

# Lezione E11

## Scheduler per job interrompibili



Sistemi operativi open-source, embedded e real-time

21 novembre 2017

[Schema della lezione](#)

[Cambio di contesto](#)

[Creazione di un task](#)

[Scheduler \(1<sup>a</sup> vers.\)](#)

[\\_irq\\_handler\(\)](#)

[\\_irq\\_schedule\(\)](#)

[\\_sys\\_schedule\(\)](#)

[Scheduler \(2<sup>a</sup> vers.\)](#)

Marco Cesati

Dipartimento di Ingegneria Civile e Ingegneria Informatica  
Università degli Studi di Roma Tor Vergata

# Di cosa parliamo in questa lezione?



In questa lezione scriviamo il codice dello scheduler per job interrompibili

- 1 Il cambio di contesto
- 2 Creazione di un nuovo task
- 3 Scheduler (prima versione)
- 4 Modifiche alla funzione `_irq_handler()`
- 5 La funzione `_irq_schedule()`
- 6 La funzione `_sys_schedule()`
- 7 Scheduler (seconda versione)

Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

`_irq_handler()`

`_irq_schedule()`

`_sys_schedule()`

Scheduler (2<sup>a</sup> vers.)

## Contesto prima dell'esecuzione di `_bsp_irq()`



Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

`_irq_handler()`

`_irq_schedule()`

`_sys_schedule()`

Scheduler (2<sup>a</sup> vers.)

- La CPU è in modalità **SYSTEM**
- I registri AAPCS-clobbered del flusso di esecuzione interrotto salvati sullo stack sono `r0–r3, r12, r14 (=lr)`
- Nel registro `r12` è caricato l'indirizzo del gestore di medio livello dell'interruzione `_bsp_irq()`

	r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11	r12	sp r13	lr r14	pc r15	cpsr	spsr
<b>SYS</b>														N-32	#	#	*	///
IRQ	N-32	Q	C	D	E	F	G	H	I	J	K	L	Ω	A	B	#	*	Q
STK	Ⓐ	B	C	D	M	O	P+4	Q	X	Y	Z							

## Contesto dopo la terminazione di `_bsp_irq()`



Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

`_irq_handler()`

`_irq_schedule()`

`_sys_schedule()`

Scheduler (2<sup>a</sup> vers.)

- La CPU è in modalità **SYSTEM**
- In cima allo stack vi sono i valori di alcuni registri del flusso di esecuzione da recuperare:
  - Registri AAPCS-clobbered `r0-r3, r12, r14 (=lr)`
  - Registro `r15 (=pc)`, ossia l'indirizzo dell'istruzione successiva del flusso di esecuzione da recuperare
  - Registro `cpsr`

	r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11	r12	sp	lr	pc	cpsr	spsr
<b>SYS</b>	?	?	?	?	E	F	G	H	I	J	K	L	?	N-32	?	#	*	////
IRQ														?	?			?
STK	(A)	B	C	D	M	O	P+4	Q	X	Y	Z							



[Schema della lezione](#)

[Cambio di contesto](#)

[Creazione di un task](#)

[Scheduler \(1ª vers.\)](#)

[\\_irq\\_handler\(\)](#)

[\\_irq\\_schedule\(\)](#)

[\\_sys\\_schedule\(\)](#)

[Scheduler \(2ª vers.\)](#)

- Ogni job ha un proprio stack
- I valori dei registri `r0–r3`, `r12`, `r14`, `r15` e `cpsr` sono salvati sul proprio stack
- È necessario salvare altrove i valori dei registri `r4–r11` e `r13 (=sp)`
  - Salvataggio nel descrittore del task!
- Effettuare il cambio di contesto significa modificare `sp` in modo che faccia riferimento allo stack di un task differente
- Il cambio di contesto viene effettuato subito dopo la terminazione di `_bsp_irq()`
  - Le istruzioni macchina seguenti in `_irq_handler()` recuperano il contesto di un flusso di esecuzione (job) diverso da quello interrotto

Il cambio di contesto avviene solo se si sta per tornare all'esecuzione di un job (attenzione alle interruzioni annidate)

## Stack specifico per ogni task



Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

\_irq\_handler()

\_irq\_schedule()

\_sys\_schedule()

Scheduler (2<sup>a</sup> vers.)

- Gli stack sono allocati nella sezione `.bss` in un unico vettore
  - È necessario evitare di inizializzare a zero gli stack, altrimenti viene riscritto l'indirizzo di ritorno di `fill_bss()`
  - Utilizziamo una sottosezione chiamata `.bss.stack`
- Ogni stack ha lunghezza predefinita
- Ogni stack è allineato a pagine di memoria
- Gli stack crescono per indirizzi decrescenti

```
#define STACK_SIZE 4096
char stacks[MAX_NUM_TASKS*STACK_SIZE]
    __attribute__((aligned (STACK_SIZE)))
    __attribute__((section (.bss.stack)));
const char *stack0_top =
    stacks + MAX_NUM_TASKS*STACK_SIZE;
```

## Il descrittore del task

```
struct task {
    int valid;
    unsigned long releasetime;
    int released;
    int period;
    int priority;
    job_t job;
    void *arg;
    const char *name;
    unsigned long sp;
    unsigned long regs[8];
};
```

- La variabile `current` contiene l'indirizzo del task in esecuzione
- Se nessun job è eseguibile, `current` punta al descrittore del task con TID 0 (idle task)



Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

\_irq\_handler()

\_irq\_schedule()

\_sys\_schedule()

Scheduler (2<sup>a</sup> vers.)

## La funzione `_switch_to()`

```
void __attribute__((naked))
_switch_to(struct task *to)
{
    irq_disable();
    save_regs(current->regs);
    load_regs(to->regs);
    switch_stacks(current, to);
    current = to;
    irq_enable();
    naked_return();
}
```



Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

`_irq_handler()`

`_irq_schedule()`

`_sys_schedule()`

Scheduler (2<sup>a</sup> vers.)





[Schema della lezione](#)

[Cambio di contesto](#)

[Creazione di un task](#)

[Scheduler \(1ª vers.\)](#)

[\\_irq\\_handler\(\)](#)

[\\_irq\\_schedule\(\)](#)

[\\_sys\\_schedule\(\)](#)

[Scheduler \(2ª vers.\)](#)

- L'attributo `naked` è una estensione del compilatore `gcc` per l'architettura ARM
- Forza il compilatore a non emettere codice per il prologo e l'epilogo della funzione
- Nessuna istruzione generata automaticamente per salvare il contenuto di registri sullo stack
- Tuttavia non viene nemmeno emessa l'istruzione di “return” per terminare la funzione
  - È necessario generare esplicitamente l'istruzione di “return”

```
#define naked_return() \  
    __asm__ __volatile__ (“bx lr”);
```

## Salvataggio e ripristino dei registri r4–r11

Per salvare il contenuto dei registri r4–r11:

```
#define save_regs(regs) \  
    __asm__ __volatile__ ("stmia %0, {r4-r11}" \  
        : : "r" (regs) : "memory");
```

Per ripristinare il contenuto degli stessi registri:

```
#define load_regs(regs) \  
    __asm__ __volatile__ ("ldmia %0, {r4-r11}" \  
        : : "r" (regs) : "r4", "r5", "r6", \  
        "r7", "r8", "r9", "r10", "r11", "memory");
```



[Schema della lezione](#)

[Cambio di contesto](#)

[Creazione di un task](#)

[Scheduler \(1ª vers.\)](#)

[\\_irq\\_handler\(\)](#)

[\\_irq\\_schedule\(\)](#)

[\\_sys\\_schedule\(\)](#)

[Scheduler \(2ª vers.\)](#)



La macro `switch_stacks` salva e ripristina il contenuto del registro `r13 (=sp)`

```
#define switch_stacks(from, to) \  
    __asm__ __volatile__ ("str sp,%0\n\t" \  
                          "ldr sp,%1\n\t" \  
                          : "m" ((from)->sp), "m" ((to)->sp) \  
                          "sp", "memory");
```

[Schema della lezione](#)

[Cambio di contesto](#)

[Creazione di un task](#)

[Scheduler \(1ª vers.\)](#)

[\\_irq\\_handler\(\)](#)

[\\_irq\\_schedule\(\)](#)

[\\_sys\\_schedule\(\)](#)

[Scheduler \(2ª vers.\)](#)

## Inizializzazione del contesto di un nuovo task



Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

\_irq\_handler()

\_irq\_schedule()

\_sys\_schedule()

Scheduler (2<sup>a</sup> vers.)

- Determinare la posizione dello stack del task
- Definire l'entry point del job
  - Non coincide con la funzione da eseguire nel job
- Inizializzare il contesto salvato nel descrittore del task
- Inizializzare lo stack del task
  - Deve avere la stessa struttura dello stack di un job interrotto da una interruzione

STK

r0	r1	r2	r3	r12	lr	pc	spsr
----	----	----	----	-----	----	----	------

N    +4    +8    +12    +16    +20    +24    +28

## La funzione `init_task_context()`

```
void init_task_context(struct task *t,
                      int ntask) {
    int i;
    unsigned long *sp = (unsigned long *)
        (stack0_top-ntask*STACK_SIZE);
    * (--sp) = SYS_MODE | NO_FIQ; // spsr
    * (--sp) = (unsigned long)
        task_entry_point; // RET
    * (--sp) = 0ul; // lr
    * (--sp) = 0ul; // r12
    * (--sp) = 0ul; // r3
    * (--sp) = 0ul; // r2
    * (--sp) = 0ul; // r1
    * (--sp) = (unsigned long) t; // r0
    t->sp = (unsigned long) sp;
    for (i = 0; i < 8; ++i)
        t->regs[i] = 0ul; /* r4-r11 */
}
```



## La funzione `init_task_context()`



[Schema della lezione](#)

[Cambio di contesto](#)

[Creazione di un task](#)

[Scheduler \(1ª vers.\)](#)

[\\_irq\\_handler\(\)](#)

[\\_irq\\_schedule\(\)](#)

[\\_sys\\_schedule\(\)](#)

[Scheduler \(2ª vers.\)](#)

- I registri `pc` e `r0` vengono configurati in maniera da saltare alla funzione `task_entry_point()` passando come argomento il puntatore al descrittore del task nel registro `r0`
- I registri da `r1` ad `r12` (compresi `r4-r11`) assumeranno valore 0
- Il registro `lr` viene impostato a 0 in quanto come si vedrà la funzione `task_entry_point()` e' senza ritorno.
- Il registro `spsr` dev'essere impostato in maniera che la modalità di esecuzione sia **SYSTEM** e gli **IRQ** siano abilitati.

## La funzione `task_entry_point()`

`task_entry_point()` è il punto di ingresso di ogni task

L'argomento (in `r0`) è l'indirizzo del descrittore di task

```
void __attribute__((naked))
task_entry_point(struct task *t)
{
    for (;;) {
        if (!t->valid || !t->released)
            panic0();
        irq_enable();
        t->job(t->arg);
        irq_disable();
        --t->released;
        _sys_schedule();
    }
}
```



## La funzione `task_entry_point()` (2)



Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

`_irq_handler()`

`_irq_schedule()`

`_sys_schedule()`

Scheduler (2<sup>a</sup> vers.)

- Esegue un ciclo senza fine
  - Una iterazione per ciascun rilascio di un job del task
- La funzione del job è eseguita con interruzioni abilitate
- Per il resto il ciclo è eseguito con interruzioni disabilitate
- Quando un job termina si decrementa `t->released`
  - È il numero di job del task rilasciati e non completati
- La funzione `_sys_schedule()` seleziona un nuovo task da eseguire quando un job del task termina
  - È eseguita nel contesto di esecuzione dei job
  - La funzione “ritornerà” solo quando questo task verrà nuovamente selezionato dallo scheduler per l'esecuzione



## La funzione `create_task()`

`create_task()` deve “saltare” il task 0 (idle task)

```
int create_task(job_t job, void *arg,
               int period, int delay,
               int priority, const char *name)
{
    int i;
    struct task *t;
    for (i=1 ;i<MAX_NUM_TASKS; ++i)
        if (!taskset[i].valid)
            break;
    [...]
    t->releasetime = ticks + delay;
    t->released = 0;
    init_task_context(t, i);
    __memory_barrier();
    irq_disable();
    [...]
}
```



## Lo scheduler dei job

Costituito da diverse procedure:

- La funzione `check_periodic_tasks()`
  - Eseguita ad ogni tick in contesto interruzione
  - Controlla i rilasci dei job dei task periodici
- La funzione `schedule()`:
  - Eseguita nel contesto dei job
  - Scansiona il vettore di task per selezionare il job rilasciato di priorità massima

Chi invoca lo scheduler:

- La funzione Assembly `_irq_handler()`:
  - Alla conclusione della gestione dell'interruzione, esegue se necessario `schedule()`
- La funzione Assembly `_sys_schedule()`:
  - Invoca `schedule()` alla terminazione di un job



Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

`_irq_handler()`

`_irq_schedule()`

`_sys_schedule()`

Scheduler (2<sup>a</sup> vers.)

## La funzione `check_periodic_tasks()`

```
void check_periodic_tasks(void) {
    unsigned long now = ticks;
    struct task *f;
    int i;
    for (i=0, f=taskset+1; i<num_tasks; ++f) {
        if (f - taskset > MAX_NUM_TASKS)
            panic0();
        if (!f->valid)
            continue;
        if (time_after_eq(now, f->releasetime)) {
            ++f->released;
            f->releasetime += f->period;
            trigger_schedule = 1;
            ++globalreleases;
        }
        ++i;
    }
}
```



## La funzione `schedule()` – 1<sup>a</sup> versione

`schedule()` restituisce l'indirizzo del descrittore del task da eseguire, oppure `NULL` se il task corrente è già il migliore

```
struct task *schedule(void)
{
    struct task *best;
    unsigned long oldreleases;
    do {
        oldreleases = globalreleases;
        best = select_best_task();
    } while (oldreleases != globalreleases);
    trigger_schedule = 0;
    return (best != current ? best : NULL);
}
```



Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

\_irq\_handler()

\_irq\_schedule()

\_sys\_schedule()

Scheduler (2<sup>a</sup> vers.)

## La funzione `select_best_task()`

```
static inline
struct task *select_best_task(void) {
    int maxprio, i;
    struct task *best, *f;
    maxprio = MININT;
    best = &taskset[0];
    for (i=0, f=taskset+1; i<num_tasks; ++f) {
        if (f - taskset > MAX_NUM_TASKS)
            panic0();
        if (!f->valid)
            continue;
        ++i;
        if (f->released > 0 &&
            f->priority > maxprio)
            maxprio = f->priority, best = f;
    }
    return best;
}
```



Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

\_irq\_handler()

\_irq\_schedule()

\_sys\_schedule()

Scheduler (2<sup>a</sup> vers.)

## Modifica alla funzione `_irq_handler()` (1)

`_irq_handler()` deve consentire di invocare `schedule()` se necessario

```
[...]  
bx      r12 /* call to _bsp_irq() */  
msr     cpsr_c, #(SYS_MODE | NO_INT)
```

**∴ Nuove istruzioni ∴**

```
mov     r0, sp  
add     sp, sp, #(8*4)  
msr     cpsr_c, #(IRQ_MODE | NO_INT)  
mov     sp, r0  
ldr     r0, [sp, #(7*4)]  
msr     spsr_cxsf, r0  
ldmfd  sp, {r0-r3, r12, lr} ^  
nop  
ldr     lr, [sp, #(6*4)]  
movs   pc, lr
```



Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

`_irq_handler()`

`_irq_schedule()`

`_sys_schedule()`

Scheduler (2<sup>a</sup> vers.)

## Modifica alla funzione `_irq_handler()` (2)

Non si deve invocare `schedule()` se il flusso interrotto non appartiene a nessun job (ad es. interruzione annidata)

Introduciamo la variabile globale `irq_level` che tiene traccia del livello di annidamento raggiunto

```
unsigned long irq_level = 0ul;
void _bsp_irq(void) {
    u32 irqno;
    isr_t isr;
    ++irq_level;
    for (;;) {
        if (iomem(...) == 0 && ...) {
            --irq_level;
            return;
        }
        [...]
    }
}
```



## Modifica alla funzione `_irq_handler()` (3)

Non si deve invocare `schedule()` se

- `irq_level` è diverso da zero
- `trigger_schedule` è uguale a zero

```
msr      cpsr_c, #(SYS_MODE|NO_INT)
ldr      r0, =irq_level
ldr      r0, [r0]
tst      r0, r0
bne     .Lrestore
ldr      r0, =trigger_schedule
ldr      r0, [r0]
tst      r0, r0
beq     .Lrestore
```

[...]

`.Lrestore:`

```
mov     r0, sp
```



Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

`_irq_handler()`

`_irq_schedule()`

`_sys_schedule()`

Scheduler (2<sup>a</sup> vers.)



[Schema della lezione](#)[Cambio di contesto](#)[Creazione di un task](#)[Scheduler \(1ª vers.\)](#)[\\_irq\\_handler\(\)](#)[\\_irq\\_schedule\(\)](#)[\\_sys\\_schedule\(\)](#)[Scheduler \(2ª vers.\)](#)

## La funzione `_irq_schedule()` (1)

- Il valore restituito da `schedule()` è nel registro `r0`
- Se `r0` è nullo non occorre effettuare un cambio di contesto

```
_irq_schedule:  
    ldr    r12,=schedule  
    blx   r12  
    tst   r0,r0  
    beq   .Lrestore  
    ldr   r12,=_switch_to  
    blx  r12  
    cpsid if  
.Lrestore:  
[...]
```

Al termine di `_switch_to`:

- `sp` punta ora alla cima dello stack del task che deve essere eseguito
- Le interruzioni sono abilitate: forzare la disabilitazione attraverso `cpsid`

## La funzione `_sys_schedule()` (1)

- Invocata al termine di un job
  - Lo stack non contiene i valori salvati da `_irq_handler()`
- È necessario salvare i registri AAPCS-clobbered sullo stack
  - ... ma non è possibile utilizzare i registri della modalità **IRQ**

```
_sys_schedule:  
str    lr, [sp, #-(4*2)]!  
mrs    lr, cpsr  
str    lr, [sp, #4]  
ldr    lr, [sp]  
[...]
```

- Il valore di ritorno in `lr` viene salvato sullo stack
  - Punta all'interno del ciclo in `task_entry_point()`
  - `sp` viene aggiornato in modo da lasciare una posizione libera sotto la cima
- `cpsr` viene salvato sullo stack nella posizione libera
- `lr` viene recuperato dallo stack (la cima non è modificata)



## La funzione `_sys_schedule()` (2)



- Si salvano sullo stack i registri `r0-r3`, `r12` e `lr`
- Dopo il salvataggio la struttura dei valori sullo stack è esattamente identica a quella che si ha in seguito ad una interruzione
- La procedura termina eseguendo un salto entro la funzione `_irq_schedule()`

[Schema della lezione](#)

[Cambio di contesto](#)

[Creazione di un task](#)

[Scheduler \(1ª vers.\)](#)

[\\_irq\\_handler\(\)](#)

[\\_irq\\_schedule\(\)](#)

[\\_sys\\_schedule\(\)](#)

[Scheduler \(2ª vers.\)](#)

```
[...]  
ldr    lr, [sp]  
stmfd  sp!, {r0-r3, r12, lr}  
b      _irq_schedule
```

## Correzione della funzione `_switch_to()`

Poiché `_switch_to()` può essere invocata con le interruzioni già disabilitate, non è corretto ri-abilitarle al termine: è necessario ripristinare lo stato precedente.

Sostituire `irq_disable()` e `irq_enable()` con `irq_save()` e `irq_restore()`:

```
void __attribute__((naked))
_switch_to(struct task *to) {
    unsigned long flags;
    irq_save(flags);
    save_regs(current->regs);
    load_regs(to->regs);
    switch_stacks(current, to);
    current = to;
    irq_restore(flags);
    naked_return();
}
```



## Le macro `irq_save()` e `irq_restore()`

Per salvare lo stato corrente del registro `cpsr` e disabilitare le interruzioni:

```
#define irq_save(flags) \  
    __asm__ __volatile__ ("mrs %0, cpsr\n\t" \  
                          "cpsid i\n\t" \  
                          : "=r" ((flags)) : : "memory");
```

Per ripristinare lo stato precedente di `cpsr` (la "c" consente di preservare i flag di condizione, eventualmente aggiornati):

```
#define irq_restore(flags) \  
    __asm__ __volatile__ ("msr cpsr_c, %0" \  
                          : : "r" ((flags)) : "memory");
```



Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

`_irq_handler()`

`_irq_schedule()`

`_sys_schedule()`

Scheduler (2<sup>a</sup> vers.)

## La nuova funzione `_irq_schedule()`



- Alla luce delle modifiche a `_switch_to()`, non è più necessario forzare la disabilitazione delle interruzioni dopo l'invocazione:

```
_irq_schedule:  
    ldr    r12, =schedule  
    blx   r12  
    tst   r0, r0  
    beq   .Lrestore  
    ldr   r12, =_switch_to  
    blx   r12  
    cpsid if  
.Lrestore:  
[...]
```

Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

`_irq_handler()`

`_irq_schedule()`

`_sys_schedule()`

Scheduler (2<sup>a</sup> vers.)

## Analisi interrompibilità dello scheduler

- Adesso `schedule()` è sempre invocata con le interruzioni disabilitate, tuttavia tenerle disabilitate durante la scansione del vettore di task porterebbe a tempi di blocco eccessivi
- Scegliamo dunque di abilitarle durante la sola esecuzione della funzione `select_best_task()`

```
struct task *schedule(void) {  
    [...]  
    do {  
        oldreleases = globalreleases;  
        irq_enable();  
        best = select_best_task();  
        irq_disable();  
    } while (oldreleases != globalreleases);  
    [...]  
}
```



## Race condition in `schedule()`



- Problema: adesso la funzione `schedule()` è rientrante
  - Ad esempio: quando un job termina viene invocata
  - Durante la sua esecuzione avviene una interruzione di tick
- Utilizziamo come meccanismo di sincronizzazione una variabile “sentinella” `do_not_enter`

Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

`_irq_handler()`

`_irq_schedule()`

`_sys_schedule()`

Scheduler (2<sup>a</sup> vers.)



## La funzione `schedule()` – versione finale

```
struct task *schedule(void) {
    static int do_not_enter = 0;
    struct task *best;
    unsigned long oldreleases;

    if (do_not_enter)
        return NULL;
    do_not_enter = 1;
    do {
        oldreleases = globalreleases;
        irq_enable();
        best = select_best_task();
        irq_disable();
    } while (oldreleases != globalreleases);
    trigger_schedule = 0;
    do_not_enter = 0;
    return (best != current ? best : NULL);
}
```



Schema della lezione

Cambio di contesto

Creazione di un task

Scheduler (1<sup>a</sup> vers.)

`_irq_handler()`

`_irq_schedule()`

`_sys_schedule()`

Scheduler (2<sup>a</sup> vers.)