Redes de sensores inalámbricos (RSI)

Plataforma de software: Contiki-NG (parte 1)

Leonardo Steinfeld

Inst. de Ingeniería Eléctrica, Fac. de Ingeniería
Universidad de la República (Uruguay)

NEON
Network of Competence on Internet of Things

Co-funded by the Erasmus+ Programme of the European Union

FACULTAD DE INGENIERÍA

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

Objetivos

- Introducir principales conceptos de OS
- Escribir procesos utilizando eventos y timers
- Familiarizarse con el proceso de compilación (muy básico)
- Comprender el funcionamiento interno de procesos
- Comprender las limitaciones de la implmentación

Agenda

- Introducción OS
- Contiki-NG: características
- Protothreads
- Procesos
- Eventos
- Timers
- Post de eventos
- Implementación de procesos
- Estructura de directorios

Introducción

- Propiedades deseadas para un OS
  - Gestión de concurrencia (multi-tarea)
  - Gestión de recursos hardware
  - Gestión de los modos de bajo consumo
  - Estrategias para bajo ciclo de trabajo

Introducción

- Ventajas
  - Pila de comunicación y drivers integrados
  - Portabilidad – HAL
  - Modularidad y reuso
  - Facilita programación para aplicaciones complejas

OS para IoT


µC/OS-III®
The Real-Time Kernel


RIOT


Zephyr®
Project


freeRTOS


CONTIKI
NEXT GENERATION

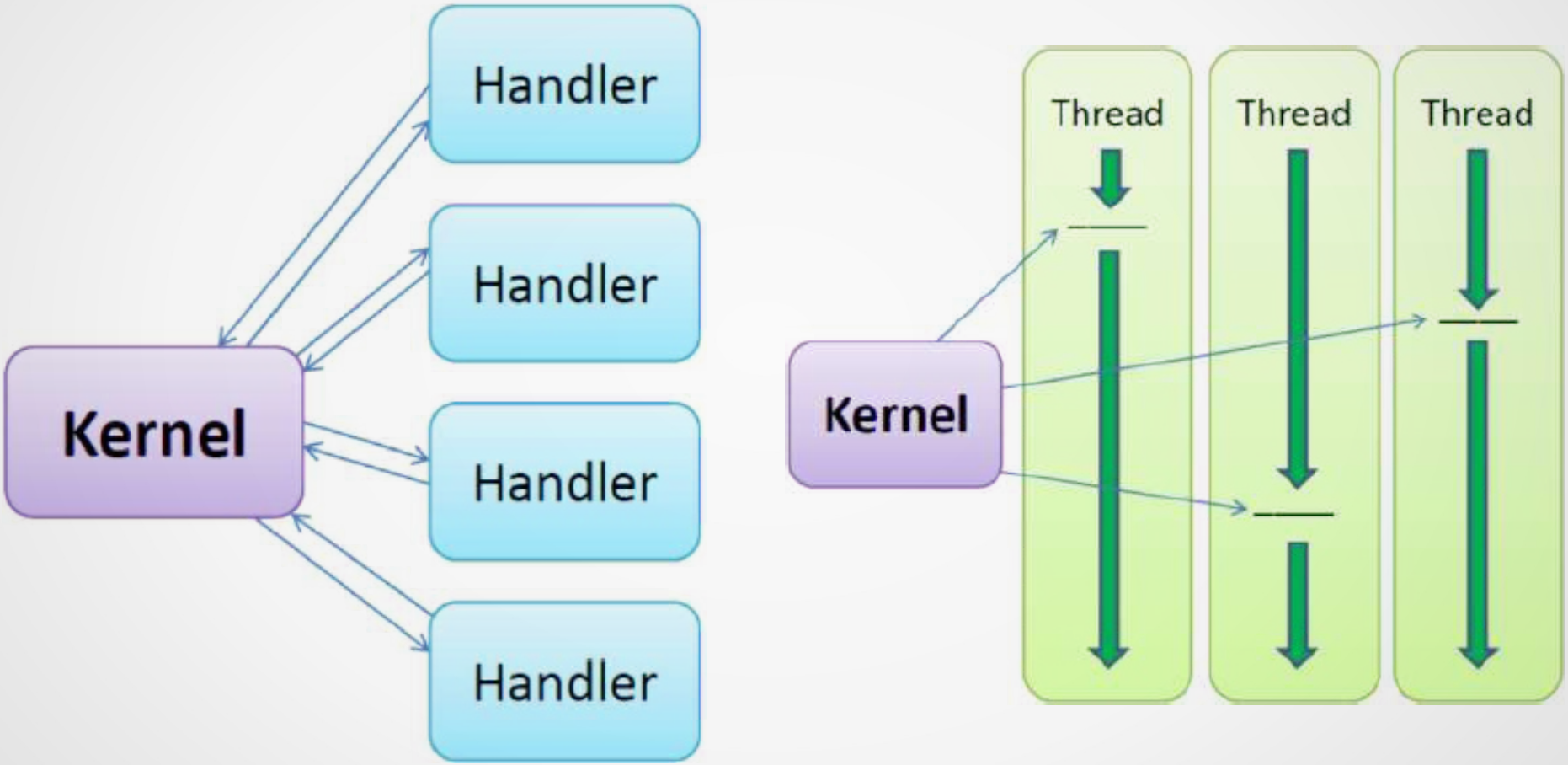
Introducción: Contiki-NG

- OS para dispositivos de recursos limitados de IoT
- Pila de comunicación (6LoWPAN, RFC-compliant)
  - Ipv6 / RPL, UDP (DTLS) / TCP, CoAP/MQTT/LWM2M, entre otros.
- Plataformas
  - ARM Cortex-M3/M4 (CC1350, CC2650, nrf52840, CC2538, gecko)
  - Texas Instruments MSP430
  - nRF
- Footprint
  - código ~ 100 kB
  - memoria ~10 kB
- Open source

Introducción

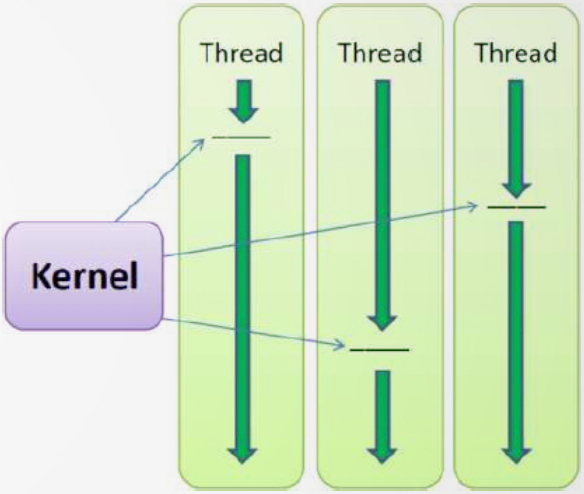
- Características: *scheduling*
  - colaborativo
  - expropiativo (preemtive scheduler)
- Arquitecturas
  - Event-driven
  - Multithreading

Comparación: event-driven & multithreading



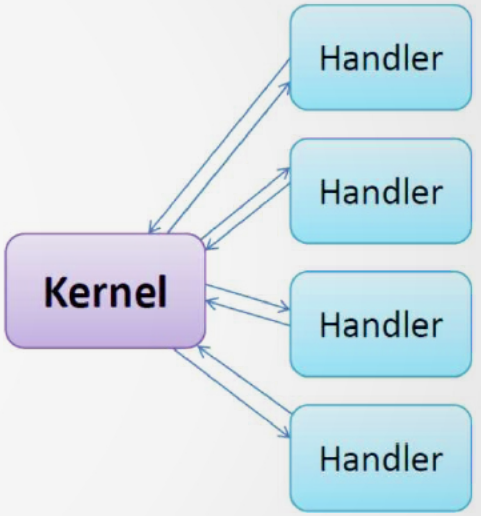
Motivación: multithreading

- ¿Cómo implementa concurrencia?
  - Procesos: implementados como hilos “concurrentes” y el procesador conmuta entre ellos.
- Pros:
  - Flujo secuencial de código.
  - Es posible expropiar el procesador
  - wait() es válido.
- Contras:
  - Overhead (código y tiempo ejecución)
  - Mecanismo de bloqueo: necesita código reentrante
  - Gran consumo de memoria (stack por thread)



Motivación: event-driven

- Como implementar concurrencia?
  - Procesos: implementados *handlers* de eventos que corren hasta el final (“run-to-completion”)
- Pros:
  - Uso eficiente de la memoria (stack único)
  - Overhead bajo de conmutación de contexto.
  - No necesitan mecanismo de bloqueo.
- Contra
  - No es posible expropiar el procesador
  - Programación state-driven difícil de manejar (callbacks recursivos)
  - No todos los programas son fácilmente escritos como FSM



Contiki-NG propone: protothreads

- Protothreads: una nueva abstracción de programación
  - Punto diseño intermedio entre event-driven y multithreading
  - Idea muy simple pero potente.
- Aspecto de multithreading
  - Primitiva de prog.: espera bloqueante condicional

    PT_WAIT_UNTIL(condition)
  - Provee control de flujo secuencial
- Aspecto de event-driven
  - Usa un solo stack
  - Requerimientos de memoria similares

Contiki-NG: prototheads

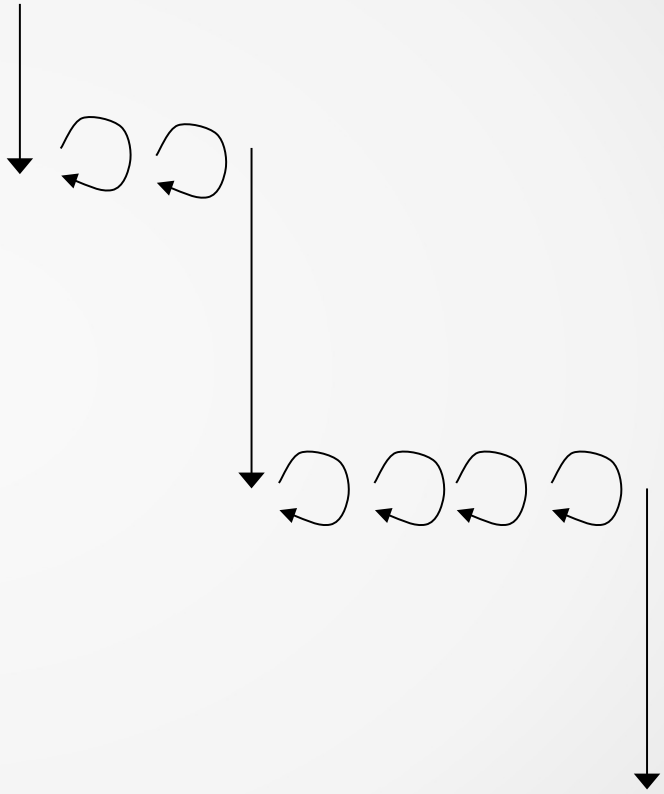
- Real Academia Española (www.rae.es)
  - **proto**-.

    (Del gr. πρωτο-, primero).

    1. elem. compos. Indica prioridad, preeminencia o superioridad. Protomártir, protomédico, prototipo.
- The American Heritage Dictionary of the English (answers.com)
  - **Proto or prot**-
    - *pref*.
      1. First in time; earliest: protolithic.
      2. First formed; primitive; original: protohuman.
      3. Proto- Being a form of a language that is the ancestor of a language or group of related languages: Proto-Germanic.
      4. Having the **least amount of a specified element** or radical: protoporphyrin. [Greek prōto-, from prōtos.]

Contiki-NG: 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);
}
```



ATENCIÓN:
al final vemos cómo se implementa y funciona internamente

Contiki-NG: procesos

- Procesos: asociado a un protothread
  - guardados en una lista encadenada
    - Información: nombre, estado, puntero al *thread*
    - estados: *desactivado*, *activado*, *ejecutando*

```
PROCESS(primer_proceso, "Nombre del primer proceso");
AUTOSTART_PROCESSES(&primer_proceso);

PROCESS_THREAD(primer_proceso, ev, data){
    PROCESS_BEGIN();
    printf("Primer proceso, imprimo y me voy.\n");
    PROCESS_END();
}
```

Contiki-NG: eventos

- Evento: desencadena la ejecución de un proceso
  - guardados en una cola circular
  - generados por procesos o interrupciones
  - *scheduler* llama al proceso, recibe: ID del evento, datos

```
PROCESS(primer_proceso, "Nombre del primer proceso");
AUTOSTART_PROCESSES(&primer_proceso);

PROCESS_THREAD(primer_proceso, ev, data){
    PROCESS_BEGIN();
    printf("Primer proceso, imprimo y me voy.\n");
    PROCESS_END();
}
```

Contiki-NG: eventos

- Tipos
  - asincrónicos
    - se encolan y después se despachan a procesos de forma *round-robin*
    - posteado a proceso específico o *broadcast*
    - función: **process_post()**
  - sincrónicos
    - se llama proceso inmediatamente (equivalente a llamar una función)
    - función: **proccess_post_synch()**

Contiki-NG: ejemplo proceso

```
PROCESS_THREAD(hello_world_process, ev, data){
    static struct etimer etimer1;

    PROCESS_BEGIN();

    etimer_set(&etimer1, CLOCK_CONF_SECOND);
    for(;;){
        PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
        printf("Hello, world\n");
        etimer_reset(&etimer1);
    }

    PROCESS_END();
}
```

- Proceso:
  - inactivo hasta que recibe un evento
  - proceso ejecuta
  - suspende su propia ejecución (se “bloquea”) hasta nuevo evento

Contiki-NG: arranque de procesos

```
PROCESS_THREAD(hello_world_process, ev, data){
    static struct etimer etimer1;

    PROCESS_BEGIN();

    etimer_set(&etimer1, CLOCK_CONF_SECOND);
    for(;;){
        PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
        printf("Hello, world\n");
        etimer_reset(&etimer1);
    }

    PROCESS_END();
}
```

- Arranque:
  - otro proceso llama
    - **process_start()**
  - se incluye
    - **AUTOSTART_PROCESS()**
- Se le envía evento
    - **PROCESS_EVENT_INIT**

Contiki-NG: finalización de procesos

```
PROCESS_THREAD(hello_world_process, ev, data){
    static struct etimer etimer1;

    PROCESS_BEGIN();

    etimer_set(&etimer1, CLOCK_CONF_SECOND);
    for(;;){
        PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
        printf("Hello, world\n");
        etimer_reset(&etimer1);
    }

    PROCESS_END();
}
```

- Finalización:
  - otro proceso llama
    - **process_exit()**
  - cuando se llega a
    - **PROCESS_END()**
  - Se envía a demás procesos
    - **PROCESS_EVENT_EXITED**

Ejercicio: compilación primera aplicación

1. Crear carpeta **rsi** en **contiki-ng/examples**
2. Crear **hola.c**
3. Crear **Makefile**
4. Compilar

   $make
5. Ejecutar

   $./hola.native

```
# archivo: Makefile
CONTIKI_PROJECT = hola
all: $(CONTIKI_PROJECT)

CONTIKI = /home/leo/work/contiki-ng/
include $(CONTIKI)/Makefile.include
```

```
// archivo: hola.c
#include "contiki.h"
#include <stdio.h>

PROCESS(primer_proceso, "Nombre del primer proceso");
AUTOSTART_PROCESSES(&primer_proceso);

PROCESS_THREAD(primer_proceso, ev, data){
    PROCESS_BEGIN();
    printf("Primer proceso, imprimo y me voy.\n");
    PROCESS_END();
}
```

Contiki-NG: timers

- Bibliotecas timers: **5 tipos**
  - utilizadas por OS y aplicaciones (usuarios)
  - funcionalidad común:
    - chequear si transcurrió un cierto período de tiempo.
  - construidas a partir de módulo **clock**
- funciones comunes (x= _, s, e, c, r)
  - setting: void **x**timer_set(struct timer *t, clock_time_t interval)
  - resetting: void **x**timer_reset(struct timer *t)
  - restarting: void **x**timer_restart(struct timer *t)
  - checking: int **x**timer_expired(struct timer *t)

Contiki-NG: timers

- timer:
  - simple, no se notifica el usuario consulta
- stimer:
  - igual a timer, pero en segundos (tiempos mayores)
- etimer:
  - genera un evento PROCESS_EVENT_TIMER cuando expira
- ctimer:
  - llama a una función cuando expira
- rtimer
  - ejecución tareas de tiempo real (ahora usado por TSCH)

Contiki-NG: etimers

- Módulo para manejo de tiempos
  - Estructura etimers
    - tiempo inicial
    - Intervalo
    - puntero al proceso a despertar
  - Guardada como lista encadenada de etimers
  - Módulo implementado como proceso
    - Despertado desde una interrupción
    - Recorre lista de etimers en busca de alguno que haya expirado

Contiki-NG: etimers

- Funciones
  - void etimer_set(struct etimer *t, clock_time_t interval)
  - void etimer_reset(struct etimer *t)
  - void etimer_restart(struct etimer *t)
  - void etimer_stop(struct etimer *t)
  - int etimer_expired(struct etimer *t)
  - int etimer_pending()
  - clock_time_t etimer_next_expiration_time()
  - void etimer_request_poll()` setting
- Variable de usuario:
  - static struct etimer etempo;

Contiki-NG: ejemplo etimers

```c
#include "sys/etimer.h"

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

  /* Delay 1 second */
  etimer_set(&et, CLOCK_SECOND);

  while(1) {
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
    /* Reset the etimer to trig again in 1 second */
    etimer_reset(&et);
    ...
  }
  PROCESS_END();
}
```

Ejercicio: timers

- Para los dos ejercicios partir del examples/hello-world
  1. Modificar hello-world.c para que
     - el timer cuente 2 seg.
     - imprima los segundos y la cantidad de veces que expira.
  2. En lugar de usar **etimer** implmentar la misma funcionalidad con un **ctimer**.
- Preguntas:
  - ¿Qué pasa si la variable del tiempo no es estática?
  - ¿Qué pasa si inicializo el timer antes de la declaración de PROCESS_BEGIN()?

Contiki-NG: post de eventos

- Proceso **origen**: post de un evento a otro proceso
  1. Declarar el evento:

     process_event_t my_event;
  2. Asignar memoria para el evento mediante la función:

     ```
     my_event  = process_alloc_event();
     ```
  3. Postear el proceso pasando un puntero a datos:

     ```
     process_post(&proceso_destino, mi_evento, &var);
     ```
- Proceso **destino**: recibe el evento y dato como parámetros
  - Para verificar recepción de evento: ev == my_event.

Ejercicio: post eventos

- Modificar el hello-world.c para agregar un nuevo proceso

1. Proceso: hello-world-origen

   - Cada 5 segundos postea un evento

2. Proceso: hello-world-destino

   - Espera el evento y cuando lo recibe imprime.

Notas:

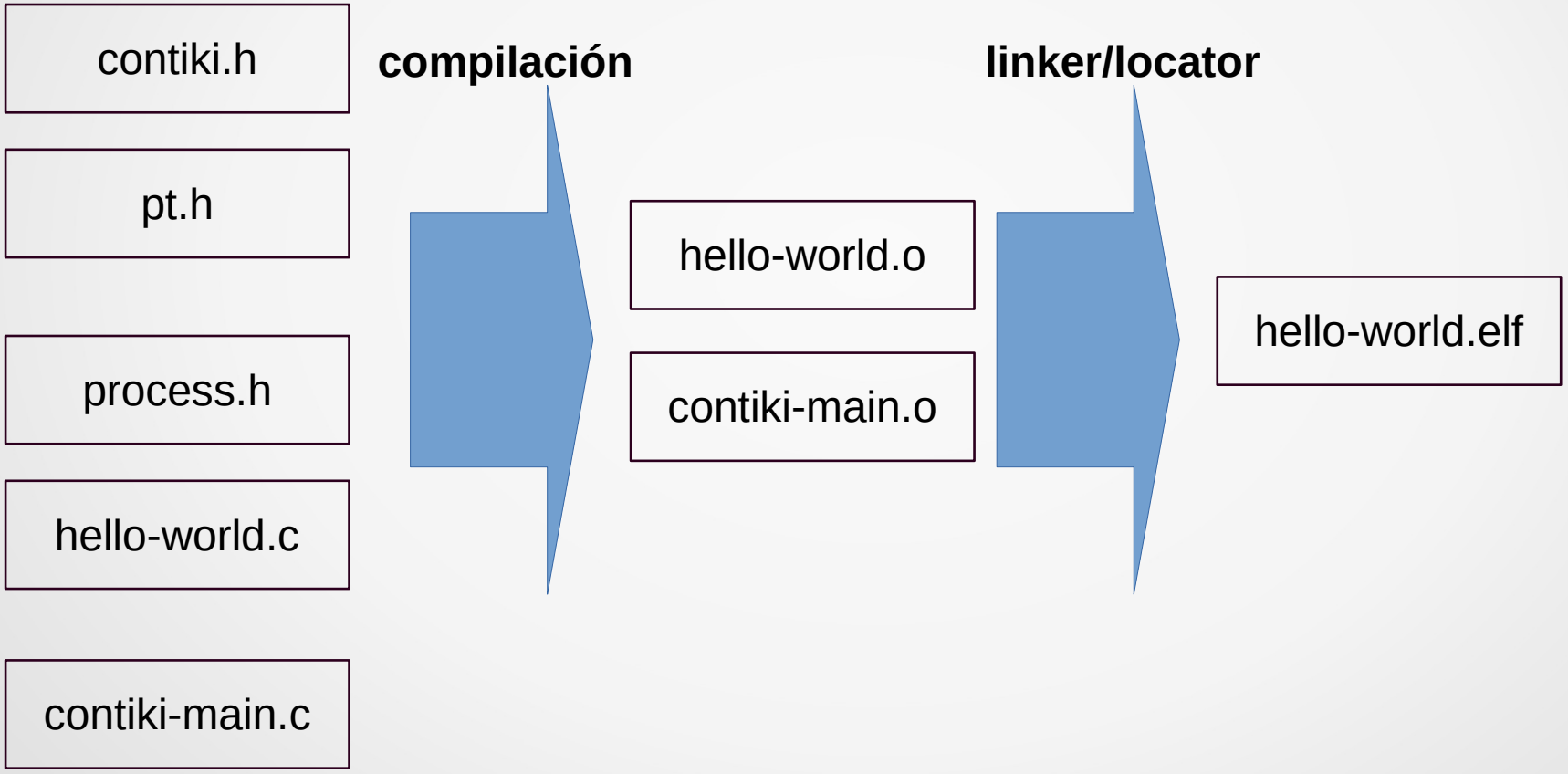
1. Seguir las instrucciones de la página anterior.
2. Si entendieran necesario, agregar prints adicionales.
3. [Opcional] Enviar como dato la cantidad de veces itera (vence el timer y realiza un post). Se debe *castear* al tipo de datos correcto.

   Ejemplo: si el dato es un entero de 16 bits

```
int16_t num = *(int16_t*)data;
```

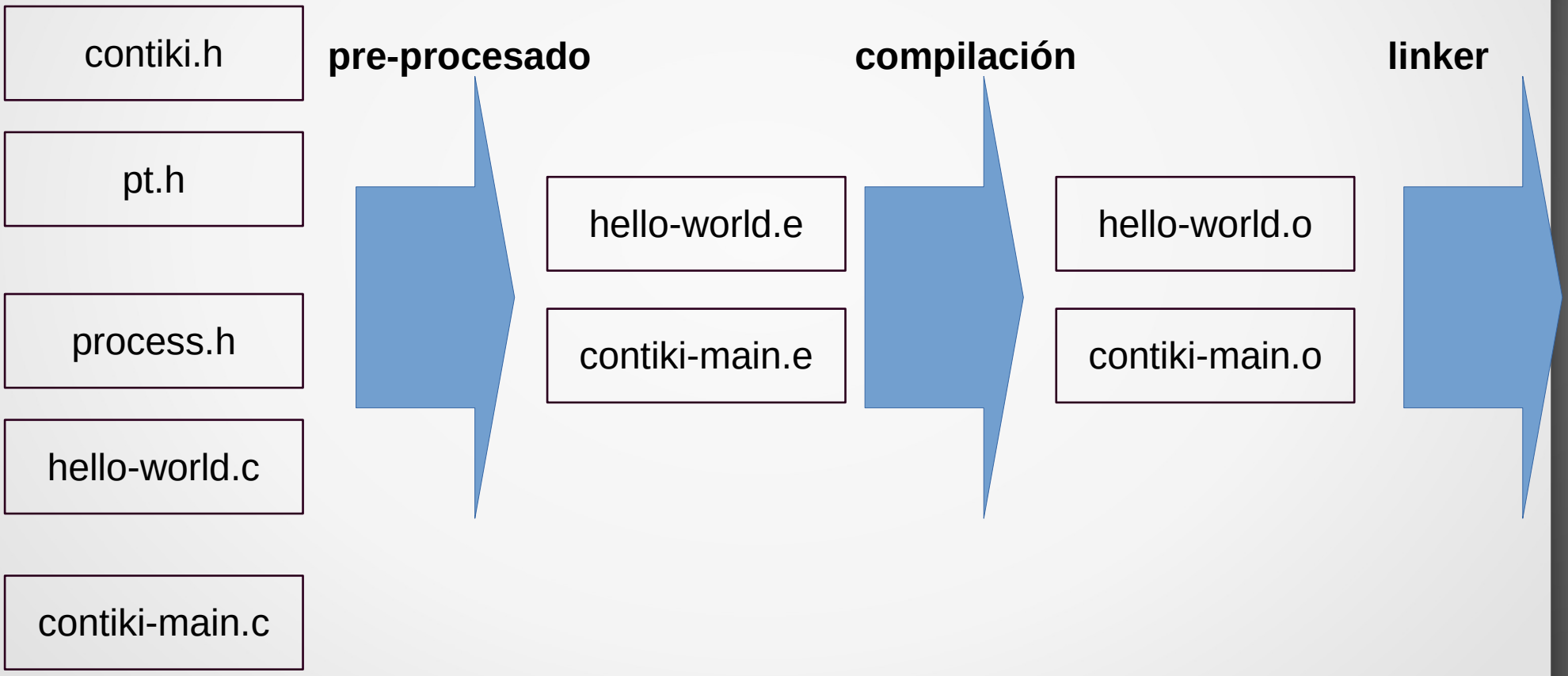
Contikti-NG: implementación protothreads

- Explicación simplificada



Contikti-NG: implementación protothreads

- Explicación simplificada



Contiki-NG: finalización de procesos

```
PROCESS_THREAD(hello_world_process, ev, data){
    static struct etimer etimer1;

    PROCESS_BEGIN();

    etimer_set(&etimer1, CLOCK_CONF_SECOND);
    for(;;){
        PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
        printf("Hello, world\n");
        etimer_reset(&etimer1);
    }

    PROCESS_END();
}
```

Contiki-NG: “magia”

```
// Fragmentos de los siguientes archivos:
//   pt.h
//   process.h
//   lc-switch

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

struct pt {
  lc_t lc;
};

#define PROCESS_BEGIN() { char PT_YIELD_FLAG = 1; if (PT_YIELD_FLAG) {;} LC_RESUME((pt)->lc)

#define PT_WAIT_UNTIL(pt, condition)           \
  do {                        \
    LC_SET((pt)->lc);                \
    if(!(condition)) {               \
      return PT_WAITING;             \
    }                         \
  } while(0)

#define PROCESS_END() LC_END((pt)->lc); PT_YIELD_FLAG = 0; \
                      PT_INIT(pt); return PT_ENDED; }
```

Contiki-NG: proceso pre-procesado

```
static char process_thread_hello_world_process(
    struct pt *process_pt, process_event_t ev, process_data_t data){

static struct etimer etimer1;
{
  char PT_YIELD_FLAG = 1;
  switch((process_pt)->lc) {
  case 0:;
    etimer_set(&etimer1, 128);
    for(;;){
      do {
        PT_YIELD_FLAG = 0;
        (process_pt)->lc = 28;
  case 28:;
    if((PT_YIELD_FLAG == 0) || !(ev == 0x88 {
      return 1;
    }
      } while(0);
    printf("Hello, world\n");
    etimer_reset(&etimer1);
    }
};
PT_YIELD_FLAG = 0;
(process_pt)->lc = 0;
return 3
}
```

Contiki-NG: resultado

```
#include "pt.h"
#include <stdio.h>
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);


PROCESS_THREAD(hello_world_process, ev, data){

  static struct etimer etimer1;

  PROCESS_BEGIN();


  etimer_set(&etimer1, CLOCK_CONF_SECOND);
  for(;;){
    PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
    printf("Hello, world\n");
    etimer_reset(&etimer1);
  }

  PROCESS_END();
}
```

```
// omitido
// para
// simplificar

static char process_thread_hello_world_process(
    struct pt *process_pt, process_event_t ev, process_data_t data){

static struct etimer etimer1;
{
  char PT_YIELD_FLAG = 1;
  switch((process_pt)->lc) {
  case 0:;
    etimer_set(&etimer1, 128);
    for(;;){
      do {
        PT_YIELD_FLAG = 0;
        (process_pt)->lc = 28;
  case 28:;
    if((PT_YIELD_FLAG == 0) || !(ev == 0x88 {
      return 1;
    }
      } while(0);
    printf("Hello, world\n");
    etimer_reset(&etimer1);
    }

};
PT_YIELD_FLAG = 0;
(process_pt)->lc = 0;
return 3
}
```

Contiki-NG: fragmento loop principal

```
int main(){

// Mucho código

    for(;;){
            do {
                // Devuelve cantidad de eventos pendientes
                r = process_run();
            } while(r > 0);
            // Enter in Low Power Mode
            __bis_SR_register(LPM3_bits);
    }
// Sigue más código

}
```

Contiki-NG: estructura de directorios

- os
  - primitivas del sistema (procesos y timers), pila de comunicación, etc.
- arch
  - dependiente de hardware (incluye CPU, drivers de platiormas)
- examples
  - proyectos prontos para usar (empezar basándose en un ejemplo)
- tools
  - no incluidos en el firmware del dispositivos, sino para PC
- tests
  - para integración contínua (se ejectura con cada pull request / merge)

Contiki-NG: estructura de directorios

- os
  - sys
    - protothreads, procesos, timers, logs, etc.
  - dev
    - drivers generales (restantes en arch).
  - lib
    - bibliotecas: listas, colas, etc.
  - net
    - implementación de los protocolos de cada capa
  - services
    - módulos con procesos, usados por Conitiki-NG o usuarios
  - storage
    - sistema de archivos coffee

Referencias

- Programming Contiki-NG
  - Processes and events
  - Repository structure
  - Timers
- Tutoriales
  - Hello, World!
  - Timers and events

Planificación clases

1) Introducción RSI
2) Plataformas de hardware
3) Arquitectura 6LoWPAN (IPv6)
4) **Plataforma de software: Contiki-NG (parte 1)**
5) Plataforma de software: Contiki-NG (parte 2)
6) Capa de aplicación: CoAP / MQTT
7) Capa de red: RPL
8) MAC
9) IEEE 802.15.4 / 6lowpan
10) Capa Fisica & antenas
11) IoT y las RSI

¿más preguntas?