Contiki OS

Giacomo Tanganelli
PhD student @ University of Pisa
g.tanganelli@iet.unipi.it

WSN Operating Systems

- The OS hides many HW details
  - Simplify the programmer life
- Contains drivers to radio and sensors, scheduling, network stacks, process & power management
- TinyOS, Contiki, FreeRTOS, Mantis OS

Contiki overview

- Contiki is a dynamic operating system for constrained devices
- Event driven kernel
  - Protothreds on top of it
- Support for many platform
  - Tmote Sky, Zolertia Z1, MicaZ …
- Support for many CPU

Contiki core

- Is based on an enhanced event handler:
  - Loop that just takes the next event and processes it
  - Nothing to do->goes to sleep (MCU low power mode)
- Set of services
  - Networking, storage, timers …

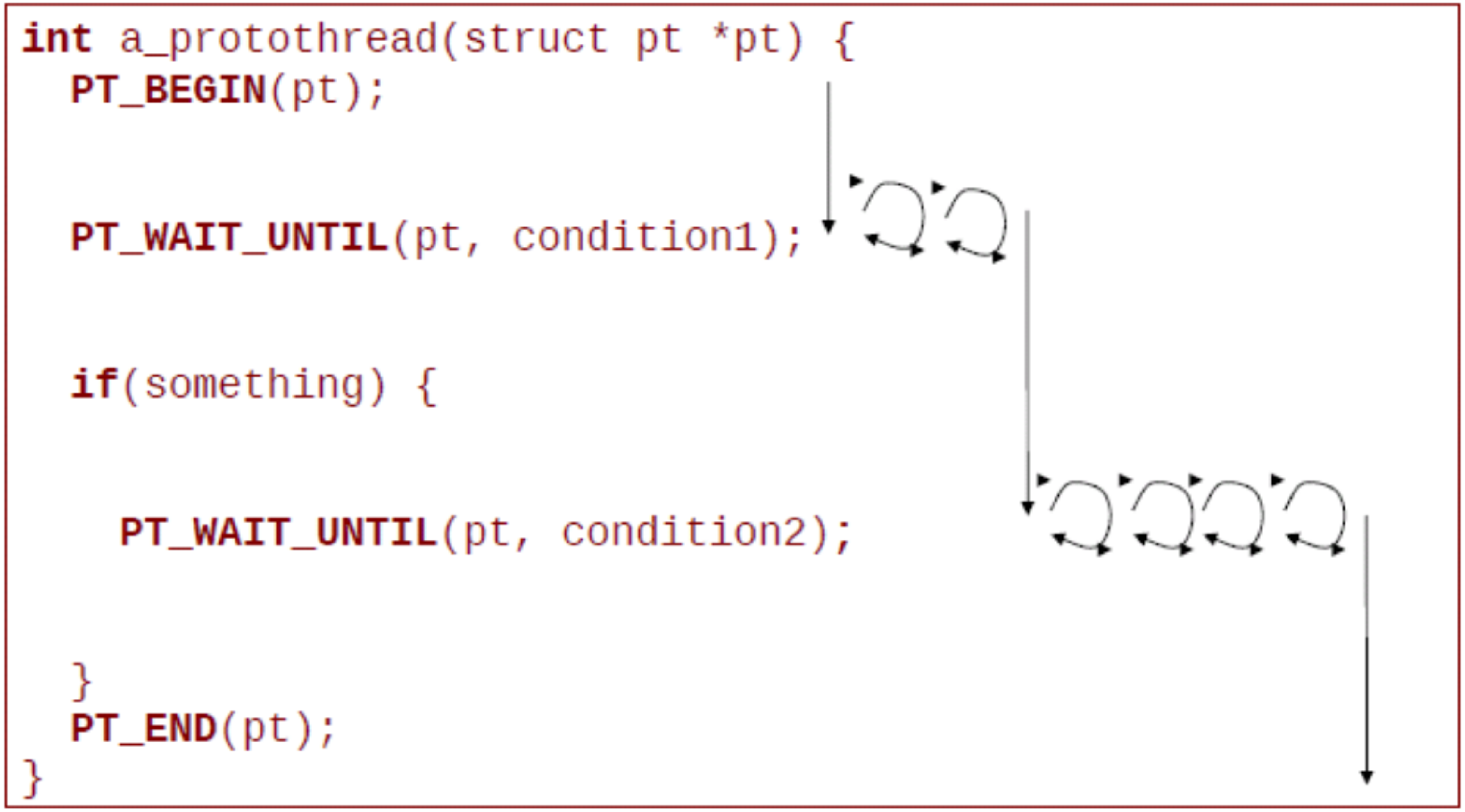
Event vs Thread

- Event driven kernel only use events
  - Difficult to program
  - No sequential flow of control
  - Low overhead
- Threads
  - Easy to program
  - Sequential flow of control
  - High overhead (more stacks)

Protothreads

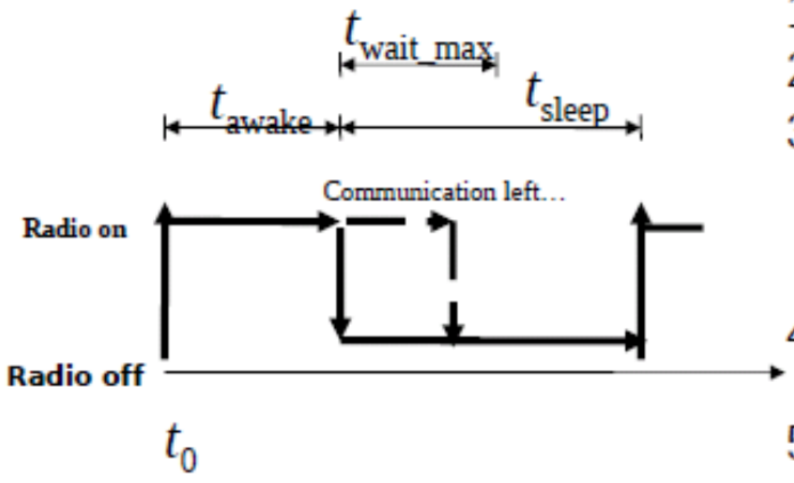
- Single stack
- Sequential flow of control
- Do not save local variable state across blocking calls
    - Overcome by use static variables

Protothreads



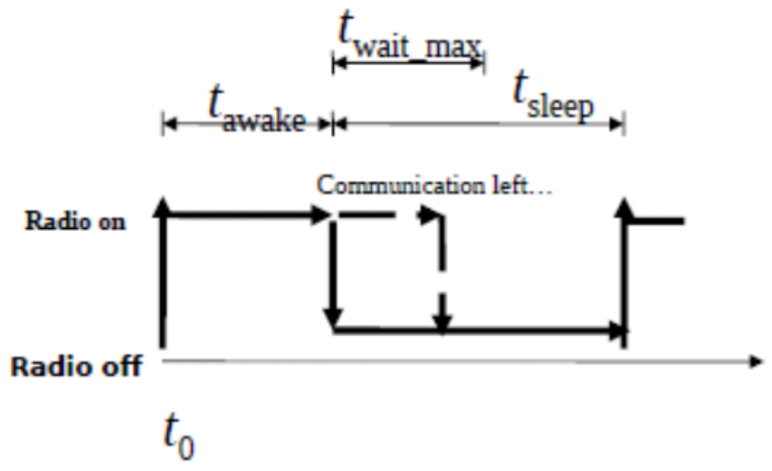
```
int a_protothread(struct pt *pt) {
  PT_BEGIN(pt);


  PT_WAIT_UNTIL(pt, condition1);


  if(something) {


    PT_WAIT_UNTIL(pt, condition2);


  }
  PT_END(pt);
}
```

Protothreads



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.

Protothreads



```
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);
}
```

Processes

- Processes are protothreads
- PROCESS_THREAD defines a new process
- Must start with PROCESS_BEGIN()
- Must end with PROCESS_END()
- Wait for new event:
  - PROCESS_WAIT_EVENT()
  - PROCESS_WAIT_EVENT_UNTIL(condition c)

Contiki directories

- core
  - System source code
- apps
  - System apps
- platform
  - Platform-specific code
    - Default mote configuration
- cpu
  - CPU-specific code
- example
  - Lots of examples. **USE** it as a starting point.
- tools
  - Cooja and other useful stuff

Hello World

```
#include “contiki.h”
#include <stdio.h>
/* Declare the process */
PROCESS(hello_world_process, “Hello world”);
/* Make the process start when the module is loaded */
AUTOSTART_PROCESSES(&hello_world_process);

/* Define the process code */
PROCESS_THREAD(hello_world_process, ev, data) {
        PROCESS_BEGIN(); /* Must always come first */
        printf(“Hello, world!\n”); /* code goes here. All printf must end
with \n */
        PROCESS_END(); /* Must always come last */
}
```

Makefile

```
CONTIKI_PROJECT = hello-world

all: $(CONTIKI_PROJECT)

CONTIKI = /home/user/contiki

include $(CONTIKI)/Makefile.include
```

project-conf.h

- Used to override default configurations
- Add to Makefile

```
CFLAGS += -DPROJECT_CONF_H=\"project-conf.h\“
```

- Example: change RDC protocol

```
#define NETSTACK_CONF_RDC nullrdc_driver
```

- See platform/z1/contiki-conf.h

Run a program on a mote

- Compile
  - make TARGET=z1 hello-world
- Upload to the mote
  - make TARGET=z1 hello-world.upload
- Useful stuff
  - make TARGET=z1 savetarget
    - Save a default target for the project
  - make motelist
    - Display the motes connected to the PC
  - make login
    - View the program output

Timers

- struct timer
  - Passive timer, only keeps track of its expiration time
- struct etimer
  - Active timer, sends an event when it expires
- struct ctimer
  - Active timer, calls a function when it expires
- struct rtimer
  - Real-time timer, calls a function at an exact time. Reserved for OS internals

POST and WAIT

- PROCESS_WAIT_EVENT();
  - Waits for an event to be posted to the process

- PROCESS_WAIT_EVENT_UNTIL(condition c);
  - Waits for an event to be posted to the process, with an extra condition. Often used: wait until timer has expired
  - PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));

- PROCESS_POST(...) and PROCESS_POST_SYNCH(..)
  - Post (a)synchronous event to a process.
  - The other process usually waits with PROCESS_WAIT_EVENT_UNTIL(ev == EVENTNAME);

Sensors

```
#include “dev/button-sensor.h”
#include “dev/leds.h”

SENSORS_ACTIVATE(button_sensor);

PROCESS_WAIT_EVENT_UNTIL(ev==sensors_event &&
data==&button_sensor);

leds_toggle(LEDS_ALL);
```

See example/sky/test-button.c

Exercise 1

- Write a program that loops indefinitely, check if a button has been pressed, and if so, toggles LEDs and prints out a message.

Timer functions

```
static struct etimer et;
etimer_set(&et, CLOCK_SECOND*4);


PROCESS_WAIT_EVENT();
If(etimer_expired(&et)){
        etimer_reset(&et);
}
```

Exercise 2 & 3

- Write a program that loops indefinitely, check if the timer has expired, and if so, toggles LEDs and prints out a message.
- Write a program that loops indefinitely, waits for an event, check if a button has been pressed, toggles LEDs and prints out “Button Press!”. If, instead, the timer has expired toggles LEDs and prints out “Timer!”