Realizzare un sistema multitasking in C

Anatomia di un ambiente multitasking e sua implementazione per l'impiego con applicazioni embedded prive di sistema operativo.

di Antonino Calderone

I microcontrollori sono ampiamente impiegati per la realizzazione di dispositivi elettronici in ambito domestico e industriale. Quando non è richiesta grande capacità di calcolo, sono adoperati dispositivi con architetture tipicamente a 8 e a 16 bit, dotati di pochi KB di RAM. L'approccio comunemente usato per la programmazione di questi sistemi, consiste nello scrivere applicativi (privi di sistema operativo) governati da un loop principale e da una serie di routine che eseguono delle macchine a stati.
Non di rado la parte infrastrutturale si confonde con il dominio del campo di applicazione. Unica concessione alla modernità è l'utilizzo di cross-compilatori C, corredati di tool e ambienti di sviluppo che tipicamente offrono l'implementazione (di un sottoinsieme) delle primitive della libreria ANSI del C.

I sistemi embedded complessi (per esempio quelli impiegati nel campo degli apparati per le telecomunicazioni) sono invece generalmente dotati di sistema operativo multitasking con prerogative di hard e soft real-time.

Tra i più noti citiamo VxWorks e pSOS, entrambi della WindRiver. Parliamo, in ogni caso, di macchine con megabyte e megahertz da spendere, non di micro-sistemi poveri di risorse.

In molti casi, anche quelli che abbiamo definito micro-sistemi potrebbero giovarsi di una soluzione multitasking.

Antonino Calderone acalderone@infomedia.it

Lavora come software engineer presso Marconi Communications SpA nella divisione Optical Networks. Inoltre collabora con la DigiFacta SW Engineering S.r.l. Si occupa principalmente di sistemi e protocolli per le telecomunicazioni, firmware, device driver, internals di sistemi operativi.

Tra i principali vantaggi, individuiamo un disegno architetturale più strutturato e rispettoso dei ruoli dominio-soluzione del problema.

Verrà di seguito descritta un'implementazione "a basso costo" in termini di risorse, quindi adatta a un micro-sistema, di un ambiente multitasking, facilmente portabile (scritta prevalentemente in C), e con la prerogativa di mantenere inalterata (o consentire) la possibilità di simulazione su sistemi desktop come Linux o Windows. Gli obiettivi primari della nostra implementazione sono quelli di realizzare:

FIGURA 1 Output del programma di prova generato compilando il codice sorgente del listato 1

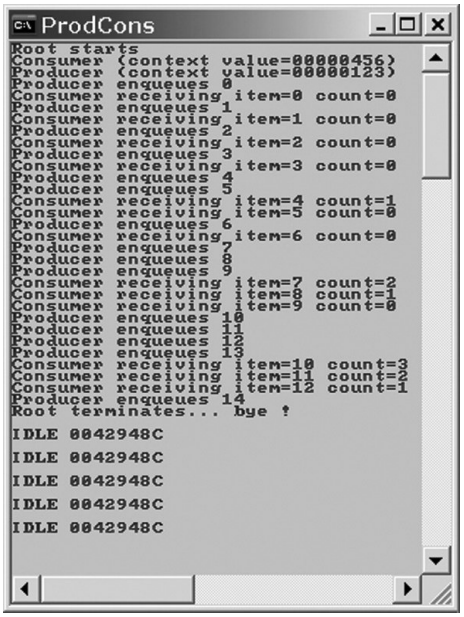
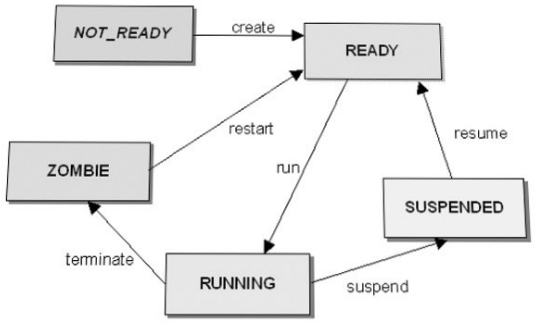


FIGURA 2 Diagramma degli stati di un task. Gli archi orientati indicano le operazioni che conducono allo stadio successivo



- un kernel per un ambiente multitasking cooperative ed event driven;
- un API completa, che consenta la gestione dei task e degli oggetti per la sincronizzazione.

Il **Listato 1** implementa un'applicazione di prova, scritta per questo ambiente, che realizza una soluzione del problema produttore-consumatore. In **Figura 1** è mostrato l'output prodotto dal programma compilato ed eseguito su sistema desktop.

La struttura del Kernel

La nostra descrizione riguarda principalmente il kernel del sistema, responsabile della gestione (creazione, scheduling e dispatching) dei task e degli oggetti per la sincronizzazione degli stessi.

Diciamo subito che la gestione delle risorse quali memoria, I/O, ecc., non rientra nelle prerogative della nostra architettura, poiché troppo dipendente dal sistema target.

L'implementazione delle funzioni di sistema (syscall) riflette in genere questa scelta.

Il kernel opera su aree di memoria allocate dall'utente (per esempio lo stack dei task, o i buffer per la gestione delle code), ad eccezione delle strutture primarie che ne costituiscono l'infrastruttura.

L'obiettivo della multiprogrammazione è quello di massimizzare l'utilizzo della CPU

In generale sappiamo che un sistema multiprogrammato, o meglio multitasking, consente l'esecuzione "concorrente" di più task in base a criteri denotati comunemente come politiche di scheduling. Noi adottiamo uno scheduling di tipo cooperative: il task in esecuzione si sospende cedendo volontariamente il controllo al sistema.

A questa scelta consegue quella di avere un approccio event driven, con una gestione degli eventi (signal) simile, per certi aspetti, a quella dei sistemi POSIX like.

TABELLA 1 Elenco delle principali syscall implementate. Per una descizione completa si faccia riferimento al file incluso nel progetto kernel.h

Nome	Oggetto	Descrizione
t_create	task	Crea un nuovo task. Lo stato di un nuovo task è READY.
t_delete	task	Cancella un task esistente. t_delete(0) cancella il task chiamante.
t_suspend	task	Sospende il task chiamante finché non risvegliato da t_resume.
tm_wkafter	task	Sospende il task chiamante per un numero di tick di sistema. tm_wkafter(0) restituisce il controllo allo scheduler, riottenendolo se non ci sono contemporaneamente altri task nello stato READY.
t_resume	task	Riattiva un task sospeso da t_suspend.
t_waitfor_signal	task	Sospende il task in attesa di un signal di tipo SIGUSRx
t_restart	task	Riesegue un task non cancellato che si trovi in stato ZOMBIE
mu_init	mutex	Inizializza un mutex
mu_lock	mutex	Acquisisce il mutex se già non fatto da un altro task, in tal caso sospende il task chiamante in attesa che il task proprietario del mutex invochi la mu_unlock.
mu_unlock	mutex	Rilascia il mutex.
q_init	queue	Inizializza un coda di eventi
q_send	queue	Aggiunge un elemento nella coda.
q_count	queue	Restituisce il numero di elementi accodati.
q_receive	queue	Sospende il task chiamante finchè almeno un messaggio non venga accodato.

Più precisamente il kernel contiene strutture e algoritmi per l'implementazione di:

- task e oggetti per la sincronizzazione (signal, mutex, queue);
- scheduler;
- dispatcher;
- le funzioni di sistema (le principali sono elencate in **Tabella 1**).

I task e la loro rappresentazione nel Kernel

I task sono rappresentati, all'interno del kernel, da una struttura denotata col nome di task descriptor o TD (**Listato 2**). Un TD si compone per difetto dei seguenti attributi:

- entry point della routine di esecuzione (il puntatore alla funzione di start-up del task), e parametri di start-up;
- una struttura utilizzata per salvare lo stato della CPU;
- lo stato del Task: (ready, running, ecc…);
- il puntatore allo stack (proprietario) del task;
- i campi per la gestione degli eventi (signal management);
- altri campi utilizzabili dallo scheduler per la gestione delle risorse associate al task.

Il kernel è proprietario della tabella dei TD, la cui dimensione è fissata e dipende dal numero massimo di task concorrenti istanziabili nel sistema (deciso in compile time). Una speciale entry con indice 0, individua la routine di idle. Questa routine è eseguita in caso di "inattività" del sistema.

Lo scheduler

La tabella dei TD è la struttura principale utilizzata per le attività di scheduling.

Lo scheduler processa ciascuna entry della tabella dei TD, applicando per quelli "validi" azioni scandite da una macchina a stati finiti il cui diagramma è mostrato in **Figura 2**.

Gli stati possibili per un task sono: READY, SUSPENDED, RUNNING, ZOMBIE e NOT_READY.

READY è lo stato di un task pronto all'esecuzione.

LISTATO 1 Codice sorgente dell'implementazione dell'applicazione di prova che realizza una soluzione del problema produttore-consumatore usando l'API del nostro ambiente multitasking (*continua*)

```
/* Listato 1 - Esempio: produttore / consumatore.
 * Il task di root, inizializza la coda usata da
 * produttore e consumatore. Crea entrambi i task e si sospende per
 * 50 tick di sistema. Il produttore, finché game_over
 * rimane 0, inserisce nella coda un valore
 * che viene scodato dal consumatore. Entrambi acquisiscono un
 * proprio mutex usato per sincronizzarsi con il task di root in fase
 * di terminazione. Il consumatore, se riceve un numero divisibile
 * per 3, si sospende per un numero di tick pari a quel numero.
 * Il task di root, scaduto il timeout dei 50 tick, decreta la fine del
 * gioco, quindi aspetta che produttore e consumatore terminino la propria
 * esecuzione (rilasciando il mutex usato per la sincronizzazione);
 * solo allora cancella i due task e se stesso. Il sistema rimane in IDLE.
 *
 * A. Calderone
 */
#include <stdio.h>
#include "kernel.h"

#define STACK_SIZE 2048
#define QUEUE_LEN 10

/* Mutex usati per sincronizzare roottask con consumer e producer */
static mutex_t sync_mutex_consumer = MU_INIT;
static mutex_t sync_mutex_producer = MU_INIT;

/* Istanziazione della coda usata da producer e consumer */
static queue_t aQueue;

/* Allocazione del pool di slot per la coda */
static queue_item_t aQueuePool[QUEUE_LEN];

static int game_over = 0; /* Il gioco puo' aver inizio ... */

/* Inizializzazione delle aree di memoria usate
   come stack per i task roottask, producer e consumer */
static char roottask_stack[STACK_SIZE];
static char producer_stack[STACK_SIZE];
static char consumer_stack[STACK_SIZE];

/* PRODUTTORE */
void producer(task_param_t context) {
  static queue_item_t aValue = 0;

  /* acquisisce il mutex */
  mu_lock(&sync_mutex_producer);

  printf("Producer (context value=%p)\n", context);

  while (! game_over) {
    printf("Producer enqueues %i\n", aValue);

    /* accoda il valore, e prepara il successivo */
    q_send(&aQueue, aValue++);

    /* consente ad altri task di andare in esecuzione */
    tm_wkafter(0);
  }
```

LISTATO 1 *(segue)*

```
 /* rilascia il mutex atteso da root */
  mu_unlock(&sync_mutex_producer);
};

/* CONSUMATORE */
void consumer(task_param_t context) {
  /* acquisisce il mutex */
  mu_lock(&sync_mutex_consumer);

  printf("Consumer (context value=%p)\n", context);

  while (! game_over) {
    queue_item_t item;
    queue_size_t count = q_receive(&aQueue, &item);
    printf("Consumer receiving item=%i count=%i\n", item, count);

    /* se item e' divisibile per 3, si sospende per il valore di item */
    if (item % 3 == 0) tm_wkafter(item);
  }

  /* rilascia il mutex atteso da root */
  mu_unlock(&sync_mutex_consumer);
};

/* ROOT */
void roottask(task_param_t context) {
  task_id_t consumer_tid, producer_tid;
  printf("Root starts\n");

  /* Inizializza la coda usata da producer e consumer */
  q_init(&aQueue, aQueuePool, (sizeof(aQueuePool)/sizeof(queue_item_t)));

  /* Crea i task producer e consumer */
  producer_tid = t_create(producer, (task_param_t) 0x123, /*priority */ 10,
                          producer_stack, sizeof(producer_stack));

  consumer_tid = t_create(consumer, (task_param_t) 0x456, /*priority */ 20,
                          consumer_stack, sizeof(consumer_stack));

  tm_wkafter(50); /* si sospende per 50 tick di sistema causando la ... */
  game_over = 1;  /* ... terminazione dei task produttore e consumatore */
  tm_wkafter(0);  /* cede il controllo allo scheduler  */
  mu_lock(&sync_mutex_producer); /* attende la terminazione del task produttore */
  mu_lock(&sync_mutex_consumer); /* attende la terminazione del task consumatore */

  /* Cancella i task, compreso se stesso, liberando i TD associati */
  t_delete(producer_tid);
  t_delete(consumer_tid);
  t_delete(0);

  printf("Root terminates... bye !\n");
}

void main() {
  /* Cede il controllo al kernel che fa partire il roottask */
  jmp_to_kernel(roottask, /* parametro */ 0, /* priorità */ 30,
                roottask_stack, sizeof(roottask_stack));
}
```

È applicato dallo scheduler un criterio di priorità per il quale se più task sono contemporaneamente nello stato READY, viene schedulato quello a priorità più alta. Se più task a più alta priorità hanno priorità identica, allora viene schedulato il task che attende da più "tempo" la CPU (starvation avoidance).

SUSPENDED è lo stato di un task che attende un signal (i signal sono essenzialmente dei flag del TD, usati per marcare il verificarsi di eventi). La sospensione di un task consiste nel salvare lo stato della CPU in un campo del TD associato al task correntemente in esecuzione. Un task si sospende invocando una primitiva che causa sotto certe condizioni (per esempio il locking di un mutex già "lockato") la cessione del controllo allo scheduler. Nel seguito chiameremo queste primitive (potenzialmente) "bloccanti".

RUNNING è lo stato di un task che è in esecuzione. Lo scheduler ottiene il controllo della CPU solo se il task lo cede sospendendosi.

ZOMBIE è lo stato di un task che termina la propria routine principale di esecuzione, ma non viene cancellato. Un task può cancellare se stesso prima di ritornare dalla routine di start-up, oppure può essere cancellato da un altro task mediante apposita syscall. Il kernel non è responsabile della decisione di cancellare un task, tale prerogativa spetta all'utente. Per altro esiste, nel nostro sistema, la possibilità di far ripartire un task (zombie); questo giustifica la transizione di stato da ZOMBIE a READY nel diagramma di **Figura 2**.

Esiste uno speciale stato detto NOT_READY che denota una entry nella tabella dei TD inutilizzata. Lo scheduler considera "non valido o riutilizzabile" un TD con stato NOT_READY e/o con entry_point nullo. In fase di inizializzazione ogni entry viene posta in questo stato.

La macchina a stati eseguita dallo scheduler tratta le transizioni di stato secondo azioni ben precise. Le abbiamo indicate nel diagramma di **Figura 2** associandole agli archi orientati. La creazione di un nuovo task (*create*) è l'associazione di una

entry non utilizzata della tabella dei TD al task stesso. L'indice di tale entry diventa l'identificatore del task (TID). La creazione del task che definiamo di root (il primo task eseguito dal kernel) è affidata alla routine *jmp_to_kernel*, usata per mandare in esecuzione il kernel stesso. Il TID del task di root vale 1.

Il dispatching di un task (*run*) in stato READY selezionato dallo scheduler, è ottenuto con l'operazione detta "cambio di contesto" (context switch). Quando la routine di esecuzione principale del task termina (*terminate*) lo scheduler ottiene il controllo: lo stato del task viene impostato a ZOMBIE; e quel task non viene più processato, ovvero il corrispondente descrittore non viene riassegnato, fino alla cancellazione (oppure alla riesecuzione) del task stesso. Un task può sospendersi invocando una syscall (*suspend*). Un task si sospende in attesa di un qualche evento. Il kernel periodicamente monitora il verificarsi dell'evento atteso.

Se tale evento si verifica, lo scheduler rimette il task in stato READY (*resume*), in attesa di cedergli appena possibile la CPU. Se un task viene cancellato il suo descrittore è reimpostato a NOT_READY. Il descrittore torna a essere riassegnabile per la creazione di un nuovo task. La cancellazione può essere effettuata invocando una specifica syscall (t_delete) da qualunque task del sistema. Un task può cancellare se stesso passando 0 a t_delete.

Comunque la cancellazione non avviene per scelta dello scheduler, quindi non è stata esplicitamente indicata nel diagramma di **Figura 2** (si potrebbe pensare di aggiungere un arco da qualunque stato a NOT_READY, poiché il task può essere cancellato qualunque sia il suo stato).

LISTATO 2 Definizione implementativa del task descriptor

```
/* TASK descriptor
 */
typedef unsigned int task_stack_size_t;
typedef unsigned int reg_t;
typedef unsigned char task_signal_mask_t;
typedef unsigned char task_signal_t;
typedef unsigned char task_priority_t;
typedef int task_timer_counter_t;
typedef void (STDCALL * task_entry_point_t)(task_param_t);
typedef struct __task_t {
  task_entry_point_t entry_point;   /* task entry point */
  task_param_t param;               /* user's context cookie */

  task_priority_t prio;             /* task priority */

  task_status_t status;             /* task status: READY, RUNNING, ... */

  union {                           /* special purpose flags*/
    task_flag_t flags;
    task_flag_mask_t i_flags;
  };

  task_timer_counter_t timer_count; /* used by tm_wkafter syscall */

  task_signal_mask_t signal_mask;    /* bit oriented signal mask */
  task_signal_mask_t signal_pending; /* pending signals */
  task_signal_mask_t signal_waiting; /* waiting signals */

  registers_state_t reg_state;       /* cpu regs status */
  reg_t saved_stack_pointer;

  char* task_stack;
  task_stack_size_t stack_size;

} task_t;
```

Il context switch in C: setjmp e longjmp

Quando una primitiva bloccante "decide" di sospendere il task chiamante, deve essenzialmente preservarne lo stato e restituire il controllo al kernel.

La nostra implementazione si giova di due primitive della libreria ANSI del C: setjmp e longjmp.

Ne riportiamo i prototipi, definiti nel file header setjmp.h

```
int setjmp( jmp_buf env );
void longjmp( jmp_buf env, int value );
```

La routine setjmp fotografa lo stato del processore, salvando lo stato dei registri (compresi flag e program counter) in un'istanza della variabile di tipo jmp_buf. La primitiva longjmp, che accetta come parametro tale istanza, recupera lo stato salvato, saltando nel punto di chiamata della setjmp.

Task può definirsi un'attività del sistema la cui esecuzione ha la prerogativa di essere sequenziale

Controllando il valore di setjmp è possibile determinare se si proviene da un salto o da una normale

invocazione. Nel primo caso setjmp restituisce il secondo argomento passato a longjmp (con l'eccezione che se il parametro fosse 0 il valore restituito sarebbe 1). Nel secondo caso setjmp restituisce zero. Questa semantica ricorda per certi aspetti quella della syscall fork dello standard POSIX. Il meccanismo è semplice e soprattutto portabile. Queste primitive vengono usate dal dispatcher per effettuare lo scambio di contesto, ogni volta che un processo precedentemente sospeso viene rischedulato. Le primitive bloccanti restituiscono il controllo allo scheduler usando una longjmp per ripristinarne lo stato precedentemente salvato.

Il dispatcher

Oltre allo stato dei registri, è necessario garantire a ogni task e al kernel una copia privata dello stack. Il dispatcher dunque, prima di mandare in esecuzione un task, deve ottenere che questi operi su un proprio stack. In questo caso non abbiamo altra possibilità che ricorrere all'assembly. Il **Listato 3** mostra l'implementazione delle due primitive SAVE_AND_SET_STACK_POINTER e RESTORE_STACK_POINTER, fatta in assembly x86. Il dispatcher, usando le due primitive, è in grado di preservare lo stato dello stack del kernel salvando lo stack pointer (SP) in memoria per ripristinarlo un volta ottenuto nuovamente il controllo. Il dispatcher cede il controllo a un task in due modi distinti:

- esecuzione (o riesecuzione) di un "nuovo" task (chiamando la funzione entry point del task);
- ripristino dell'esecuzione di un task sospeso (salto con longjmp al marcatore della setjmp fatta dalla chiamata sospensiva).

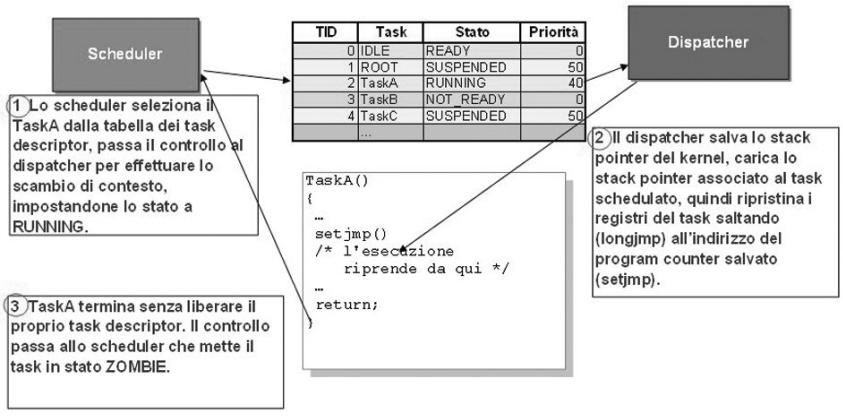
LISTATO 3 Implementazione in assembly x86 delle primitive SAVE_AND_SET_STACK_POINTER e RESTORE_STACK_POINTER

```
#if (defined(_MSC_VER) || defined(__BORLANDC__))
/* If you are using Visual C++ ** DO NOT ** compile with /GZ option,
 * GX- must replace /GX, and avoid /YX
 */
#define SAVE_AND_SET_STACK_POINTER(__OLD_SP, __STACK_P)\
  __asm { mov ebx, [__OLD_SP] }\
  __asm { mov eax, esp }\
  __asm { mov [ebx], eax }\
  __asm { mov eax, __STACK_P }\
  __asm { mov esp, eax }

#define RESTORE_STACK_POINTER(__OLD_SP)\
  __asm { mov eax, __OLD_SP }\
  __asm { mov esp, [eax] }

#elif defined (__GNUC__)

#define SAVE_AND_SET_STACK_POINTER(__OLD_SP, __STACK_P)\
__asm__ __volatile__ ( \
                    "movl " # __OLD_SP ", %ebx\n\t"\
                    "movl %esp, %eax\n\t"\
                    "movl %eax, (%ebx)\n\t"\
                    "movl " # __STACK_P ", %eax\n\t"\
                    "movl %eax, %esp\n\t")

#define RESTORE_STACK_POINTER(__OLD_SP)\
__asm__ __volatile__ ( \
                    "movl " # __OLD_SP ", %eax\n\t"\
                    "movl %eax, %esp\n\t")
#else
  #error No compiler/os supported !!!
#endif
```

Nel primo caso, il dispatcher salva lo SP del kernel; imposta il nuovo stack, assegnando allo SP register un puntatore all'area dello stack il cui indirizzo è un attributo del TD, quindi esegue la funzione di start-up del task. Nel secondo caso, si tratta di un task la cui esecuzione era stata sospesa.

Lo stato dei registri è contenuto nel TD, quindi per restituire il controllo procede con l'esecuzione della primitiva longjmp, simmetricamente a quanto fatto dalla routine bloccante che ha ceduto il controllo allo scheduler. Il punto di ritorno di un task (quando la routine di start-up termina) è gestito dal

FIGURA 3 Esempio schematico delle interazioni tra i moduli del kernel durante l'esecuzione



LISTATO 4 Implementazione del dispatcher

```
#define SAVE_CPU_REGS setjmp
#define RESTORE_CPU_REGS(x) longjmp(x, 1)

/* ... */

dispatcher:
  saved_stack_pointer = &p_task->saved_stack_pointer;
  top_of_the_stack = &p_task->task_stack[p_task->stack_size];

  /* Before executing a task, save current scheduler state */
  if (SAVE_CPU_REGS(KERNEL_ENV.scheduler_registers_state)) {
    /* Returning from a scheduled TASK:
     * restore the stack pointer and restart scheduling */
    RESTORE_STACK_POINTER(saved_stack_pointer);

    candidate_task_prio = 0;
    candidate_p_task = 0;

    goto scheduler;
  }

  /* Task was suspended ?  */
  if (p_task->flags.wkup) {
    p_task->flags.wkup = 0; /* Ok, reset wkup flag */
    RESTORE_CPU_REGS(p_task->reg_state);/* Resume a suspended task
*/
  }
  else {
    /* First execution: set up the task stack frame */
    SAVE_AND_SET_STACK_POINTER(saved_stack_pointer, top_of_the_
stack);
    /* Run the task */
    p_task->entry_point( p_task->param );
  }

  goto scheduler;
}
```

dispatcher a valle della chiamata della routine stessa. Il dispatcher in questo caso ripristina lo SP del kernel e cede il controllo allo scheduler. L'implementazione del dispatcher è mostrata nel **Listato 4**.

> task condividono il medesimo spazio d'indirizzamento, ma preservano una copia proprietaria dello stack

Interrupt e timer

I sistemi real-time devono garantire che certi compiti siano svolti entro un tempo prestabilito.

In genere, nelle applicazioni embedded prive di sistema operativo, si ottiene il rispetto di tali prerogative svolgendo questi compiti all'interno delle routine di interrupt associate ad eventi scatenati dall'hardware. Se decidiamo di mantenere quest'approccio, possiamo ragionevolmente pensare di affidarci alla nostra architettura, invece, per attività meno critiche, anche se comunque scandite dal tempo.

L'attuale implementazione non affronta queste problematiche, che per la loro complessità richiederebbero una trattazione specifica.

In generale, una efficace iterazione dell'ambiente multitasking con eventi asincroni ed esterni ad esso come gli interrupt, richiede una sincronizzazione che si può ottenere abilitando e disabilitando questi eventi durante l'esecuzione del codice in regioni considerate critiche, cioè per le quali si considera requisito indispensabile la non interrompibilità con eventuale cambio di contesto.

Potremmo pensare di dotarci quindi al minimo di due primitive, DISABLE_INTERRUPT ed ENABLE_INTERRUPT, da implementare opportunamente, attraverso le quali proteggere le strutture del kernel accedute all'interno delle Interrupt Service Routine (ISR).

Conclusioni

Disporre di un "sistema multi-tasking on process" può essere utile in molti altri contesti. Abbiamo enfatizzato l'applicazione sui micro-sistemi.

Il ricorso a simili paradigmi in campi differenti da quello suggerito è meno raro di quanto si possa immaginare, per esempio nell'ambito della progettazione del software per gli apparati per le telecomunicazioni.

In tal caso i principali vantaggi riscontrati sono la portabilità e la controllabilità del sistema (e non solo), usando un'implementazione certamente meno snella e più attenta a prerogative legate alle prestazioni e alla capacità di gestire efficacemente eventi nel dominio del tempo.

Non ci resta che rimandare il lettore ai sorgenti dell'implementazione completa. Riteniamo troverà molto interessante anche gli aspetti implementativi qui solo accennati.

CODICE ALLEGATO

ftp.infomedia.it  **MultiC**