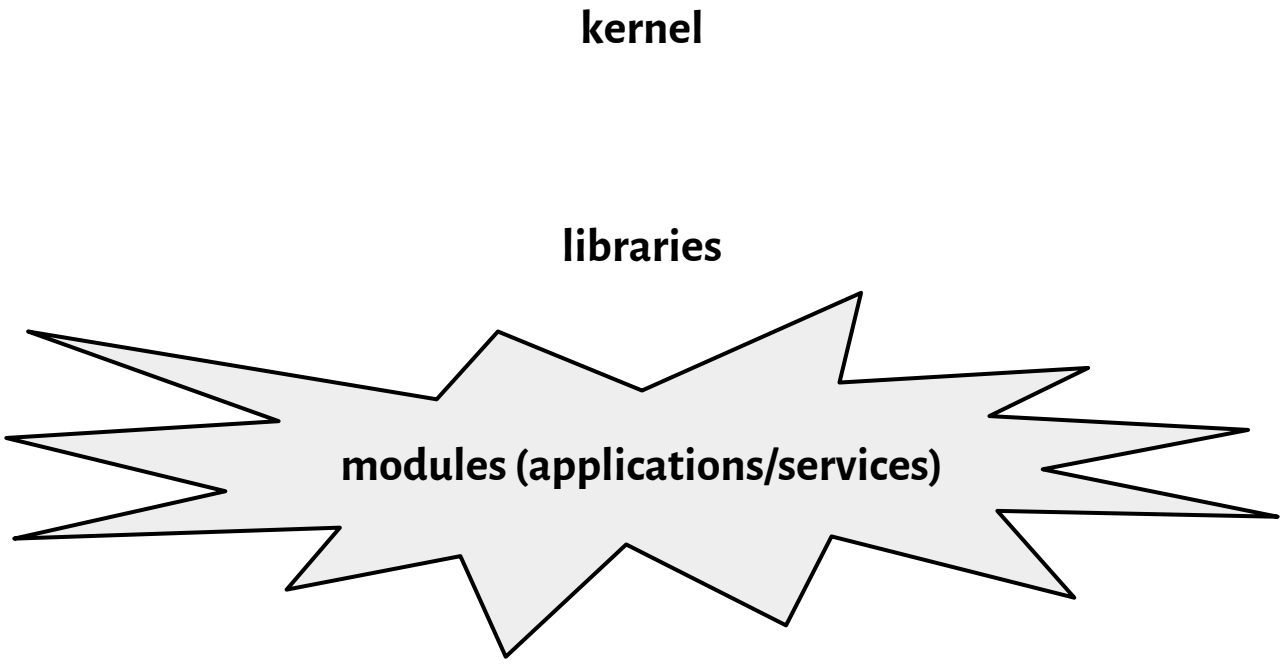
Contiki

kernel

libraries

modules (applications/services)

Contiki

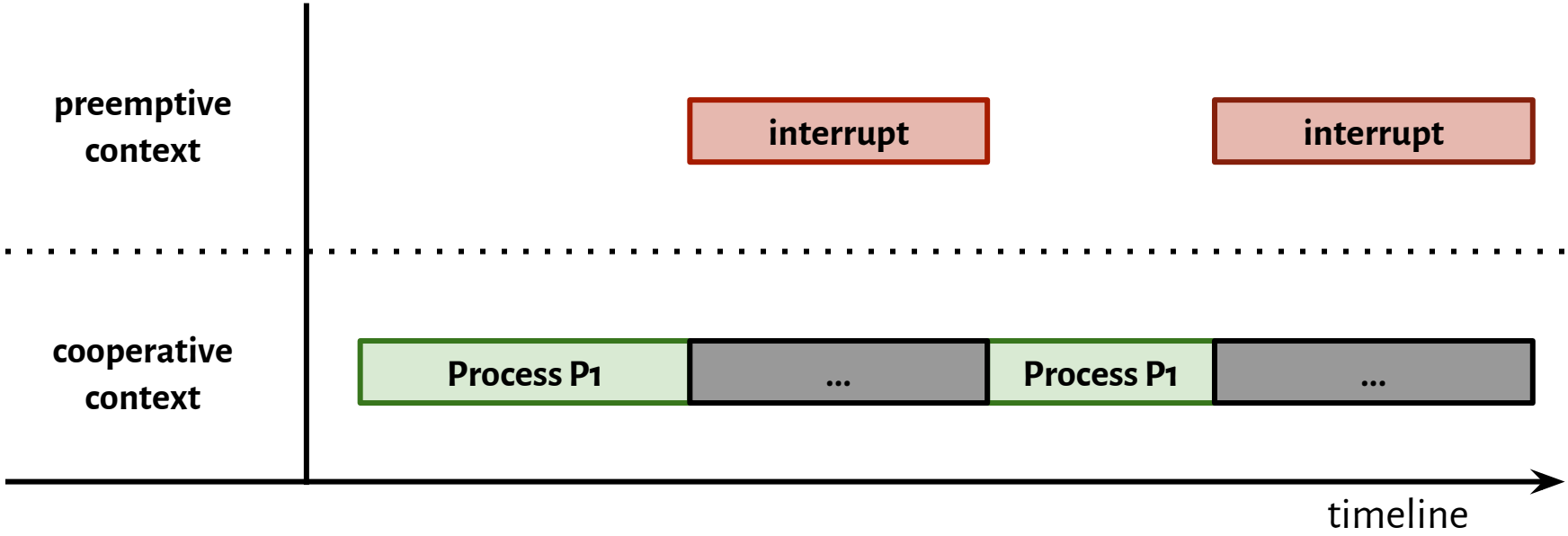


https://github.com/contiki-os/contiki/wiki/The-dynamic-loader

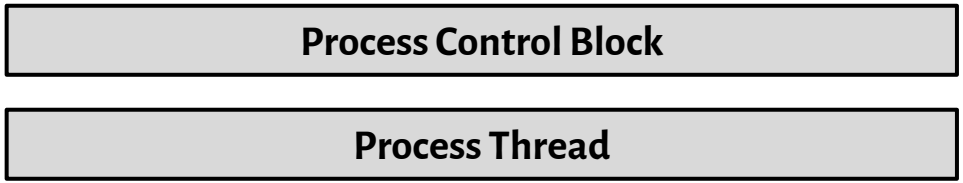
Contiki - processes

All Contiki programs are processes.

A ***process*** is a piece of code that is executed regularly by the Contiki system.



Contiki - processes



Contiki - processes


```
// kept in RAM
struct process {
   struct process *next;
   const char *name;
   int (* thread)(struct pt *,
                  process_event_t,
                  process_data_t);
   struct pt pt;
   unsigned char state, needspoll;
};
```

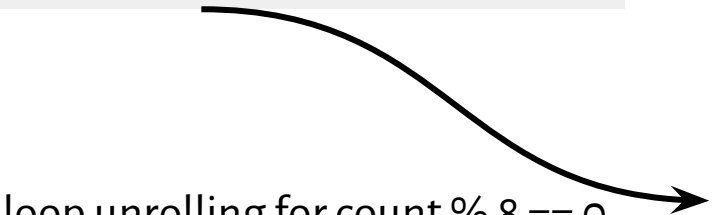
Contiki - processes

- ★ is stored in ROM
- ★ contains the code of the process
- ★ is a single ***protothread*** that is invoked from the process scheduler

Protothreads And Duff's Device

Problem: copy 16-bit units from an array into a memory-mapped output register

```
send(to, from, count)
register short *to, *from;
register count;
{
    do {
        *to = *from++;
    } while(--count > 0);
}
```




```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = count / 8;
    do {
    *to = *from++;
    *to = *from++;
    *to = *from++;
    *to = *from++;
    *to = *from++;
    *to = *from++;
    *to = *from++;
    *to = *from++;
    } while (--n > 0);
}
```

Protothreads And Duff's Device

Problem: copy 16-bit units from an array into a memory-mapped output register

```
send(to, from, count)
register short *to, *from;
register count;
{
     register n = (count + 7) / 8;
     switch (count % 8) {
     case 0: do { *to = *from++;
     case 7:   *to = *from++;
     case 6:   *to = *from++;
     case 5:   *to = *from++;
     case 4:   *to = *from++;
     case 3:   *to = *from++;
     case 2:   *to = *from++;
     case 1:   *to = *from++;
          } while (--n > 0);
     }
}
```

Protothreads And Duff's Device

Problem: copy 16-bit units from an array into a memory-mapped output register

```
send(to, from, count)
register short *to, *from;
register count;
{
     register n = (count + 7) / 8;
     switch (count % 8) {
     case 0: do { *to = *from++;
     case 7:   *to = *from++;
     case 6:   *to = *from++;
     case 5:   *to = *from++;
     case 4:   *to = *from++;
     case 3:   *to = *from++;
     case 2:   *to = *from++;
     case 1:   *to = *from++;
          } while (--n > 0);
     }
}
```

a protothread is just a C function

Protothreads And Duff's Device

Problem: copy 16-bit units from an array into a memory-mapped output register

```
send(to, from, count)
register short *to, *from;
register count;
{
     register n = (count + 7) / 8;
     switch (count % 8) {
     case 0: do { *to = *from++;
     case 7:   *to = *from++;
     case 6:   *to = *from++;
     case 5:   *to = *from++;
     case 4:   *to = *from++;
     case 3:   *to = *from++;
     case 2:   *to = *from++;
     case 1:   *to = *from++;
          } while (--n > 0);
     }
}
```

a protothread is just a C function

that has many entry points to the code

Protothreads And Duff's Device

Problem: copy 16-bit units from an array into a memory-mapped output register

```
send(to, from, count)
register short *to, *from;
register count;
{
     register n = (count + 7) / 8;
     switch (count % 8) {
     case 0: do { *to = *from++;
     case 7:   *to = *from++;
     case 6:   *to = *from++;
     case 5:   *to = *from++;
     case 4:   *to = *from++;
     case 3:   *to = *from++;
     case 2:   *to = *from++;
     case 1:   *to = *from++;
          } while (--n > 0);
     }
}
```

a protothread is just a C function

that has many entry points to the code

switching the context is jumping to the proper case

Protothreads And Duff's Device

Problem: copy 16-bit units from an array into a memory-mapped output register

```
send(to, from, count)
register short *to, *from;
register count;
{
     register n = (count + 7) / 8;
     switch (count % 8) {
     case 0: do { *to = *from++;
     case 7:   *to = *from++;
     case 6:   *to = *from++;
     case 5:   *to = *from++;
     case 4:   *to = *from++;
     case 3:   *to = *from++;
     case 2:   *to = *from++;
     case 1:   *to = *from++;
          } while (--n > 0);
     }
}
```

a protothread is just a C function

that has many entry points to the code

switching the context is jumping to the proper case

determined by a global variable

Protothreads And Duff's Device

Problem: copy 16-bit units from an array into a memory-mapped output register

```
send(to, from, count)
register short *to, *from;
register count;
{
     register n = (count + 7) / 8;
     switch (count % 8) {
     case 0: do { *to = *from++;
     case 7:   *to = *from++;
     case 6:   *to = *from++;
     case 5:   *to = *from++;
     case 4:   *to = *from++;
     case 3:   *to = *from++;
     case 2:   *to = *from++;
     case 1:   *to = *from++;
          } while (--n > 0);
     }
}
```

a protothread is just a C function

that has many entry points to the code

switching the context is jumping to the proper case

determined by a global variable

well, all variables should be global

Contiki - Hello world

```
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------*/
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data)
{
  PROCESS_BEGIN();

  printf("Hello, world\n");

  PROCESS_END();
}
/*---------------------------------*/
```

declare a process

Contiki - Hello world

```
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------*/
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data)
{
  PROCESS_BEGIN();

  printf("Hello, world\n");

  PROCESS_END();
}
/*---------------------------------*/
```

start the process automatically

Contiki - Hello world

```
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------*/
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data)
{
  PROCESS_BEGIN();

  printf("Hello, world\n");

  PROCESS_END();
}
/*---------------------------------*/
```

define the body of a process

Contiki - Hello world

```
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------*/
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data)
{
  PROCESS_BEGIN();

  printf("Hello, world\n");

  PROCESS_END();
}
/*---------------------------------*/
```

specify the beginning of a process

specify the end of a process

Contiki - Counter

```
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------*/
static struct etimer et_hello;
static uint16_t count;
/*---------------------------------*/
PROCESS(counter_process, "Hello world process");
AUTOSTART_PROCESSES(&counter_process);
/*---------------------------------*/
PROCESS_THREAD(counter_process, ev, data)
{
  PROCESS_BEGIN();
  ...
  PROCESS_END();
}
/*---------------------------------*/
```

declare variables: a timer and a counter

Contiki - Counter

```
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------*/
static struct etimer et_hello;
static uint16_t count;
/*---------------------------------*/
PROCESS(counter_process, "Hello world process");
AUTOSTART_PROCESSES(&counter_process);
/*---------------------------------*/
PROCESS_THREAD(counter_process, ev, data)
{
  PROCESS_BEGIN();
  ...
  PROCESS_END();
}
/*---------------------------------*/
```

here we can *autostart* our previous process as well:

```
AUTOSTART_PROCESSES(
&hello_world_process, &counter_process);
```

Contiki - Counter

```
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------*/
static struct etimer et_hello;
static uint16_t count;
/*---------------------------------*/
PROCESS(counter_process, "Hello world process");
AUTOSTART_PROCESSES(&counter_process);
/*---------------------------------*/
PROCESS_THREAD(counter_process, ev, data)
{
  PROCESS_BEGIN();
  ...
  PROCESS_END();
}
/*---------------------------------*/
```

initialize the timer: it will fire after 4 seconds

```
etimer_set(&et_hello, CLOCK_SECOND * 4);
count = 0;

while(1) {
    PROCESS_WAIT_EVENT();

    if(ev == PROCESS_EVENT_TIMER) {
        printf("Sensor says #%u\n", count);
        count++;

        etimer_reset(&et_hello);
    }
}
```

Contiki - Counter

```
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------*/
static struct etimer et_hello;
static uint16_t count;
/*---------------------------------*/
PROCESS(counter_process, "Hello world process");
AUTOSTART_PROCESSES(&counter_process);
/*---------------------------------*/
PROCESS_THREAD(counter_process, ev, data)
{
  PROCESS_BEGIN();
  ...
  PROCESS_END();
}
/*---------------------------------*/
```

the process is blocked and waits for an event

```
etimer_set(&et_hello, CLOCK_SECOND * 4);
count = 0;

while(1) {
   PROCESS_WAIT_EVENT();

   if(ev == PROCESS_EVENT_TIMER) {
      printf("Sensor says #%u\n", count);
      count++;

      etimer_reset(&et_hello);
   }
}
```

Contiki - Counter

```
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------*/
static struct etimer et_hello;
static uint16_t count;
/*---------------------------------*/
PROCESS(counter_process, "Hello world process");
AUTOSTART_PROCESSES(&counter_process);
/*---------------------------------*/
PROCESS_THREAD(counter_process, ev, data)
{
  PROCESS_BEGIN();
  ...
  PROCESS_END();
}
/*---------------------------------*/
```

verify if the event comes from the timer

```
etimer_set(&et_hello, CLOCK_SECOND * 4);
count = 0;

while(1) {
   PROCESS_WAIT_EVENT();

   if(ev == PROCESS_EVENT_TIMER) {
      printf("Sensor says #%u\n", count);
      count++;

      etimer_reset(&et_hello);
   }
}
```

Contiki - Counter

```c
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------*/
static struct etimer et_hello;
static uint16_t count;
/*---------------------------------*/
PROCESS(counter_process, "Hello world process");
AUTOSTART_PROCESSES(&counter_process);
/*---------------------------------*/
PROCESS_THREAD(counter_process, ev, data)
{
  PROCESS_BEGIN();
  ...
  PROCESS_END();
}
/*---------------------------------*/
```

reset the timer so it will fire again after 4 seconds

```c
etimer_set(&et_hello, CLOCK_SECOND * 4);
count = 0;

while(1) {
   PROCESS_WAIT_EVENT();

   if(ev == PROCESS_EVENT_TIMER) {
      printf("Sensor says #%u\n", count);
      count++;

      etimer_reset(&et_hello);
   }
}
```

Contiki - Counter

```
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------*/
static struct etimer et_hello;
static uint16_t count;
/*---------------------------------*/
PROCESS(counter_process, "Hello world process");
AUTOSTART_PROCESSES(&counter_process);
/*---------------------------------*/
PROCESS_THREAD(counter_process, ev, data)
{
  PROCESS_BEGIN();
  ...
  PROCESS_END();
}
/*---------------------------------*/
```

reset the timer so it will fire again after 4 seconds

```
etimer_set(&et_hello, CLOCK_SECOND * 4);
count = 0;

while(1) {
   PROCESS_WAIT_EVENT();

   if(ev == PROCESS_EVENT_TIMER) {
      printf("Sensor says #%u\n", count);
      count++;

      etimer_reset(&et_hello);
   }
}
```

Contiki - Counter

```c
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------*/
static struct etimer et_hello;
static uint16_t count;
/*---------------------------------*/
PROCESS(counter_process, "Hello world process");
AUTOSTART_PROCESSES(&counter_process);
/*---------------------------------*/
PROCESS_THREAD(counter_process, ev, data)
{
  PROCESS_BEGIN();
  ...
  PROCESS_END();
}
/*---------------------------------*/
```

reset the timer so it will fire again after 4 seconds

```c
etimer_set(&et_hello, CLOCK_SECOND * 4);
count = 0;

while(1) {
   PROCESS_WAIT_EVENT_UNTIL(
      ev == PROCESS_EVENT_TIMER);

   printf("Sensor says #%u\n", count);
   count++;

   etimer_reset(&et_hello);

}
```

Contiki - Counter

```c
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------*/
static struct etimer et_hello;
static uint16_t count;
/*---------------------------------*/
PROCESS(counter_process, "Hello world process");
AUTOSTART_PROCESSES(&counter_process);
/*---------------------------------*/
PROCESS_THREAD(counter_process, ev, data)
{
  PROCESS_BEGIN();
  ...
  PROCESS_END();
}
/*---------------------------------*/
```

reset the timer so it will fire again after 4 seconds

```c
etimer_set(&et_hello, CLOCK_SECOND * 4);
count = 0;

while(1) {
   PROCESS_WAIT_EVENT_UNTIL(
      etimer_expired(&et_hello));

   printf("Sensor says #%u\n", count);
   count++;

   etimer_reset(&et_hello);

}
```

```
static
PT_THREAD(example(struct pt *pt))
{
  PT_BEGIN(pt);

  while(1) {
    PT_WAIT_UNTIL(pt,
      counter == 1000);
    printf("Threshold reached\n");
    counter = 0;
  }

  PT_END(pt);
}
```

```
static
char example(struct pt *pt)
{
  switch(pt->lc) { case 0:

  while(1) {
    pt->lc = 12; case 12:
    if(!(counter == 1000)) return 0;
    printf("Threshold reached\n");
    counter = 0;
  }

  } pt->lc = 0; return 2;
}…
```

```
PROCESS_BEGIN(); // Declares the beginning of a process' protothread.
PROCESS_END(); // Declares the end of a process' protothread.
PROCESS_EXIT(); // Exit the process.
PROCESS_WAIT_EVENT(); // Wait for any event.
PROCESS_WAIT_EVENT_UNTIL(); // Wait for an event, but with a condition.
PROCESS_YIELD(); // Wait for any event, equivalent to PROCESS_WAIT_EVENT().
PROCESS_WAIT_UNTIL(); // Wait for a given condition; may not yield the process.
PROCESS_PAUSE(); // Temporarily yield the process.
```

The Clock Module and Timers

```
clock_time_t clock_time();        // Get the system time.
unsigned long clock_seconds();    // Get the system time in seconds.

CLOCK_SECOND;                     // The number of ticks per second.
void clock_init(void);            // Initialize the clock module (called when booting).
void clock_wait(int delay);       // Delay the CPU for a number of clock ticks.
```

Why do we need to delay CPU?

Learn more about *Delaying Execution*:
https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch07.html

```
void clock_wait(clock_time_t i) {
     clock_time_t start;
     start = clock_time();
     while(clock_time() - start < (clock_time_t)i);
}
```

The Clock Module and Timers

```
struct timer {
     clock_time_t start;
     clock_time_t interval;
};
```

```
void timer_set(struct timer *t, clock_time_t interval); // Start the timer.
void timer_reset(struct timer *t); // Restart from the previous expiration time.
void timer_restart(struct timer *t); // Restart the timer from current time.
int timer_expired(struct timer *t); // Check if the timer has expired.
clock_time_t timer_remaining(struct timer *t); // Get the time till the timer expires.
```

```
void timer_set(struct timer *t, clock_time_t interval) {
     t->interval = interval;
     t->start = clock_time();
}
```

The Clock Module and Timers

```
struct timer {
     clock_time_t start;
     clock_time_t interval;
};
```

```
void timer_set(struct timer *t, clock_time_t interval); // Start the timer.
void timer_reset(struct timer *t); // Restart from the previous expiration time.
void timer_restart(struct timer *t); // Restart the timer from current time.
int timer_expired(struct timer *t); // Check if the timer has expired.
clock_time_t timer_remaining(struct timer *t); // Get the time till the timer expires.
```

```
void timer_reset(struct timer *t, clock_time_t interval) {
     t->start += t->interval;
}
```

The Clock Module and Timers

```
struct timer {
     clock_time_t start;
     clock_time_t interval;
};
```

```
void timer_set(struct timer *t, clock_time_t interval); // Start the timer.
void timer_reset(struct timer *t); // Restart from the previous expiration time.
void timer_restart(struct timer *t); // Restart the timer from current time.
int timer_expired(struct timer *t); // Check if the timer has expired.
clock_time_t timer_remaining(struct timer *t); // Get the time till the timer expires.
```

```
void timer_reset(struct timer *t, clock_time_t interval) {
     t->start += t->interval;
}

void timer_restart(struct timer *t, clock_time_t interval) {
     t->start = clock_time();
}
```

The Clock Module and Timers

```
struct timer {
     clock_time_t start;
     clock_time_t interval;
};
```

```
void timer_set(struct timer *t, clock_time_t interval); // Start the timer.
void timer_reset(struct timer *t); // Restart from the previous expiration time.
void timer_restart(struct timer *t); // Restart the timer from current time.
int timer_expired(struct timer *t); // Check if the timer has expired.
clock_time_t timer_remaining(struct timer *t); // Get the time till the timer expires.
```

```
interrupt(UART1RX_VECTOR) uart1_rx_interrupt(void) {
     if (timer_expired(&t)) {
          /* … */
     }
     timer_restart(&t);
     /* … */
 }
```

The Clock Module and Timers

```
struct timer {
     clock_time_t start;
     clock_time_t interval;
};
```

```
void timer_set(struct timer *t, clock_time_t interval); // Start the timer.
void timer_reset(struct timer *t); // Restart from the previous expiration time.
void timer_restart(struct timer *t); // Restart the timer from current time.
int timer_expired(struct timer *t); // Check if the timer has expired.
clock_time_t timer_remaining(struct timer *t); // Get the time till the timer expires.
```

```
struct stimer {
     unsigned long start;
     unsigned long interval;
};
```

The Clock Module and Timers

```
struct timer {
     clock_time_t start;
     clock_time_t interval;
};
```

```
void timer_set(struct timer *t, clock_time_t interval); // Start the timer.
void timer_reset(struct timer *t); // Restart from the previous expiration time.
void timer_restart(struct timer *t); // Restart the timer from current time.
int timer_expired(struct timer *t); // Check if the timer has expired.
clock_time_t timer_remaining(struct timer *t); // Get the time till the timer expires.
```

timer

stimer

The Clock Module and Timers

```
struct timer {
     clock_time_t start;
     clock_time_t interval;
};
```

```
void timer_set(struct timer *t, clock_time_t interval); // Start the timer.
void timer_reset(struct timer *t); // Restart from the previous expiration time.
void timer_restart(struct timer *t); // Restart the timer from current time.
int timer_expired(struct timer *t); // Check if the timer has expired.
clock_time_t timer_remaining(struct timer *t); // Get the time till the timer expires.
```

timer

stimer

etimer

Contiki - Counter

```c
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------*/
static struct etimer et_hello;
static uint16_t count;
/*---------------------------------*/
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data)
{
  PROCESS_BEGIN();
  ...
  PROCESS_END();
}
/*---------------------------------*/
```

reset the timer so it will fire again after 4 seconds

```c
etimer_set(&et_hello, CLOCK_SECOND * 4);
count = 0;

while(1) {
   PROCESS_WAIT_EVENT_UNTIL(
      etimer_expired(&et));

   printf("Sensor says #%u\n", count);
   count++;

   etimer_reset(&et_hello);

}
```

The Clock Module and Timers

```
struct timer {
     clock_time_t start;
     clock_time_t interval;
};
```

```
void timer_set(struct timer *t, clock_time_t interval); // Start the timer.
void timer_reset(struct timer *t); // Restart from the previous expiration time.
void timer_restart(struct timer *t); // Restart the timer from current time.
int timer_expired(struct timer *t); // Check if the timer has expired.
clock_time_t timer_remaining(struct timer *t); // Get the time till the timer expires.
```

timer

stimer

etimer

ctimer

CTimer

```
#include "sys/ctimer.h"
static struct ctimer t;

static void
callback(void *ptr)
{
  ctimer_reset(&t);
  /* ... */
}

void
init(void)
{
  ctimer_set(&t, CLOCK_SECOND,
             callback, NULL);
}
```

```
void ctimer_stop(struct ctimer *t);
// Stop the timer.
```

Like etimer, ctimer **cannot** be used safely from interrupts.

The Clock Module and Timers

```
struct timer {
     clock_time_t start;
     clock_time_t interval;
};
```

```
void timer_set(struct timer *t, clock_time_t interval); // Start the timer.
void timer_reset(struct timer *t); // Restart from the previous expiration time.
void timer_restart(struct timer *t); // Restart the timer from current time.
int timer_expired(struct timer *t); // Check if the timer has expired.
clock_time_t timer_remaining(struct timer *t); // Get the time till the timer expires.
```

timer

stimer

etimer

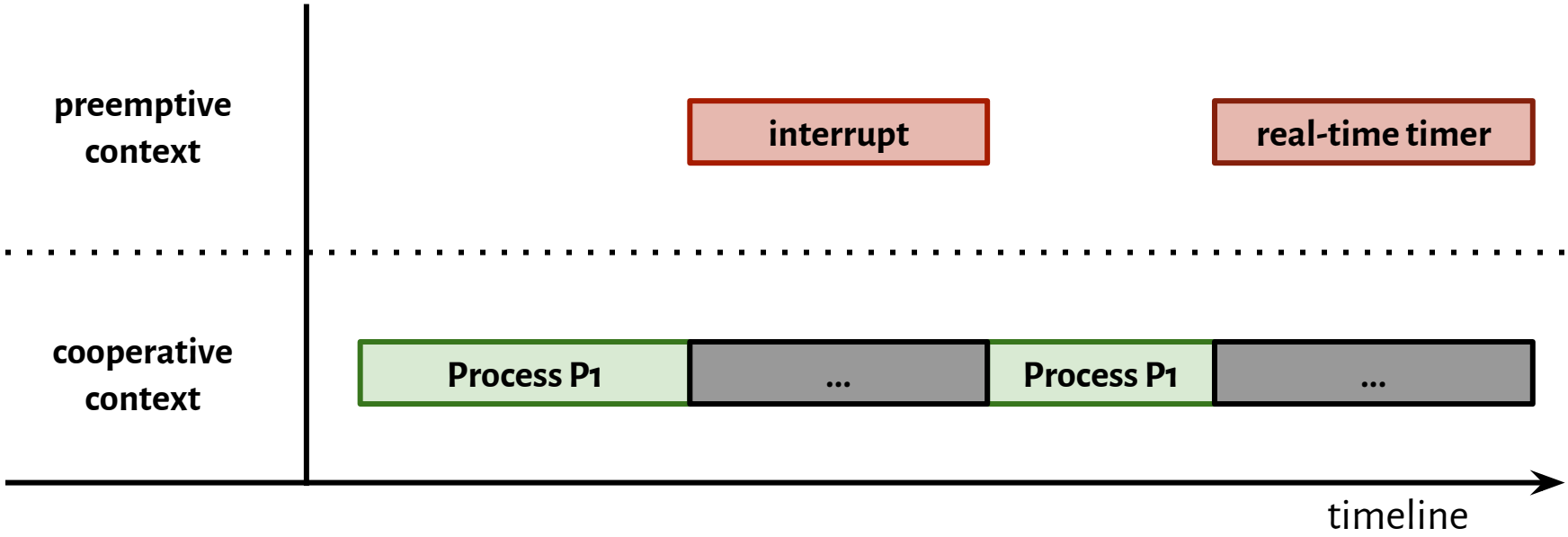
ctimer

rtimer

Contiki - processes

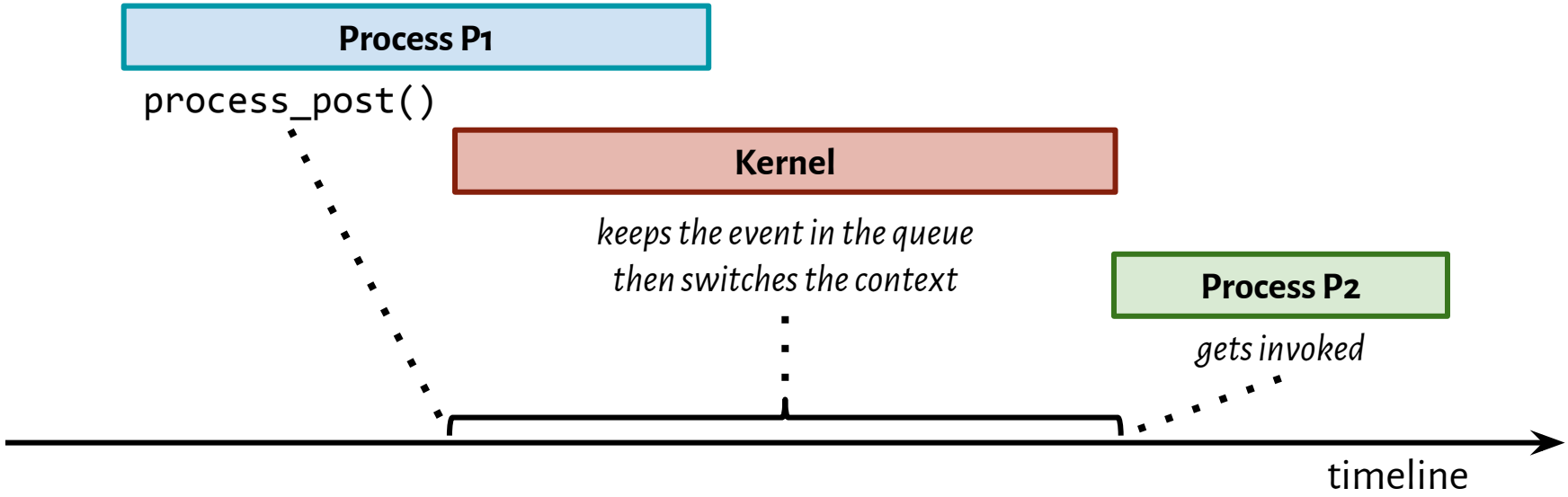
All Contiki programs are processes.

A ***process*** is a piece of code that is executed regularly by the Contiki system.



Contiki - Events

Asynchronous

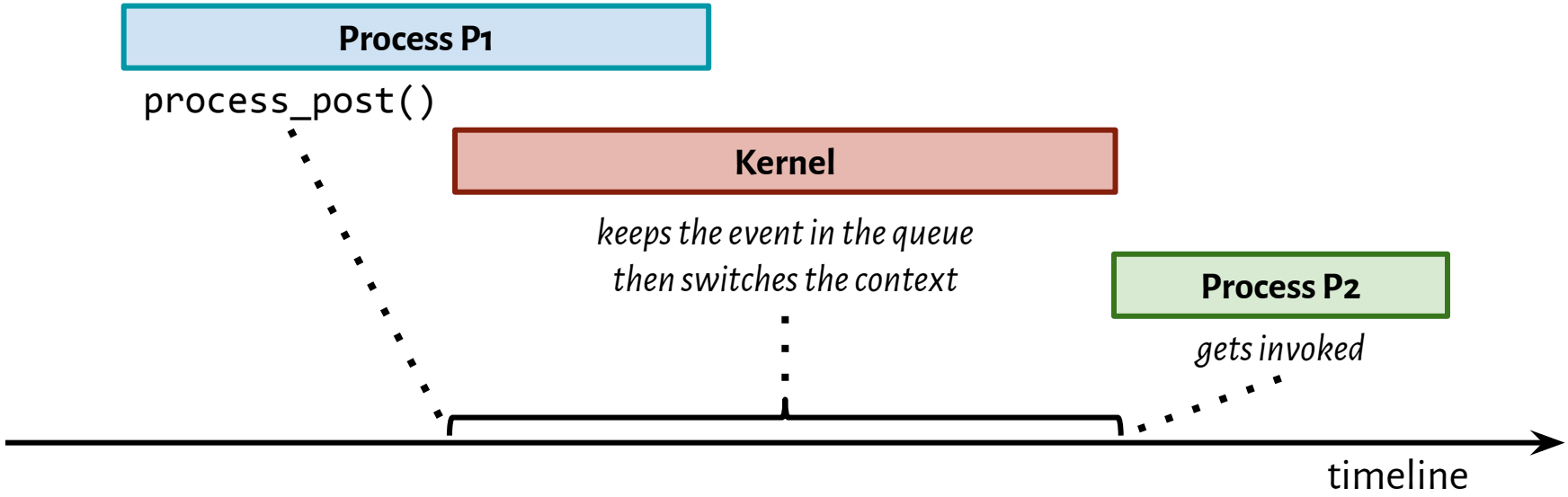


Contiki - Events

Asynchronous

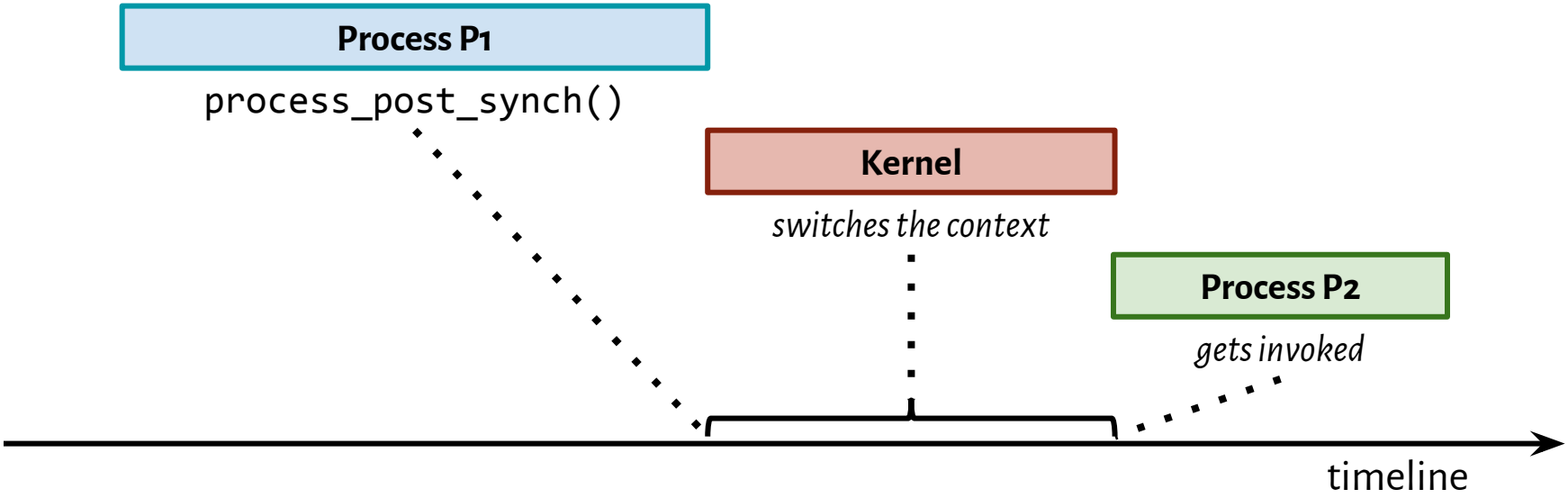
1. An asynchronous event can be delivered either to a specific process or to all running process.

2. If the event queue is full process_post() returns an error.



Contiki - Events

Synchronous



Contiki - Events

Identified with a 8-bit number.

<= 127 can be freely used within a user process

> 127 are intended to be used between different processes, managed by the kernel

The first numbers over 128 are statically allocated by the kernel, to be used for a range of different purposes.

```
#define PROCESS_EVENT_NONE          128
#define PROCESS_EVENT_INIT          129
#define PROCESS_EVENT_POLL          130
#define PROCESS_EVENT_EXIT          131
#define PROCESS_EVENT_CONTINUE      133
#define PROCESS_EVENT_MSG           134
#define PROCESS_EVENT_EXITED        135
#define PROCESS_EVENT_TIMER         136
```

Contiki - Events

`PROCESS_EVENT_POLL`

- ★ A process is polled by calling the function `process_poll()`.
- ★ It causes the process to be scheduled as quickly as possible.
- ★ Polling is the way to make a process run from an **interrupt**.

`PROCESS_EVENT_EXITED`

- ★ When a process exits the Contiki kernel sends a synchronous event to all other processes.
- ★ This can be used to free up any resource allocations made by the process that is exiting.
- ★ For example, the uIP TCP/IP stack will close and remove any active network connections that the exiting process has.