Protothreads e sistemi operativi

Nicoletta Triolo

Dip. di Informatica

15 Maggio 2012

Controllo del flusso

- programmazione non strutturata (the evil GOTO)
- programmazione strutturata

Usi del GOTO (1)

- uscita dal mezzo di un ciclo

```
1    while not cond1 do
2     do_something();
3     if cond2 goto 35;
4    end;
5  35: continue_somethingOther();
```

- uscita preventiva da una subroutine

```
1   procedure myProcedure()
2   ...
3    if cond1 then goto 35;
4   ...
5  35:
6   end;
```

Usi del GOTO (2)

- ritorni multilivello (nonlocal gotos)

```
1   function myFun()
2    var rtn : string;
3    ...
4    procedure myProc()
5     ...
6     if cond2 (x)
7          rtn := x;
8          goto 35;
9     ...
10   end;
11  ...
12   for ...
13    myProc();
14  ...
15 35: return rtn;
16  end;
```

  - stack unwinding : deallocazione degli stack frame delle subroutine da cui siamo usciti, ripristino dei registri

Usi del GOTO (3)

- errori

```
 1   function myFun()
 2    ...
 3    procedure myProc()
 4     ...
 5     if cond2 goto 35;
 6     ...
 7    end;
 8   ...
 9    myProc();
10   ...
11  35:
12   end;
```

Usi del GOTO (4)

Per ognuno di questi pattern diversi linguaggi di programmazione hanno introdotto costrutti ad hoc rendendo la programmazione piu' facile ed elegante es: `continue`, `return`, `try-catch`

Continuazioni (1)

nonlocal gotos

- indirizzo di codice e ambiente da ristabilire quando si salta all'indirizzo a cui e' associato nella continuazione
- si comporta come una chiusura

Le chiusure legano la funzione al suo contesto locale

```
fun f()
{
        int x=1;
        g=fun(void){
           print(x);
           return x++;
        }
        return g;
}
let new_g = f();
new_g(); //stampa 1
new_g(); //stampa 2
new_g(); //stampa 3
```

Continuazioni (2)

- in alcuni linguaggi come Scheme esiste la funzione `call-with-current-continuation`: prende in input una funzione `f` e la applica passando come argomento una continuazione `c` contenente il program counter corrente e l'ambiente riferito.
- in qualsiasi momento `f` puo' chiamare `c` per ristabilire il contesto catturato.
- le continuazioni possono essere salvate in variabili e ritornate esplicitamente da subroutine o chiamate ripetutamente anche dopo che il controllo e' stato restituito da f.

Coroutines (1)

- simili a continuazioni, ma in piu'
- la coroutine chiamata ricorda il vecchio program counter
- contesti di esecuzione che esistono concorrentemente ma che vengono eseguiti uno alla volta, il controllo viene trasferito esplicitamente per nome
- a seconda dell'annidamento permesso dal linguaggio si hanno stack disgiunti, uno per ogni coroutine, oppure cactus stack
- istruzione `transfer` per il trasferimento del controllo alla coroutine passata per nome come parametro

Coroutines (2)

usi

- iteratori
- **threads** livello utente
- simulazione di eventi

Coroutines in C

implementazione di Simon Tatham (o di Tom Duff)

Da `http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html` per implementare una semantica ritorna e continua e' possible usare il costrutto `switch` mischiato ad un `while` o un `for`

```
int function(void) {
    static int i, state = 0;
    switch (state) {
        case 0: /* start of function */
        for (i = 0; i < 10; i++) {
            state = 1; /* so we will come back to "case 1" */
            return i;
            case 1:; /* resume control straight after the return */
        }
    }
}
```

Coroutines in C

implementazione di Simon Tatham (o di Tom Duff) (2)

l'implementazione puo' essere resa piu' usabile usando delle macro

```
#define crBegin static int state=0; switch(state) { case 0:
#define crReturn(x) do { state = __LINE__; return x; case __LINE__:;} while
      (0)
#define crFinish }
int function(){
    static int i;
    crBegin;
    for (i=0;i<10;i++)
      crReturn(i);
      crFinish;
}
```

da notare che l'uso delle variabili statiche rende questo codice non rientrante; ovvia usando una struttura rappresentante il contesto

Dalle coroutine ai thread

- implementazione di uno scheduler che eviti all'utente di passare esplicitamente il controllo da un thread (coroutine) all'altro
- implementazione di un meccanismo di prerilascio
- condivisione delle strutture dati che rappresentano i thread tra piu' processi, possibilmente su processori diversi, cosicche' i thread possano girare su qualsiasi processo

Sistemi guidati da eventi (1)

- programmi = insieme di gestori di eventi
- i gestori di eventi sono invocati in risposta ad eventi esterni
- sono delle subroutine che eseguono un'azione e ritornano al chiamante

Sistemi guidati da eventi (1)

- programmi = insieme di gestori di eventi
- i gestori di eventi sono invocati in risposta ad eventi esterni
- sono delle subroutine che eseguono un'azione e ritornano al chiamante
- semantica run-to-completion : non possono eseguire un'attesa bloccante, ma...

Sistemi guidati da eventi (1)

- programmi = insieme di gestori di eventi
- i gestori di eventi sono invocati in risposta ad eventi esterni
- sono delle subroutine che eseguono un'azione e ritornano al chiamante
- semantica run-to-completion : non possono eseguire un'attesa bloccante, ma...
- il sistema puo' utilizzare un unico stack condiviso, pero'...

Sistemi guidati da eventi (1)

- programmi = insieme di gestori di eventi
- i gestori di eventi sono invocati in risposta ad eventi esterni
- sono delle subroutine che eseguono un'azione e ritornano al chiamante
- semantica run-to-completion : non possono eseguire un'attesa bloccante, ma...
- il sistema puo' utilizzare un unico stack condiviso, pero'...
- c'e' maggior difficolta' nel programmare operazioni di alto livello: macchine a stati

Sistemi guidati da eventi (2)

esempio state machine vs thread-like

```
enum {
  ON,
  WAITING,
  OFF
} state;

void radio_wake_eventhandler() {
  switch(state) {

  case OFF:
    if(timer_expired(&timer)) {
      radio_on();
      state = ON;
      timer_set(&timer, T_AWAKE);
    }
    break;

  case ON:
    if(timer_expired(&timer)) {
      timer_set(&timer, T_SLEEP);
      if(!communication_complete()) {
        state = WAITING;
      } else {
        radio_off();
        state = OFF;
      }
    }
    break;

  case WAITING:
    if(communication_complete()
       || timer_expired(&timer)) {
      state = ON;
      timer_set(&timer, T_AWAKE);
    } else {
      radio_off();
      state = OFF;
```

```
PT_THREAD(radio_wake_thread(struct pt *pt)) {
  PT_BEGIN(pt);

  while(1) {
    radio_on();
    timer_set(&timer, T_AWAKE);
    PT_WAIT_UNTIL(pt, timer_expired(&timer));

    timer_set(&timer, T_SLEEP);
    if(!communication_complete()) {
      PT_WAIT_UNTIL(pt, communication_complete()
                        || timer_expired(&timer));
    }

    if(!timer_expired(&timer)) {
      radio_off();
      PT_WAIT_UNTIL(pt, timer_expired(&timer));
    }
  }

  PT_END(pt);
}
```

Protothread

- introdotti da Adam Dunkels e Oliver Schmidt in Protothreads : lightweight, stackless, threads in C
- home page del progetto (sorgenti e documentazione) http://www.sics.se/~adam/pt/index.html
- implementati in C, ispirandosi notevolamente all'implementazione in C delle coroutine di Tatham
- obiettivo : efficienza dell'implementazione a eventi + astrazione/programmabilita' dei thread

Protothread

- i protothread forniscono *conditional blocking*, implementati al di sopra di un sistema guidato da eventi
- non hanno necessità di uno stack per ogni (proto)thread
- si semplifica notevolmente la programmazione perchè si puo' ridurre il numero di macchine a stati programmate esplicitamente

Torniamo all'esempio

rispetto alla programmazione a eventi

```
enum {
  ON,
  WAITING,
  OFF
} state;

void radio_wake_eventhandler() {
  switch(state) {

  case OFF:
    if(timer_expired(&timer)) {
      radio_on();
      state = ON;
      timer_set(&timer, T_AWAKE);
    }
    break;

  case ON:
    if(timer_expired(&timer)) {
      timer_set(&timer, T_SLEEP);
      if(!communication_complete()) {
        state = WAITING;
      } else {
        radio_off();
        state = OFF;
      }
    }
    break;

  case WAITING:
    if(communication_complete()
       || timer_expired(&timer)) {
      state = ON;
      timer_set(&timer, T_AWAKE);
    } else {
      radio_off();
      state = OFF;
```

```
PT_THREAD(radio_wake_thread(struct pt *pt)) {
  PT_BEGIN(pt);

  while(1) {
    radio_on();
    timer_set(&timer, T_AWAKE);
    PT_WAIT_UNTIL(pt, timer_expired(&timer));

    timer_set(&timer, T_SLEEP);
    if(!communication_complete()) {
      PT_WAIT_UNTIL(pt, communication_complete()
                        || timer_expired(&timer));
    }

    if(!timer_expired(&timer)) {
      radio_off();
      PT_WAIT_UNTIL(pt, timer_expired(&timer));
    }
  }

  PT_END(pt);
}
```

Torniamo all'esempio

rispetto alla programmazione a eventi

- il codice è piu' corto, ma non solo
- e' piu' comprensibile
- segue piu' linearmente la specifica
- si usano i costrutti del linguaggio (C in questo caso) come il `while`

Protothread

rispetto ai thread

- i protothread sono thread piu' leggeri, perche'...

Protothread

rispetto ai thread

- i protothread sono thread piu' leggeri, perche'...
- non hanno necessità di uno stack per ogni (proto)thread

Protothread

rispetto ai thread

- i protothread sono thread piu' leggeri, perche'...
- non hanno necessità di uno stack per ogni (proto)thread
- condividono lo stesso unico stack -> commutazione di contesto veloce

Protothread

rispetto ai thread

- i protothread sono thread piu' leggeri, perche'...
- non hanno necessità di uno stack per ogni (proto)thread
- condividono lo stesso unico stack -> commutazione di contesto veloce
- richiede solo 2 byte per l'implementazione

Protothread

rispetto ai thread

- i protothread sono thread piu' leggeri, perche'...
- non hanno necessità di uno stack per ogni (proto)thread
- condividono lo stesso unico stack -> commutazione di contesto veloce
- richiede solo 2 byte per l'implementazione
- possono chiamare funzioni ma non possono bloccarsi all'interno di una funzione chiamata, piuttosto...

Protothread

rispetto ai thread

- i protothread sono thread piu' leggeri, perche'...
- non hanno necessità di uno stack per ogni (proto)thread
- condividono lo stesso unico stack -> commutazione di contesto veloce
- richiede solo 2 byte per l'implementazione
- possono chiamare funzioni ma non possono bloccarsi all'interno di una funzione chiamata, piuttosto...
- viene generato un nuovo protothread per ogni funzione bloccante (protothread gerarchici)

Protothread

rispetto ai thread

- i protothread sono thread piu' leggeri, perche'...
- non hanno necessità di uno stack per ogni (proto)thread
- condividono lo stesso unico stack -> commutazione di contesto veloce
- richiede solo 2 byte per l'implementazione
- possono chiamare funzioni ma non possono bloccarsi all'interno di una funzione chiamata, piuttosto...
- viene generato un nuovo protothread per ogni funzione bloccante (protothread gerarchici)
- la sospensione e' esplicita, il programmatore deve sapere quale funzione si puo' bloccare

Protothread

rispetto ai thread

- i protothread sono thread piu' leggeri, perche'...
- non hanno necessità di uno stack per ogni (proto)thread
- condividono lo stesso unico stack -> commutazione di contesto veloce
- richiede solo 2 byte per l'implementazione
- possono chiamare funzioni ma non possono bloccarsi all'interno di una funzione chiamata, piuttosto...
- viene generato un nuovo protothread per ogni funzione bloccante (protothread gerarchici)
- la sospensione e' esplicita, il programmatore deve sapere quale funzione si puo' bloccare
- la preemption non e' possibile, come per gli eventi, è possibile una wait senza condizione

Protothread

rispetto ai thread

- i protothread sono thread piu' leggeri, perche'...
- non hanno necessità di uno stack per ogni (proto)thread
- condividono lo stesso unico stack -> commutazione di contesto veloce
- richiede solo 2 byte per l'implementazione
- possono chiamare funzioni ma non possono bloccarsi all'interno di una funzione chiamata, piuttosto...
- viene generato un nuovo protothread per ogni funzione bloccante (protothread gerarchici)
- la sospensione e' esplicita, il programmatore deve sapere quale funzione si puo' bloccare
- la preemption non e' possibile, come per gli eventi, è possibile una wait senza condizione
- le variabili automatiche non vengono salvate tra due attese bloccanti successive, ma

Protothread

rispetto ai thread

- i protothread sono thread piu' leggeri, perche'...
- non hanno necessità di uno stack per ogni (proto)thread
- condividono lo stesso unico stack -> commutazione di contesto veloce
- richiede solo 2 byte per l'implementazione
- possono chiamare funzioni ma non possono bloccarsi all'interno di una funzione chiamata, piuttosto...
- viene generato un nuovo protothread per ogni funzione bloccante (protothread gerarchici)
- la sospensione e' esplicita, il programmatore deve sapere quale funzione si puo' bloccare
- la preemption non e' possibile, come per gli eventi, è possibile una wait senza condizione
- le variabili automatiche non vengono salvate tra due attese bloccanti successive, ma
- vanno salvate esplicitamente (*state object*)

Protothread

rispetto ai thread

- i protothread sono thread piu' leggeri, perche'...
- non hanno necessità di uno stack per ogni (proto)thread
- condividono lo stesso unico stack -> commutazione di contesto veloce
- richiede solo 2 byte per l'implementazione
- possono chiamare funzioni ma non possono bloccarsi all'interno di una funzione chiamata, piuttosto...
- viene generato un nuovo protothread per ogni funzione bloccante (protothread gerarchici)
- la sospensione e' esplicita, il programmatore deve sapere quale funzione si puo' bloccare
- la preemption non e' possibile, come per gli eventi, è possibile una wait senza condizione
- le variabili automatiche non vengono salvate tra due attese bloccanti successive, ma
- vanno salvate esplicitamente (*state object*)
- oppure e' possibile usare variabili statiche

Protothread

implementazione (1)

- continuazioni locali
- come continuazioni, ma non catturano l'ambiente (lo stack)
- implementazione possibile con uno *switch* C
- supportano due operazioni : `set` e `resume`
- `set` : salva lo stato della funzione (ad esclusione dello stack)
- `resume` : ristabilisce lo stato della continuzione precedentemente salvato dalla `set`

Protothread

implementazione (2)

- il protothread e' una funzione + la continuazione locale
- prima di mettersi in attesa su una condizione il protothread invoca la `set`
- se la condizione e' vera il protothread esegue un ritorno esplicito al chiamante
- alla successiva chiamata del protothread lo stato salvato dalla `set` viene ristabilito
- la condizione viene rivalutata e se falsa la funzione continua

Protothread

implementazione (3)

viene usato in modo non ovvio il costrutto `switch` , all'utente vengono fornite delle macro :

```
#define LC_INIT(lc) lc = 0
#define LC_RESUME(lc) switch(lc) {case 0: //se la continuazione locale e'
    0 non fa niente
#define LC_SET(lc) lc=__LINE__; case __LINE__:
#define LC_END(lc) }
```

```
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
```

Protothread

yield e gerarchie

- `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 é schedulato dal padre ogni volta che il padre viene chiamato
  - lo stato del figlio é memorizzato un una variabile locale del padre

Protothread

PT_SPAWN

```
father_fun
     child_state: protothread_state
     PT_BEGIN
      while (running)
              PT_WAIT_UNTIL(cond)
              PT_SPAWN(child_fun(par), child_state)
     PT_END
child_fun(par):
     PT_BEGIN
      do
         ....
         PT_WAIT_UNTIL(cond1 )
      until (cond1)
      PT_END
```

Protothread

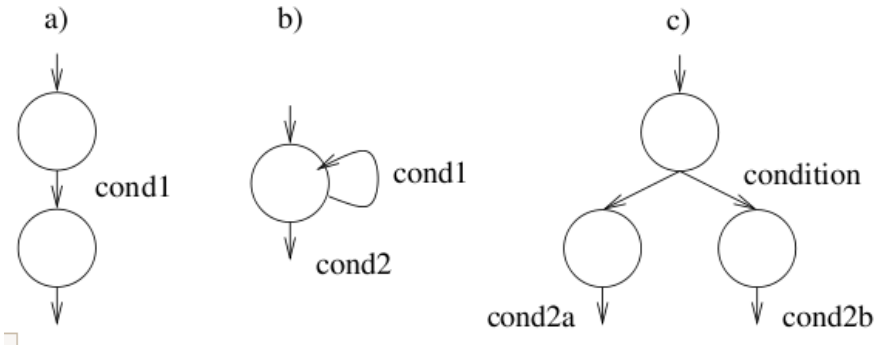
implementazione (4)

- in quest'implementazione serve soltanto un unsigned integer per memorizzare la linea di codice all'interno della funzione
- in un'implementazione alternativa (labels-as-values C extension) si usa un puntatore la cui dimensione dipende dalla piattaforma (2-3 byte)
- i programmi che usano i protothread non possono usare costrutti `switch` per evitare errori di compilazione e a tempo d'esecuzione
- l'uso del costrutto switch non e' ovvio perche' si hanno dei `case` annidati dentro un `while` interno allo `switch`

Protothread

da macchine a stati a protothread

E' possibile individuare 3 principali pattern per la composizione di macchine a stati : sequence (a), iteration (b), selection (c)



Protothread

da macchine a stati a protothread

Ognuno di questi pattern puo' essere associato a uno scheletro di implementazione tramite protothread

```
a_sequence:
   PT_BEGIN
   (* ... *)
   PT_WAIT_UNTIL(cond1)
   (* ... *)
   PT_END
```

```
an_iteration:
   PT_BEGIN
   (* ... *)
   while (cond1)
      PT_WAIT_UNTIL(cond1 or
                    cond2)
   (* ... *)
   PT_END
```

Figure 8. Pseudocode implementation of the sequence and iteration patterns with protothreads.

```
a_selection:
   PT_BEGIN
   (* ... *)
   if (condition)
      PT_WAIT_UNTIL(cond2a)
   else
      PT_WAIT_UNTIL(cond2b)
   (* ... *)
   PT_END
```

Figure 9. Pseudocode implementation of the selection pattern with a protothread.

Protothread

da macchine a stati a protothread

- quindi e' possibile trasformare del codice event-driven in un codice a protothread applicando un numero limitato di regole
- dopo aver analizzato la macchina a stati che implementa

Protothread

implementazione - esempio di riscrittura (1)

Da *Dunkels,Schmidt Protothreads : lightweight, stackless, threads in C*

```
PT_THREAD(radio_wake_thread(struct pt *pt)) {
  PT_BEGIN(pt);

  while(1) {
    radio_on();
    timer_set(&timer, T_AWAKE);
    PT_WAIT_UNTIL(pt, timer_expired(&timer));

    timer_set(&timer, T_SLEEP);
    if(!communication_complete()) {
      PT_WAIT_UNTIL(pt, communication_complete()
                        || timer_expired(&timer));
    }

    if(!timer_expired(&timer)) {
      radio_off();
      PT_WAIT_UNTIL(pt, timer_expired(&timer));
    }
  }

  PT_END(pt);
}
```

```
void radio_wake_thread(struct pt *pt) {
  switch(pt->lc) {
  case 0:

  while(1) {
    radio_on();
    timer_set(&timer, T_AWAKE);

    pt->lc = 8;
  case 8:
    if(!timer_expired(&timer)) {
      return;
    }

    timer_set(&timer, T_SLEEP);
    if(!communication_complete()) {

      pt->lc = 13;
  case 13:
      if(!(communication_complete() ||
           timer_expired(&timer))) {
        return;
      }

    }

    if(!timer_expired(&timer)) {
      radio_off();

      pt->lc = 18;
  case 18:
      if(!timer_expired(&timer)) {
        return;
      }

    }
  }
```

Protothread

implementazione - esempio di riscrittura (2)

- da un codice in uno stile thread-like riotteniamo una macchina a stati dopo la precompilazione

Protothread

Quando vengono invocati?

- non e' specificata una politica di invocazione o scheduling
- implementazione a carico del sistema che usa i protothread
- ad esempio, in caso di sistema a eventi la schedulazione del protothread puo' avvenire quando l'handler dell'evento e' chiamato dallo scheduler degli eventi (Contiki)

Protothread

Come viene gestito lo stato?

- non e' specificata una politica gestione della memoria
- se si conosce il numero di protothread puo' essere allocata staticamente
- altrimenti e' possibile una gestione dinamica
- in Contiki lo stato e' mantenuto nel process control block, tipicamente e' allocato staticamente

Protothread

Benefici

- gli autori hanno reimplementato numerose applicazioni guidate da eventi (macchine a stati complesse) usando i protothread
- per valutare la riduzione della complessita' del codice
- usando come metriche di complessita'
  - il numero di stati espliciti
  - il numero di transizioni di stati esplicite
  - le linee di codice necessarie
- nella maggior parte dei casi e' possibile eliminare completamente le macchine a stati esplicite
- e ridurre significativamente il numero di transizioni e di linee di codice

Protothread

Risultati

Program	States, before	States, after	Transitions, before	Transitions, after	Lines of code, before	Lines of code, after	Reduction, percentage
XNP	25	-	20	-	222	152	32%
TinyDB	23	-	24	-	374	285	24%
Mantis CC1000 driver	15	-	19	-	164	127	23%
SOS CC1000 driver	26	9	32	14	413	348	16%
Contiki TR1001 driver	12	3	32	3	152	77	49%
uIP SMTP client	10	-	10	-	223	122	45%
Contiki code propagation	6	4	11	3	204	144	29%

Table 1. The number of explicit states, explicit state transitions, and lines of code before and after rewriting with protothreads.

Protothread

Risultati

Program	Code size, before (bytes)	Code size, after (bytes)	Increase
XNP	931	1051	13%
TinyDB DBBufferC	2361	2663	13%
Mantis CC1000	994	1170	18%
SOS CC1000	1912	2165	13%
Contiki TR1001	823	836	2%
uIP SMTP	1106	1901	72%
Contiki code prop.	1848	1426	-23%

Table 2. Code size before and after rewriting with protothreads.

Protothread

Risultati

	State machine	Proto-thread
MSP430	9	12
AVR	23	34

Table 4. Machine code instructions overhead for a state machine, a protothread, and a yielding protothread.

Contiki

introduzione (1)

- sistema operativo per sistemi embedded memory-constrained in rete (es. reti di sensori!), ma anche per vecchi computer con poca memoria
- configurazione tipica 2KB ram 40KB rom
- progetto open source http://www.sics.se/contiki/
- implementato in C da Adam Dunkels e Networked Embedded Systems group @ Swedish Institute of Computer Science
- lo stesso dei protothread!
- le applicazioni si programmano in C

Contiki

introduzione (2)

- basato su kernel event-driven
- la prima versione e' stata sviluppata prima dell'introduzione dei protothread
- in seguito parti riscritte usando i protothread
- dalla prima versione offre una libreria con l'implementazione dei thread veri e propri (se sono proprio necessari!)
- l'astrazione di programmazione principale e' quella dei processi come protothread (nel seguito ci riferiamo a versioni successive a 2.1)
- permette allocazione dinamica della memoria (`malloc`)

Contiki

Esempi di hardware supportato

- ESB (embedded sensor board): microcontrollore bassa potenza Texas Instruments MSP430, 2k RAM, 60k flash ROM, TR1001 radio transceiver, 32k serial EEPROM, porta RS232 , porta JTAG , un beeper, passive IR, active IR sender/receiver, vibration/tilt, microfono, temperatura ...
- Tmote Sky Board : microcontrollore bassa potenza Texas Instruments MSP430, 10k RAM, 802.15.4-compatible CC2420 radio chip, 1 MB memoria flash esterna, 2 sensori di luce

Contiki

esempio Hello World

Da *Programming Contiki (processes, protothreads, uIP, Rime), Adam Dunkels, the Contiki Hands-On Workshop 2007, Stockholm, Sweden, March 2007.*

```
/* Declare the process */
PROCESS(hello_world_process, "Hello world");
/* Make the process start when the module is loaded */
AUTOSTART_PROCESSES(&hello_world);

/* Define the process code */
PROCESS_THREAD(hello_world_process, ev, data) {
  PROCESS_BEGIN();              /* Must always come first */
  printf("Hello, world!\n");    /* Initialization code goes here */
  while(1) {                   /* Loop for ever */
    PROCESS_WAIT_EVENT();       /* Wait for something to happen */
  }
  PROCESS_END();                /* Must always come last */
}
```

Contiki

esecuzione dei processi

Per eseguire i processi si hanno due alternative:

- invio di un evento
  - `process_post(process_ptr, eventno, ptr);` : l'evento sara' gestito alla prossima schedulazione del processo
  - `process_post_synch(process_ptr, eventno, ptr);` : invocazione sincrona
- poll del processo
  - `process_poll(process_ptr);` : puo' essere invocato da un gestore di interruzione
- operano sulle due strutture dati principali: lista di processi e coda di eventi
- nel main loop viene testata la presenza di processi di cui è stato richiesto il poll e la presenza di eventi ed eseguito il processo associato

Contiki

esecuzione dei processi

```
struct process {
  struct process *next;
#if PROCESS_CONF_NO_PROCESS_NAMES
#define PROCESS_NAME_STRING(process) ""
#else
  const char *name;
#define PROCESS_NAME_STRING(process) (process)->name
#endif
  PT_THREAD((* thread )(struct pt *, process_event_t, process_data_t));
  struct pt pt;
  unsigned char state, needspoll;
};

struct event_data {
  process_event_t ev;
  process_data_t data;
  struct process *p;
};
```

Contiki

esecuzione dei processi

Il main loop eseguito dall'avvio

```
....
while(1) {
    int r;
    do {
        ....
      r = process_run();
    } while(r > 0)
        .....
}
```

Contiki

esecuzione dei processi

Dal file *process.c*

```
int
process_run(void)
{
  /* Process poll events. */
  if(poll_requested) {
    do_poll();
  }

  /* Process one event from the queue */
  do_event();

  return nevents + poll_requested;
}
```

Contiki

esecuzione dei processi

```
static void do_event(void)
{
  ....
  if(nevents > 0) {
   ev = events[fevent].ev;
   data = events[fevent].data;
    receiver = events[fevent].p;
     .....
   --nevents;
   /* If this is a broadcast event, we deliver it to all events, in
      order of their priority. */
   if(receiver == PROCESS_BROADCAST) {
     for(p = process_list; p != NULL; p = p->next) {
       ....
       call_process(p, ev, data); // <|==
     }
   } else {
      if(ev == PROCESS_EVENT_INIT) {
receiver->state = PROCESS_STATE_RUNNING;
     }
     call_process(receiver, ev, data); <|==
   }
```

Contiki

esecuzione dei processi

```
static void
do_poll(void)
{
  struct process *p;

  poll_requested = 0;
  for(p = process_list; p != NULL; p = p->next) {
    if(p->needspoll) {
      p->state = PROCESS_STATE_RUNNING;
      p->needspoll = 0;
      call_process(p, PROCESS_EVENT_POLL, NULL); <|==
    }
  }
}
```

Contiki

esecuzione dei processi

```
static void
call_process(struct process *p, process_event_t ev, process_data_t data)
{
  int ret;
  if((p->state & PROCESS_STATE_RUNNING) &&
     p->thread != NULL) {
    process_current = p;
    p->state = PROCESS_STATE_CALLED;
    ret = p->thread(&p->pt, ev, data); // <|==
    if(ret == PT_EXITED ||
       ret == PT_ENDED ||
       ev == PROCESS_EVENT_EXIT) {
      exit_process(p, p);
    } else {
      p->state = PROCESS_STATE_RUNNING;
    }
  }
}
```

Contiki

esecuzione dei processi

```
void process_poll(struct process *p)
{
  if(p != NULL) {
    if(p->state == PROCESS_STATE_RUNNING ||
       p->state == PROCESS_STATE_CALLED) {
      p->needspoll = 1;
      poll_requested = 1;
    }
  }
}
int process_post(struct process *p, process_event_t ev, process_data_t data)
{
  static process_num_events_t snum;
  snum = (process_num_events_t)(fevent + nevents) % PROCESS_CONF_NUMEVENTS;
  events[snum].ev = ev;
  events[snum].data = data;
  events[snum].p = p;
  ++nevents;
   .....
  return PROCESS_ERR_OK;
}
```

Contiki

altre caratteristiche

- i processi comunicano per scambio di messaggi tramite eventi (`process_post`)
- la comunicazione di rete e' supportata da due stack: uIP (TCP/IP) e Rime (per radio a bassa potenza)
- caricamento dinamico dei programmi
- Coffee flash file system POSIX un po' semplificato
- librerie di sistema linkabili opzionalmente
- ambiente di simulazione e debugging
- sistema di build che usa Make
- ha una shell e un browser web per l'interazione coi nodi
- esiste una GUI con desktop, finestre e screen saver

Contiki

Comunicazione di rete con uIP

- uIP fornisce un'implementazione leggera di TCP UDP IP
- meno 100KB di codice, poche decine di KB di RAM
- i nodi della rete possono comunicare direttamente con il resto del mondo!

Contiki

Comunicazione di rete con uIP - come funziona

Il loop principale consiste in:

- controllare se un pacchetto e' arrivato dalla rete
  - se e' arrivato viene invocata la funzione `uip_input()` (non bloccante)
  - i pacchetti di risposta sono prodotti quando il processo ritorna
- controllare se e' scaduto un timeout (per ack ritardati o ritrasmissioni)

Contiki

Comunicazione di rete con uIP - gestione della memoria

- non si usa allocazione dinamica di memoria ma un unico buffer globale e una tabella per mantenere lo stato della connessione
- quindi i dati devono essere gestiti subito o copiati in altri buffer, altrimenti vengono sovrascritti
- i pacchetti che arrivano durante la gestione dei dati vengono accodati in altri buffer
- se i buffer sono pieni il nuovo pacchetto viene perso -> le prestazioni degradano

Contiki

Comunicazione di rete con uIP - API

uIP offre due API:

- event-driven API
  - vanno bene per piccoli programmi, perche'...
  - richiedono programmazione di macchine a stati esplicite!
- protosockets : programmazione socket-like basata su protothread
  - vanno bene per grandi programmi, perche'...
  - offrono astrazione di programmazione sequenziale
  - es. macro `PSOCK_READTO` e `PSOCK_SEND` wrappano due protothread

Contiki

Comunicazione di rete con uIP - API - esempio smtp con protosocket

```
PT_THREAD(smtp(struct psock *p))
{
  PSOCK_BEGIN(s);

  PSOCK_READTO(s, '\n');

  if(strncmp(inputbuffer, "220", 3) != 0) {
    PSOCK_CLOSE(s);
    PSOCK_EXIT(s);
  }

  PSOCK_SEND(s, "HELO ", 5);
  PSOCK_SEND(s, hostname, strlen(hostname));
  PSOCK_SEND(s, "\r\n", 2);
  PSOCK_READTO(s, '\n');

  if(inputbuffer[0] != '2') {
    PSOCK_CLOSE(s);
    PSOCK_EXIT(s);
  }
```

Contiki

Protosocket macro

```
#define PSOCK_BEGIN(psock) PT_BEGIN(&((psock)->pt))
#define PSOCK_READTO(psock, c) \
        PT_WAIT_THREAD(&((psock)->pt), psock_readto(psock, c))
#define PSOCK_CLOSE(psock) uip_close()
#define PSOCK_EXIT(psock) PT_EXIT(&((psock)->pt))
#define PSOCK_SEND(psock, data, datalen) \
        PT_WAIT_THREAD(&((psock)->pt), psock_send(psock, data, datalen))
```

implementazione

Comunicazione di rete con uIP - API - riscrittura con protosocket

Dopo il preprocessing...

```
char smtp(struct psock *s){
  { char PT_YIELD_FLAG = 1; switch((&((s)->pt))->lc) { case 0:;
  do { (((&((s)->pt))))->lc = 12; case 12:; if (!(!((( psock_readto(s, '\n')) < 2)))) {
      return 0; } } while(0);
   if(strncmp(inputbuffer, "220", 3) != 0) {
     (uip_flags = 16); do { (&((s)->pt))->lc = 0;; return 2; } while(0);
  }
  do { (((&((s)->pt))))->lc = 19; case 19:; if (!(!((( psock_send(s, "HELO ", 5)) < 2)))) {
       return 0; } } while(0);
  do { (((&((s)->pt))))->lc = 20; case 20:; if (!(!((( psock_send(s, hostname, strlen(
      hostname))) < 2)))) { return 0; } } while(0);
  do { (((&((s)->pt))))->lc = 21; case 21:; if (!(!((( psock_send(s, "\r\n", 2)) < 2)))) {
      return 0; } } while(0);
  do { (((&((s)->pt))))->lc = 23; case 23:; if (!(!((( psock_readto(s, '\n')) < 2)))) {
      return 0; } } while(0);
   if( inputbuffer[0] != '2') {
     (uip_flags = 16);
     do { (&((s)->pt))->lc = 0;; return 2; } while(0);
  }
 }; PT_YIELD_FLAG = 0; (&((s)->pt))->lc = 0;; return 3; };
}
```

Contiki

Caricamento dinamico dei programmi (2)

- le reti di sensori possono essere a larga scala
- non e' pensabile in caso di riprogrammazione, come patch o estensioni, agire individualmente su ogni sensore
- esistono diversi meccanismi di distribuzione di codice su WSN
- e' importante ridurre il numero di byte da trasferire perchè la comunicazione consuma buona parte dell'energia del nodo
- in molti sistemi ogni volta viene generato il binario che include il sistema operativo e tutte le applicazioni e riscaricato su ogni nodo
- Contiki riesce a caricare e scaricare programmi individuali a runtime (si parla di code propagation)
- un unico programma e' molto piu' piccolo di tutto il sistema

Contiki

vs TinyOS

- ad es in TinyOS dopo il linking non si puo' piu' modificare il programma
- Mate' e' una macchina virtuale per TinyOS che consente di scaricare il codice a runtime sui nodi
- il codice e' piu' piccolo, quindi minor consumo energetico per il trasferimento, ma si consuma molta energia per l'interpretazione
- Contiki sostituisce direttamente il codice nativo

TinyOs

brevemente...

- kernel event-driven
- paradigma di programmazione a eventi
- applicazioni costituite da componenti connessi tra loro da interfacce
- si usa linguaggio NesC, che non e' proprio C
- vieta allocazione dinamica e ricorsione
- Active Messages come primitive di comunicazione

LiteOs

- obiettivo: fornire astrazione Unix-like ai programmatori di reti di sensori
- dimensioni del codice piccole rispetto ai sistemi operativi embedded convenzionali
- perche' i sensori sono destinati a diventare sempre piu' piccoli (indossabili!)
- gira su MicaZ e IRIS
- comunicazione basata su file, un file per ogni device

LiteOs

componenti principali

- **LiteShell** : shell che gira su PC per interagire con la rete
  - comandi Unix-like (file, processi, debugging, ambiente, device)
  - gira sul PC della base station
  - i nodi rispondono semplicemente a dei messaggi (non mantengono lo stato)
- **LiteFS** : file system gerarchico che fornisce un mecchanismo per il mount wireless dei nodi nel raggio di copertura della base station
  - un nodo e' visto come una directory
  - una rete di sensori e' una directory con delle sottodirectory corrispondenti ai nodi che vi appartengono
  - necessari meccanismi di autenticazione
- **Kernel** :
  - multi-threaded : scheduling a piorita' o round robin
  - permette anche di gestire eventi tramite callback
  - caricamento / scaricamento dinamico delle applicazioni
  - permette allocazione dinamica di memoria (`malloc` e `free`)

LiteOs

programmazione - thread

- i thread mantengono il contesto di esecuzione

```
1  int main()
2  {
3   while (1) {
4    radioSend_string("Hello, world!\n");
5    greenToggle();
6    sleepThread(100);
7   }
8   return 0;
9  }
```

LiteOs

programmazione - callback

- per motivi di efficienza e' fornita anche la primitiva callback, ad es:

```
1  /* applicazione : dopo aver inviato un messaggio si attende l'arrivo dell'ack, se non
       arriva il messaggio e' perso */
2  bool wakeup = FALSE;
3  uint8_t currentThread;
4  currentThread = getCurrentThreadIndex();
5  registerRadioEvent(MYPORT, msg, length, packetReceived); /* si mette in ascolto su
       una porta e alloca un buffer */
6  sleepThread(T_timeout); /* sospensione */
7  /* dopo la sveglia */
8  unregisterRadioEvent(MYPORT);
9  if (wakeup == TRUE) {/* caso callback eseguita */ }
10 else {/* e' scattato il timeout */}
11 /* callback, invocata quando il kernel ha copiato il messaggio nel buffer */
12 void packetReceived(){
13   _atomic_start();
14   wakeup = TRUE;
15   wakeupThread(currentThread);
16   _atomic_end();
17 }
```

LiteOs

Risultati

- i thread rendono programmazione piu' semplice, quanto meno nel numero di linee di codice, ma
- maggior consumo di memoria statica (uno stack per ogni thread)

Conclusioni

Confronto

	LiteOS	TinyOS	Mantis	Contiki	SOS
Current license	GPL	BSD	BSD	BSD	Modified BSD
Website	www.liteos.net	www.tinyos.net	mantis.cs.colorado.edu	www.sics.se/contiki	projects.nesl.ucla.edu/public/sos-2x/doc/
Remote scriptable wireless shell	Yes (on the base PC, Unix commands supported)	No (application specific shell such as SimpleCmd exists)	No (on-the-mote shell is supported)	No (on-the-mote shell is supported)	No
Remote file system interface for networked nodes	Yes	No	No	No	No
File system	Hierarchical Unix-like	Single level (ELF, Matchbox)	No (will be available in 1.1)	Single Level	No
Thread support	Yes	Partial (through TinyThreads)	Yes	Yes (also supports ProtoThreads)	No
Event based programming	Yes (through callback functions)	Yes	No	Yes	Yes
Remote Debugging(e.g. watch and breakpoints)	Yes	Yes (through Clairvoyant)	Partial (through NodeMD)	No	No
Wireless reprogramming	Yes (application level)	Yes (whole system image replacement)	No (will be available in 1.1)	Yes	Yes (module level)
Dynamic memory	Yes	Yes (in 2.0 or through TinyAlloc for 1.x)	No	Yes	Yes
First publication/release date	2007	2000	2003	2003	2005
Platform support	MicaZ and AVR series MCU	Mica, Mica2, MicaZ, Telos, Tmote, XYZ, Iris (among others)	Mica2, MicaZ, Telos	Tmote, ESB, AVR series MCU, certain old computers	Mica2, MicaZ, XYZ
Simulator	Through AVRORA	TOSSIM, PowerTossim	Through AVRORA	Netsim, Cooja, MSPSim	Source level Simulator/ Through AVRORA

Riferimenti

- Michael L. Scott, Programming Language Pragmatics, Morgan Kaufmann
- Simon Tatham, Coroutines in C, http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html
- Adam Dunkels and Oliver Schmidt and Thiemo Voigt and Muneeb Ali, Protothreads: Simplifying event-driven programming of memory-constrained embedded systems,In Proc. 2006 SenSys
- Adam Dunkels and Oliver Schmidt and Thiemo Voigt, Using Protothreads for Sensor Node Programming, In Proceedings of the REALWSN 2005 Workshop on RealWorld Wireless Sensor Networks

Riferimenti

- Adam Dunkels and Oliver Schmidt, Protothreads - Lightweight, Stackless Threads in C, 2005
- The Contiki Operating System. Web page. http://www.sics.se/~adam/contiki
- Programming Contiki (processes, protothreads, uIP, Rime), Adam Dunkels, the Contiki Hands-On Workshop 2007, Stockholm, Sweden, March 2007
- Xu Dingxin, Contiki tutorial II. From What to How
- The Contiki User Manual, http://www.sics.se/~adam/contiki/docs/
- Cao, Qing and Abdelzaher, Tarek and Stankovic, John and He, Tian, The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks, Proceedings of the 7th international conference on Information processing in sensor networks, 2008