

Introduzione a Contiki, Processi e Protothread, Eventi

Ing. Simone Brienza

E-mail: simone.brienza@for.unipi.it

Pervasive Computing & Networking Lab (PerLab) <http://www.perlab.it>

Dipartimento di Ingegneria dell'Informazione, Università di Pisa



Outline Esercitazioni



- **Introduzione a Contiki**
- **Processi e Protothread**
- **Eventi**
- **Ambiente di simulazione Cooja**
- **Implementazione su Tmote Sky**
- **Gestione Input e Output da nodi sensori**
- **Timer**
- **Allocazione di Memoria e Librerie**
- **Comunicazioni di rete (IPv4)**
- **Esercizi e applicazioni per Smart Cities**



Introduzione a Contiki

Wireless Sensor Networks



- **Insiemi di nodi sensori, ossia dispositivi elettronici integrati, che comprendono:**
 - **Sensori (e attuatori)**
 - **Microcontrollore**
 - **Memoria limitata**
 - **Radio per comunicazioni wireless**
 - **Alimentazione (batterie)**
- **I nodi formano reti che trasmettono i dati misurati dai sensori ad una base station, in modo single-hop o multi-hop**

Sistemi operativi per WSNs



- Il SO è l'interfaccia tra l'hardware e il programmatore
 - Nasconde molti dettagli implementativi
- Gestisce:
 - driver per la radio e i sensori
 - scheduling
 - gli stack di rete
 - i processi
 - power management
- Limitata interazione con l'utente
 - A causa dei vincoli di memoria e piattaforma (embedded)
- Esempi di SO per WSN:
Contiki, TinyOS, LiteOS, Nano-RK, Mantis OS

Contiki: Caratteristiche



Contiki:

Un sistema operativo dinamico per sistemi embedded in rete

- Piccola impronta di memoria
- Kernel event-driven
- Progettato per la portabilità
 - Supporto per molte piattaforme (Tmote Sky, Zolertia, RedBee etc.)
- I programmi per Contiki sono scritti in C
- Usato sia nel mondo accademico che nell'industria
 - Cisco e Atmel fanno parte del *Contiki project*

Contiki: Funzioni principali



- **Protothreads** (Multithreading opzionale)
- **uIP**: stack TCP/IP per comunicazioni wireless low power
 - 6lowpan
 - IPV4 e IPv6 ready
- **Allocazione dinamica della memoria**
- **Power profiling**
- **Dynamic loading e over-the-air programming**
- **IPsec, on-node database Antelope, Coffee file system, ...**

Contiki: System Overview (1/2)



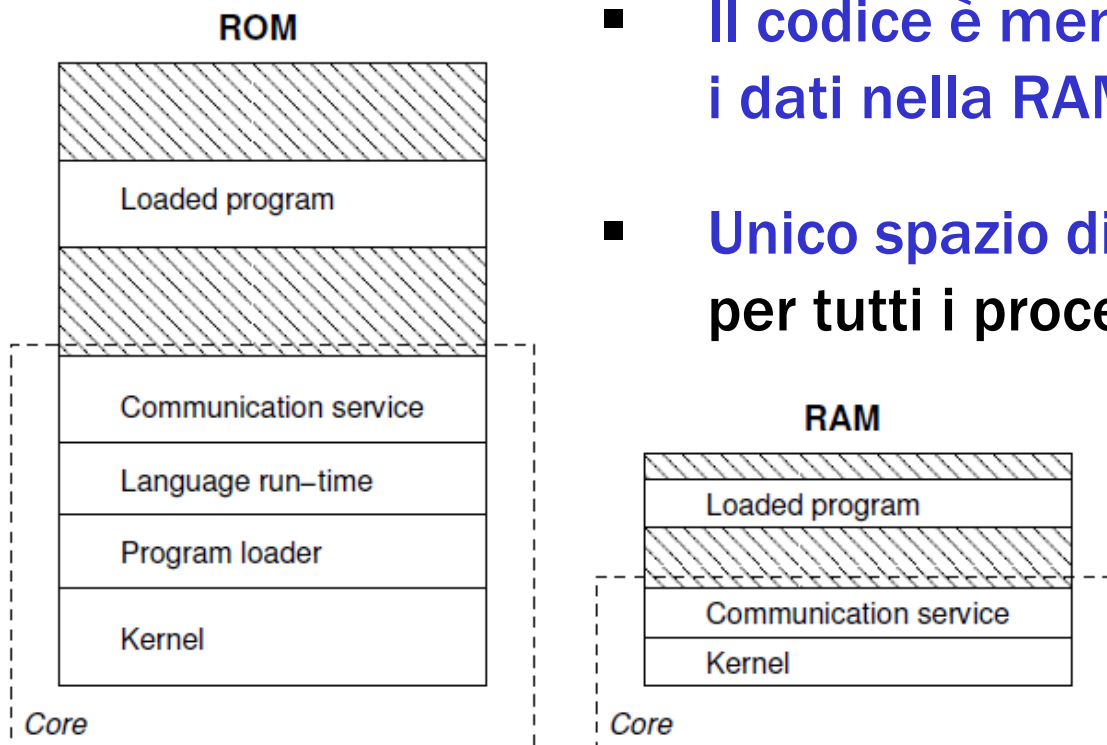
Un sistema **Contiki** in esecuzione consiste in:

- **Kernel**, in particolare uno **scheduler** (*event handler*)
 - Inter-process communication mediante eventi
- **Librerie**
- **Program loader**
- un insieme di processi
 - **Servizi e applicazioni** (caricabili anche dinamicamente)
- **Bootloader** per l'avvio di Contiki

Contiki: System Overview (2/2)



- Si distinguono due parti:
 - Core
 - Loaded programs
- Il codice è memorizzato nella ROM, i dati nella RAM
- Unico spazio di indirizzamento per tutti i processi



Contiki: Code size



Module	Code size (AVR)	Code size (MSP430)	RAM usage
Kernel	1044	810	$10 + 4e + 2p$
Service layer	128	110	0
Program loader	-	658	8
Multi-threading	678	582	$8 + s$
Timer library	90	60	0
Replicator stub	182	98	4
Replicator	1752	1558	200
Total	3874	3876	$230 + 4e + 2p + s$

Size of the compiled code, in bytes.

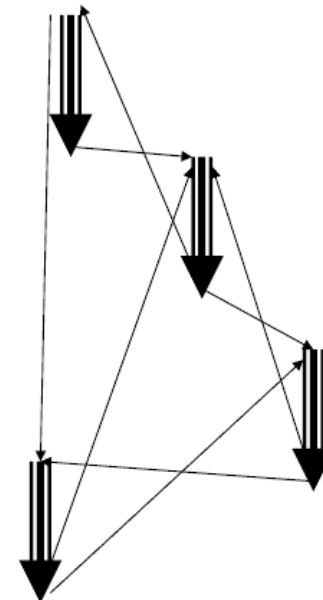
- e dimensione massima della coda degli eventi asincroni
- p numero massimo di processi per cui il sistema è stato configurato
- s dimensione degli stack dei thread (se il multithreading è abilitato)

Protothreads

Sistemi Event-driven



- programmi → insieme di *event handler*
- gli *event handler* sono invocati in risposta ad **eventi esterni**:
 - sono delle subroutine: eseguono un'azione e ritornano al chiamante
 - semantica *run-to-completion*: non possono eseguire un'attesa bloccante
- difficoltà nel programmare operazioni di alto livello: **macchine a stati**



Sistemi Event-driven



- Problema: in un programma event-driven il *code flow* è complesso
- Esempio: stop-and-wait sender

```
void reliableSend(pkt) {  
    call Unreliable.send(pkt);  
}
```

```
event void Unreliable.sendDone(pkt) {  
    call Timer.start(timeout);  
}
```

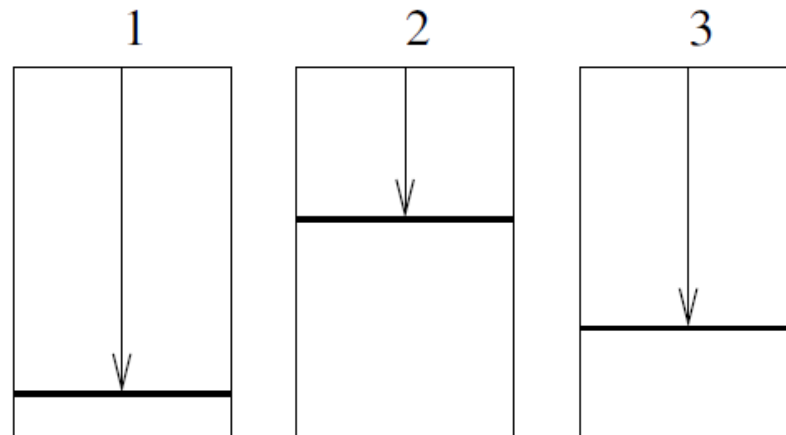
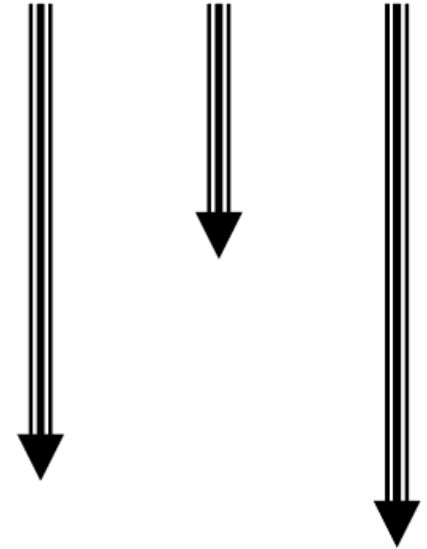
```
event void Timer.fired() {  
    call Unreliable.send(pkt);  
}
```

```
event void Receiver.receive(r) {  
    if (is_ack(r)) call Timer.stop();  
}
```

Sistemi Multi-threaded



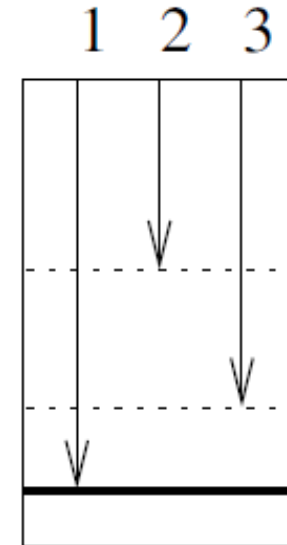
- **Code flow sequenziale** e più intuitivo
- Possono bloccarsi in attesa che una condizione si avveri
- Problema:
ogni thread richiede il suo proprio stack



Protothread: Motivazioni



- **Thread-style programming**
per sistemi con risorse limitate
- **Unico stack per tutti i protothread**
- ***Conditional blocking***,
implementato al di sopra di un
event-driven kernel
- si semplifica notevolmente la programmazione perché si
può ridurre il numero di macchine a stati programmate
esplicitamente



- Esemplio: stop-and-wait sender

```
PROCESS_THREAD(reliable_sender, ...) {  
    PROCESS_THREAD_BEGIN();  
  
    do {  
        PROCESS_WAIT_UNTIL(data_to_send());  
        send(pkt);  
        timer_start();  
        PROCESS_WAIT_UNTIL((ack_received() || timer_expired()));  
    } while (!ack_received());  
  
    PROCESS_THREAD_END();  
}
```


Esempio: MAC protocol (2/3)



```
state: {ON, WAITING, OFF}
```

```
radio_wake_eventhandler:
```

```
  if (state = ON)
```

```
    if (expired(timer))
```

```
      timer  $\leftarrow$   $t_{sleep}$ 
```

```
      if (not communication_complete())
```

```
        state  $\leftarrow$  WAITING
```

```
        wait_timer  $\leftarrow$   $t_{wait\_max}$ 
```

```
    else
```

```
      radio_off()
```

```
      state  $\leftarrow$  OFF
```

```
  elseif (state = WAITING)
```

```
    if (communication_complete() or  
        expired(wait_timer))
```

```
      state  $\leftarrow$  OFF
```

```
      radio_off()
```

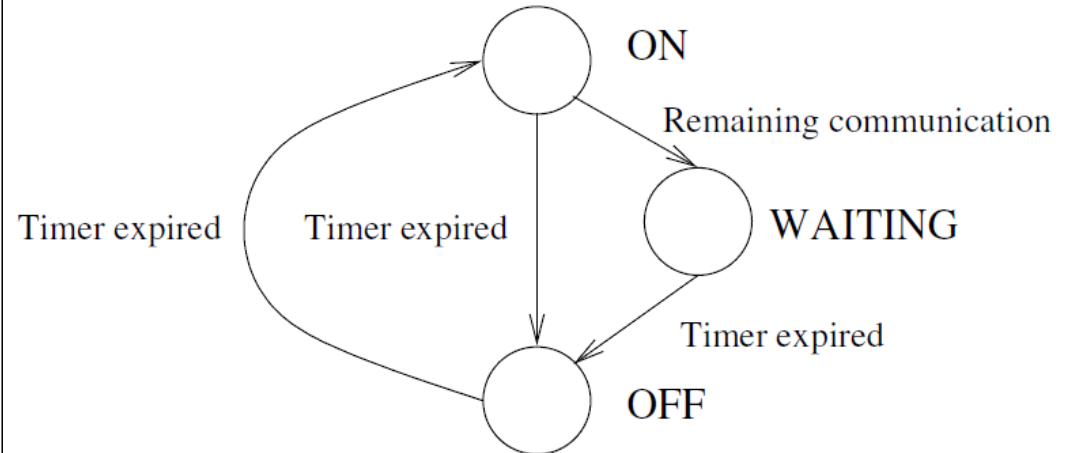
```
  elseif (state = OFF)
```

```
    if (expired(timer))
```

```
      radio_on()
```

```
      state  $\leftarrow$  ON
```

```
      timer  $\leftarrow$   $t_{awake}$ 
```



- **Macchina a stati finiti e implementazione con eventi**

Esempio: MAC protocol (3/3)



```
radio_wake_protthread:  
PT_BEGIN  
while (true)  
    radio_on()  
    timer ←  $t_{awake}$   
    PT_WAIT_UNTIL(expired(timer))  
    timer ←  $t_{sleep}$   
    if (not communication_complete())  
        wait_timer ←  $t_{wait\_max}$   
        PT_WAIT_UNTIL(communication_complete() or  
                      expired(wait_timer))  
  
    radio_off()  
    PT_WAIT_UNTIL(expired(timer))  
PT_END
```

- implementazione con protothread

- *PT_THREAD()* **Dichiara un protothread.**
- *PT_INIT()* **Inizializza il protothread.**
- *PT_BEGIN()* **Dichiara l'inizio del protothread nella funzione che lo implementa.**
- *PT_END()* **Dichiara la fine del protothread.**
- *PT_EXIT()* **Esce dal protothread.**
- *PT_WAIT_UNTIL()* **Aspetta il verificarsi di una certa condizione. Può essere bloccante.**
- *PT_YIELD()* **Blocca il protothread incondizionatamente. Continua alla successiva chiamata.**
- *PT_SPAWN()* **Un protothread può lanciare un protothread figlio che esegue un'altra funzione bloccante.**
 - *il padre si blocca fino a che il figlio non termina*
 - *il figlio è schedulato dal padre ogni volta che il padre viene chiamato*
 - *lo stato del figlio è memorizzato in una variabile locale del padre*

- Quando un protothread si blocca, lo stato del protothread è memorizzato in una **continuazione locale**
- **Lo stato è il punto attualmente in esecuzione** della funzione che costituisce il protothread
- **Lo stack non viene salvato**
- **2 operazioni:**
 - ***set***: salva lo stato della funzione
 - ***resume*** : ristabilisce lo stato della continuazione precedentemente salvato dalla set

Implementazione C (1/3)



- **Gli statement per i protothread sono implementati come macro del preprocessore C**

```
struct pt { lc_t lc };
#define PT_WAITING 0
#define PT_EXITED 1
#define PT_ENDED 2
#define PT_INIT(pt) LC_INIT(pt->lc)
#define PT_BEGIN(pt) LC_RESUME(pt->lc)
#define PT_END(pt) LC_END(pt->lc); \
return PT_ENDED
#define PT_WAIT_UNTIL(pt, c) LC_SET(pt->lc); \
if(!(c)) \
return PT_WAITING
#define PT_EXIT(pt) return PT_EXITED
```

Implementazione C (2/3)



- **Le continuazioni locali sono implementate utilizzando in modo non banale il costrutto `switch`**

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

Implementazione C (3/3)



- Esempio di codice C espanso con continuazioni locali implementate tramite costrutto switch

```
1 int sender(pt) {
2     PT_BEGIN(pt);
3
4     /* ... */
5     do {
6
7         PT_WAIT_UNTIL(pt,
8                     cond1);
9
10    } while(cond);
11    /* ... */
12    PT_END(pt);
13
14 }
```

```
int sender(pt) {
    switch(pt->lc) {
    case 0:
        /* ... */
        do {
            pt->lc = 8;
        case 8:
            if(!cond1)
                return PT_WAITING;
        } while(cond);
        /* ... */
    }
    return PT_ENDED;
}
```


Protothread: Limitazioni



- **le variabili locali automatiche non vengono salvate tra due chiamate successive!**
 - è possibile usare variabili statiche o globali
- **Il programmatore non può usare costrutti switch**
- **Possono chiamare funzioni ma non possono bloccarsi all'interno di una funzione chiamata**
 - Deve essere generato un nuovo protothread per ogni funzione bloccante (protothread gerarchici)
- **La sospensione è esplicita: il programmatore deve sapere quale funzione si può bloccare**
- ***Preemption* non possibile tra protothread**

Processi

- Una sezione di codice eseguito da Contiki
- Avviato al boot del sistema o quando un modulo che contiene un processo è caricato nel sistema
- È eseguito **in seguito ad un evento**
 - l'invocazione dei processi avviene da parte di un apposito **scheduler**
- Consiste in 2 parti:
 - *Process control block*
 - *Process thread*

Contesti di esecuzione (1/2)

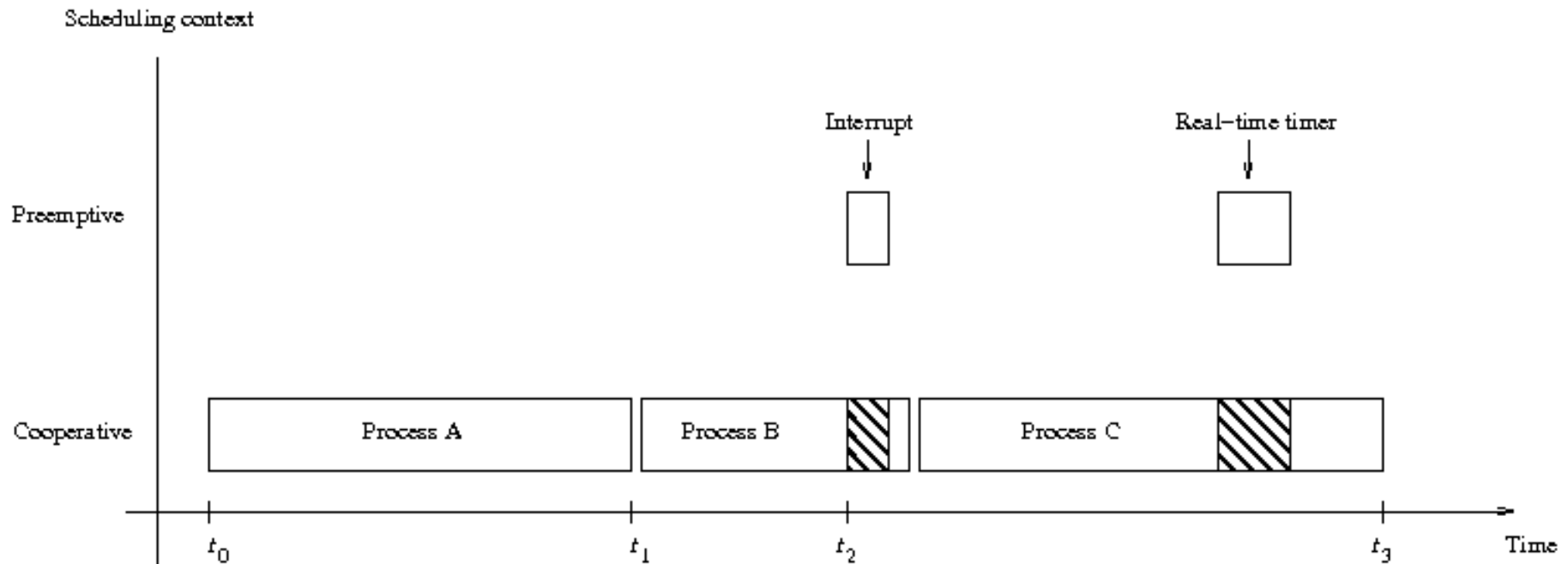


- 2 contesti di esecuzione:
 - **Cooperativo:** il codice è eseguito sequenzialmente rispetto ad altro codice cooperativo
 - ✓ I processi sono eseguiti sempre nel contesto cooperativo
 - ✓ Semantica *run-to-completion*
 - **Preemptive:** il codice può interrompere temporaneamente il codice cooperativo, in qualsiasi momento
 - ✓ Il codice cooperativo riprende al termine del codice preemptive
 - ✓ *Interrupt handlers* dei driver (hardware)
 - ✓ *Real-time task* schedulati entro una certa deadline (timer)

Contesti di esecuzione (2/2)



- Esempio dei due livelli di scheduling
 - Contiki non disabilita mai le interruzioni



Process Control Block



- Contiene **informazioni sui processi in esecuzione**
 - Memorizzato in RAM
 - Usato internamente dal kernel
 - Il codice utente non vi dovrebbe mai accedere

```
struct process {  
    struct process *next;           -> Prossimo PCB nella lista dei processi attivi  
    const char *name;              -> Nome testuale del processo  
    int (* thread)(struct pt *,    -> Puntatore a funzione: punta al process thread  
    process_event_t, process_data_t); del processo  
    struct pt pt;                  -> Tiene lo stato del protothread associato  
    unsigned char state;           -> Stato del processo  
    unsigned char needspoll;       -> Indica che il processo è stato polled  
};
```

- Dichiarato con la macro `PROCESS(variable_PCB, nome_processo)`
 - E.g.: `PROCESS (hello_world_process, «Hello world process»);`

Process Thread



- Contiene il codice del processo
- È un Protothread
 - Invocato dallo scheduler dei processi
- È definito tramite la macro `PROCESS_THREAD()`
 - `variable_PCB` la variabile associata al Process Control Block
 - `ev` identificatore dell'evento che ha causato l'invocazione del processo
 - `data` dati passati insieme all'evento `ev`

```
PROCESS_THREAD (hello_world_process, ev, data)
{
    PROCESS_BEGIN();
    printf(«Hello, world\n»);
    PROCESS_END();
};
```

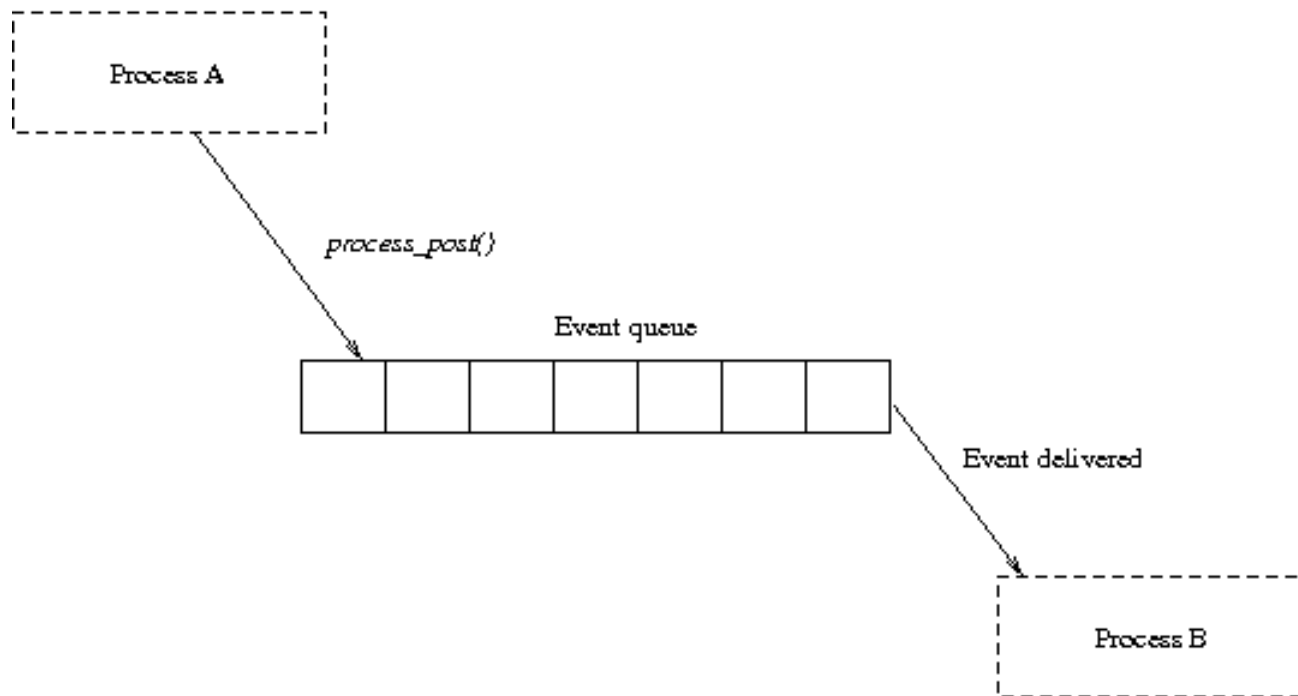
- **Gli statement sono implementati mediante macro**
 - **Richiamano le macro dei protothread**
 - **In aggiunta, gestiscono il Process Control Block**
- *PROCESS_BEGIN()* **Dichiara l'inizio del protothread di un processo**
- *PROCESS_END()* **Dichiara la fine del protothread del processo**
- *PROCESS_EXIT()* **Chiude il processo**
- *PROCESS_WAIT_EVENT()* **Aspetta per un evento qualsiasi**
- *PROCESS_WAIT_EVENT_UNTIL()* **Aspetta per un evento, ma con una condizione**
- *PROCESS_YIELD()* **Equivalente a *PROCESS_WAIT_EVENT()***
- *PROCESS_WAIT_UNTIL()* **Aspetta per una data condizione.
Può non essere bloccante**
- *PROCESS_PAUSE()* **Mette temporaneamente in pausa il processo**

Eventi

Eventi Asincroni (1/3)



- Quando un evento asincrono viene postato, **l'evento è messo nella coda degli eventi del kernel** e consegnato al processo destinatario dopo qualche tempo



Eventi Asincroni (2/3)



- È il kernel (scheduler) che si occupa di consegnare gli eventi
 - La gestione dell'evento è ritardata fino a quando il kernel (lo scheduler) non schedula il processo destinatario
- Il destinatario può essere
 - Uno specifico processo
 - Tutti i processi in esecuzione
 - ✓ Il kernel consegna lo stesso evento a tutti i processi, uno dopo l'altro
- Gli eventi asincroni sono postati con la funzione *process_post()*
 - Controlla che la coda degli eventi non sia piena
 - Aggiunge l'evento alla coda e ritorna
 - Il processo destinatario solitamente è bloccato su `PROCESS_WAIT_EVENT_UNTIL (ev == EVENTNAME) ;`

Eventi Asincroni (3/3)



```
int process_post    (struct process *p,  
                    process_event_t ev,  
                    void * data)
```

■ Parametri:

- *p* Il processo destinatario dell'evento. **PROCESS_BROADCAST** se deve essere postato a tutti i processi
- *ev* L'evento da postare
- *data* I dati aggiuntivi da inviare con l'evento

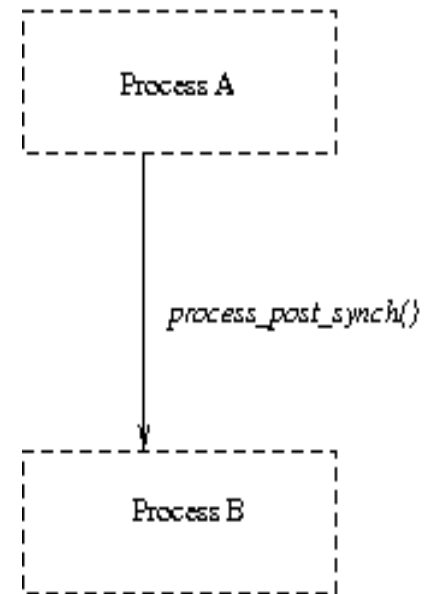
■ Valori restituiti:

- **PROCESS_ERR_OK**
L'evento è stato correttamente inserito nella coda.
- **PROCESS_ERR_FULL**
La coda degli eventi era piena e l'evento non è stato postato.

Eventi Sincroni (1/2)



- Sono consegnati direttamente quando vengono postati
- Possono essere **diretti solo ad uno specifico processo**
- Postare un evento sincrono è **funzionalmente equivalente ad una chiamata di funzione:**
 - Il processo destinatario (B) è invocato direttamente dal processo mittente (A)
 - Il processo A si blocca finché il processo B non ha terminato di processare l'evento
- Gli eventi sincroni sono postati con la funzione `process_post_synch()`



Eventi Sincroni (2/2)



```
int process_post_synch    (struct process *p,  
                           process_event_t   ev,  
                           void * data)
```

- **Parametri:**
 - *p* Un puntatore al PCB del processo
 - *ev* L'evento da postare
 - *data* Un puntatore verso dati aggiuntivi da postare insieme con l'evento

- È un tipo speciale di evento
- Il processo che riceve un poll è schedato il prima possibile
 - Inoltre riceve un evento di tipo `PROCESS_EVENT_POLL`
- È l'unico modo con cui è possibile chiamare un processo da un interrupt
- Un processo riceve un poll tramite la funzione `process_poll()`
 - La funzione setta il flag *needspoll* nel PCB del processo

```
void process_poll (struct process *p)
```

- Parametri:
 - *p* Un puntatore al PCB del processo

Event Identifiers



- **Gli eventi sono indicati con un identificatore:**
 - Un numero intero su 8 bit
 - È passato al processo che riceve l'evento
- **Gli eventi con identificatore...**
 - **< 128** possono essere usati all'interno di uno stesso processo
 - **≥ 128** sono usati tra processi distinti e sono gestiti dal kernel
- **I primi numeri oltre il 128 sono allocati staticamente dal kernel**

Event Identifiers: ID riservati



- `#define PROCESS_EVENT_NONE` 128
- `#define PROCESS_EVENT_INIT` 129
Inviato ai nuovi processi quando sono inizializzati
- `#define PROCESS_EVENT_POLL` 130
Inviato a un processo che ha ricevuto un poll
- `#define PROCESS_EVENT_EXIT` 131
Inviato a un processo che sta per essere terminato dal kernel
- `#define PROCESS_EVENT_CONTINUE` 133
Inviato a un processo in attesa su un `PROCESS_YIELD()`
- `#define PROCESS_EVENT_MSG` 134
Inviato a un processo che ha ricevuto un messaggio (e.g. uIP)
- `#define PROCESS_EVENT_EXITED` 135
Inviato a tutti i processi quando un altro processo sta per terminare
- `#define PROCESS_EVENT_TIMER` 136
Inviato a un processo quando un timer evento è scattato

- In aggiunta, i processi possono allocare identificatori sopra il 128 da usare per la comunicazione con altri processi

1. Si dichiara la variabile associata all'evento custom

```
static process_event_t evento_di_test;
```

2. Si alloca l'identificatore all'evento

```
evento_di_test = process_alloc_event();
```

3. L'evento può essere normalmente postato e ricevuto:

e.g.

```
process_post(&process, evento_di_test, &data);  
PROCESS_WAIT_EVENT_UNTIL(ev == evento_di_test);
```

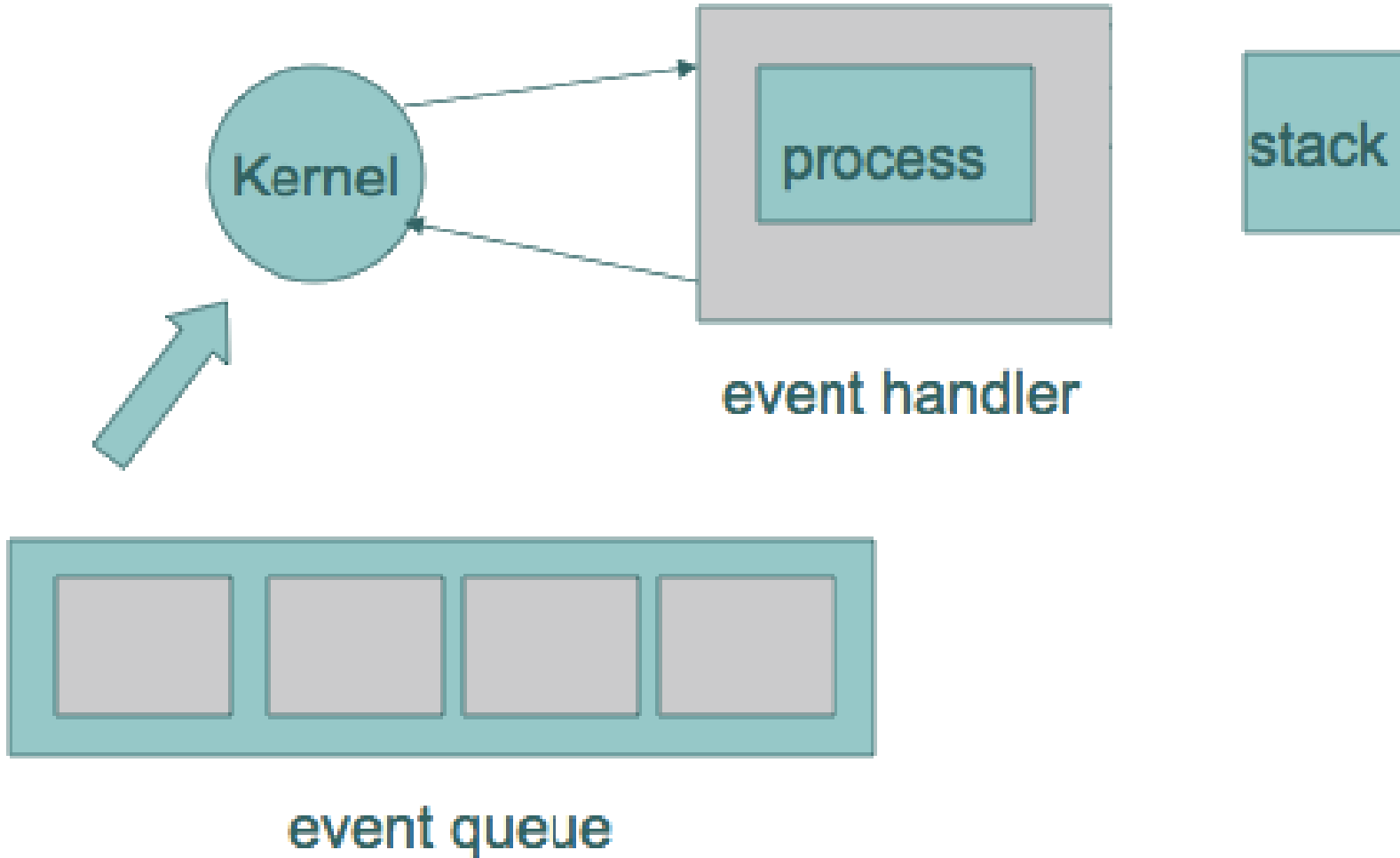
Lo scheduler dei processi

Scheduler dei processi (1/2)



- Ha lo scopo di invocare i processi
 - Un **Loop** che prende un evento dalla coda degli eventi
 - Chiama il/i **PROCESS_THREAD** a cui l'evento è destinato
 - Se non c'è niente da fare, la CPU va in *sleep* (*MCU low power mode*)
- **Ogni invocazione deriva da un evento**
 - Eventi esterni (interrupt hardware)
 - Timer
 - Eventi postati da altri processi
 - *Poll* richiesti da altri processi
- **Lo scheduler passa al processo chiamato**
 - L'identificatore dell'evento
 - I dati associati all'evento

Scheduler dei processi (2/2)



Il System Loop (1/2)



- Al boot si avviano
 - I processi essenziali (e.g. che gestiscono i timer, lo stack TCP/IP, ...)
 - I processi per cui è previsto l'avvio automatico
- Dopodiché si chiama ripetutamente la funzione *process_run()*
 - Se è stato richiesto un poll, lo si esegue
 - Processa un singolo evento dalla coda degli eventi
 - Restituisce il numero degli eventi rimasti in coda
- Se la coda degli eventi è vuota, si va in *Low Power Mode*
 - Il sistema può essere svegliato da una interruzione esterna
 - ✓ Si ripete il ciclo

Il System Loop (2/2)



```
int main(void) {
    initialize();
    while(1) {
        while(process_run() > 0);
        LPM_SLEEP();
    }
    return 0;
}
/* ----- */
process_run(void) {
    if(poll_requested)
        do_poll();
    do_event();
    return nevents + poll_requested;
}
```

-> Dal main source file

-> Inizializza sistema, avvia processi base

-> System Loop infinito

-> Ritorna quando una interruzione chiama la LPM_AWAKE()

-> Dal file process.c

-> Processa i poll events

-> Processa un evento dalla coda

Starting processes



- I processi sono avviati con il comando *process_start()*
 - Mette il processo nella lista dei processi attivi del kernel (controllando prima che non sia già presente)
 - Inizializza il Process Control Block (lo stato del processo diventa `PROCESS_STATE_RUNNING`)
 - Inizializza il `Process_thread` con il comando `PT_INIT()`
 - Dopo il comando, il processo è subito eseguito
 - Il processo avviato riceve un evento sincrono `PROCESS_EVENT_INIT`

Exiting and Killing processes



- Un processo termina autonomamente
 - Chiamando la funzione `PROCESS_EXIT()`
 - Quando il Process Thread raggiunge lo statement `PROCESS_END()`
- Il processo può essere terminato da un altro processo
 - Chiamando la funzione `process_exit()`
 - Il processo chiuso riceve l'evento `PROCESS_EVENT_EXIT`
- Quando un processo è terminato, il kernel invia un evento sincrónico (`PROCESS_EVENT_EXITED`) a tutti gli altri processi
 - Per informarli riguardo al processo uscente
 - Per liberare eventuali risorse occupate (e.g. uIP TCP/IP stack)
- Infine, il processo è eliminato dalla coda dei processi attivi

Funzioni di scheduling



1. `void process_start (struct process *p,
const char * arg)`

- Parametri:

- ✓ `p` Un puntatore al PCB del processo
- ✓ `arg` Un puntatore all'argomento che può essere passato al nuovo processo

2. `void process_exit (struct process *p)`

- Parametri:

- ✓ `p` Un puntatore al PCB del processo

Autostarting Processes



- L'autostart module permette di avviare automaticamente i processi
 - Al boot del sistema
 - Quando il modulo contenente i processi è caricato
- **Si crea una lista dei processi da avviare**
 - Tramite la macro `AUTOSTART_PROCESSES (variabili_PCB)`
 - È il metodo più comune per avviare processi utente
- Un modulo può informare il sistema circa i processi attivi contenuti
 - La lista indica quali processi avviare quando il modulo è caricato
 - La lista è usata anche per terminare i processi quando il modulo è unloaded

Processo di esempio

Processo di esempio (1/2)



```
1.  #include "contiki.h"
2.
3.  PROCESS(example_process, "Example process");
4.  AUTOSTART_PROCESSES(&example_process);
5.
6.  PROCESS_THREAD(example_process, ev, data)
7.  {
8.      PROCESS_BEGIN();
9.
10.     while(1) {
11.         PROCESS_WAIT_EVENT();
12.         printf("Got event number %d\n", ev);
13.     }
14.
15.     PROCESS_END();
16. }
```

Processo di esempio (2/2)



```
static char msg[] = "Data";
static void example_function(void)
{
    /* Start "Example process", and send it a NULL pointer. */
    process_start(&example_process, NULL);

    /* Send the PROCESS_EVENT_MSG event synchronously to "Example process",
       with a pointer to the message in the array 'msg'. */
    process_post_synch(&example_process, PROCESS_EVENT_CONTINUE, msg);

    /* Send the PROCESS_EVENT_MSG event asynchronously to "Example process",
       with a pointer to the message in the array 'msg'. */
    process_post(&example_process, PROCESS_EVENT_CONTINUE, msg);

    /* Poll "Example process". */
    process_poll(&example_process);
}
```

- [1] A. Dunkels, B. Gronvall, T. Voigt, “**Contiki - a lightweight and flexible operating system for tiny networked sensors**”, IEEE International Conference on Local Computer Networks, 16-18 November 2004
- [2] A. Dunkels, O. Schmidt, T. Voigt, M. Ali, “**Protothreads: simplifying event-driven programming of memory-constrained embedded systems**”, ACM International Conference on Embedded Networked Sensor Systems (SenSys), 2006
- [3] Contiki, Processes. Available Online at:
<https://github.com/contiki-os/contiki/wiki/Processes>
- [4] A. Dunkels, “**Contiki 2.6: The Contiki Operating System**” (*Doxygen Doc*). Available Online at:
<http://contiki.sourceforge.net/docs/2.6/>
- [5] T. Voigt, “**Contiki COOJA Crashcourse**”, The International School on Cooperative Robots and Sensor Networks (RoboSense School 2012), Hammamet, Tunisia, December 2012
- [6] F. Hermans, “**A Practical Introduction to Sensor Network Programming**”. Available Online at:
<http://www.it.uu.se/edu/course/homepage/wcnes/vt11/schedule/slides-lab-intro-lecture-wcnes.pdf>
- [7] N. Triolo, “**Protothreads e sistemi operativi**”. Available Online at:
http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/rhs/protothread_2012.pdf
- [8] A. Dunkels, “**Programming Contiki (processes, protothreads, uIP, Rime)**”, the Contiki Hands-On Workshop 2007, Stockholm, Sweden, March 2007