Analisi di RTLinux

Fabio Dalla Libera - 547226-IF

Progetto per il corso di Sistemi operativi 2 del Professor Sergio Congiu

Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Specialistica in Ingegneria Informatica

Anno accademico 2006/2007

Indice

Capitolo 1

Introduzione

Lo scheduler di Linux, sebbene sia stato fortemente modificato e migliorato nel passaggio dal kernel 2.4 al kernel 2.6 (si veda [3] o [4] a riguardo) non offre garanzie nel tempo di terminazione dei processi, caratteristica fondamentale di ogni sistema real-time.

Sono però disponibili moltissimi sistemi operativi real-time basati su Linux, alcuni dei quali sono elencati in [5], differenziabili in base alla soluzione adottata per avere determinismo nei tempi di risposta. In particolare un primo approccio, utilizzato da uCLinux (sistema operativo nato per i microcontrollori privi di MMU) consiste nell'eliminazione di funzionalità dal kernel Linux standard che introducono non determinismo. La seconda soluzione, impiegata da MontaVista Linux e da KURT (Kansas University Real-Time Linux), prevede la sostituzione dello scheduler di Linux con un algoritmo più deterministico e la modifica del kernel in modo da dare garanzie sui tempi di esecuzione, limitando ad esempio il più possibile la durata delle sezioni non interrompibili ed effettuando più spesso chiamate allo scheduler. In RTAI (RTAI - the RealTime Application Interface for Linux) ed RTLinux, invece, viene creato un nucleo real-time che si frappone tra l'hardware ed il kernel di Linux, il quale viene eseguito come processo a priorità inferiore rispetto ai job real-time ed anziché agire sull'hardware per la gestione degli interrupt agisce su un'emulazione fornita dal kernel real-time.

RTLinux, inizialmente sviluppato da Victor Yodaiken al New Mexico Institute of Mining and Technology è ora disponibile in due versioni, una free, denominata appunto RTLinux Free, ed una commerciale, chiamata RTLinux Pro. La versione Free è rilasciata sotto una specifica licenza, denominata Open RTLinux Patent License e reperibile sul sito [15], che rinforza ulteriormente le condizioni della licenza GPL Versione 2 nel richiedere che i lavori di modifica ed aggiunta derivati da RTLinux siano rilasciati con licenza GPL. La versione Pro supporta le architetture x86, x86 a 64 bit, PowerPC, Fujitsu FR-V, Arm e StrongARM, XScale, MIPS e Alpha mentre la versione free supporta solo x86, PowerPC, MIPS ed Alpha, anche se sullo stesso sito di RTLinux Free viene indicato che sostanzialmente l'unica architettura supportata dalla versione Free è x86. Emblematico è il commento " TODO How will this work on PPC" che si trova in alcuni file sorgente di RTLinux. Nel seguito si farà sempre riferimento alla versione Free per x86 di RTLinux, avendo a disposizione per i test solo architetture x86, ed in particolare alla versione 3.1 per kernel Linux 2.4.29, scelta secondo i criteri riportati nella sezione relativa all'installazione a pagina 37.

Nella seguente relazione, come si potrà intuire fin da subito, ci si è prefissati l'obiettivo di riportare solo ciò che è stato personalmente prodotto o esaminato in maniera scrupolosa, evitando di realizzare una fusione di vario materiale non verificato, magari incoerente, e tralasciando introduzioni alle nozioni che si danno per scontate per uno studente che abbia seguito il corso di Sistemi Operativi 2.

Capitolo 2

Caratteristiche di RTLinux

In molte delle fonti riportate in bibliografia (ad esempio [26]) si afferma che RTLinux effettua una virtualizzazione dell'hardware utilizzata dal kernel di Linux, il quale diviene uno strato software che agisce sopra di essa, e vengono fornite rappresentazioni come quella riportata in figura 2.1. In realtà l'unica cosa che viene effettivamente virtualizzata quando è in esecuzione RTLinux è la gestione degli interrupt; il kernel di Linux viene modificato mediante una patch in modo che l'abilitazione e disabilitare delle interruzioni sia solo virtuale e non hardware, ma a parte questo, comunque, il kernel continua ad operare direttamente sull'hardware, ed infatti in [27] viene riportata lo schema di figura 2.2. Addirittura RTLinux potrebbe essere visto come un gestore di interrupt caricato nel kernel di Linux come un qualsiasi altro driver.

RTLinux è infatti costituito da un insieme di moduli del kernel[1]; quello principale, una volta caricato, gestisce le interruzioni hardware e permette di definire handler (ossia le routine di servizio) per tali interrupt, invocati con una latenza molto bassa. Grazie a questo modulo è possibile in particolare intercettare gli interrupt del timer, e quindi un secondo modulo può fungere da scheduler per i processi real-time. Tale modulo contiene inoltre tutte le funzioni necessarie a creare thread, a sincronizzarli ed a gestire la mutua esclusione. Vi sono poi due moduli utilizzabili per scambiare dati tra i thread real-time di RTLinux ed i processi utente di Linux, uno che permette di definire aree di memoria condivisa ed uno che consente la creazione di code FIFO. A loro volta, i programmi realizzati per RTLinux saranno costituiti da uno o più moduli del kernel ed eventualmente da alcuni programmi che girano in user space e svolgono operazioni che non hanno deadline hard.

2.1 Gestione delle interruzioni

Un'imprecisione presente in molti siti su RTLinux, come [22],[23] o [24], riguarda il fatto che si afferma che la virtualizzazione appena descritta è ottenuta mediante ridefinizione delle macro *cli()* e *sti()* di Linux con delle macro *S_CLI*, *S_STI* ed *S_IRET*, utilizzate per simulare rispettivamente la disabilitazione degli interrupt, la loro abilitazione ed il ritorno da interruzione (la "*S* " che precede i tre nomi indica appunto che si tratta di operazioni "soft" ossia non eseguite effettivamente sull'hardware). Queste tre macro sono descritte nel lavoro di Michael Barabanov, ex studente di V. Yodaiken, "A Linux based Operating System" reperibile ad esempio da [25] e riportate in figura 2.3.

Tuttavia osservando il sorgente del kernel di Linux modificato o addirittura il solo file di patch si può notare che tali macro non sono più presenti nella versione 3.1 (l'articolo [25] fa infatti riferimento alla versione 0.5a), e sono state sostituite da varie parti di codice. Più in dettaglio

[1] Il kernel di Linux permette l'inserimento, runtime, di moduli, costituiti da normali file oggetto (file ".o") che, una volta caricati mediante il comando *insmod,* diventano parte integrante del kernel. I moduli hanno due funzioni, *int init_module(void)* e void *cleanup_module(void),* che vengono chiamate, rispettivamente, al momento dell'inserimento e della rimozione (comando *rmmod*) del modulo stesso; si può immaginare che la funzione *init_module* sia per un modulo ciò che la funzione *main* rappresenta per i programmi comuni.

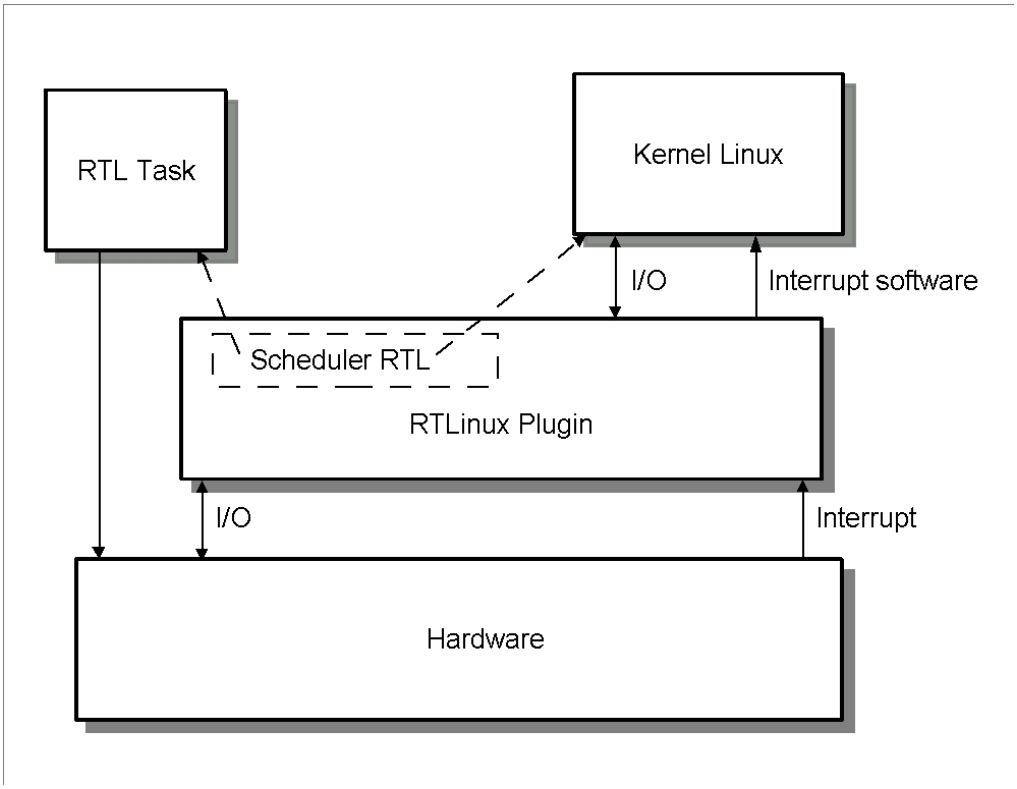

Figura 2.1: Rappresentazione del funzionamento di RTLinux riportata in [26].

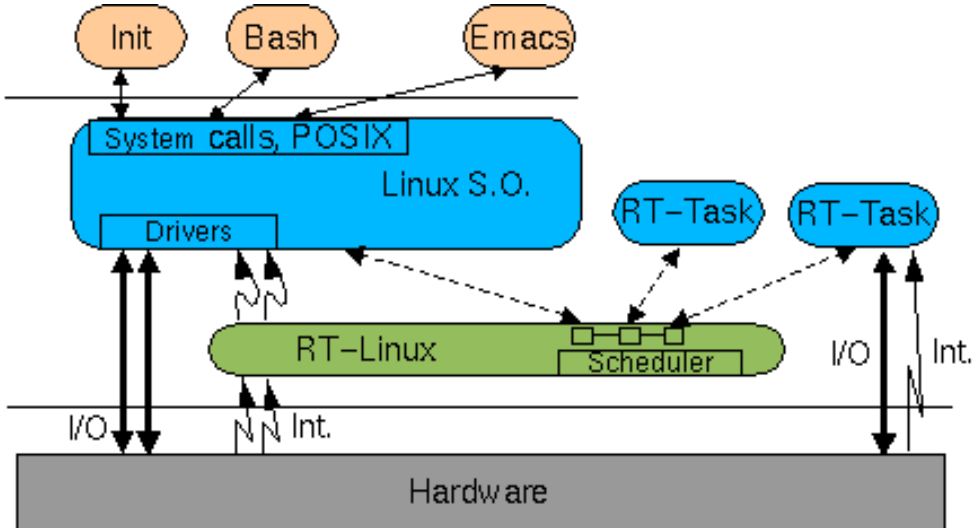

Figura 2.2: Rappresentazione del funzionamento di RTLinux riportata in [27].

```
S_CLI:  movl $0, SFIF

S_IRET: push %ds
        pushl %eax
        pushl %edx
        movl $KERNEL_DS, %edx
        mov %dx,%ds
        cli
        movl SFREQ,%edx
        andl SFMASK,%edx
        bsrl %edx,%eax
        jz not_found
        movl $0,SFIF
        sti
        jmp SFIDT (,%eax,4)
not_found:
        movl $1,SFIF
        sti
        popl %edx
        popl %eax
        pop %ds
        iret

S_STI:  pushfl
        pushl $KERNEL_CS
        pushl $done_STI
        S_IRET
done_STI:
```

Figura 2.3: Le tre macro definite nel lavoro "A Linux based RealTime Operating System" di Michael Barabanov.

nel file[2] *linux/include/asm-i386/rtlinux_cli.h* viene definita la seguente struttura dati denominata *irq_control_s*:

```
struct irq_control_s {
        void (*do_save_flags)(unsigned long *);
        void (*do_restore_flags)(unsigned long);
        void (*do_cli)(void);
        void (*do_sti)(void);
        void (*do_local_irq_save)(unsigned long *);
        void (*do_local_irq_restore)(unsigned long);
};
```

Come si può notare, tale struttura è un insieme di puntatori a funzione che include le funzioni *do_cli* e *do_sti*. Nel file *linux/arch/i386/kernel/rtlinux.c* viene dichiarata la variabile *irq_control*, che viene inizializzata in questo modo:

```
struct irq_control_s irq_control = {
        rtl_hard_save_flags_f,
        rtl_hard_restore_flags_f,
        rtl_hard_cli_f,
        rtl_hard_sti_f,
        rtl_hard_local_irq_save_f,
        rtl_hard_local_irq_restore_f
 };
```

dove *rtl_hard_cli_f* e *rtl_hard_sti_f* sono due funzioni presenti in *linux/arch/i386/kernel/rtlinux.c* che realizzano, come indica il nome, le funzione *cli* e *sti* direttamente sull'hardware:

```
static void rtl_hard_cli_f(void) { rtl_hard_cli_kernel(); }
static void rtl_hard_sti_f(void) { rtl_hard_sti_kernel(); }
```

visto che *rtl_hard_cli_kernel()* e *rtl_hard_sti_kernel()* sono macro definite in *linux/include/asm-i386/rtlinux_cli.h* come (per le altre funzioni di *irq_control* la struttura del codice è del tutto analoga):

```
#define rtl_hard_cli_kernel() __asm__ __volatile__("cli": : :"memory")
#define rtl_hard_sti_kernel() __asm__ __volatile__("sti": : :"memory")
```

Tali macro si possono ritrovare con un'altro nome in un kernel di Linux a cui non sia applicata la patch. Infatti nel file *linuxnp/include/asm-i386/system.h* è presente la sezione di direttive

```
#ifdef CONFIG_SMP

extern void __global_cli(void);
extern void __global_sti(void);
extern unsigned long __global_save_flags(void);
extern void __global_restore_flags(unsigned long);
#define cli() __global_cli()
#define sti() __global_sti()
#define save_flags(x) ((x)=__global_save_flags())
#define restore_flags(x) __global_restore_flags(x)

#else

#define cli() __cli()
#define sti() __sti()
#define save_flags(x) __save_flags(x)
```

[2]Tutti i percorsi di file riportati fanno riferimento alla directory radice dei sorgenti di RTLinux Free 3.1, supposto che in tale directory sia presente una sottodirectory (o un link simbolico ad una directory) denominata *linux* e contenente i sorgenti del kernel di Linux a cui è stata applicata la patch per l'installazione di RTLinux; per indicare invece i sorgenti del kernel di Linux prima dell'applicazione della patch viene utilizzato come nome di directory iniziale *linuxnp*. Qualora risulti chiaro, ad esempio perché se ne è già parlato, nel seguito spesso verrà omesso il percorso dei vari file per non appesantire eccessivamente la trattazione.

```
#define restore_flags(x) __restore_flags(x)

#endif
```

che definisce *cli()* e *sti()* come *_global_cli()* e *_global_sti()* o come *__cli()* e *__sti()* a seconda che si compili o meno con il supporto per architetture multiprocessore(*_global_cli()* e *_global_sti()* sono definite nel file *linuxnp/arch/i386/kernel/irq.c* e contengono l'acquisizione ed il rilascio di un lock oltre all'invocazione di *__cli()* e *__sti()*); sempre nel file *system.h* si trova la definizione di *__cli()* e *__sti()*, identica a quella vista per *rtl_hard_cli_kernel()* e *rtl_hard_sti_kernel()* :

```
 #define __cli()   __asm__ __volatile__("cli": : :"memory")
 #define __sti()   __asm__ __volatile__("sti": : :"memory")
```

Nel kernel di Linux a cui è applicata la patch per RTLinux, invece, in *linux/include/asm-i386/system.h* dopo la definizione di *__cli()* e *__sti()* appena riportata le seguenti direttive

```
#ifdef CONFIG_RTLINUX
#include <asm/rtlinux_cli.h>
#endif
#ifdef CONFIG_SMP
```

comportano l'inclusione del file *linux/include/asm-i386/rtlinux_cli.h*; Questo file fa in modo che le invocazioni di *cli()* e *sti()* coincidano con le chiamate delle corrispondenti funzioni presenti in *irq_control* mediante le seguenti ridefinizioni:

```
#undef __cli
#undef __sti
#undef __save_flags
#undef __restore_flags
#undef local_irq_save
#undef local_irq_restore
#undef local_irq_disable
#undef local_irq_enable
#define __save_flags(x) irq_control.do_save_flags(&x)
#define __restore_flags(x) irq_control.do_restore_flags(x)
#define __cli()         irq_control.do_cli()
#define __sti()         irq_control.do_sti()
#define local_irq_save(x)       irq_control.do_local_irq_save((&x))
#define local_irq_restore(x)    irq_control.do_local_irq_restore(x)
#define local_irq_disable()     irq_control.do_cli()
#define local_irq_enable()      irq_control.do_sti()
```

Dato che, come si è appena visto, le definizioni di *rtl_hard_cli_kernel()* e *rtl_hard_sti_kernel()* corrispondono a quelle di *__cli()* e *__sti()* del kernel di Linux originale, l'inizializzazione della struttura *irq_control* non comporta alcuna variazione del comportamento del kernel di Linux, il quale non subisce alcuna virtualizzazione nel passaggio degli interrupt. Tuttavia l'aver definito tale struttura permette al nucleo di RTLinux di attuare tale virtualizzazione quando viene avviato. Grazie ad un meccanismo simile anche l'emulazione dell'istruzione di ritorno da interrupt può essere virtualizzata, infatti in *linux/arch/i386/kernel/entry.S*, file assembly che contiene le chiamate di sistema e le routine di gestione delle eccezioni, è possibile notare il seguente codice

```
 * (c) Victor Yodaiken 1999
 * RTLINUX_IRET emulates the turn on interrupts effect of
 * iret since, under RTLinux we may have pended interrupts
 * that will not be automatically enabled on an iret.
 */
#ifdef CONFIG_RTLINUX
#define RTLINUX_IRET    \
        movl  rtl_emulate_iret,%eax; \
        testl %eax, %eax; \
        je  1f; \
        call  *%eax;    \
```

```
        1:
#else
#define RTLINUX_IRET
#endif
```

dove *rtl_emulate_iret* è un puntatore a funzione definito nella seguente sezione di codice di *linux/arch/i386/kernel/rtlinux.c*:

```
void *rtl_emulate_iret = 0;
void *rtl_exception_intercept = 0;
void *rtl_syscall_intercept = 0;

extern void *__start_rtlinux_patch,*__stop_rtlinux_patch;
extern asmlinkage void do_IRQ(struct pt_regs );
extern void ret_from_intr(void);
RTLINUX_EXPORT(__start_rtlinux_patch);
RTLINUX_EXPORT(__stop_rtlinux_patch);
RTLINUX_EXPORT(rtl_emulate_iret);
RTLINUX_EXPORT(irq_desc);
RTLINUX_EXPORT(do_IRQ);
RTLINUX_EXPORT(ret_from_intr);
RTLINUX_EXPORT(rtl_exception_intercept);
```

e *RTLINUX_EXPORT* è la seguente macro presente in *linux/include/asm-i386/rtlinux.h*:

```
#define RTLINUX_EXPORT(x)\
asm (".section␣rtlinux_funcs,\"a\"\n␣.align␣4\n"\
".long␣"#x"\n␣.previous\n" );
```

Si può osservare che il puntatore *rtl_emulate_iret* è inizializzato a zero, cosicché nel codice della macro *RTLINUX_IRET* viene esegue il salto all'etichetta *"1"* ossia non si esegue nulla; qualora invece il puntatore punti ad una funzione (l'indirizzo sia non nullo) questa viene eseguita. Il commento riportato subito prima della definizione di tale macro specifica anche che normalmente subito dopo l'esecuzione di *iret* se ci sono interruzioni pendenti si può avere l'invocazione di interrupt service routine, ma nel caso si introduca una virtualizzazione dell'hardware tale effetto deve essere emulato dal software. La macro *RTLINUX_IRET* viene impiegata nel file *linux/arch/i386/kernel/entry.S* nella entry *ret_from_sys_call:*

```
ENTRY(ret_from_sys_call)
        call *(SYMBOL_NAME(irq_control)+8) # cli from irq_control structure
        cmpl $0,need_resched(%ebx)
        jne reschedule
        cmpl $0,sigpending(%ebx)
        jne signal_return
        call *(SYMBOL_NAME(irq_control)+12) # sti from irq_control
           structure

restore_all:
        RTLINUX_IRET
        RESTORE_ALL

        ALIGN
signal_return:
        call *(SYMBOL_NAME(irq_control)+12) # sti from irq_control
           structure
        RTLINUX_IRET
        testl $(VM_MASK),EFLAGS(%esp)
        movl %esp,%eax
        jne v86_signal_return
        xorl %edx,%edx
        call SYMBOL_NAME(do_signal)
        jmp restore_all
```

```
        ALIGN
v86_signal_return:
        call SYMBOL_NAME(save_v86_state)
        movl %eax,%esp
        xorl %edx,%edx
        call SYMBOL_NAME(do_signal)
        jmp restore_all

        ALIGN
tracesys:
        movl $-ENOSYS,EAX(%esp)
        call SYMBOL_NAME(syscall_trace)
        movl ORIG_EAX(%esp),%eax
        cmpl $(NR_syscalls),%eax
        jae tracesys_exit
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)             # save the return value
tracesys_exit:
        call SYMBOL_NAME(syscall_trace)
        jmp ret_from_sys_call
badsys:
        movl $-ENOSYS,EAX(%esp)
        jmp ret_from_sys_call
```

Confrontanto il kernel originale e quello modificato in questa sezione di codice si può notare come si siano sostituiti *cli* e *sti* assembly (per cui la modifica delle funzioni *cli()* e *sti()* non avrebbe avuto effetto) con le invocazioni delle rispettive funzioni in *irq_control*.

Anche altre entry vengono modificate dalla patch di RTLinux, in particolare

- la entry *system_call:*

```
ENTRY(system_call)
        pushl %eax                      # save orig_eax
        SAVE_ALL
#ifdef CONFIG_RTLINUX
        movl  rtl_syscall_intercept,%ebx
        testl %ebx, %ebx
        je  991f
        call  *%ebx
        movl ORIG_EAX(%esp), %eax
991:
#endif
        GET_CURRENT(%ebx)
        testb $0x02,tsk_ptrace(%ebx)    # PT_TRACESYS
        jne tracesys
        cmpl $(NR_syscalls),%eax
        jae badsys
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)             # save the return value
```

- la entry *device_not_available*

```
ENTRY(device_not_available)
        pushl $-1               # mark this as an int
        SAVE_ALL
#ifdef CONFIG_RTLINUX
        movl  rtl_exception_intercept,%eax
        testl %eax, %eax
        je  8f
        movl %esp,%edx
```

```
        pushl $0                # no error code
        pushl %edx
        pushl $7                # vector
        call  *%eax
        addl $12,%esp
        testl %eax, %eax
        je 8f
        RESTORE_ALL
8:
#endif
        GET_CURRENT(%ebx)
        movl %cr0,%eax
        testl $0x4,%eax                 # EM (math emulation bit)
        jne device_not_available_emulate
        call SYMBOL_NAME(math_state_restore)
        jmp ret_from_exception
device_not_available_emulate:
        pushl $0                # temporary storage for ORIG_EIP
        call  SYMBOL_NAME(math_emulate)
        addl $4,%esp
        jmp ret_from_exception
```

- e la entry *divide_error*

```
#define RTL_PUSH_VECTOR(vector,routine) \
1:      cmpl $routine, %edi; \
        jne 1f; \
        pushl $vector; \
        jmp 7f;

ENTRY(divide_error)
        pushl $0                # no error code
        pushl $ SYMBOL_NAME(do_divide_error)
        ALIGN
error_code:
        pushl %ds
        pushl %eax
        xorl %eax,%eax
        pushl %ebp
        pushl %edi
        pushl %esi
        pushl %edx
        decl %eax                       # eax = -1
        pushl %ecx
        pushl %ebx
        cld
        movl %es,%ecx
        movl ORIG_EAX(%esp), %esi       # get the error code
        movl ES(%esp), %edi             # get the function address
        movl %eax, ORIG_EAX(%esp)
        movl %ecx, ES(%esp)
        movl %esp,%edx
        pushl %esi                      # push the error code
        pushl %edx                      # push the pt_regs pointer
        movl $(__KERNEL_DS),%edx
        movl %edx,%ds
        movl %edx,%es
#ifdef CONFIG_RTLINUX
        movl  rtl_exception_intercept,%eax
        testl %eax, %eax
```

```
        je  8f
        RTL_PUSH_VECTOR(0,do_divide_error)
        RTL_PUSH_VECTOR(1,do_debug)
        RTL_PUSH_VECTOR(3,do_int3)
        RTL_PUSH_VECTOR(4,do_overflow)
        RTL_PUSH_VECTOR(5,do_bounds)
        RTL_PUSH_VECTOR(6,do_invalid_op)
        RTL_PUSH_VECTOR(8,do_double_fault)
        RTL_PUSH_VECTOR(9,do_coprocessor_segment_overrun)
        RTL_PUSH_VECTOR(10,do_invalid_TSS)
        RTL_PUSH_VECTOR(11,do_segment_not_present)
        RTL_PUSH_VECTOR(12,do_stack_segment)
        RTL_PUSH_VECTOR(13,do_general_protection)
        RTL_PUSH_VECTOR(14,do_page_fault)
        RTL_PUSH_VECTOR(15,do_spurious_interrupt_bug)
        RTL_PUSH_VECTOR(16,do_coprocessor_error)
        RTL_PUSH_VECTOR(17,do_alignment_check)
        RTL_PUSH_VECTOR(18,do_machine_check)
        RTL_PUSH_VECTOR(19,do_simd_coprocessor_error)
1:      push $-1
7:      mov %ecx, %ebx                  # ebx will not be corrupted by
   a C routine
        call  *%eax
        popl %ecx
        mov %ebx, %ecx
        testl %eax, %eax
        je 8f
        addl $8,%esp
        RESTORE_ALL
8:
#endif
        GET_CURRENT(%ebx)
        call *%edi
        addl $8,%esp
        jmp ret_from_exception
```

Dove *rtl_syscall_intercept* ed *rtl_exception_intercept* sono, similmente a quanto visto per *rtl_emulate_iret*, dei puntatori a funzione inizializzati a valore nullo definiti in *linux/arch/i386/kernel/rtlinux.c* ed a cui è applicata la macro *RTLINUX_EXPORT*, come si può vedere dalle linee di codice riportate a pagina 8. In tale sorgente è anche possibile notare la presenza della dichiarazione di funzione (extern) *do_IRQ,* che fa riferimento alla funzione *do_IRQ* implementata in *linux/arch/i386/kernel/irq.c,* la quale viene chiamata, nel normale kernel di Linux, ad ogni interrupt hardware. Grazie alla macro *RTLINUX_EXPORT* è poi possibile ottenere, durante l'avvio di RTLinux, l'indirizzo di tale funzione, per poter intercettare gli interrupt e passarli al kernel di Linux. Infatti andando ad analizzare il codice del modulo principale, *rtl.o*, possiamo notare che in *rtl_core.c* all'inizio di *init_module* vi sono le istruzioni

```
int ret;

if ( arch_takeover() )
{
        printk("arch_takeover␣failed\n");
        return -1;
}
```

dove *arch_takeover* è una funzione di *main/arch/arch.h*[3] che a sua volta chiama la funzione *patch_kernel*, definita nello stesso file ed utilizzata, oltre che per sostituire le funzioni di *irq_control*

[3]Nonostante l'estensione, questo file contiene vario codice e non solo definizioni di funzioni e costanti.

prima viste con la loro versione soft, per fare in modo che ad ogni interruzione venga chiamata la funzione *rtl_intercept* di *rtl_core.c.*

Una volta caricato il modulo *rtl.o* tutti gli interrupt continuano ad essere passati al kernel di Linux, tuttavia tale modulo fornisce la funzione

*int rtl_request_global_irq(unsigned int irq, unsigned int (*handler)(unsigned int, struct pt_regs *))*

che permette di registrare come handler per l'irq indicato dal primo parametro la funzione il cui indirizzo è fornito come secondo argomento. Qualora in un secondo momento non sia più necessario intercettare gli interrupt è possibile invocare la funzione

int rtl_free_global_irq(unsigned int irq)

che fa in modo che l'interrupt torni ad essere passato al kernel di Linux. E' anche possibile mandare un interrupt a Linux (si deve ricordare infatti che quando è in esecuzione *rtl.o* si ha una virtualizzazione degli interrupt passati a Linux, grazie alle funzioni di *irq_control*) mediante un'ulteriore funzione di *rtl_core.c,*

void rtl_global_pend_irq(int ix)

Infine si possono installare e rimuovere *handler* per irq software (attivabili sempre con *rtl_global_pend_-*
irq) utilizzando, rispettivamente,

*int rtl_get_soft_irq(void (*handler)(int, void *, struct pt_regs *), const char * devname)*;

e

void rtl_free_soft_irq(unsigned int irq)

La parola *global,* presente nel nome di tutte le funzioni appena citate, indica che le funzioni si riferiscono a tutte le cpu nel caso di sistema multiprocessore, tuttavia molti dei commenti lasciano intuire che il supporto per le architetture SMP nella versione 3.1 sia ancora poco affidabile, ad esempio prima di *arch_takeover* si trova il commento "TODO Must test smp synchronization!!!" ed in *rtl_core.c* è possibile leggere "TODO soft smp_processor_id doesn't work here???? – Michael" e "TODO resolve the smp synchronization problem here".

2.2 Gestione dei timer

L'unità di misura del tempo maggiormente utilizzata nel codice di RTLinux è il nanosecondo, ed in particolare viene impiegato il tipo di dato *hrtime_t* definito in */include/i386/rtl_time.h* come *long long* ossia come intero a 64bit; il riferimento temporale, ossia l'"istante 0" viene fatto coincidere con il momento in cui è stato acceso il sistema. Per misurare i tempi su architettura x86 RTLinux si utilizza, se disponibile sul processore, il Time-Stamp Counter, un contatore incrementato ad ogni ciclo di clock presente su Pentium e superiori(poiché tale contatore è a 64bit l'overflow su una CPU a 3Ghz si può avere solo dopo oltre 190 anni); se tale componente non è presente viene utilizzato il timer 8254, disponibile in tutti i processori x86 ma avente risoluzione minore visto che la frequenza del suo clock è di 1193180 Hz.

Per programmare la generazione di interrupt RTLinux utilizza invece il timer 8254 nel caso di architettura a singolo processore ed il timer fornito dall'architettura APIC (acronimo di Advanced

Programmable Interrupt Controllers) nel caso di SMP, infatti in *schedulers/i386/rtl_time.c* si può trovare la funzione *rtl_getbestclock* così definita:

```
clockid_t rtl_getbestclock (unsigned int cpu)
{
#ifdef CONFIG_X86_LOCAL_APIC
        if (smp_found_config) {
                return &_apic_clock[cpu];
        } else {
                return &_i8254_clock;
        }
#else
        return &_i8254_clock;
#endif
}
```

dove *_apic_clock[cpu]* e *_i8254_clock* sono strutture corrispondenti ai timer indicati dal nome ed istanziate nello stesso file. Tali variabili sono di tipo *rtl_clock*, una struttura definita in *include/rtl_time.h* ed utilizzata per la gestione di un generico timer (contiene, ad esempio, un puntatore alla funzione che lo inizializza ed uno alla funzione che permette di leggerne il valore); come si potrà intuire dal codice appena riportato, il tipo *clockid_t* è semplicemente un puntatore a tale struttura.

Evidentemente, visto che il TSC procede alla frequenza della CPU, è necessario effettuare una conversione per ottenere il tempo in nanosecondi; sfruttando il timer utilizzato per la generazione di interruzioni il fattore di conversione viene impostato automaticamente dalla funzione *do_calibration* presente in *rtl_time.c* ed invocata da *init_hrtime*.

Il modulo *rtl_time.o* si occupa di fornire varie funzioni per la gestione dei timer, in particolare per leggere il tempo è possibile utilizzare una delle seguenti funzioni

- *clock_gethrtime* che opera utilizzando uno dei campi *rtl_clock* cui si è appena accennato ed è definita in *rtl_time.h* come

  ```
  static inline hrtime_t clock_gethrtime (clockid_t clock_id)
  {
          return clock_id->gethrtime (clock_id);
  }
  ```

- *int clock_gettime(clockid_t clock_id, struct timespec *tp)*
  del tutto analoga, che utilizza una struttura dati (*timespec*) per rappresentare il tempo, mediante due campi che indicano rispettivamente i secondi ed i nanosecondi.

- *hrtime_t gethrtime(void)*, definita in *rtl_time.c* da

  ```
  hrtime_t gethrtime(void)
  {
          return rtl_do_get_time();
  }
  ```

  dove *rtl_do_get_time* è un puntatore a funzione impostato (dalla funzione *init_hrtime* del file *schedulers/i386/rtl_time.c*, invocata da *init_module* dello stesso file) in modo da ottenere il tempo dal TSC se questo è presente o dall'8254 in caso contrario.

Per mdificare il valore del timer *clock_id* è invece possibile invocare la funzione

*int clock_settime(clockid_t clock_id, const struct timespec *tp)*

che ritorna il valore -1 nel caso l'operazione non sia possibile[4].

[4]La seconda delle funzioni per la lettura e quest'ultima funzione sono conformi allo standard POSIX 1003.1b (chiamato anche "Real-time Extensions") come molte delle funzioni che verranno presentate in seguito. Ciò rende le applicazioni scritte per RTLinux facilmente portabili su altri sistemi operativi conformi a tale standard.

Oltre ai tre clock descritti (8254, TSC ed APIC) se in fase di compilazione si abilita il supporto per il Real Time Clock (direttiva *CONFIG_RTL_CLOCK_GPOS*) viene compilato un ulteriore variabile di tipo *rtl_clock*, chiamata *clock_gpos*, che corregge il tempo ritornato da *gethrtime* utilizzando il Real Time Clock in modo da avere il tempo misurato a partire da mezzanotte del primo Gennaio 1970.

2.3 Gestione dei thread e Scheduler

La struttura dati utilizzata per definire un thread, presente in *include/rtl_sched.h,* è chiamata *rtl_thread_struct* ed è definita nel seguente modo:

```
struct rtl_thread_struct {
        int *stack;     /* hardcoded */
        int fpu_initialized;
        RTL_FPU_CONTEXT fpu_regs;
        int uses_fp;
        int *kmalloc_stack_bottom;
        struct rtl_sched_param sched_param;
        struct rtl_thread_struct *next;
        int cpu;
        hrtime_t resume_time;
        hrtime_t period;
        hrtime_t timeval;
        struct module *creator;
        void (*abort)(void *);
        void *abortdata;
        int threadflags;
        rtl_sigset_t pending;
        rtl_sigset_t blocked;
        void *user[4];
        int errno_val;
        struct rtl_cleanup_struct *cleanup;
        int magic;
        struct rtl_posix_thread_struct posix_data;
        void *tsd [RTL_PTHREAD_KEYS_MAX];
};
```

mentre il tipo di dato corrispondente ad un puntatore a tale struttura, *pthread_t*, è dato semplicemente da:

```
typedef struct rtl_thread_struct *pthread_t;
```

Tra i campi di *rtl_thread_struct* è possibile notare la presenza dei seguenti elementi

- *stack* e *kmalloc_stack_bottom* sono puntatori utilizzati per la gestione di uno stack indipendente per ciascun thread;
- *uses_fp, fpu_initialized* e *fpu_regs* riguardano la gestione dell'unità floating point della CPU, visto che per default si suppone che i thread di RTLinux non la utilizzino, in modo da velocizzare i cambi di contesto (se un thread ne fa uso bisogna specificarlo esplicitamente);
- *sched_param* fa riferimento ai parametri di schedulazione; tale campo è di tipo *struct rtl_sched_param,* che per la macro

  #define rtl_sched_param sched_param;

  coincide con la struttura *sched_param* di *include/rtl_sched.h* contenente il solo campo intero *sched_priority*;
- *cpu* indica su quale CPU il processo deve essere schedulato (nel caso di SMP);

- *resume_time* indica il momento in cui il prossimo job del task verrà rilasciato[5];
- *period* se nullo indica che il task è costituito da un solo job, altrimenti rappresenta il periodo del task;
- *timeval* viene usato quando il task viene posto in attesa (ad esempio nella funzione *clock_nanosleep* di *schedules/rtl_sched.c)*;
- *threadflags* viene impiegato per salvare vari flag relativi alla cancellazione (terminazione) ed al join (attesa della terminazione di un altro thread) dei thread, oltre che il flag *RTL_THREAD_TIMERARMED,* utilizzato per indicare che il tempo di rilascio del prossimo job è stato impostato e che tale job non è ancora stato rilasciato;
- *posix_data* contiene altri dati necessari ad effettuare il join tra thread, infatti la struttura *rtl_posix_thread_struct* definita in include/rtl_posix.h è così composta:

```
struct rtl_posix_thread_struct
{
        struct rtl_thread_struct *joining_thread;
        void *retval;
        void *joined_thread_retval;
        pthread_spinlock_t exitlock;
};
```

- *pending* e *blocked* sono utilizzati, rispettivamente, per memorizzare quali segnali sono attivi[6] e quali si intendono ignorare;
- *magic* è un intero impostato per ogni thread a *RTL_THREAD_MAGIC*, una costante del tutto arbitraria e pari a 0x79433743, in modo da poter controllare, quando si sta operando su un thread, se questo è effettivamente un thread di RTLinux;

Questa struttura dati (o meglio il suo puntatore *pthread_t*) è alla base dell'implementazione delle funzioni POSIX per la gestione dei thread, ad esempio per creare un thread è necessario invocare

*int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)*

dove *thread* è un puntatore al thread creato che viene restituito, *start routine* è la funzione che deve essere eseguita dal thread (analoga al metodo *run()* dei Thread di Java), *arg* è l'argomento che viene passato a tale funzione e *attr* è un puntatore ad una struttura utilizzata per impostare alcuni degli attributi del thread, definita il *rtl_sched.h* come

```
typedef struct STRUCT_PTHREAD_ATTR {
        size_t stack_size;
        void *stack_addr;
        struct rtl_sched_param sched_param;
        int cpu;
        int use_fp;
        rtl_sigset_t initial_state;
        int detachstate;
} pthread_attr_t;
```

Si può notare come sia possibile ritrovare il nome di vari campi della struttura *rtl_thread_struct* infatti questi vengono semplicemente copiati da *pthread_create* nel thread che viene creato; uniche eccezioni sono il campo initial_state, che viene copiato come campo *pending,* e *detachstate*, utilizzato

[5] Con i termini job e task, ora come in seguito, si fa riferimento alle definizioni di tali concetti date in [1].

[6] I segnali di RTLinux funzionano in modo simile a quelli di Linux, tuttavia RTLinux (almeno nella versione analizzata) non prevede che un processo (in questo caso thread) possa registrare handler per i vari segnali. Questo argomento sarà ripreso ed affrontato nella sezione relativa a sincronizzazione e comunicazione.

per impostare *threadflags*. Particolare attenzione va dedicata ai parametri relativi allo stack, infatti mentre all'interno delle funzioni *init_module* è possibile impostare il puntatore *stack_addr* al valore 0 in modo che venga allocato automaticamente dello spazio per lo stack, se un thread real-time ne crea un altro deve fornire anche un puntatore ad uno spazio utilizzabile come stack, altrimenti *pthread_create* ritorna errore ed il nuovo task non viene creato.

Pur essendo possibile definire una variabile di tipo *pthread_attr_t* e impostare i vari campi prima di invocare *pthread_create*, vengono fornite varie funzioni[7], il cui nome inizia con *pthread_attr_* che sono utilizzabili per leggere modificare i singoli campi con un approccio simile alla programmazione ad oggetti (funzioni *get* e *set*).

Per terminare un thread è possibile utilizzare la funzione

int pthread_cancel(pthread_t thread)

che sostanzialmente invia un segnale chiamato *RTL_SIGNAL_CANCEL* al thread *thead*; invocando invece la funzione

*void pthread_exit(void *retval)*

il thread chiamante viene terminato. Un ulteriore opzione consiste nell'utilizzo della funzione

int pthread_delete_np (pthread_t thread)

che però non è POSIX, come indicano le due lettere *np* acronimo di "Not Portable".

Come si sarà intuito dalla presenza dei campi *period* e *resume_time* nell'implementazione di RTLinux vi è poi particolare attenzione ai task periodici. Infatti le API comprendono la funzione

int pthread_make_periodic_np (pthread_t p, hrtime_t start_time, hrtime_t period)

che imposta per il thread *p* il tempo di rilascio del prossimo job al valore *start_time* ed il periodo *period* (utilizzando la convenzione sul periodo nullo riportata nella descrizione dell'attributo *period* di *rtl_thread_struct*); Viene poi fornita la funzione

int pthread_wait_np(void)

utilizzabile all'interno del codice dei thread per far sì che l'esecuzione si arresti fino al rilascio del prossimo job.

Le API contengono anche tre funzioni per bloccare l'esecuzione del thread chiamante per un determinato periodo di tempo, in particolare

int usleep (useconds_t interval)

che permette di indicare il tempo in microsecondi,

*int nanosleep(const struct timespec *rqtp, struct timespec *rmtp)*

che utilizza la rappresentazione del tempo mediante la struttura *timespec* cui si è accennato precedentemente e

*int clock_nanosleep(clockid_t clock_id, int flags, const struct timespec *rqtp, struct timespec *rmtp)*

che permette di specificare molte opzioni, descritte in [37]. Un thread può invece sospenderne

[7] Una lista completa delle API di RTLinux è fornita in [36].

un'altro (eventualmente anche se stesso) invocando

int pthread_suspend_np (pthread_t thread)

o risvegliarlo mediante

int pthread_wakeup_np (pthread_t thread)

2.3.1 Analisi dello scheduler

Lo scheduler di RTLinux, implementato dal modulo *rtl_sched.o,* è uno scheduler a priorità fissa che esegue come task a priorità più bassa i processi gestiti dal kernel di Linux, il quale potrebbe essere visto a tutti gli effetti come un background server[8] che esegue processi aperiodici, visto che i processi di Linux non presentano deadline hard. Per ciascun thread real-time può essere specificata una priorità, memorizzata come *int* a 32 bit nel campo *sched_priority* della struttura *sched_param* vista prima. A Linux viene assegnata una priorità pari a -1, quindi le priorità utilizzabili per i processi real-time vanno da 0 a $2^{31} - 1$, dato che si intende come più prioritario un task avente *sched_priority* maggiore e se si assegnasse una priorità inferiore a -1 il thread non verrebbe mai eseguito. E' perciò possibile assegnare ad ogni thread una diversa priorità, almeno se il processore è a 32 bit, visto che in tal caso i possibili valori di priorità sono pari addirittura a metà della memoria indirizzabile. Ciò è un po' in contrasto con quanto riportato nelle pagine 166 e 167 di [1] dove si afferma che per i sistemi operativi real-time si usano solitamente meno di 256 livelli di priorità perché se si utilizza un opportuna mappatura tra le priorità dei processi e le priorità effettivamente disponibili nel sistema operativo, la diminuzione della schedulabità (di RM) causata dall'avere un numero limitato di livelli di priorità è bassissima qualora vi siano almeno 256 livelli di priorità.

Il file principale dello scheduler è *schedulers/rtl_sched.c*, dove viene dichiarato un array, chiamato *rtl_sched,* avente tanti elementi quante sono le cpu del sistema; ogni elemento è una struttura di tipo *rtl_sched_cpu_struct* (definita in *rtl_sched.h*) che contiene, tra i vari campi, i seguenti:

```
struct rtl_thread_struct *rtl_current;
struct rtl_thread_struct rtl_linux_task;
struct rtl_thread_struct *rtl_task_fpu_owner;   /* the task whose FP
   context is currently in the FPU unit */
struct rtl_thread_struct *rtl_tasks; /* the queue of  RT tasks */
clockid_t clock;
```

dove *rtl_current* è un puntatore al thread in stato di running, *rtl_linux_task* è un puntatore al thread che corrisponde all'esecuzione di Linux, *rtl_task_fpu_owner* è un puntatore all'ultimo thread che ha utilizzato l'unità floating point, *rtl_tasks* è, come indica il commento, un puntatore alla cosa dei task e *clock* è il timer utilizzato per la generazione di interrupt necessari ad invocare lo scheduler. La creazione del task per Linux e l'inizializzazione del campo *clock* per ciascuna CPU avviene nella funzione *init_module* mediante le istruzioni

```
for (i = 0; i < rtl_num_cpus(); i++) {
        cpu_id = cpu_logical_map (i);
        s = &rtl_sched [cpu_id];
        s -> rtl_current = &s->rtl_linux_task;
        s -> rtl_tasks = &s->rtl_linux_task;
        s -> rtl_new_tasks = 0;

        rtl_spin_lock_init (&s->rtl_tasks_lock);

        s -> rtl_linux_task . magic = RTL_THREAD_MAGIC;
        rtl_sigemptyset(&s -> rtl_linux_task . pending);
        rtl_sigaddset(&s -> rtl_linux_task . pending, RTL_SIGNAL_READY);
```

[8] Con la definizione data dal libro di testo [1] a pagina 192.

```
        s -> rtl_linux_task . blocked = 0;
        s -> rtl_linux_task . threadflags = 0;
        s -> rtl_linux_task . sched_param . sched_priority = -1;
        s -> rtl_linux_task . next = 0;
        s -> rtl_linux_task . uses_fp = 1;
        s -> rtl_linux_task . fpu_initialized = 1;
        s -> rtl_linux_task . creator = 0;
        s -> rtl_linux_task . abort = 0;

        s -> rtl_task_fpu_owner = &s->rtl_linux_task;
        s -> sched_flags = 0;
        rtl_posix_init (&s->rtl_linux_task);

        s-> clock =  rtl_getbestclock (cpu_id);
        if (s->clock && rtl_setclockhandler (s->clock,
           rtl_sched_timer_interrupt) == 0) {
                s->clock->init(s->clock);
        } else {
                rtl_printf("Can't␣get␣a␣clock␣for␣processor␣%d\n",cpu_id);
                rtl_restore_interrupts (interrupt_state);
                return  -EINVAL;
        }
}
```

Le righe appena riportate oltre che impostare *clock* al valore ritornato da *rtl_getbestclock* (funzione analizzata in precedenza) registrano la funzione *rtl_sched_timer_interrupt* come handler per tale timer e lo inizializzano. Nel caso di architettura a singolo processore, come spiegato nella sezione sulla gestione del tempo, il timer utilizzato è l'8254 perciò la funzione effettivamente invocata è *_8254_init* di *schedulers/i386/rtl_time.c,* nella quale compaiono le seguenti righe

```
        rtl_request_global_irq(0, _8254_irq);
        _8254_settimermode (clock, RTL_CLOCK_MODE_ONESHOT);
        _i8254_clock.settimer (clock, HRTIME_INFINITY);
```

che dopo aver registrato la funzione *_8254_irq* come gestore dell'interrupt 0 (corrispondente appunto al timer 8254 nell'architettura suddetta) impostano il timer nella modalità one-shot (ossia il timer deve essere riprogrammato ogni volta per generare un interrupt). Il tempo in cui deve essere generata l'interruzione viene poi posto a *HRTIME_INFINITY*, una costante utilizzata per indicare il massimo tempo misurabile dal timer. Andando ad osservare la funzione *_8254_setoneshot*, a cui punta *_i8254_clock.settimer* in questo caso, si evince che un controllo iniziale limita il tempo impostato al valore *MAX_LATCH_ONESHOT*:

```
static int _8254_setoneshot (clockid_t c, hrtime_t interval)
{
        rtl_irqstate_t flags;
        long t;
        rtl_spin_lock_irqsave (&lock8254, flags);
        if (interval > MAX_LATCH_ONESHOT) {
                interval = MAX_LATCH_ONESHOT;
        }

        t = RTIME_to_8254_ticks (interval); /* - _8254_latency); */
        if (t < 1) {
                t = 1;
        }
        WRITE_COUNTER_ZERO_ONESHOT(t);
        _i8254_clock.arch.istimerset = 1;

        rtl_spin_unlock_irqrestore(&lock8254, flags);
        return 0;
}
```

Il valore di *MAX_LATCH_ONESHOT* viene calcolato in *init_hrtime* secondo la formula

*MAX_LATCH_ONESHOT = LATCH_NS * 3 / 4*

dove *LATCH_NS* è il tempo (in nanosecondi) dell'interrupt del timer utilizzato in Linux e pari a dieci millisecondi; quindi, quando si utilizza la costante *HRTIME_INFINITY* in realtà si riceve un interrupt dopo soli 7.5 millisecondi. All'arrivo di un'interruzione del timer viene invocata la funzione *_8254_irq*, che a sua volta

- riabilita l'interrupt 0,
- invoca l'handler del clock, in questo caso la funzione *rtl_sched_timer_interrupt,*
- segnala a Linux l'interruzione 0 (interrupt software) invocando *_8254_checklinuxirq()* (in questa funzione si effettua un controllo sul fatto che siano trascorsi almeno dieci millisecondi dall'ultima interruzione).

La funzione *rtl_sched_timer_interrupt* è data semplicemente da

```
static void rtl_sched_timer_interrupt( struct pt_regs *regs)
{
        clear_bit (RTL_SCHED_TIMER_OK, &LOCAL_SCHED->sched_flags); \
        rtl_schedule();
}
```

dove *rtl_schedule* è la funzione riportata al termine di questa sezione e presente in *rtl_sched.c* che implementa lo scheduler. Essa viene invocata in vari momenti, in particolare quando

- arriva un interruzione del timer, come appena visto,
- un thread viene creato mediante l'invocazione di *pthread_create* (che a sua volta chiama *rtl_startup*, dove è effettivamente contenuta l'invocazione dello scheduler),
- un thread invoca *pthread_exit* per terminare la sua esecuzione,
- si rende un thread periodico eseguendo *pthread_make_periodic_np* o se ne imposta il periodo con *pthread_setperiod_np,*
- si attende la terminazione di un thread con *pthread_join,*
- si sospende un thread con *pthread_suspend_np, usleep, nanosleeep* o *clock_nanosleep,*
- si risveglia un thread con *pthread_wakeup_np,*
- si fa lock su un mutex già acquisito o si attende su un semaforo non libero mediante, rispettivamente, le funzioni *pthread_mutex_lock* di *schedulers/rtl_mutex.c* e *sem_wait di schedulers/rtl_sema.c*, due funzioni che invocano la funzione *rtl_wait_sleep* di *rtl_mutex.c.*

Analizzando il codice dello scheduler è possibile notare che ad ogni sua invocazione le interruzioni della CPU che sta eseguendo il codice vengono disabilitate mediante l'esecuzione di

rtl_no_interrupts(interrupt_state)

e riattivate subito prima di uscire dallo scheduler con

rtl_restore_interrupts(interrupt_state)

dove la variabile *interrupt_state* viene utilizzata per salvare e poi ripristinare i flag relativi agli

interrupt. Nelle prime righe è possibile notare che lo scheduling avviene indipendentemente per ciascun processore (cosa che si poteva già intuire dal fatto che vi è una coda di processi per ogni elemento dell'array *rtl_sched*), infatti una volta individuata la CPU che sta eseguendo il codice mediante *int cpu_id = rtl_getcpuid()* viene selezionato con la variabile *sched* l'elemento di *rtl_sched* corrispondente a tale processore.

Dopo aver letto dal timer di *sched* (rappresentato dall'8254 nella situazione che si considera, architettura x86 a singolo processore) il tempo corrente ed averlo immagazzinato nella variabile *now* si procede ad esaminare tutti i task presenti in coda, e per ciascuno di essi se il task ha un job che deve essere rilasciato (*test_bit(RTL_THREAD_TIMERARMED, &t->threadflags)*) allora si controlla se il tempo corrente è maggiore del tempo di rilascio del job (*now >= t->resume_time*); se ciò è vero si indica che il job è stato rilasciato (*clear_bit(RTL_THREAD_TIMERARMED, &t->threadflags)*), si aggiunge il segnale *RTL_SIGNAL_TIMER* a quelli pendenti del task (*rtl_sigaddset (&t->pending, RTL_SIGNAL_TIMER)*) e si imposta il tempo di rilascio del prossimo job. Questo viene posto a *HRTIME_INFINITY* se il task è composto da un singolo job (*t->period* nullo) e pari al tempo di rilascio del job appena rilasciato più il periodo nel caso contrario. Il ciclo che segue quest'ultima assegnazione poi continua ad aggiungere un tempo pari al periodo finché il tempo di rilascio del prossimo job non è nel futuro (questo comportamento è disabilitabile abilitando *CONFIG_RTL_OLD_TIMER_BEHAVIOUR* in fase di compilazione, tuttavia tale direttiva non viene presentata come opzione nelle fase di configurazione dell'installazione e tanto meno è indicata nella documentazione ufficiale); ciò fa sì che RTLinux si comporti in modo diverso rispetto al modello a job periodici adottato in [1], ed un esempio di questo fatto verrà fornito nella sezione relativa agli esercizi. Il flag *RTL_THREAD_TIMERARMED,* come si può intuire, viene modificato dalle funzioni *pthread_make_periodic_np* o *pthread_wait_np* che lo attivano mediante l'utilizzo della macro *__rtl_setup_timeout* di *rtl_sched.h.*

Sempre nel ciclo che esamina tutti i task della coda si eseguono le operazioni necessarie ad individuare il prossimo thread che deve diventare running, in particolare la variabile *new_task*, inizializzata a zero prima del ciclo, viene posta al task in esame se questo ha almeno un segnale non bloccato (si ricordi che anche il rilascio di un job viene indicato da un segnale attivo) e se la variabile *new_task* ha valore nullo o punta ad un task di priorità strettamente inferiore:

```
if ((t->pending & ~t->blocked) && (!new_task ||
                        (t->sched_param.sched_priority > new_task->
                           sched_param.sched_priority))) {
                                new_task = t;
                }
```

Osservando il modo in cui viene selezionato il task da eseguire si può capire che nel caso di due thread alla stessa priorità (eventualità facilmente evitabile, dato l'altissimo valore di livelli disponibili) non si adotta una scelta di tipo FIFO ma semplicemente si opta per il primo thread della coda, ossia, visto che la coda è una linked list semplice e l'inserzione avviene in testa (funzione *add_to_task_list*), il thread scelto sarà quello creato per ultimo.

Il passo successivo consiste nel reimpostare il timer per una nuova interruzione qualora la modalità di funzionamento sia one-shot (come avviene per il 8254 date le inizializzazioni prima descritte) e non lo si sia ancora fatto dall'ultima interruzione:

```
if (sched->clock->mode == RTL_CLOCK_MODE_ONESHOT && !test_bit (
   RTL_SCHED_TIMER_OK, &sched->sched_flags)) {
        if ( (preemptor = find_preemptor(sched,new_task))) {
                (sched->clock)->settimer(sched->clock, preemptor->
                   resume_time - now);
        } else {
                (sched->clock)->settimer(sched->clock, (HRTICKS_PER_SEC /
                   HZ) / 2);
        }
        set_bit (RTL_SCHED_TIMER_OK, &sched->sched_flags);
 }
```

Il codice appena riportato illustra che sotto tali condizioni viene invocata la routine *find_preemptor,* la quale ritorna il task che per primo potrebbe effettuare pre-emption sul task che sta per diventare running, ossia restituisce il task a priorità più elevata di *new_task* avente tempo di rilascio del suo prossimo job inferiore. Come si può intuire dal codice, se tale task non esiste viene ritornato zero, ed in tal caso il prossimo interrupt viene programmato *(HRTICKS_PER_SEC / HZ) / 2* nanosecondi dopo, pari cinque millisecondi visto che *HZ* è il normale valore 100 utilizzato da Linux per indicare che lo scheduler deve essere invocato con una frequenza di 100Hz; nell'ipotesi contraria il timer viene programmato in modo da invocare lo scheduler al tempo di rilascio del prossimo job del task indicato dalla funzione *find_preemptor*. Ultima operazione necessaria è evidentemente quella di memorizzare il fatto che il timer è stato programmato mediante l'istruzione *set_bit (RTL_SCHED_TIMER_OK, &sched->sched_flags).*

Designato il prossimo task che diventerà running, si controlla se esso è diverso da quello attualmente in esecuzione, ed in tal caso si provvede ad effettuare il context switch, gestendo i casi particolari relativi al fatto che il nuovo task può essere Linux o che, se è stato compilato il supporto per l'unità floating point, il task potrebbe richiederla e non esserne in possesso. Infine, se il thread che ha eseguito lo scheduler presenta almeno un segnale attivo ad eccezione di *RTL_SIGNAL_READY* viene invocata su di esso la funzione *do_signal*, che provvede a sistemare i flag del thread, attivando ad esempio il segnale *RTL_SIGNAL_READY* e rimuovendo il segnale *RTL_SIGNAL_TIMER* qualora quest'ultimo sia attivo. Come preannunciato viene ora riportato il codice di rtl_schedule. Si può notare come non vi sia alcun controllo sulle deadline, come si poteva già supporre dal fatto che non vi sono campi di *rtl_thread_struct* o funzioni nel file *rtl_sched.c* che facciano riferimento a tale concetto.

```
int rtl_schedule (void)
{
    schedule_t *sched;
    struct rtl_thread_struct *t;
    struct rtl_thread_struct *new_task;
    struct rtl_thread_struct *preemptor = 0;
    unsigned long interrupt_state;
    int cpu_id = rtl_getcpuid();
    hrtime_t now;
    rtl_sigset_t mask;

    rtl_no_interrupts(interrupt_state);
    rtl_trace2 (RTL_TRACE_SCHED_IN, (long) pthread_self());
    /* new_task = &sched->rtl_linux_task;*/

    new_task = 0;
    sched = &rtl_sched[cpu_id];

    now = sched->clock->gethrtime(sched->clock);

    if (sched->clock->mode == RTL_CLOCK_MODE_ONESHOT) {
            sched->clock->value = now;
    }

    for (t = sched->rtl_tasks; t; t = t->next) {
            /* expire timers */

            if (test_bit(RTL_THREAD_TIMERARMED, &t->threadflags)) {
                    if (now >= t->resume_time) {
                            clear_bit(RTL_THREAD_TIMERARMED, &t->
                               threadflags);
                            rtl_sigaddset (&t->pending, RTL_SIGNAL_TIMER);
                            if (t->period != 0) { /* periodic */
                                    t->resume_time += t->period;
                                    /* timer overrun */
#ifndef CONFIG_RTL_OLD_TIMER_BEHAVIOUR
```

```
                                        while (now >= t->resume_time) {
                                                t->resume_time += t->period;
/*                                              rtl_printf("overrun"); */
                                        }
#endif
                                } else {
                                        t->resume_time = HRTIME_INFINITY;
                                }
                        }
                }

                /* and find highest priority runnable task */
                if ((t->pending & ~t->blocked) && (!new_task ||
                        (t->sched_param.sched_priority > new_task->sched_param.
                           sched_priority))) {
                             new_task = t;
                }
        }

        if (sched->clock->mode == RTL_CLOCK_MODE_ONESHOT && !test_bit (
           RTL_SCHED_TIMER_OK, &sched->sched_flags)) {
                if ( (preemptor = find_preemptor(sched,new_task))) {
                        (sched->clock)->settimer(sched->clock, preemptor->
                           resume_time - now);
                } else {
                        (sched->clock)->settimer(sched->clock, (HRTICKS_PER_SEC
                            / HZ) / 2);
                }
                set_bit (RTL_SCHED_TIMER_OK, &sched->sched_flags);
        }

        if (new_task != sched->rtl_current) { /* switch out old, switch in new
           */
                if (new_task ==  &sched->rtl_linux_task) {
                        rtl_make_rt_system_idle();

                } else {
                        rtl_make_rt_system_active();
                }

                rtl_trace2 (RTL_TRACE_SCHED_CTX_SWITCH, (long) new_task);
                rtl_switch_to(&sched->rtl_current, new_task);
                /* delay switching the FPU context until it is really needed */
#ifdef CONFIG_RTL_FP_SUPPORT
                if (sched->rtl_current-> uses_fp &&\
                                sched->rtl_task_fpu_owner != sched->rtl_current
                                   )
                {
                        if (sched->rtl_task_fpu_owner)
                        {
                                rtl_fpu_save (sched,sched->rtl_task_fpu_owner);
                        }
                        rtl_fpu_restore (sched,sched->rtl_current);
                        sched->rtl_task_fpu_owner = sched->rtl_current;
                }
#endif /* CONFIG_RTL_FP_SUPPORT */
        }
/*  if (RTL_CURRENT==rtl_get_linux_thread(rtl_getcpuid())) {
                unsigned long flags;
                rtl_allow_interrupts();
```

```
            __save_flags(flags);
            __restore_flags(flags);
    }*/
    mask = pthread_self()->pending;

    if (pthread_self()->pending & ~(1 << RTL_SIGNAL_READY)) do_signal(
       pthread_self());

    rtl_trace2 (RTL_TRACE_SCHED_OUT, (long) pthread_self());
    rtl_restore_interrupts(interrupt_state);
    return mask;
}
```

2.4 Sincronizzazione e comunicazione

2.4.1 Segnali

Nella trattazione precedente si è già accennato al fatto che RTLinux implementa i segnali, una primitiva di comunicazione tra processi molto semplice e normalmente utilizzata nei sistemi UNIX. In un normale programma per Linux mediante la funzione *sigaction*[9] è possibile registrare per quasi tutti i segnali la funzione da utilizzare come handler, la quale viene eseguita in modo asincrono rispetto al resto del programma, in modo simile a quanto avviene quando un interrupt hardware provoca l'esecuzione di una ISR. In RTLinux ciò non avviene, infatti nel codice dello scheduler appena descritto non vi è alcun meccanismo per l'invocazione di routine di gestione dei segnali. I segnali di RTLinux scambiabili tra i thread sono definiti in *rtl_sched.h* e sono solamente i seguenti, il cui significato si può facilmente desumere direttamente dal nome:

```
#define RTL_SIGNAL_NULL 0 /* posix wants signal=0 to simply check */
#define RTL_SIGNAL_WAKEUP 1
#define RTL_SIGNAL_CANCEL 2
#define RTL_SIGNAL_SUSPEND 3
#define RTL_SIGNAL_TIMER 5
#define RTL_SIGNAL_READY 6
```

Per inviare il segnale *signal* al thread *thread* è possibile utilizzare la funzione

int pthread_kill(pthread_t thread, int signal)

fornita da *rtl_sched.c*. Qualora il thread a cui si invia il segnale sia Linux, è possibile inviare il segnale *x+RTL_LINUX_MIN_SIGNAL,* dove *x* è un valore intero e *RTL_LINUX_MIN_SIGNAL* è una costante di *rtl_sched.h*, ottenendo come effetto quello di invocare *rtl_global_pend_irq (x).* Anche in RTLinux in *schedulers/signal.c* viene definita una funzione chiamata *sigaction*

*int sigaction(int sig, const struct sigaction *act, struct sigaction *oact)*

di prototipo identico a quella di Linux. Tuttavia questa funzione si limita a fornire un'altra modalità di registrazione di handler per interrupt hardware, infatti il codice invoca *rtl_free_global_irq (irq)* per rimuovere eventuali handler precedenti e chiama *rtl_request_global_irq (irq, rtl_sig_interrupt),* dove

- *irq* è pari a *sig* se si attiva il flag *SA_IRQ* di *act->sa_flags* e pari a *sig-RTL_SIGIRQMIN* (costante definita *include/posix/signal.h*) altrimenti
- *rtl_sig_interrupt* è una funzione che esegue *act->sa_handler* se questo campo non è pari a *SIG_IGN* o *SIG_DFL*, il nome di due costanti normalmente utilizzate per indicare, rispettivamente, di ignorare il segnale o di utilizzare l'handler di default.

[9] I dettagli su questa funzione sono disponibili ad esempio in [38].

Ulteriori funzioni per i segnali sono fornite da un modulo di RTLinux opzionale e non compilato per default, chiamato PSC, acronimo di Portable Signal Code. Tale modulo, presentato come *Userspace Realtime* in fase di configurazione dell'installazione, permette sostanzialmente di registrare handler degli interrupt che vengono mantenuti real-time pur essendo eseguiti in spazio utente. Tuttavia vi sono forti limiti sullo sviluppo di tali applicazioni in spazio utente, in particolare

- devono essere lanciate da root,
- non possono invocare chiamate di sistema,
- deve essere effettuato un link statico di tutte le librerie utilizzate,
- visto che oltre ad avere uno spazio di indirizzamento protetto possono accedere allo spazio di indirizzamento del kernel nel caso di errori possono comportare danni alle strutture dati del sistema operativo alla stregua di quanto avviene per i moduli.

2.4.2 Spinlock

Un'altra primitiva molto semplice che viene fornita sono gli spinlock, implementati direttamente nel file di definizione *include/rtl_spinlock.h.* Per inizializzare e distruggere un lock è possibile utilizzare, rispettivamente

*int pthread_spin_init(pthread_spinlock_t *lock, int pshared)*

*int pthread_spin_destroy(pthread_spinlock_t *lock)*

anche se in realtà la chiamata a *pthread_spin_destroy* è priva di effetti dato che il codice è il seguente

```
static inline int pthread_spin_destroy(pthread_spinlock_t *lock)
{
        return 0;
}
```

Per effettuare un lock, per rilasciarlo e per tentare un lock ed acquisirlo se è libero o ritornare subito in caso contrario è necessario utilizzare, rispettivamente,

*int pthread_spin_lock(pthread_spinlock_t *lock)*,

*int pthread_spin_unlock(pthread_spinlock_t *lock)*

e

*int pthread_spin_trylock(pthread_spinlock_t *lock).*

Risulta opportuno utilizzare i lock il meno possibile, sia per l'inefficienza del busy waiting sia per il fatto che si disabilitano gli interrupt della CPU su cui viene eseguito (*spin_lock* è una delle funzioni implementate dall'usuale kernel di Linux):

```
static inline int pthread_spin_lock(pthread_spinlock_t *lock)
{
        rtl_irqstate_t flags;
        rtl_no_interrupts (flags);
        spin_lock (&lock->lock);
        lock->flags = flags;
        return 0;
}
```

2.4.3 Mutex

Ben più interessante è l'implementazione dei mutex, fornita dal file *schedulers/rtl_mutex.c*. In fase di inizializzazione di un mutex, mediante la chiamata a

*int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)*

si possono specificare due tipi di mutex, mediante il campo *type* di attr[10] (per essere precisi il campo *type* della struttura puntata da *attr*):

- *PTHREAD_MUTEX_SPINLOCK_NP* che rende il funzionamento del mutex identico a quello di uno spinlock,
- *PTHREAD_MUTEX_NORMAL* che rende le operazioni del mutex dipendenti dal campo *protocol* di *attr*, il quale a sua volta può valere *PTHREAD_PRIO_NONE* o *PTHREAD_PRIO_PROTECT.*

Nel caso il protocollo sia *PTHREAD_PRIO_NONE* la risorsa viene allocata al processo richiedente non appena è disponibile, mentre utilizzando *PTHREAD_PRIO_PROTECT*, se si è scelto di includere il supporto "*POSIX Priority Protection*" in fase di configurazione dell'installazione, l'allocazione del mutex avviene secondo modalità simili al Ceiling-priority protocol presentato a pagina 301 di [1]. Tuttavia vi sono due differenze significative che possono portare al deadlock o all'impossibilità di acquisire una risorsa (di questo fatto non sia accenna minimamente nella documentazione). Più in dettaglio la regola 1a di pagina 303 del libro di testo afferma che i job con la stessa priorità devono essere schedulati con modalità FIFO, ma come sottolineato nella sezione relativa all'analisi dello scheduler, in RTLinux se più job hanno la stessa priorità quello che viene mandato in esecuzione è quello che è stato creato per ultimo. Risulta facile costruire un caso che porti a deadlock: si immagini di creare un task T_1 di priorità π_1 avente come suo primo job J_1 rilasciato all'istante 0 e successivamente un task T_2 avente priorità[11] π_2 con $\pi_2 \geq \pi_1$. Si supponga che (solo) questi due task utilizzino due risorse, A e B; il priority ceiling delle risorse è lo stesso e vale $\Pi_A = \Pi_B = \pi_2$. Sia vero che il job J_1 acquisisce la risorsa A prima che venga rilasciato J_2. La priorità del job J_1 è pari a $\Pi_A = \pi_2$ al momento di rilascio di J_2, quindi se si seguisse la regola FIFO J_2 non verrebbe eseguito, ma nel caso di RTLinux J_2 effettua preemption su J_1 essendo stato creato dopo. Come si sarà già intuito, è ora possibile supporre che J_2 richieda la risorsa B e poi la risorsa A; in tal momento l'esecuzione di J_1 riprende, ed è possibile che quest'ultimo richieda la risorsa B generando una situazione di deadlock.

Una soluzione che si potrebbe adottare per risolvere questo problema sarebbe associare alle risorse un priority-ceiling maggiore di tutte le priorità dei job che la utilizzano, ad esempio si potrebbe impostare a valori dispari le priorità dei job e definire il priority-ceiling di una risorsa come massimo delle priorità del job che la utilizzano più uno.

Tuttavia permane il secondo dei problemi citati: un processo, per definizione di priority-ceiling di una risorsa, non può utilizzare risorse a priorità inferiore, come assicura il primo dei controlli di *__pthread_mutex_trylock,* una funzione invocata quando si effettua un lock su un mutex (si presti attenzione ai due underscore iniziali nel nome, che la distinguono dalla funzione *pthread_mutex_trylock* presentata successivamente)

```
static inline int __pthread_mutex_trylock(pthread_mutex_t *mutex)
{
#ifdef _RTL_POSIX_THREAD_PRIO_PROTECT
/* if _RTL_POSIX_THREAD_PRIO_PROTECT, this code is protected by a spinlock
   */
```

[10] Come fatto durante la discussione dei thread si evita qui di elencare le varie funzioni set e get, come *pthread_mutexattr_settype*, che permettono di impostare i valori in modo orientato agli oggetti e si rimanda alla documentazione.

[11] Si è preferito in questo caso non utilizzare la convenzione normalmente utilizzata in [1] dove task di indice minore hanno priorità maggiore per denotare con indice superiore un task creato dopo.

```
        if (mutex->protocol == PTHREAD_PRIO_PROTECT) {
                if (RTL_PRIO(RTL_CURRENT) > mutex->prioceiling) {
                        return EINVAL;
                }

                mutex->oldprio = RTL_PRIO (RTL_CURRENT);
                RTL_PRIO (RTL_CURRENT) = mutex->prioceiling;
        }
#endif
        if (test_and_set_bit(0, &mutex->busy)) {
                return EBUSY;
        }
        return 0;
}
```

Un job però può momentaneamente aumentare la propria priorità secondo il ceiling-priority protocol se fa utilizzo di risorse con priority-ceiling più elevato della sua priorità. Risulta quindi necessario distinguere tra la priorità assegnata ad un job e la sua priorità istantanea, e rifiutare l'acquisizione di una risorsa per violazione del protocollo solo quando il lock su una risorsa di priority-ceiling Π_R viene effettuato da parte di un job di priorità assegnata π' maggiore di Π_R, e non quando un job di priorità istantanea π chiede R. Se infatti si memorizza una sola priorità per i job, corrispondente alla priorità istantanea, e si effettua il controllo del rispetto del protocollo su di essa, diviene impossibile acquisire le risorse in ordine decrescente di priorità; questo è proprio ciò che avviene in RTLinux, come si desume facilmente dal codice appena riportato, poiché *RTL_PRIO* è una macro, definita in *rtl_sched.h* che cambia il campo *sched_param.sched_priority* del thread passato come argomento (come intuibile dal nome, RTL_CURRENT è una macro che restituisce il puntatore al thread che sta eseguendo il codice). Un esempio di questo comportamento è riportato nella sezione relativa alle applicazioni sviluppate.

Le API per la gestione dei mutex, che comprendono la funzione *pthread_mutex_init* prima citata, sono molto intuitive:

*int pthread_mutex_destroy(pthread_mutex_t *mutex)*

*int pthread_mutex_lock(pthread_mutex_t *mutex)*

*int pthread_mutex_unlock(pthread_mutex_t *mutex)*

*int pthread_mutex_trylock(pthread_mutex_t *mutex)*

*int pthread_mutex_timedlock(pthread_mutex_t *mutex, const struct timespec *abstime)*

Analogamente a quanto visto prima per *pthread_spin_trylock, pthread_mutex_trylock* acquisisce il mutex solo se questo è disponibile altrimenti ritorna subito la costante EBUSY (anziché zero, come avviene quando il mutex è stato acquisito); *pthread_mutex_timedlock* invece ritorna non appena si è acquisito il mutex o scade il timeout specificato da *abstime.* Risulta interessante sottolineare che quando si esegue unlock su un mutex non si invoca lo scheduler, ed un commento nel codice della funzione *pthread_mutex_unlock* spiega che questa è una scelta implementativa:

```
/* XXX we do not call the scheduler here by design;
  * for fast wakeups, use semaphores & pthread_wakeup_np */
```

2.4.4 Semafori

Il file *schedulers/rtl_sema.c*, sfruttando il codice di gestione delle code dei processi in attesa implementate per i mutex, fornisce le primitive per i semafori. In particolare viene implementato solo il semaforo numerico, e non quello binario (che comunque si può sostituire con un mutex); non è

inoltre specificabile il massimo valore assumibile, dato che nel codice della funzione signal, chiamata *sem_post* in RTLinux, si incrementa semplicemente il valore del semaforo senza verificare che si superi una determinata soglia. Nonostante il semaforo sia numerico non viene fornita nessuna funzione per incrementare il valore del semaforo di un valore arbitrario, ed è quindi necessario invocare *post* tante volte quanto è il numero di incrementi che si vogliono effettuare. Le API fornite sono, come si potrà già supporre, le seguenti, e del tutto analoghe a quelle presentate per i mutex

*int sem_init(sem_t *sem, int pshared, unsigned int value)*

*int sem_destroy(sem_t *sem)*

*int sem_wait(sem_t *sem)*

*int sem_post(sem_t *sem)*

*int sem_getvalue(sem_t *sem, int *sval)*

*int sem_trywait(sem_t *sem)*

*int sem_timedwait(sem_t *sem, const struct timespec *abstime)*

2.4.5 Variabili condizionali

In *rtl_mutex.c* vengono poi definite, sempre sfruttando il codice delle code di thread in attesa dei mutex, le variabili condizionali, utilizzabili in modo simile alle regioni critiche condizionali viste a Sistemi Operativi o ai monitor di Java. Una volta dichiarata una struttura di tipo *pthread_cond_t* è possibile invocare su di essa la funzione

*int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)*

dove il parametro *attr* è un parametro ignorato dal codice della versione di RTLinux in esame. Un thread a questo punto può mettersi in attesa della condizione *cond* invocando

*int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)*

Quando questa funzione ritorna si è certi che la condizione *cond* sia verificata e che il thread sia in possesso del mutex puntato da *mutex* (concetto analogo al poter eseguire un metodo *synchronized* in Java). Un altro thread può segnalare il verificarsi della condizione *cond* mediante

*int pthread_cond_broadcast(pthread_cond_t *cond)*

e come suggerisce il nome *broadcast*, tutti i thread in attesa vengono risvegliati. Similmente a quanto visto precedentemente viene fornita una funzione per specificare un tempo di attesa massimo (timeout)

*int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)*

ed una funzione per terminare l'utilizzo della variabile condizionale (il cui corpo è ancora un semplice *return 0*)

*int pthread_cond_destroy(pthread_cond_t *mutex)*

2.4.6 Code FIFO

Come è stato precedentemente accennato per permettere lo scambio di dati tra thread e tra thread e processi utente (di Linux) RTLinux mette a disposizione due modalità (se i rispettivi moduli vengono abilitati in fase di installazione), code (buffer) FIFO e memoria condivisa.

Le code FIFO, implementate in *fifos/rtl_fifo.c,* vengono fornite dal modulo *rtl_fifo.o* che a sua volta richiede che sia caricato il modulo *rtl_posixio.o*. Ciascuna fifo corrisponde ad un file in */dev* creato in file di installazione di nome *rtf#*, dove # è un numero intero. Per inciso questo consente, ad esempio, di leggere i dati inseriti da un processo (anche real-time) nella coda con un semplice *cat*. Per iniziare ad utilizzare una coda FIFO è necessario invocare

int rtf_create(unsigned int fifo, int size)

dove *fifo* indica quale coda utilizzare e *size* indica la dimensione (in byte) del buffer che costituisce la coda. Similmente a quanto visto per lo stack dei thread, visto che non è consentito allocare dinamicamente spazio ai processi real-time, questa funzione non può essere impiegata all'interno di thread real-time ma può essere utilizzata da *init_module*. Si fa notare che la documentazione consiglia di invocare la funzione di deallocazione della coda *fifo*

int rtf_destroy(unsigned int fifo)

prima di chiamare la funzione *rtf_create* sulla coda *fifo* in modo da far perdere il controllo della coda ad altri processi che la stiano utilizzando.

Per inserire dati in una coda i processi real-time possono utilizzare

*int rtf_put(unsigned int fifo, void * buf, int count)*

dove *fifo* è la coda da utilizzare, *buf* è un puntatore al primo dei byte da inviare e *count* ne indica il numero. Il valore ritornato è pari a *count* nel caso di successo ed è negativo nel caso di errore, ad esempio viene restituito *-ENODEV* se *fifo* è superiore al numero massimo di FIFO impostato in fase di installazione o *-ENOSPC* se non vi è sufficiente spazio nel buffer per inserire *count* dati. Quest'ultimo errore può ovviamente essere ritornato quando *count* è maggiore di *size* specificato in fase di creazione, ma normalmente il motivo per cui si ottiene è che il processo lettore è eseguito per troppo poco tempo rispetto al flusso dei dati prodotti dallo scrittore.

Per leggere dati da una coda un processo real-time può utilizzare

*int rtf_get(unsigned int fifo, void *buf, int count)*

di sintassi analoga alla funzione *rtf_put*. Non è necessario però effettuare un polling dei dati, visto che viene fornita la possibilità di registrare una funzione *handler* invocata all'arrivo di nuovi dati mediante

*int rtf_create_handler(unsigned int fifo, int (*handler) (unsigned int fifo))*

La funzione

int rtf_flush(unsigned int fifo)

libera il buffer da tutti i dati eventualmente presenti, che vanno persi (osservando il codice si nota che viene eseguito *RTF_LEN(minor) = 0* dove *RTF_LEN(minor)* corrisponde alla dimensione dei

dati presenti nel buffer non ancora letti dal processo lettore)

Vengono poi fornite le funzioni

int rtf_isempty(unsigned int fifo)

e

int rtf_isused(unsigned int fifo)

utilizzabili, rispettivamente, per individuare se la coda *fifo* contiene dati o se è gia in uso da parte di un altro processo. I processi utente possono gestire i devices */dev/rtf#* utilizzando le funzioni

*int open(const char *pathname, int flags)*

*ssize_t write(int fd, const void *buf, size_t count)*

*ssize_t read(int fd, void *buf, size_t count)*

dove *fd* è un file descriptor, un identificativo di ogni file aperto ritornato da open ed utilizzabile da tutte le funzioni che devono operare su tale file. Anche in questo caso per la lettura non è necessario utilizzare un busy waiting ma si può invocare

*int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout)*

una funzione che può essere utilizzata per attendere dati da uno o più file. Per una trattazione più dettagliata su queste funzioni di uso generale si rimanda a [41].

Le code FIFO sono unidirezionali, infatti se si scrive un'applicazione real-time che predispone una funzione come handler per l'arrivo di dati e nello stesso tempo ne invia si ottiene come risultato che l'handler viene invocato solo quando un altro processo scrive dei dati nella FIFO, ma i dati ottenuti corrispondono a tutti i dati inviati da entrambi i processi; per comodità le API includono quindi la funzione

int rtf_make_user_pair (unsigned int fifo, unsigned int fifo_put)

che crea due code FIFO.

2.4.7 Memoria condivisa

La condivisione di memoria viene fornita dal modulo opzionale *mbuff.o*; come per le code FIFO, i dati possono essere acceduti tramite un device, in questo caso */dev/mbuff.* Tuttavia è più semplice impiegare le API fornite, utilizzabili sia dai processi utente che dai processi real-time. Mediante la funzione

*void * mbuff_alloc(const char *name, int size)*

si alloca un'area di memoria di dimensione *size* con il nome *name* (un'usuale stringa null terminated), e viene ritornato un puntatore a tale area. Se esiste già un'area col nome specificato e *size* non è superiore alla dimensione dell'area di tal nome precedentemente allocata (nel qual caso viene ritornato un errore) viene ritornato un puntatore a tale area, che diviene condivisa tra tutti i processi che hanno invocato la funzione *mbuff_alloc* utilizzando lo stesso nome. Ogni qualvolta un processo esegue *mbuff_alloc* viene incrementato un contatore (osservando il codice si nota che tutto è realizzato

sfruttando la funzione *mmap* fornita dal kernel di linux, per cui si rimanda a [41]). Tale contatore viene decrementato alla chiamata di

*void mbuff_free(const char *name, void * mbuf)*

utilizzata per indicare che non si fa più uso dell'area di memoria, cosicché è possibile tener traccia di quanti processi utilizzano l'area di memoria condivisa e liberarla effettivamente solo quando nessuno vi accede. E' anche possibile, utilizzando

*void * mbuff_attach(const char *name, int size)*

accedere ad un'area di memoria già creata senza incrementare il contatore in modo che non sia necessario eseguire *mbuff_free* per liberare l'area di memoria. Se si desidera accelerare la liberazione della memoria (ovviamente solo nel caso in cui tutti i processi che hanno eseguito *mbuff_alloc* abbiano già invocato *mbuff_free*) è possibile chiamare

*void mbuff_detach(const char *name, void * mbuf)*

2.4.8 Seriale

RTLinux prevede infine un modulo opzionale per la gestione della seriale, *rt_com.o*, che fornisce le primitive per lo scambio di dati

*void rt_com_write(unsigned int com, char *ptr, int cnt)*

e

*int rt_com_read(unsigned int com, char *ptr, int cnt)*

dove, similmente a quanto visto prima, *rt_com_read* ritorna il numero di byte letti. Prima di poter utilizzare tali funzioni, tuttavia, è necessario modificare il sorgente del modulo *drivers/rt_com/rt_com.c* e ricompilarlo per adattarlo al proprio hardware. Infatti l'inizializzazione delle varie porte seriali presenti sul sistema avviene in *init_module* mediante la lettura di un array chiamato *rt_com_table* avente *RT_COM_CNT* elementi:

```
#define RT_COM_CNT 1
struct rt_com_struct rt_com_table[ RT_COM_CNT ] =
{
        { 0, BASE_BAUD, 0x3f8, 4, STD_COM_FLAG, rt_com0_isr },
};
```

dove *rt_com_struct* è una struttura definita in *drivers/rt_com/rt_comP.h* utilizzata per contenere i vari parametri di ciascuna porta seriale, tra cui l'IRQ ad essa associata e l'indirizzo dell'ISR che si vuole invocare.

Infine chiamando

void rt_com_setup(unsigned int com, int baud, unsigned int parity, unsigned int stopbits, unsigned int wordlength)

è possibile impostare i parametri della trasmissione dati.

Capitolo 3

Installazione

Il primo passo per l'utilizzo di RTLinux consiste nell'installazione di una distribuzione di Linux, che può essere usata anche per la compilazione di RTLinux stesso. Tutti i programmi installati potranno poi essere utilizzati anche quando è in esecuzione RTLinux dato che la virtualizzazione della gestione degli interrupt operata da RTLinux non impedisce, come si è già detto, il normale funzionamento del kernel di Linux.

La maggior parte delle guide per l'installazione di RTLinux prevede l'installazione su una Red Hat, per questo si è scelto di provare una distribuzione diversa e recente, in particolare si è deciso di installare RTLinux su Debian Etch (attuale release in testing), snapshot del 22 gennaio 2007. Conviene probabilmente aprire qui una parentesi. In ogni momento sono mantenute tre versioni di Debian denominate "stable", "testing" ed "unstable". Il software dapprima viene inserito in unstable e quando è passato un determinato tempo e si è verificata l'assenza di grossi bug viene portato in testing (per le altre condizioni che devono essere soddisfatte si veda [42]). Quando poi si decide di rilasciare una release stabile si entra in uno stato denominato "freeze" (un periodo di circa sei mesi), in cui i pacchetti di unstable non vengono più portati in testing e si risolvono tutti i bug segnalati per i pacchetti della versione testing. Dopo questo periodo tutti i pacchetti testing vengono portati nella release stable, e non vengono più aggiornati se non in caso di bug. Dal sito http://cdimage.debian.org/cdimage/weekly-builds/i386/iso-dvd/ è possibile scaricare quello che viene definito uno "snapshot", ossia le immagini dei CD/DVD contenenti tutti i pacchetti presenti nella distribuzione testing in un determinato momento (lo snapshot viene in realtà aggiornato settimanalmente, come indica l'indirizzo).

Una volta installata una distribuzione di Linux bisogna compilare un kernel di Linux a cui sia già stata applicata la patch per l'esecuzione di RTLinux, o, alternativamente, scaricare un kernel originale da *www.kernel.org* per cui sia disponibile una tale patch ed applicarla. Sul sito di RTLinux Free [16] sono[1] disponibili tre versioni di RTLinux Free, versione 3.1, versione 3.1 per kernel 2.6 e versione 3.2-rc1, e tre versioni del kernel di Linux a cui è gia stata applicata la corrispondente patch per l'utilizzo con RTLinux, 2.4.20, 2.4.29 e 2.6.9; si sono utilizzati perciò i sorgenti del kernel di Linux con patch già applicata disponibili su tale sito ed il kernel originale (versione 2.4.29) di cui si parla a pagina 6 è stato scaricato a *www.kernel.org* solo per poter individuare più facilmente le modifiche che sono state apportate per permettere l'esecuzione di RTLinux. Tutte le versioni del kernel e di RTLinux Free sono state provate sulla distribuzione indicata, in modo da determinare quale presentasse maggior facilità di installazione.

Dopo aver estratto il kernel prepatched in una directory a scelta ed aver creato un link simbolico (con il comando *ln -s*) da */usr/src/linux* alla directory radice dei file estratti (ciò non è strettamente necessario, ma il Makefile di alcuni programmi, in genere moduli del kernel, cerca gli header del kernel in */usr/scr/linux/include*) si devono scegliere le parti del kernel da compilare come statiche, quelle da compilare come modulo e di quelle da non compilare. Per fare questo esistono tre interfacce, una testuale, una che fa uso di menu su interfaccia a caratteri ed una grafica (i kernel recenti, tra cui 2.6.9 dispongono anche di una quarta interfaccia, realizzata utilizzando le librerie *GTK*). Si è

[1] O meglio, *erano* disponibili, visto che la terza settimana di Febbraio il sito è stato chiuso.

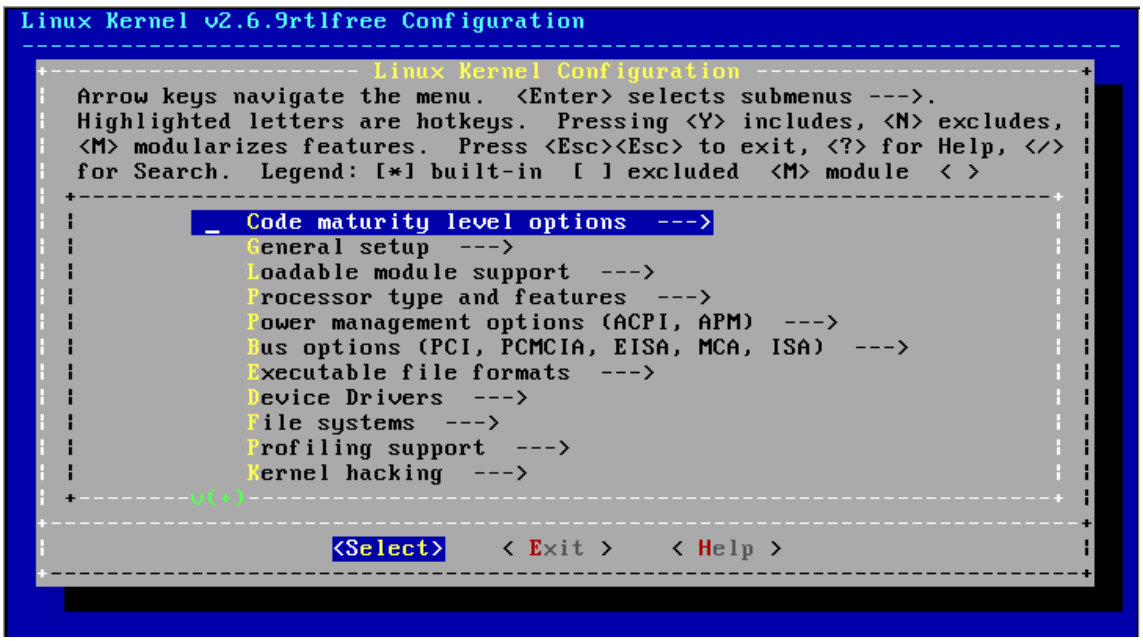

Figura 3.1: Interfaccia Ncurses per la configurazione del kernel

optato per la seconda scelta, che presuppone siano installate le librerie development di *Ncurses*, una libreria per la realizzazione di interfacce testuali di cui si possono trovare informazioni in [43] o in [44]. Nel caso dello snapshot utilizzato mediante il comando

apt-get install ncurses-dev

si sono installare le librerie *Ncurses* versione 5.5-5.

Per avviare il menu a caratteri appena descritto basta avviare make con target *menuconfig* ossia (per l'interfaccia testuale di deve usare *make config* e per quella grafica *make xconfig*)

make menuconfig

ed appare una schermata come quella riportata in figura 3.1.

In questa fase, visto che si è scelto di non utilizzare initrd[2], bisogna includere come statico il supporto per il dispositivo che contiene il root file system (ad esempio disco IDE o SATA), il supporto per il file system di root (che nel test effettuato era *Ext3*), e, nel caso si selezioni *Advanced partition selection* è necessario selezionare anche il tipo di tabella delle partizioni, in genere *PC BIOS (MSDOS partition tables) support*. Qualora si trascuri uno di questi aspetti all'avvio del sistema si ottiene kernel panic. Inoltre è bene disabilitare i supporti al power management ed al frequency scaling, che sono fonti di non determinismo (se la frequenza della CPU viene rallentata per risparmiare energia i tempi di esecuzione ovviamente si allungano).

Il passo successivo consiste nel fornire il comando

make bzImage

che compila l'immagine compressa del kernel (un file di nome *bzImage*). A questo punto mediante

[2] *initrd*, il cui nome è l'abbreviazione di "initial ramdisk", è un file che viene montato da Linux nella RAM come RAM disk e contiene ciò che è necessario al funzionamento del dispositivo in cui è presente il file system radice. Una volta che sono stati caricati i moduli per l'accesso al root file system tale file system viene montato e si esegue la normale partenza del sistema operativo (per kernel successivi al 2.9.16, è preferibile sostituire *inird* con alternative più recenti quali *initramfs-tools* o *Yaird*).

make modules

è possibile avviare la compilazione dei moduli del kernel.

Una volta terminato il processo di compilazione è necessario copiare il file *linux/arch/i386/boot/bzImage* in una locazione opportuna. Solitamente si utilizza */boot*, ed infatti si è dato il comando

cp linux/arch/i386/boot/bzImage /boot/*bzImage-rtl.*

Risulta poi necessario aggiornare il bootloader (ad esempio LILO o GRUB), e nello specifico siccome si è utilizzato GRUB si sono aggiunte le seguenti righe in */boot/grub/menu.lst:*

```
title           RTLinux
root            (hd0,2)
kernel          /boot/bzImage-rtl root=/dev/hda3 ro
savedefault
```

mediante le quali si indica di aggiungere una nuova voce "RTLinux" nel menu di scelta presentato all'avvio da GRUB e di utilizzare l'immagine /boot/bzImage-rtl che si trova sul root file system presente nella terza partizione del primo disco IDE (si possono notare le due rappresentazioni di tale partizione, *(hd0,2)* e */dev/hda*3, si veda [45] per maggiori dettagli).

Conviene poi copiare anche il file dei simboli del kernel *linux/System.map* in */boot/System.map*; una descrizione abbastanza approfondita dell'uso di tale file, non strettamente necessario, è reperibile in [46].

Una volta installati i moduli mediante il comando

make modules_install

è possibile notare la presenza della directory */lib/modules/versioneKernel* (dove *versioneKernel* è, ad esempio, *2.4.29-rtl-3.1*) contenente nelle sue varie sottodirectory i file oggetto costituenti i moduli compilati.

Dopo aver riavviato il sistema per verificare il funzionamento del nuovo kernel è necessario compilare RTLinux, e visto che quest'ultimo è sostanzialmente un insieme di moduli del kernel, è bene utilizzare lo stesso compilatore utilizzato per la compilazione del kernel di Linux, come indicato in [47], [48] o [49]. Una volta estratti i sorgenti di RTLinux bisogna creare nella directory radice dei sorgenti un link simbolico chiamato *linux* che punta alla directory dei sorgenti utilizzati per la compilazione del kernel di Linux, in modo da poter includere gli header di tale kernel durante la compilazione. A questo punto per la configurazione delle opzioni di RTLinux è possibile scegliere, come per il kernel, una delle tre interfacce di cui si è accennato precedentemente ed invocare rispettivamente *make config*, *make menuconfig* o *make xconfig* (la prima volta che si esegue la configurazione di RTLinux vengono presentate le condizioni imposte dalla licenza Open RTLinux che si devono accettare per poter procedere).

Per la compilazione si sono usate tutte le configurazioni di default tranne per il fatto che

- si è attivato il supporto *POSIX Priority Protection*
- si è disabilitato il supporto per la versione 1 delle API di RTLinux perché la documentazione sconsiglia di utilizzarle per nuovi progetti

Per compilare basta dare il comando

make

Infine lanciando

make install

viene creato un link simbolico */usr/rtlinux* che punta ad una directory di nome *rtlinux-3.x* (dove *x* è 1 o 2 a seconda della versione) creata sempre in */usr* e contenente i file necessari alla compilazione di programmi real-time, oltre che alcuni esempi. Lanciando *make install*, inoltre, si esegue automaticamente anche il target *devices_install* (che alcune guide lanciano manualmente) il quale si occupa di creare i file in */dev* relativi alle code FIFO ed alla memoria condivisa.

Per compilare gli esempi forniti, tra cui il più semplice è un classico *hello world* basta entrare in */usr/rtlinux/examples/nome-esempio* (dove *nome-esempio* è, nel caso appena citato, *hello*) e dare

make

Per provare poi ad eseguire ciascun esempio è sufficiente lanciare

make test

nella corrispondente directory. Questo target provvede anche a lanciare gli script necessari a caricare i moduli di RTLinux richiesti per l'esecuzione (tra cui vi è almeno *rtl-core*). Per fare in modo che tali script vengano trovati è però necessario copiare la directory *scripts* presente nella directory radice dei sorgenti di RTLinux in */usr/rtlinux*, altrimenti all'esecuzione di *make test* si ottiene il seguente errore

```
/bin/sh: scripts/rmrtl: No such file or directory
make: *** [test] Error 127
```

3.1 Incompatibilità tra versioni

Si è notato che varie combinazioni di versioni del kernel di Linux a cui è già applicata la patch per RTLinux, di *gcc*, dei *binutils* e di *make* fornivano successo o meno in modo molto strano, e si è perciò deciso di studiare il problema in maniera sistematica. Nello snapshot di Debian utilizzato il compilatore *gcc* di default è la versione 4.1.2, tuttavia è possibile installare contemporaneamente più versioni di *gcc*, ed in particolare le versioni fornite sono 2.95.4, 3.3.6, 3.4.6 e 4.1.2. La soluzione più veloce per passare da una versione dall'altra è modificare il link simbolico */usr/bin/gcc* facendolo puntare alla versione voluta (ad esempio a */usr/bin/gcc-2.95* o a */usr/bin/gcc-3.3*).

Il kernel 2.4.20 non risulta compilabile con le versioni 4.1.2 e 3.4.6 infatti dopo alcuni warning compare l'errore

```
In file included from /rtlinux/linux-2.4.20-rtl/include/linux/sched.h:23,
                 from /rtlinux/linux-2.4.20-rtl/include/linux/mm.h:4,
                 from /rtlinux/linux-2.4.20-rtl/include/linux/slab.h:14,
                 from /rtlinux/linux-2.4.20-rtl/include/linux/proc_fs.h:5,
                 from init/main.c:15:
/rtlinux/linux-2.4.20-rtl/include/linux/smp.h:29: error: conflicting types
for 'smp_send_reschedule'
/rtlinux/linux-2.4.20-rtl/include/asm/smp.h:43: error: previous
declaration of 'smp_send_reschedule' was here
In file included from /rtlinux/linux-2.4.20-rtl/include/linux/unistd.h:9,
                 from init/main.c:17:
/rtlinux/linux-2.4.20-rtl/include/asm/unistd.h:375: warning: conflicting
types for built-in function '_exit'
make: *** [init/main.o] Error 1
```

e nemmeno la versione 3.3.6 è in grado di compilare tale kernel a causa del seguente errore

```
In file included from ide-cd.c:318:
ide-cd.h:440: error: long, short, signed or unsigned used invalidly for
‘slot_tablelen’
make[3]: *** [ide-cd.o] Error 1
make[2]: *** [first_rule] Error 2
make[1]: *** [_subdir_ide] Error 2
make: *** [_dir_drivers] Error 2
```

Tuttavia è possibile compilarlo con il compilatore 2.95.4, anche se la compilazione viene interrotta a causa dell'assembler dei *binutils 2.17* (unica versione disponibile sulla distribuzione adottata), che fornisce il seguente errore

```
{standard input}: Assembler messages:
{standard input}:943: Error: suffix or operands invalid for ‘mov’
{standard input}:944: Error: suffix or operands invalid for ‘mov’
{standard input}:1037: Error: suffix or operands invalid for ‘mov’
{standard input}:1038: Error: suffix or operands invalid for ‘mov’
{standard input}:1096: Error: suffix or operands invalid for ‘mov’
{standard input}:1097: Error: suffix or operands invalid for ‘mov’
{standard input}:1099: Error: suffix or operands invalid for ‘mov’
{standard input}:1111: Error: suffix or operands invalid for ‘mov’
make[1]: *** [process.o] Error 1
make: *** [_dir_arch/i386/kernel] Error 2
```

Per risolvere tale problema

- si è scaricato il pacchetto *binutils* versione 2.15 presente in Debian stabile da `http://packages.debian.org/stable/devel/binutils`,
- si è estratto il contenuto del pacchetto con il seguente comando

  dpkg-deb -x binutils_2.15-6_i386.deb binutils
- si sono copiati tutti i file creati nella directory *binutils/usr/bin* in */usr/local/bin*: tale directory non è utilizzata da Debian per l'installazione di programmi, tuttavia è presente nel path di default, prima di */usr/bin* quindi evitando di sovrascrivere i componenti di *binutils 2.17* si sono potuti utilizzare i *binutils* 2.15 senza specificarne ogni volta il percorso; qualora si ritenga che questa soluzione sia poco elegante è possibile rinominare i vari programmi dei *binutils* 2.17 e dei *binuils* 2.15 facendo seguire la versione al nome del programma e creare dei link simbolici, come fatto per *gcc*.
- si sono copiati i file di *binutils/usr/lib* in */usr/local/lib* e si è eseguito

  ldconfig /usr/local/lib

  per permettere al sistema di caricare le librerie necessarie ai *binutils* 2.15

Anche il kernel 2.4.29 non è compilabile con *gcc* 4.1.2 dato che viene riportato il seguente errore

```
In file included from /rtlinux/linux-2.4.29-rtl/include/linux/prefetch.h
   :13,
                 from /rtlinux/linux-2.4.29-rtl/include/linux/list.h:6,
                 from /rtlinux/linux-2.4.29-rtl/include/linux/wait.h:14,
                 from /rtlinux/linux-2.4.29-rtl/include/linux/fs.h:12,
                 from /rtlinux/linux-2.4.29-rtl/include/linux/capability.h
                    :17,
                 from /rtlinux/linux-2.4.29-rtl/include/linux/binfmts.h:5,
                 from /rtlinux/linux-2.4.29-rtl/include/linux/sched.h:9,
                 from /rtlinux/linux-2.4.29-rtl/include/linux/mm.h:4,
                 from /rtlinux/linux-2.4.29-rtl/include/linux/slab.h:14,
                 from /rtlinux/linux-2.4.29-rtl/include/linux/proc_fs.h:5,
                 from init/main.c:15:
/rtlinux/linux-2.4.29-rtl/include/asm/processor.h:75: error: array type has
incomplete element type
```

ma non presenta invece problemi con *gcc* 3.4.6 e *binutils* 2.17. La situazione è ancora diversa per *gcc* 3.3.6 e 2.95.4 che permettono di compilare ma solo se si utilizza *binutils* 2.15

Il kernel 2.6.9 a differenza dei precedenti è risultato compilabile da *gcc* 4.1.2, ma solo con l'utilizzo di *binutils* 2.15

I risultati ottenuti sono riassunti nella seguente tabella, che riporta per ciascuna combinazione versione kernel/versione *gcc* quale sia la versione di *binutils* che permette la compilazione ("-" indica che non è stato possibile compilare con nessuna delle due versioni dell'assembler prese in esame).

Versione kernel	Versione *gcc*	Versione *binutils*
2.4.20	2.95.4	2.15
2.4.20	3.3.6	-
2.4.20	3.4.6	-
2.4.20	4.1.2.	-
2.4.29	2.95.4	2.15
2.4.29	3.3.6	2.15
2.4.29	3.4.6	2.15 e 2.17
2.4.29	4.1.2.	-
2.6.9	2.95.4	2.15
2.6.9	3.3.6	2.15
2.6.9	3.4.6	2.15
2.6.9	4.1.2	2.15

Si possono notare due cose

- contrariamente a quanto ci si potrebbe aspettare i compilatori non offrono "compatibilità verso il basso" ossia un kernel compilabile con una versione di *gcc* può non esserlo con una più recente, ma al contrario, almeno nei casi esaminati, se una versione del kernel è compatibile con una versione di *gcc* lo è anche con versioni più datate;
- il comportamento dei *binutils* (in particolare dell'assembler) è molto strano, perché la versione 2.17 può essere utilizzata solo con kernel 2.29 (e non con un kernel più recente o con uno più vecchio) e con *gcc* 3.4.6 (anche in questo caso, né con versioni precedenti né con versioni successive);

Per quanto riguarda la compilazione di RTLinux, questa ha avuto successo per tutte le versioni di RTLinux solo con i compilatori *gcc* 2.95 e 3.3.6 e si è riscontrata la compatibilità di *binutils* 2.17 con kernel 2.4.29 e *gcc* 3.4.6 (ovviamente solo per RTLinux per kernel 2.4). Si è presentata poi un'ulteriore incompatibilità dovuta a *make*, infatti con tutte le versioni di RTLinux si ottiene il seguente errore

```
/bin/sh: -c: line 3: syntax error near unexpected token ‘fi’
/bin/sh: -c: line 3: ‘fi’
make: *** [.depend] Error 2
```

Esistono due soluzioni a questo problema; la prima consiste nel modificare i makefile, ed in particolare è necessario inserire uno spazio prima delle "\" che indicano la continuazione della riga qualora questo manchi, ad esempio le righe

```
@if [ ! -f $(RTLINUX)/.config -o ! -f $(RTLINUX)/include/linux/version.h ];
    then\
        echo You must do a make config and make dep in $(RTLINUX); \
```

devono essere modificate in

```
@if [ ! -f $(RTLINUX)/.config -o ! -f $(RTLINUX)/include/linux/version.h ];
    then \
        echo You must do a make config and make dep in $(RTLINUX); \
```

La seconda soluzione consiste nel non utilizzare il *make* disponibile nello snapshot (versione 3.81) ma impiegare quello della versione stabile di Debian (versione 3.80), che può essere installato in modo analogo a quanto descritto per i *binutils* 2.15.

Si è poi riscontrato che nelle sole versioni di RTLinux 3.1 le directory dei sorgenti, una volta estratte, sono read-only e per compilare è quindi necessario impostare il permesso di scrittura, inoltre nel caso della versione per kernel 2.6 nella directory *scripts* i link simbolici *rtlinux rtl-config* puntano, rispettivamente, ai file *scripts/rtlinux-2.6* e *scripts/rtl-config-2.6*, con percorso relativo alla directory *scripts*, ossia puntano ai file inesistenti *scripts/scripts/rtlinux-2.6* e *scripts/scripts/rtl-config-2.6*. Il problema è stato corretto manualmente, perché altrimenti durante la fase di *make install* si ottiene un errore di file non trovato che blocca l'installazione.

La versione di RTLinux 3.2-rc1 presente sul sito di RTLinux Free non è compatibile con il kernel 2.6, ed infatti nella directory *patches* sono presenti patch per le versioni del kernel 2.4.26, 2.4.27 2.4.28 e 2.4.29; durante la compilazione di RTLinux 3.1 per kernel 2.6 si sono ottenuti molti warning del tipo

```
*** Warning: "rtl_unregister_rtldev" [/rtlinux/rtlinux-3.1-2.6/fifos/
   rtl_fifo.ko] undefined!
*** Warning: "rtl_register_rtldev" [/rtlinux/rtlinux-3.1-2.6/fifos/rtl_fifo
   .ko] undefined!
*** Warning: "rtl_free_soft_irq" [/rtlinux/rtlinux-3.1-2.6/fifos/rtl_fifo.
   ko] undefined!
*** Warning: "rtl_get_soft_irq" [/rtlinux/rtlinux-3.1-2.6/fifos/rtl_fifo.ko
   ] undefined!
```

ed anche la compilazione dei programmi di esempio ha riportato vari warning. Inoltre una volta caricato RTLinux in momenti apparentemente arbitrari il sistema si è piantato più volte. Anche la versione 3.2 di RTLinux si è dimostrata instabile, comportando sistematicamente un blocco del sistema alla rimozione dei moduli costituenti i programmi di esempio, un fatto riportato da altri utenti, ad esempio in [51]. Si è perciò deciso di utilizzare la versione 3.1 di RTLinux con il kernel 2.4.29, il più recente dei due kernel che non hanno dimostrato alcun problema.

Capitolo 4

Applicazioni di prova

Come già affermato, in RTLinux i programmi real-time non sono altro che moduli del kernel che utilizzano le API di RTLinux. Per facilitare lo sviluppo di programmi durante l'installazione di RTLinux viene creato un Makefile, */usr/rtlinux/rtl.mk,* che contiene i percorsi corretti per tutti gli include necessari alla compilazione di programmi per RTLinux. Nel caso di programmi semplici che prevedono che un modulo sia generato a partire da un solo file *nomefile.c* è sufficiente invocare

make -f rtl.mk nomefile.o

Nel caso sia necessario effettuare altre operazioni di linking, invece, è sufficiente porre nel Makefile relativo al programma la riga

include rtl.mk

4.1 Gestione di Interrupt

La prima applicazione sviluppata consente di verificare il funzionamento delle API di gestione degli interrupt. In particolare si è deciso di intercettare l'interrupt della tastiera e stampare la stringa "Interrupt tastiera" ogni qualvolta si ottiene un'interruzione. Nel caso di funzionamento normale del sistema, visti i tempi molto lunghi dell'utente umano rispetto ai tempi durante i quali gli interrupt sono disabilitati, ciascun interrupt corrisponde alla pressione o al rilascio di un tasto. Il semplice codice necessario a realizzar questo è riportato nel seguito

```
#include <linux/module.h>
#include <rtl_core.h>

MODULE_LICENSE("GPL");

unsigned int handler(unsigned int irq, struct pt_regs *regs){
  printk("Interrupt␣tastiera\n");
  rtl_global_pend_irq(1);
  return 0;
}

int init_module(void) {
  rtl_request_irq(1,&handler);
  return 0;
}

void cleanup_module(void) {
  rtl_free_irq(1);
```

```
}
```

Nonostante la brevità del programma, è possibile fare varie osservazioni. Innanzitutto si può notare l'utilizzo della macro *MODULE_LICENSE*, necessaria ad indicare il tipo di licenza del modulo. Se non viene specificato che il modulo è open source al momento dell'inserimento si ottiene il seguente messaggio

```
Warning: loading nomemodulo.o will taint the kernel: no license
See http://www.tux.org/lkml/#export-tainted for information about tainted
    modules
```

Molto utile è la funzione *printk*, presente nella funzione *handler*, una routine di sintassi e utilizzo analogo a *printf* che può essere impiegata dai moduli del kernel. L'output viene diretto sul normale output del kernel, che viene sempre stampato a video se ci si trova in una console testuale, almeno per quanto riguarda Debian. Se lo si vuole leggere dall'ambiente grafico è sufficiente invocare in un terminale il comando *dmesg* che fornisce il contenuto del buffer dei messaggi del kernel (ossia vengono restituiti gli ultimi messaggi, nel caso di default gli ultimi 16392 byte).

Si può poi vedere che nel codice riportato la funzione registrata come ISR (*handler*) segnala l'interrupt a Linux, mediante l'istruzione *rtl_global_pend_irq(1)*; se si omette tale codice le routine di gestione della tastiera di Linux non vengono invocate e, ad esempio, alla pressione dei tasti in un terminale non avviene nulla; inoltre poiché la routine non riattiva l'interrupt (hardware) il messaggio "Interrupt tastiera" viene stampato una sola volta. Se nel codice della funzione *handler* si aggiungessero le istruzioni

status = inb(0x64);scancode = inb(0x60);

sarebbe possibile individuare il tasto premuto, tuttavia, visto che tali istruzioni leggono dal buffer della tastiera cui accede anche Linux, se non si introduce qualche modifica al kernel di Linux quest'ultimo non può leggere quali siano stati i tasti premuti.

4.2 Simulazione di gruppi di task

La seconda applicazione scritta è costituita da più moduli e applicativi in user space destinati a permettere la registrazione di eventi di interesse e la creazione ed il successivo avvio di gruppi di task real-time.

4.2.1 Registrazione di eventi

Per quanto riguarda la prima di queste operazioni, RTLinux fornisce già un modulo opzionale disabilitato per default ed indicato come *Experimental* chiamato *rtl_tracer.o*. Tuttavia, anziché utilizzare tale modulo, si è deciso di implementare un programma di registrazione di eventi che svolgesse tutte e sole le operazioni necessarie, in modo da rendere semplici gli altri moduli che devono comunicare l'esecuzione di particolari istruzioni, spesso, ma non necessariamente, assieme al momento in cui queste vengono eseguite. Lo sviluppo di quest'applicazione è stato guidato anche dalla scelta di utilizzare *kiwi,* un programma open source reperibile in [52] volto alla realizzazione di grafici rappresentanti l'andamento nel tempo di vari task (l'interfaccia grafica di questo programma viene mostrata in figura 4.1).

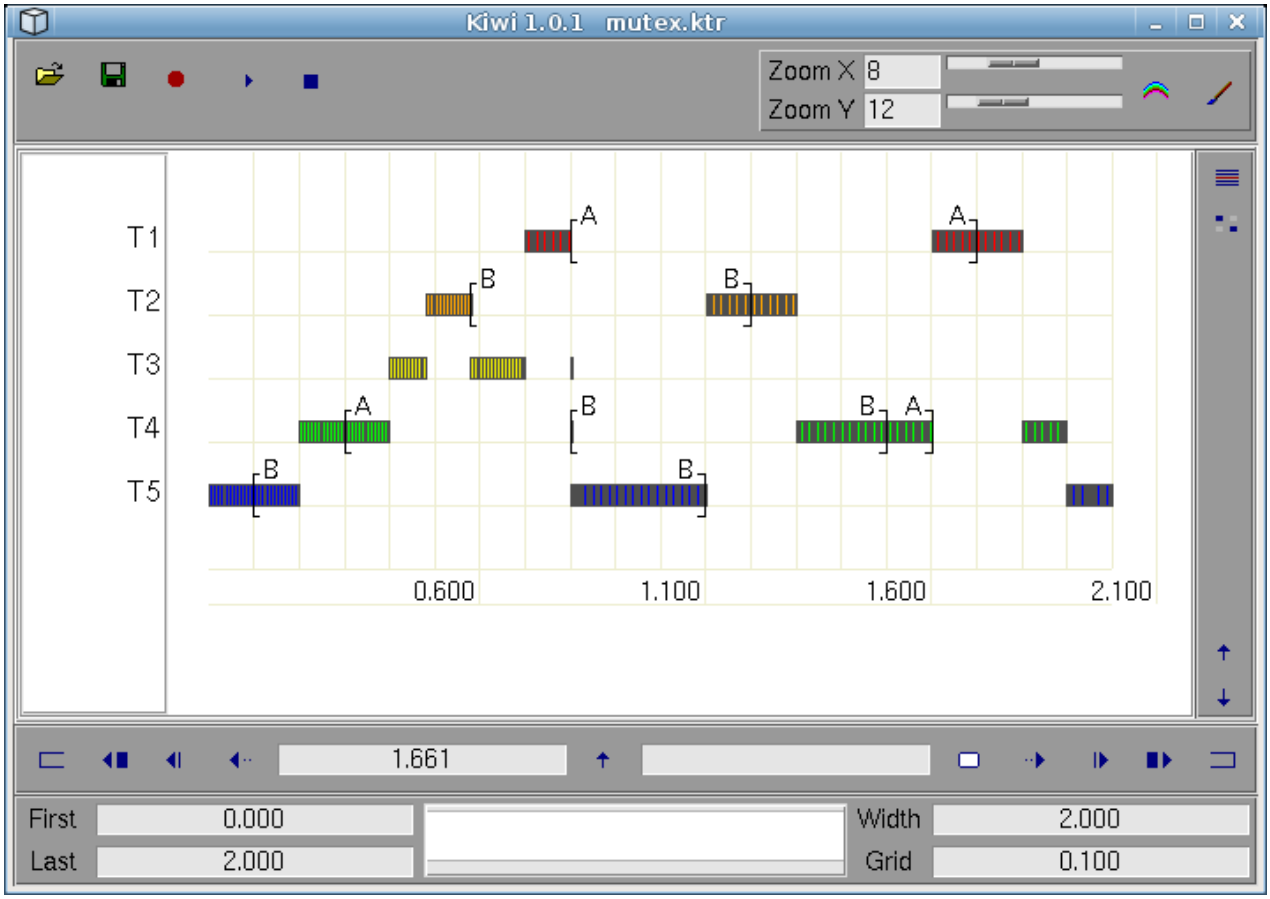

Figura 4.1: Interfaccia grafica di *kiwi*.

I file di input di *kiwi* sono costituiti da un file di testo dove ciascuna riga, a parte le istruzioni per l'impostazione di opzioni di disegno come i colori o il tracciamento di linee verticali, è costituita da almeno 3 campi separati da spazio:

- momento in cui avviene l'evento
- tipo di evento
- identificativo del task per cui avviene

Il momento in cui avviene l'evento corrisponde ad un numero da 0 a 9999.999999999, che può essere espresso anche mediante notazione esponenziale (corrispondente al *%E* di *printf*). Il tipo di evento esprime ciò che avviene e l'identificativo del task, un numero da 0 a 99, indica quale task sia interessato all'evento. Ad esempio

2.1 EXEC-B 0

indica che all'istante 2.1 il task 0 (o meglio un job del task 0) inizia la sua esecuzione. Gli eventi utilizzati dal programma sviluppato sono riportati nella seguente tabella

Nome	Significato
EXEC-B	inizio esecuzione di una porzione di job
EXEC-E	fine esecuzione di una porzione di job
LOCK	richiesta di lock su un mutex
UNLOCK	rilascio di un mutex

Dove *LOCK* e *UNLOCK* prevedono come quarto parametro una stringa indicante il nome della risorsa. Molti eventi poi prevedono la possibilità di specificare altri parametri, come il colore da utilizzare, ma per questi dettagli si rimanda a [53]. Un fatto non documentato nella guida appena citata riguarda la possibilità di inserire commenti nel file iniziando la riga con il carattere #, cosa che è stata ampiamente sfruttata nell'output del programma realizzato.

Nell'applicazione sviluppata gli eventi sono rappresentati mediante una struttura chiamata *event* definita nel file *event.h*, qui riportato:

```
#include <rtl_time.h>

struct event{
  hrtime_t time;
  int type;
  long long int info,info2;
};

extern void (*log_event)(struct event* e);
extern void default_log(struct event* e);

extern void log_simple(int type);
extern void log_now(int type,long long int info);
extern void log_notime(int type,long long int info);
extern void log_now_full(int type,long long int info,long long int info2);
extern void log_notime_full(int type,long long int info,long long int info2
   );

#define LOG_FIFOID 1
#define LOG_FIFOFILE "/dev/rtf1"
#define LOG_FIFOSIZE 10000*sizeof(struct event)

#define TEST_LOG 0
#define LOG_MODULE_ON 1
#define LOG_MODULE_OFF 2
#define TASK_START 3
#define JOB_START 4
#define JOB_END 5
#define SEARCH_TARGET 6
#define MAX_SEARCH 7
#define ITERS_SEARCH 8
#define ADD_TASK 9
#define START_TASK_ID 10
#define START_TASK_PHI 11
#define START_TASK_P 12
#define START_TASK_E 13
#define START_TASK_D 14
#define STOP_TASK 15
#define STOP_ALL_TASKS 16
#define ENTER_SCHEDULER 17
#define EXIT_SCHEDULER 18
#define NEXT_TASK 19
#define NEXT_TASK_LINUX 20
#define TIME_ZERO 21
#define MISSED_EVENTS 22
#define MUTEX_LOCK 23
#define MUTEX_UNLOCK 24
#define MUTEX_GOT 25
```

Come si può vedere leggendo il codice, si è caratterizzato ciascun evento con il momento in cui avviene, il suo tipo, descritto mediante un *int,* e con un massimo di due informazioni, ciascuna delle quali viene memorizzata mediante un intero da 64 bit. Il puntatore a funzione *log_event* permette di

cambiare durante l'esecuzione la funzione utilizzata per effettuare il log, e viene inizialmente posto alla funzione chiamata *default_log*. Viene poi definita una serie API per registrare un evento, il cui utilizzo è illustrato nella seguente tabella (nell'implementazione di tutte le funzioni, in ultima analisi, viene invocata la funzione puntata la *log_event*)

Funzione	Utilizzo
log_simple(int type)	registra l'evento *type* senza alcuna informazione aggiuntiva (nemmeno il momento in cui avviene)
log_now(int type,long long int info)	registra l'evento *type*, l'informazione *info* ed il momento in cui si effettua la registrazione
log_notime(int type,long long int info)	registra l'evento *type* e l'informazione *info* senza il momento in cui si effettua la registrazione
log_now_full(int type,long long int info,long long int info2)	registra l'evento *type*, le informazioni *info* ed *info2* ed il momento in cui si effettua la registrazione
log_notime_full(int type,long long int info,long long int info2)	registra l'evento *type* e le l'informazioni *info* ed *info2* senza il momento in cui si effettua la registrazione

A queste definizioni segue un insieme di costanti relative al modulo, descritto in seguito, che trasmette gli eventi ai programmi in user-space mediante una coda FIFO. Nelle ultime righe del file infine sono definite numerose costanti, relative agli eventi generati dagli altri moduli, che saranno presentate successivamente (la prima di queste costanti, *TEST_LOG*, è stata definita per registrare eventi senza uno specifico significato, ad esempio in fase debug).

L'implementazione delle varie funzioni è fornita dal file *evenlog.c*:

```
#include <linux/module.h>
#include "event.h"

MODULE_LICENSE("GPL");

void default_log(struct event* e){
}

struct event ev;

void (*log_event)(struct event* e)=default_log;


void inline log_simple(int type){
  ev.time=0;
  ev.type=type;
  ev.info=0;
  ev.info2=0;
  (*log_event)(&ev);

}

void inline log_notime(int type,long long int info){
  ev.time=0;
  ev.type=type;
  ev.info=info;
  ev.info2=0;
  (*log_event)(&ev);
}

void inline log_now(int type,long long int info){
  ev.time=gethrtime();
```

```
  ev.type=type;
  ev.info=info;
  ev.info2=0;
  (*log_event)(&ev);
}

void inline log_now_full(int type,long long int info,long long int info2){
  ev.time=gethrtime();
  ev.type=type;
  ev.info=info;
  ev.info2=info2;
  (*log_event)(&ev);
}

void inline log_notime_full(int type,long long int info,long long int info2
   ){
  ev.time=0;
  ev.type=type;
  ev.info=info;
  ev.info2=info2;
  (*log_event)(&ev);
}

int init_module(void) {
  return 0;
}

void cleanup_module(void) {
}
```

Si può notare che tutte le funzioni di log presenti nella precedente tabella, dopo aver impostato i vari campi della struttura *ev* (di tipo event), ponendo pari a 0 i parametri non specificati e leggendo il tempo corrente se la funzione lo prevede, invocano **log_event* che, nel caso di default, corrisponde ad una funzione dal corpo vuoto. Per fare in modo che i vari eventi vengano stampati a video è stato scritto il modulo *logtest.c,* che si limita a far puntare *log_event* ad una funzione che stampa gli eventi usando *printk* ed a notificare il suo caricamento e la sua rimozione mediante la registrazione, rispettivamente, degli eventi *LOG_MODULE_ON* e *LOG_MODULE_OFF*.

```
#include <linux/module.h>
#include "event.h"
#include "messages.h"

MODULE_LICENSE("GPL");

void printklog(struct event* e){
  printk("%Ld:␣%s␣%Ld.%Ld␣\n",e->time,messages[e->type],e->info,e->info2);
}

int init_module(void) {
  log_event=&printklog;
  log_simple(LOG_MODULE_ON);
  return 0;
}

void cleanup_module(void) {
  log_simple(LOG_MODULE_OFF);
  log_event=&default_log;
}
```

L'array *messages* utilizzato in *printklog* è un array di stringhe definito nel file *messages.h* che permette di fornire un output più comprensibile visto che al posto dell'identificativo di ciascun evento

viene stampata una descrizione di quest'ultimo. La lettura del file *messages.h* permette di intuire già il significato di molte delle costanti di *event.h* e viene quindi riportato subito:

```
 char * messages[]={
  "Test␣stampa",//0
  "Caricamento␣modulo␣log",
  "Rimozione␣modulo␣log",
  "Avvio␣task",
  "Avvio␣job",
  "Terminazione␣job",//5
  "Ricerca␣iterazioni␣per␣tempo",
  "Ricerca␣limite␣superiore",
  "Ricerca␣numero␣iterazioni",
  "Aggiunta␣task",
  "Avvio␣task␣(id)",//10
  "Avvio␣task␣(phi)",
  "Avvio␣task␣(p)",
  "Avvio␣task␣(e)",
  "Avvio␣task␣(D)",
  "Rimozione␣task",//15
  "Rimozione␣di␣tutti␣i␣task",
  "Entrata␣scheduler",
  "Uscita␣scheduler",
  "Thread␣switch",
  "Thread␣switch:␣linux",//20
  "Impostazione␣tempo␣a␣fase␣0",
  "Eventi␣non␣notificati",
  "Mutex␣lock",
  "Mutex␣unlock",
  "Mutex␣acquisito"//25
};
```

Risulta opportuno chiarire le motivazioni di alcune delle scelte fatte. In particolare, si è deciso di descrivere il tipo di ciascun evento con un intero, e non direttamente con un messaggio associato, sia per minimizzare l'overhead sia per facilitare l'utilizzo di sequenze di eventi da parte di programmi che possono, ad esempio, effettuare un semplice *switch* basato sul campo *type*. Si è poi scelto di implementare la funzione *default_log* come funzione vuota, anziché come stampa a video, perché nella maggior parte dei casi (si pensi, ad esempio, al fatto che si registrano le invocazioni dello scheduler) la velocità dei dati generati rende inutile l'output a video. Infine, sebbene i file *event.h* e *messages.h* siano strettamente collegati visto che la posizione di ciascuna stringa nell'array deve corrispondere alla rispettiva costante, si è preferito mantenere separate le due cose, visto che solo i programmi che interagiscono con l'utente hanno bisogno dell'array *messages*.

4.2.2 Creazione di task

Per rappresentare ciascun task si è utilizzata una struttura chiamata *task_data*, definita nel file *taskdata.h*:

```
#ifndef __TASK_DATA__
#define __TASK_DATA__
#include <time.h>

struct task_data{
  hrtime_t phi;
  hrtime_t p;
  hrtime_t e;
  hrtime_t D;
  int priority;
};
```

```
#define uSEC (long long int)1000
#define mSEC (long long int)1000000
#define SEC  (long long int)1000000000
#endif
```

Come si può notare si sono utilizzati i quattro elementi tipici della descrizione di un task periodico presentati in [1], fase, periodo, tempo di esecuzione e deadline relativa, ed a questo si è aggiunta la priorità di ciascun task. In realtà la deadline relativa non viene utilizzata da alcuno dei moduli sviluppati, ma si è deciso di includerla comunque per facilitare possibili sviluppi futuri. Si segue poi la convenzione di RTLinux di rappresentare i task costituiti da un solo job mediante periodo nullo, anche se concettualmente il loro periodo sarebbe infinito. Alla file del file appena presentato sono state dichiarate, per comodità, alcune costanti di conversione tra tempi espressi in microsecondi, millisecondi e secondi e tempi espressi in nanosecondi.

Parte fondamentale dell'applicazione realizzata è il modulo *tasktest.o*, generato a partire dal file *tasktest.c*. Le funzioni più importanti di tale modulo sono dichiarate nel file *tasktest.h*, in modo che altri moduli possano essere compilati facendo riferimento ad esse:

```
#define MAX_TASKS 20

extern void add_task(struct task_data * data);
extern void start_tasks(hrtime_t when);
extern void stop_task(int task);
extern void stop_tasks(void);
extern void soft_stop_task(int task);
extern void soft_stop_tasks(void);
extern void set_fun(int task,void * fp);
extern void use_cpu(int task);
```

Dopo aver aggiunto un insieme di task di parametri voluti mediante invocazioni successive di *add_task*, è possibile renderli operativi eseguendo *start_tasks*. Questa funzione prevede un parametro, *when*, utilizzato per indicare il riferimento temporale delle fasi, espresso in nanosecondi trascorsi dal momento di accensione del sistema. Ad esempio invocando *start_tasks(gethrtime())* si ottiene (concettualmente) il rilascio immediato di tutti i task a fase nulla. Per terminare l'esecuzione di un singolo task viene fornita la funzione *stop_task*, avente come parametro il task da fermare, mentre per terminare tutti i task in esecuzione viene fornita la funzione senza argomenti *stop_tasks.* Queste due funzioni invocano *pthread_delete_np*, mentre le funzioni *soft_stop_task* e *soft_stop_tasks* permettono che ciascun task termini il job corrente prima di venir distrutto. Per una piena comprensione delle ultime due funzioni, *set_fun* e *use_cpu,* conviene mostrare e commentare il contenuto del file *tasktest.c*:

```
#include <linux/module.h>
#include <rtl.h>
#include <pthread.h>
#include "event.h"
#include "taskdata.h"
#include "tasktest.h"

#define STACKSIZE 20000
#define STARTERSTACKSIZE 20000

#define TASK_IS_INVALID 0
#define TASK_IS_STARTING 1
#define TASK_IS_READY 2
#define TASK_IS_TERMINATING 3

#define MINTIME 100000

MODULE_LICENSE("GPL");
```

```
int phi=0,e=0,p=0,D=0;
MODULE_PARM (phi, "i");
MODULE_PARM (e, "i");
MODULE_PARM (p, "i");
MODULE_PARM (D, "i");

char starter_stack[STARTERSTACKSIZE];

struct task_handle {
  pthread_t id;
  char status;
  struct task_data data;
  char stack[STACKSIZE];
  void * function;
  struct{
    hrtime_t last,remaining;
  }tsim;
};

struct task_handle tasks[MAX_TASKS];

void use_cpu(int task){
  static hrtime_t now,elap;
  tasks[task].tsim.last=gethrtime();
  tasks[task].tsim.remaining=tasks[task].data.e;
  while (tasks[task].tsim.remaining>0){
    now=gethrtime();
    elap=now-tasks[task].tsim.last;
    tasks[task].tsim.last=now;
    if (elap<MINTIME) tasks[task].tsim.remaining-=elap;
  }//while
}//use_cpu


void dummy_f(void *args){
  int task=(int)args;
  log_now(TASK_START,task);
  while (tasks[task].status==TASK_IS_READY){
    pthread_wait_np();
    log_now(JOB_START,task);
    use_cpu(task);
    log_now(JOB_END,task);
  }//while
}//dummy_f


void set_fun(int task,void * fp){
  tasks[task].function=fp;
}


void add_task(struct task_data * data){
  int i=0;
  while (i<MAX_TASKS&&(tasks[i].status!=TASK_IS_INVALID)) i++;
  if (i==MAX_TASKS){
    printk("Impossibile␣inserire␣un␣nuovo␣task,␣limite␣raggiunto\n");
  }else{
    tasks[i].data=(*data);
    log_now(ADD_TASK,i);
    tasks[i].status=TASK_IS_STARTING;
```

```
    tasks[i].id=0;
    set_fun(i,&dummy_f);
  }
}//add_task


void  start_tasks_fun(void * arg){
  hrtime_t when=*((hrtime_t *)(arg));
  int i;
  log_notime(TIME_ZERO,when);
  for (i=0;i<MAX_TASKS;i++){
    if (tasks[i].status==TASK_IS_STARTING){
      tasks[i].status=TASK_IS_READY;
      pthread_attr_t attr;
      struct sched_param p;
      pthread_attr_init (&attr);
      pthread_attr_setstackaddr (&attr,tasks[i].stack);
      pthread_attr_setstacksize (&attr,STACKSIZE);
      pthread_create (&(tasks[i].id),&attr,tasks[i].function,(void *)i);
      p.sched_priority=tasks[i].data.priority;
      pthread_setschedparam (tasks[i].id, SCHED_FIFO, &p);
      pthread_make_periodic_np (tasks[i].id,when+tasks[i].data.phi,tasks[i
         ].data.p);
      log_now_full(START_TASK_ID,i,(long long int)tasks[i].id);
      log_now_full(START_TASK_PHI,i,tasks[i].data.phi);
      log_now_full(START_TASK_P,i,tasks[i].data.p);
      log_now_full(START_TASK_E,i,tasks[i].data.e);
      log_now_full(START_TASK_D,i,tasks[i].data.D);
    }//ready
  }//for
}//start_tasks_fun


void start_tasks(hrtime_t when){
  pthread_t thread;
  pthread_attr_t attr;
  pthread_attr_init (&attr);
  pthread_attr_setstackaddr (&attr,starter_stack);
  pthread_attr_setstacksize (&attr,STARTERSTACKSIZE);
  pthread_create (&thread,&attr, &start_tasks_fun,&when);
  struct sched_param p;
  p.sched_priority=sched_get_priority_max(SCHED_FIFO);
  pthread_setschedparam (thread, SCHED_FIFO, &p);
  pthread_join(thread,NULL);
}//start tasks


void stop_task(int task){
  if ((tasks[task].status==TASK_IS_READY)||(tasks[task].status==
     TASK_IS_TERMINATING)){
    pthread_delete_np (tasks[task].id);
    log_now(STOP_TASK,task);
  }
  tasks[task].status=TASK_IS_INVALID;
}


void stop_tasks(void){
  int i;
  log_now(STOP_ALL_TASKS,0);
```

```
  for(i=0;i<MAX_TASKS;i++) stop_task(i);
}


void soft_stop_task(int task){
  if (tasks[task].status==TASK_IS_READY)
    tasks[task].status=TASK_IS_TERMINATING;
}


void soft_stop_tasks(void){
  int i;
  for(i=0;i<MAX_TASKS;i++) soft_stop_task(i);
}


void init_taskset(void){
  int i;
  for (i=0;i<MAX_TASKS;i++){
    tasks[i].id=0;
    tasks[i].status=TASK_IS_INVALID;
  }
}


int init_module(void) {
  init_taskset();
  if (!e){
    printk("Nessun task avviato\n");
  }else{
    struct task_data data;
    data.phi=phi*mSEC;
    data.e=e*mSEC;
    data.p=p*mSEC;
    data.D=D*mSEC;
    add_task(&data);
    start_tasks(gethrtime());
  }
  return 0;
}


void cleanup_module(void) {
  stop_tasks();
}
```

In questo modulo viene creato un array, di dimensioni pari alla costante *MAX_TASKS* definita in *tasktest.h* di strutture *task_handle*, ciascuna delle quali contiene i dati di un task. Come immaginabile viene creato un thread per ogni task, e visto che la struttura *rtl_thread_struct* di RTLinux stranamente non contiene un campo *id* o simili, si è scelto di utilizzare come identificativo per i thread (campo *id* di *task_handle*) la locazione di memoria dove sono allocate le informazioni del thread piuttosto che modificare l'appena citata struttura di RTLinux.

Lo stato del task (campo *status*) assume sempre il valore di una delle quattro costanti *TASK_IS_INVALID*, *TASK_IS_STARTING*, *TASK_IS_READY* e *TASK_IS_TERMINATING* definite in *tasktest.c*. Tutte le strutture relative ai task sono inizializzate al valore *TASK_IS_INVALID* dalla funzione *init_taskset()* che viene eseguita come prima operazione di *init_module*. Quando si aggiunge un task mediante *add_task* i suoi parametri temporali (fase, periodo, etc.) vengono memorizzati nel campo *data* e lo stato viene commutato in *TASK_IS_STARTING*. All'esecuzione di *start_tasks* tutti i thread relativi ai task nello stato *TASK_IS_STARTING* vengono avviati, e lo

stato viene posto a *TASK_IS_READY*. Quando si invoca *soft_stop_task* se un task si trova in stato *TASK_IS_READY* viene portato a *TASK_IS_TERMINATING* mentre quando si esegue *stop_task* se lo stato è *TASK_IS_READY* o *TASK_IS_TERMINATING* viene invocata *pthread_delete_np* e si pone lo stato a *TASK_IS_INVALID*.

Visto infatti che si desidera poter avviare task da thread real-time, è necessario preallocare lo stack, come spiegato nella sezione relativa alla gestione dei thread. La dimensione è definita dalla costante *STACKSIZE* e per sicurezza si è posto tale dimensione ad un valore abbastanza elevato, 20000, quasi 20Kbyte per ogni thread, comunque, dato il modesto valore di *MAX_TASKS* (pari a 20), l'utilizzazione totale di memoria è 400Kbyte, assolutamente non problematica su una macchina odierna.

Quando viene invocata *start_tasks* viene creato un thread a priorità massima che esegue la funzione *start_tasks_fun*, la quale, a sua volta, avvia tutti i thread corrispondenti ai task in stato *TASK_IS_STARTING*. Utilizzando la massima priorità si ha la certezza che funzione possa essere interrotta solo dalle routine di gestione delle interruzioni (tra cui lo scheduler, associato al timer) ma non da altri thread, se si esclude il caso particolare di altri thread alla massima priorità creati successivamente. Questo permette di lanciare tutti i nuovi thread nel tempo più breve possibile; si è preferito non disabilitare gli interrupt, tuttavia volendo rendere l'operazione di creazione dei task atomica (nel caso di architettura a singolo processore) basta aggiungere la coppia di istruzioni *rtl_no_interrupts* e *rtl_restore_interrupts* all'inizio ed alla fine *start_tasks_fun.* La funzione costituente ciascuno dei thread creati in questa fase è quella puntata dal campo *function* della struttura *task_handle*. Tale campo viene inizializzato, al momento dell'invocazione di *add_task*, all'indirizzo della funzione *dummy_f*. Questa funzione simula un task periodico (o costituito da un solo job nel caso particolare di periodo nullo) eseguendo un ciclo, finché lo stato del task è *TASK_IS_READY*, nel quale

- ci si mette in attesa del rilascio del prossimo rilascio di un job del task (utilizzando *pthread_wait_np()*)
- si invoca la funzione *use_cpu*

Tale funzione a sua volta simula l'esecuzione di un job di durata pari al tempo di esecuzione del task specificato come argomento. *use_cpu* è una delle due funzioni tralasciate durante la spiegazione di *tasktest.h*; la seconda delle funzioni non descritte in tale occasione è *set_fun* che, se invocata dopo *add_task* e prima di *start_tasks,* permette, mediante la modifica del puntatore *function,* di sostituire la funzione *dummy_f,* normalmente eseguita dai thread, con una routine a piacere. Questo rende possibile, ad esempio, definire task che effettuino lock ed unlock su mutex, come si vedrà in seguito.

Lo sviluppo della funzione *use_cpu* ha subito varie fasi, che hanno permesso di notare fatti interessanti. In particolare il primo tentativo è stato definire un ciclo for che contasse per un numero di iterazioni specificato da un parametro *long long int* argomento della funzione. Per determinare il numero di iterazioni necessarie a fare in modo che si continuasse a ciclare per un tempo voluto si era scritto il seguente codice

```
long long int search_iter(hrtime_t t){
  long long int min,max,it,mit;
  hrtime_t et;
  int i=0,skip;
  rtl_irqstate_t flags;
  mit=0;
  for (;i<SAMPLES;i++){
    it=TEST_ITER;
    rtl_no_interrupts (flags);
    et=gethrtime();
    useCpu(it);
    et=gethrtime()-et;
    rtl_restore_interrupts (flags);
    min=0;max=(t/et>0?t/et:1)*TEST_ITER;
```

```
    do{
      max=2*max;
      et=gethrtime();
      useCpu(max);
      et=gethrtime()-et;
    }while (et<2*t);
    skip=SKIPAFTER;
    while (max-min>MAXDIFF){
      it=(max+min)/2;
      et=gethrtime();
      useCpu(it);
      et=gethrtime()-et;
      if (et>t) max=it;
      else if (et<t) min=it;
      else max=min=it;
      if (!(--skip)) break;
    }
    mit+=it;
  }
  return mit/SAMPLES;
}
```

ossia

- si misurava il tempo necessario ad effettuare *TEST_ITER* iterazioni
- si calcolava il numero di iterazioni necessario a ciclare per t nanosecondi, argomento della funzione, supponendo un rapporto di proporzionalità diretta tra numero di iterazioni e ciclo impiegato
- si continuava a raddoppiare tale numero di iterazioni, finché non si giungeva ad avere un tempo impiegato superiore ad doppio del tempo desiderato t
- si effettuava una ricerca binaria, mantenendo un estremo inferiore (*min*) ed un estremo superiore (*max*) al numero di iterazioni, e si terminava solo quando la differenza tra questi due estremi era minore di *MAXDIFF*

Tutto il processo veniva ripetuto *SAMPLES* volte, e si restituiva la media dei valori ottenuti, tuttavia si è notato che aumentare tale valore non migliorava le prestazioni. Per evitare ricerche troppo lunghe si era poi introdotto un controllo che permetteva di uscire dalla ricerca binaria dopo *SKIPAFTER* iterazioni, tuttavia si è verificato che la convergenza era sufficientemente veloce anche ponendo *MAXDIFF* a 1. Questa funzione era sfruttata all'invocazione di *add_task* per impostare un campo *iters* di *task_handle* che veniva utilizzato dalla funzione di simulazione dei task. Come per *start_tasks* anche in questo caso si creava un thread a priorità massima, in modo da non essere interrotti durante l'esecuzione dei test se non dallo scheduler. Il tempo utilizzato dallo scheduler è quasi costante vista la semplicità della funzione *rtl_schedule*, almeno se il numero di task rimane lo stesso. Si supponeva quindi che con tempi di esecuzione dei task sufficientemente lunghi (i test sono stati effettuati con tempi pari a un secondo) e quindi con numero di interruzioni dello scheduler praticamente costante il tempo di esecuzione di *use_cpu* in fase di determinazione del numero di iterazioni e durante la simulazione dell'esecuzione di un job fosse abbastanza simile. Tuttavia analizzando poco più di 100 job di un task di periodo pari a due secondi e tempo di esecuzione pari a un secondo si è misurato un tempo di esecuzione massimo superiore quasi del 10% rispetto al tempo nominale di un secondo. Si è quindi tentato di rendere meno variabile il tempo di esecuzione della la funzione *use_cpu* inserendo delle istruzioni assembly *nop* dentro al ciclo. Con vari tentativi si è determinato che il numero apparentemente migliore di *nop* era 32, con le quali, pur ovviamente diminuendo la granularità dei tempi realizzabili si è giunti ad un tempo massimo superiore al 6% rispetto a quello nominale nelle stesse condizioni del test precedente. Il passo successivo è stato scrivere la funzione *use_cpu* in assembly

```
int uc1,uc2;
void  use_cpu(long long int x){
  uc1=hi(x);uc2=lo(x);
  __asm__ __volatile__(
"␣␣␣␣␣␣␣␣movl␣␣␣␣␣%0,␣%%ebx\n\
␣␣␣␣␣␣␣␣movl␣␣␣␣␣%1,␣%%eax\n\
␣␣␣␣␣␣␣␣addl␣␣␣␣␣$1,%%eax\n\
␣␣␣␣␣␣␣␣addl␣␣␣␣␣$1,%%ebx\n\
.cicla:␣\n\
␣␣␣␣␣␣␣␣subl␣␣␣␣␣$1,%%eax\n\
␣␣␣␣␣␣␣␣jne␣␣␣␣␣␣.cicla\n\
␣␣␣␣␣␣␣␣subl␣␣␣␣␣$1,%%ebx\n\
␣␣␣␣␣␣␣␣jne␣␣␣␣␣␣.cicla\n\
":/*output*/:"r"(uc1),"r"(uc2):"eax", "ebx");
}
```

dove *__volatile__* indica al compilatore che non è possibile effettuare alcuna previsione sul valore contenuto nelle locazioni di memoria (tipicamente volatile si impiega quando si accede a locazioni corrispondenti a periferiche mappate in memoria) ossia che non deve effettuare ottimizzazioni. Tuttavia, fatto molto sorprendente, si è riscontrato un leggero peggioramento, in termini di tempo massimo, rispetto alla versione realizzata mediante ciclo *for* e *nop*.

A questo punto si è deciso di utilizzare l'approccio presente nell'attuale implementazione. Il funzionamento della funzione *use_cpu* ora inclusa in *tasktest.c* si basa sull'assunto che vi sia un tempo minimo di esecuzione dei job, chiamato *MINTIME,* che si è posto pari a 100 microsecondi. La funzione (che, si ricorda, simula un job) dapprima copia il valore del tempo di esecuzione del task passato come argomento nel campo *tsim.remaining* relativo a tale task ed inizializza la variabile *tsim.last* con il tempo corrente; poi si entra in un ciclo, dove

- si legge il tempo corrente e lo si memorizza nella variabile *now*,
- si calcola il tempo trascorso dalla lettura precedente (mediante sottrazione di *tsim.last* da *now)*, e lo si pone nella variabile *elap,*
- si salva il valore *now* nel campo *tsim.last* per la prossima esecuzione,
- si verifica se elap è minore di *MINTIME*, e se ciò avviene, vista l'ipotesi fatta sul tempo minimo di esecuzione, si assume di aver eseguito in tutto l'intervallo $[tsim.last, now]$ e si sottrae il tempo *elap* da *tsim.remaining.*

Il ciclo continua finché la variabile *tsim.remaining,* che serve ad indicare il tempo di esecuzione rimanente del job, rimane positiva. Questa soluzione, che presenta il vantaggio rispetto alle precedenti di non richiedere alcun tipo di taratura (come la ricerca binaria del numero di iterazioni), si è rivelata molto più performante di quelle precedentemente sviluppate, visto che si continuano a monitorare i tempi, ed il tempo massimo, nel caso di un solo task, può essere superiore a quello nominale solo per un'iterazione in eccesso. Se vi sono più task nel caso peggiore ogni job può subire preemption subito prima della lettura del tempo, e quindi non considerare come tempo impiegato dal job stesso il tempo tra l'ultima lettura ed il momento in cui ha subito preemption; vista la velocità del ciclo, comunque, se non si costruiscono appositamente casi in cui un task subisce continuamente preemption la sommatoria dei tempi trascurati non raggiunge mai valori molto elevati.

Inoltre, variando *MINTIME* si può decidere se includere o meno i tempi di esecuzione dello scheduler nel tempo del job. Infatti assegnando a *MINTIME* un valore inferiore al tempo di esecuzione dello scheduler non si conteggia tale tempo in quello del job, permettendo di simulare sul sistema reale job aventi un tempo di esecuzione conosciuto. Viceversa se si sceglie un valore superiore al tempo necessario alla schedulazione, come si è effettivamente fatto, si ottiene una simulazione più simile alle situazioni presentate in [1], dove si considerano trascurabili i tempi di intervento dello scheduler.

Come ultima nota su *tasktest.o* si può osservare che viene fornita la possibilità di avviare un task nel momento di inserimento del modulo (comando *insmod*) specificandone i parametri da linea di comando. Per leggere i parametri *phi, e, p* e *D* si è utilizzata l'apposita macro *MODULE_PARM*. Nel caso non venga specificato nulla il tempo di esecuzione *e* rimane pari al suo valore di inizializzazione (zero) e non si avvia alcun task. I tempi specificati si assumono espressi in millisecondi sia per comodità sia perché ciò rende possibile l'utilizzo di *int* anziché di *long long*, non disponibili con *MODULE_PARM*.

4.2.3 Comunicazione con lo spazio utente

Per poter registrare gli eventi su disco e dare all'utente la possibilità di gesire i task si è creato un modulo real-time che apre due code FIFO, ciascuna utilizzata per uno di questi due scopi.

Per eseguire la prima operazione nella funzione *init_module* del file *fifotest.c* qui riportato si fa puntare *log_event* ad una funzione chiamata *log_handler*:

```
#include <linux/module.h>
#include <rtl_fifo.h>
#include <rtl_time.h>
#include "event.h"
#include "command.h"
#include "taskdata.h"
#include "tasktest.h"

#define DELAY 2*SEC

MODULE_LICENSE("GPL");


static struct event missed;
void log_handler(struct event* e){
  static char missed_event=0;
  if (missed_event){
    if (rtf_put(LOG_FIFOID,&missed,sizeof(struct event))>0){
      printk("Perdita␣eventi␣segnalata\n");
      missed_event=0;
    }
  }
  if (rtf_put(LOG_FIFOID,e,sizeof(struct event))<0){
    missed_event=1;
    soft_stop_tasks();
  }
}


int cmd_handler(unsigned int fifo){
  struct command cmd;
  int err=0;
  while ((err = rtf_get(CMD_FIFOID, &cmd, sizeof(cmd)))== sizeof(cmd)) {
    printk("Ricevuto␣comando␣%d\n",cmd.cmd);
    switch (cmd.cmd){
    case ADD:
      add_task(&cmd.data);
      break;
    case START:
      start_tasks(gethrtime()+DELAY);
      break;
    case STOP:
      stop_task(cmd.option);
      break;
```

```
    case STOP_ALL:
      stop_tasks();
      break;
    }//switch
  }//while
  if (err) rtl_printf("Errore␣nella␣lettura␣dei␣comandi,%d\n",err);
  return err;
}


int init_module(void){
  missed.type=MISSED_EVENTS;
  missed.info=missed.info2=missed.time=0;
  rtf_destroy(LOG_FIFOID);
  rtf_create(LOG_FIFOID,LOG_FIFOSIZE);
  log_event=&log_handler;
  rtf_destroy(CMD_FIFOID);
  rtf_create(CMD_FIFOID,CMD_FIFOSIZE);
  rtf_create_handler(CMD_FIFOID,&cmd_handler);
  return 0;

}

void cleanup_module(void){
  log_event=&default_log;
  rtf_destroy(LOG_FIFOID);
  stop_tasks();
}
```

Questa funzione si occupa di inserire ciascun evento che riceve, rappresentato mediante la già analizzata struttura *event*, nella coda *LOG_FIFOID*, in modo che i processi utente, accedendo a */dev/rtf1*, possano recuperare questi oggetti. La dimensione del buffer costituente la coda è pari alla costante *LOG_FIFOSIZE* cui si è dato valore *10000*sizeof(struct event),* ossia si è scelto di avere un massimo di 10000 messaggi accumulati e non letti. Qualora si riempia il buffer la funzione *rtf_put* segnala un errore mediante il ritorno di un numero negativo, ed in tal caso il controllo presente in *log_handler* oltre a porre la variabile *missed_event* a 1 (la quale viene inizializzata a 0), invoca *soft_stop_tasks*. Questo permette di fermare i task in situazioni in cui il controllo non passa mai a Linux ed i processi che dovrebbero leggere dal buffer non sono eseguiti. Sebbene a prima vista possa apparire un comportamento scorretto, questo permette di sbloccare il sistema in situazioni in cui si sono commessi errori, ad esempio avviando un insieme di task con fattore di utilizzazione totale superiore all'unità. Inoltre, poiché ci si prefigge come scopo dell'applicazione quello di simulare un insieme di task e registrare tutti gli eventi, la perdita anche di un solo di questi eventi giustifica la terminazione dell'esperimento (se questo comportamento non è desiderato è ovviamente possibile commentare la chiamata a *soft_stop_tasks*). Dopo una situazione di questo tipo all'arrivo di nuovi eventi si tenta, per prima cosa, di inserire un evento di tipo *MISSED_EVENTS* per segnalare, come indica il nome, che si sono persi degli eventi.

La gestione della seconda coda FIFO, *CMD_FIFOID*, viene affidata alla funzione *cmd_handler,* impostata in *init_module* come routine da invocare all'arrivo di nuovi dati. Le informazioni scambiate su questa FIFO sono di tipo *cmd*, una struttura definita dal file *command.h:*

```
#include "taskdata.h"

struct command{
  int cmd;
  int option;
  struct task_data data;
};
```

```
#define CMD_FIFOSIZE 100*sizeof(struct command)
#define CMD_FIFOID 0
#define CMD_FIFOFILE "/dev/rtf0"

#define ADD 1
#define START 2
#define STOP 3
#define STOP_ALL 4
```

Come intuibile la struttura è utilizzata per inviare comandi, il cui tipo viene specificato dal campo *cmd,* che può valere

- *ADD*, per aggiungere un task di parametri indicati dal campo *data,*
- *START*, per avviare tutti i task non ancora avviati,
- *STOP*, per terminare il task indicato dal campo *option,*
- *STOP_ALL*, per terminare tutti i task.

Molte delle informazioni inviate non sono necessarie, ad esempio si invia il campo data anche per l'esecuzione di *STOP_ALL*, comunque i comandi vengono inviati con frequenza molto bassa e l'utilizzare per tutti i comandi una stessa struttura, di dimensione fissa, semplifica di molto il codice. Nella routine *cmd_handler*, infatti, ci si limita ad effettuare uno *switch* sulla base del valore di *cmd* e ad invocare la corrispondente funzione fornita da *tasktest.o.* La presenza del ciclo *while*, apparentemente non necessario, garantisce un corretto funzionamento del codice anche in situazioni, molto rare, nelle quali si accumulino più comandi prima che la funzione venga invocata.

Una scelta implementativa degna di nota riguarda il fatto che si è deciso di invocare la funzione *start_tasks* passandole come parametro *gethrtime()+DELAY*, dove *DELAY* è pari ad un tempo di due secondi; in tal modo si imposta l'avvio dei task a fase nulla in un momento due secondi successivo al ricevimento del comando, istante che con altissima probabilità è posteriore alla terminazione della creazione di tutti i task.

L'applicazione in spazio utente *printevent* si occupa di leggere gli eventi dal file *LOG_FIFOFILE* (*/dev/rtf1*) ed effettuare le operazioni necessarie a produrre su standard output un file compatibile con il formato di *kiwi.* Risulta quindi possibile invocare[1]

./printevent >nomefile

per salvare un file contenente gli eventi. Qualora si volesse minimizzare il tempo di acquisizione degli eventi sarebbe possibile salvare gli eventi (in formato binario) con un semplice

cat /dev/rtf1>nomefilebinario

modificare *printevent* in modo da fargli leggere da standard input ed invocarlo con

cat nomefilebinario | printevent >nomefile

Si è comunque riscontrato che il tempo impiegato dalla conversione di formato non crea alcun problema in condizioni normali. Il codice di *printevent.c* è il seguente:

```
#include <fcntl.h>
#include <stdio.h>
#include "event.h"
#include "command.h"
#include "messages.h"
#include "taskdata.h"
```

[1]Una breve descrizione dei comandi *bash* presenti in questa relazione è riportata in appendice.

```
#include "tasktest.h"

int task_id[MAX_TASKS];
hrtime_t timezero=0;
int runningtask=-1;

int get_task(int id){
  int i=0;
  for (;i<MAX_TASKS;i++)
    if (task_id[i]==id) return i;
  return MAX_TASKS+2;//task di servizio
}


void inline printev(hrtime_t time,char * string,int task,int resource){
  if (!timezero) timezero=time;
  if (time<timezero) printf("#");
  if (resource<0)
    printf("%.20E␣%s␣%d\n",(double)(time-timezero)/SEC,
           string,task);
  else
    printf("%.20E␣%s␣%d␣%c\n",(double)(time-timezero)/SEC,
           string,task,resource+'A');
}


int main(void){
  int fd = open(LOG_FIFOFILE, O_RDONLY);
  if (fd<0) return -1;
  fd_set fdset;
  FD_ZERO(&fdset);
  FD_SET(fd,&fdset);
  struct event e;
  while (1){
    if(select(FD_SETSIZE, &fdset, NULL, NULL, NULL)<0)
      return -1;
    else{
      if (read(fd,&e,sizeof(e))<0) return -1;
      printf("#%Ld:␣%s␣%Ld.%Ld␣\n",e.time,
           messages[e.type],e.info,e.info2);
      switch (e.type){
      case START_TASK_ID:
        task_id[(int)e.info]=e.info2;
        break;
      case TIME_ZERO:
        timezero=e.info;
        break;
      case ENTER_SCHEDULER:
        if (runningtask!=-1)
          printev(e.time,"EXEC-E",runningtask,-1);
        printev(e.time,"EXEC-B",MAX_TASKS,-1);
        break;
      case NEXT_TASK:
        runningtask=get_task((int)e.info);
        break;
      case NEXT_TASK_LINUX:
        runningtask=MAX_TASKS+1;
        break;
      case EXIT_SCHEDULER:
        printev(e.time,"EXEC-E",MAX_TASKS,-1);
```

```
        printev(e.time,"EXEC-B",runningtask,-1);
        break;
      case MUTEX_LOCK:
        printev(e.time,"LOCK",e.info,e.info2);
        break;
      case MUTEX_UNLOCK:
        printev(e.time,"UNLOCK",e.info,e.info2);
        break;
      case MISSED_EVENTS:
        printf("#####\n");
        break;
      }//switch
    }//select>=0
    }//whilex
  return 0;
}
```

Nel *main* del programma, dopo aver aperto *LOG_FIFOFILE* si entra in un ciclo infinito che attende l'arrivo di eventi nel file, stampa tutti i campi dell'evento come commento ed in base al tipo compie le operazioni necessarie. In particolare, per adattare l'identificativo usato per i thread (che si ricorda essere dalla posizione in memoria della struttura del thread stesso) al formato di *kiwi* (che prevede identificatori da 0 a 99) si gestisce un array per la conversione; osservando *start_tasks_fun*, funzione di *tasktest.c* che si occupa di avviare tutti i task con stato *TASK_IS_STARTING,* si può notare che viene eseguita l'istruzione

log_now_full(START_TASK_ID,i,(long long int)tasks[i].id)

che pone nelle due informazioni l'indice assunto dal task in *tasks* ed il suo identificatore. Il numero che si usa nel file di output prodotto da *printevent* corrisponde a tale indice, in modo che se si aggiungono dei task in sequenza e poi si avviano, i loro identificatori nel file prodotto per *kiwi* sono $0, 1, ..., n-1$; questa è la motivazione che ha portato a scrivere le istruzioni

```
case START_TASK_ID:
        task_id[(int)e.info]=e.info2;
        break;
```

di *printevent*. Per caratterizzare lo scheduler si utilizza nell'output l'identificatore *MAX_TASKS*, mentre i thread non conosciuti, corrispondenti ad esempio al thread ad alta priorità che avvia i task, vengono indicati con identificatore *MAX_TASKS+1*. All'inizio di *start_tasks_fun* inoltre viene comunicato il "tempo a fase nulla" mediante

log_notime(TIME_ZERO,when);

cosicché sia possibile, nella funzione *printev* di *printevent,* riferire i tempi prodotti in output a tale istante. Nel codice infatti viene dichiarata la variabile *timezero*, posta, alla ricezione dell'evento *TIME_ZERO*, al valore della prima informazione registrata nel messaggio:

```
  case TIME_ZERO:
        timezero=e.info;
        break;
```

Introducendo questo comportamento, utile ad aumentare la comprensibilità dei grafici prodotti, vengono però a crearsi tempi negativi, visto che il tempo a fase nulla è impostato due secondi dopo l'esecuzione di *start_tasks*; *kiwi* accetta solo tempi positivi o nulli perciò si sono resi commenti (preponendo il carattere '#') tutti gli eventi caratterizzati da tempo negativo. La produzione di output con tempi relativi al tempo *timezero* genera tuttavia un secondo inconveniente, la non monotonicità del tempo che viene a crearsi alla ricezione di eventi *TIME_ZERO*. Per risolvere questo problema è sufficiente tagliare dal file prodotto da *printevent* la prima parte, relativa ad eventi antecedenti all'ultimo tempo 0, con il comando

tail -n +‘grep -n fase nomefilein | cut -f1 -d’:’ |tail -n 1‘ nomefilein >nomefileout

Per permettere la registrazione degli interventi dello scheduler e per determinare quale sia il task in esecuzione si sono aggiunte nel file *schedulers/rtl_sched.c* le seguenti parti:

```
log_now(ENTER_SCHEDULER,0);
```

all'inizio di *rtl_schedule*,

```
  if (new_task==&sched->rtl_linux_task)
    log_now(NEXT_TASK_LINUX,0);
  else
    log_now(NEXT_TASK,(long long int)new_task);
```

subito prima della riga

```
 if (new_task != sched->rtl_current) { /* switch out old, switch in new */
```

e

```
  log_now(EXIT_SCHEDULER,0);
```

al termine della funzione. In *printevent*, quindi,

- quando si riceve un evento di tipo *NEXT_TASK* o *NEXT_TASK_LINUX* si imposta il valore di *runningtask*, una variabile (inizializzata a -1) indicante il task che entrerà in esecuzione all'uscita dallo scheduler
- quando il tipo di evento che si legge è *ENTER_SCHEDULER* si comunica in output la terminazione del task *runningtask* e l'inizio della funzione di scheduling
- quando il tipo di evento che si legge è *EXIT_SCHEDULER* si stampano gli eventi relativi all'uscita dallo scheduler e all'inizio dell'esecuzione del task *runningtask*

Si noti che in questo modo, in realtà, il tempo di context switch viene incluso in larga parte nel tempo di esecuzione dello scheduler ed in minima parte (il ritorno dalla funzione *rtl_schedule*) nel tempo di esecuzione del job successivo.

Gli ultimi casi dello *switch* di *printevent* riguardano il tentativo di acquisizione di un mutex (*MUTEX_LOCK*), il conseguente rilascio (*MUTEX_UNLOCK*) e la perdita di eventi, segnalata mediante la stringa facilmente individuabile“#####”. La convenzione adottata per i mutex consiste nel dichiarare direttamente l'identificativo del task che opera sul mutex (numero tra 0 e *MAX_TASKS*-1) come prima informazione, ed indicare su quale si risorsa si opera mediante un identificatore numerico posto nel campo *info2* dell'evento. Per generare output più facilmente leggibili, comunque, la funzione *printev* stampa in output come nomi delle risorse A per la risorsa 0, B per la risorsa 1 e così via.

La seconda applicazione in spazio utente, *sendcmd,* costituisce un'interfaccia testuale per l'aggiunta, l'avvio e la terminazione di task. In particolare si continuano a presentare cinque scelte, ciascuna delle quali corrisponde alla pressione di un tasto

- aggiunta di un task, tasto 'a'
- avvio dei task aggiunti e non ancora avviati, tasto 's'
- terminazione di un task, tasto 'e'
- terminazione di tutti i task, tasto 'r'
- uscita dal programma, tasto 'q'

Qualora si scelga di aggiungere un task vengono chiesti i parametri fase, periodo, tempo di esecuzione, deadline relativa e priorità mentre se si sceglie di terminare un singolo task viene richiesto l'identificatore di tale task. Per la specifica dei tempi è possibile far seguire al numero uno spazio ed una lettera, che può essere 'n', 'u', 'm' o 's' per indicare, rispettivamente, che si tratta di nanosecondi, microsecondi, millisecondi o secondi. Una volta specificata un'unità di misura questa viene utilizzata anche per i tempi successivi se non se ne indica una nuova. Nel caso non venga mai indicata un'unità di misura si assume che tutti i tempi siano espressi in nanosecondi. Il programma, che utilizza la libreria *Ncurses* e di cui viene fornito uno screenshot in figura 4.2, non presenta niente di concettualmente interessante e viene quindi riportato senza ulteriori commenti

```
#include "command.h"
#include "taskdata.h"
#include <stdio.h>
#include <unistd.h>
#include <ncurses.h>

int cf;

int send_cmd(int cmd,int option,struct task_data * data){
  static struct command c;
  c.cmd=cmd;
  c.option=option;
  if (data!=NULL) c.data=(*data);
  if (write(cf, &c, sizeof(c)) < 0) {
    fprintf(stderr, "Errore␣trasmissione␣del␣comando␣%d.%d\n",
            cmd,option);
    return -1;
  }
  return 0;
}


void set_unit(hrtime_t *x,char unit){
  switch (unit){
  case 'n':
    break;
  case 'u':
    (*x)*=uSEC;
    break;
  case 'm':
    (*x)*=mSEC;
    break;
  case 's':
    (*x)*=SEC;
    break;
  default:
    break;
  }
}

int main(void){
  cf = open(CMD_FIFOFILE, O_WRONLY);
  char cmd=0;
  initscr();
  raw();
  int task;
  struct task_data data;
  char unit;
  refresh();
```

```
  while (cmd != 'q'){
    clear();
    mvprintw(0, 0,"Comando(q=uscita,a=aggiunta␣task,s=avvio␣task,\
e=rimozione␣task,r=rimozione␣di␣tutti␣i␣task):");
     refresh();
     cmd = getch();
     switch (cmd){
     case 'a':
       clear();
       noraw();
       mvprintw(0,0,"Aggiunta␣task");
       refresh();
       mvprintw(1,1,"Fase␣task:");
       scanw("%Ld␣%c",&data.phi,&unit);
       set_unit(&data.phi,unit);
       mvprintw(2,1,"Periodo␣task:");
       scanw("%Ld␣%c",&data.p,&unit);
       set_unit(&data.p,unit);
       mvprintw(3,1,"T.␣esecuzione␣task:");
       scanw("%Ld␣%c",&data.e,&unit);
       set_unit(&data.e,unit);
       mvprintw(4,1,"Deadline␣relativa␣task:");
       scanw("%Ld␣%c",&data.D,&unit);
       set_unit(&data.D,unit);
       mvprintw(5,1,"Priorita'␣task:");
       scanw("%d",&data.priority);
       mvprintw(6,0,
                "Richiesta␣aggiunta␣task\(%Ld,%Ld,%Ld,%Ld),%d\n",
                data.phi,data.p,data.e,data.D,data.priority);
       refresh();
       send_cmd(ADD,0,&data);
       raw();
       cmd = getch();
       break;
     case 's':
       clear();
       mvprintw(0,0,"Avvio␣task\n");
       refresh();
       send_cmd(START,0,NULL);
       cmd = getch();
       break;
     case 'e':
       clear();
       noraw();
       mvprintw(0,0,"Rimozione␣task");
       mvprintw(1,1,"Task␣da␣rimuovere:");
       scanw("%d",&task);
       mvprintw(6,0,"Rimozione␣del␣task␣%d\n",task);
       refresh();
       send_cmd(STOP,task,NULL);
       cmd = getch();
       raw();
       break;
     case 'r':
       clear();
       mvprintw(0,0,"Rimozione␣di␣tutti␣i␣task\n");
       refresh();
       send_cmd(STOP_ALL,0,NULL);
       cmd = getch();
       break;
```

```
    default:
      clear();
      mvprintw(0,0,"Comando non valido(q,a,s,e o r):");
      refresh();
      break;
    }//switch
    refresh();
  }
  endwin();
  return 0;
}//main
```

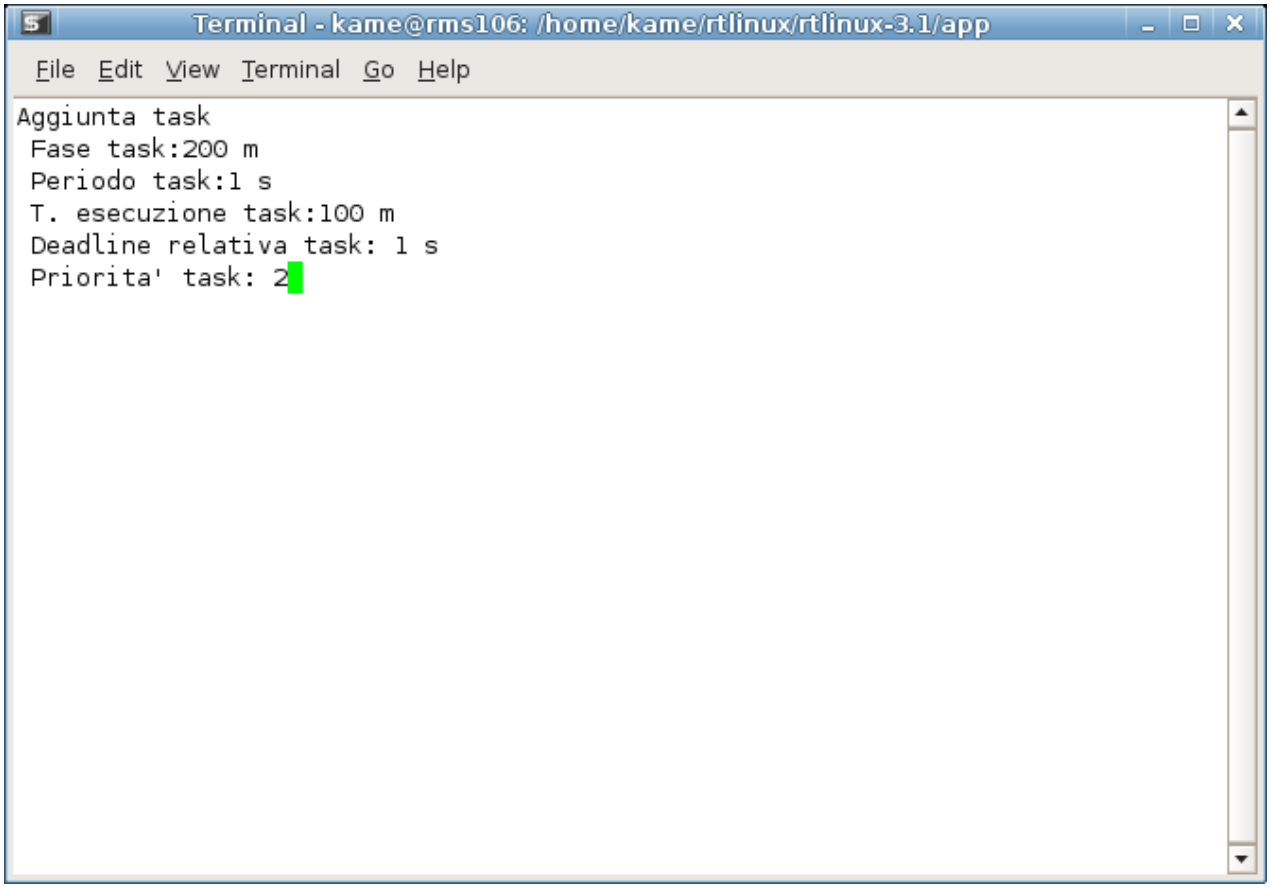

Figura 4.2: Screenshot del programma *sendcmd* in esecuzione in un terminale grafico.

4.2.4 Esempi di utilizzo

Mediante l'applicazione sviluppata è possibile verificare molte delle peculiarità di RTLinux, oltre che simulare insiemi di task aventi i parametri temporali voluti. Si presentano qui cinque possibili utilizzi.

4.2.4.1 Interrupt del timer

Grazie all'inserimento delle operazioni di log (che, evidentemente ed inevitabilmente, modificano leggermente i tempi di esecuzione) è possibile verificare le considerazioni fatte sullo scheduler. In particolare si è visto che qualora non vi siano task che potrebbero effettuare preemption sul task che entrerà in esecuzione all'uscita dello scheduler, in un sistema a singolo processore il timer 8254 viene programmato per generare un'interruzione dopo 5 millisecondi. Il caso più semplice che si possa analizzare è quello di avere il solo task di default, Linux. Caricando i moduli necessari, *eventlog.o*,

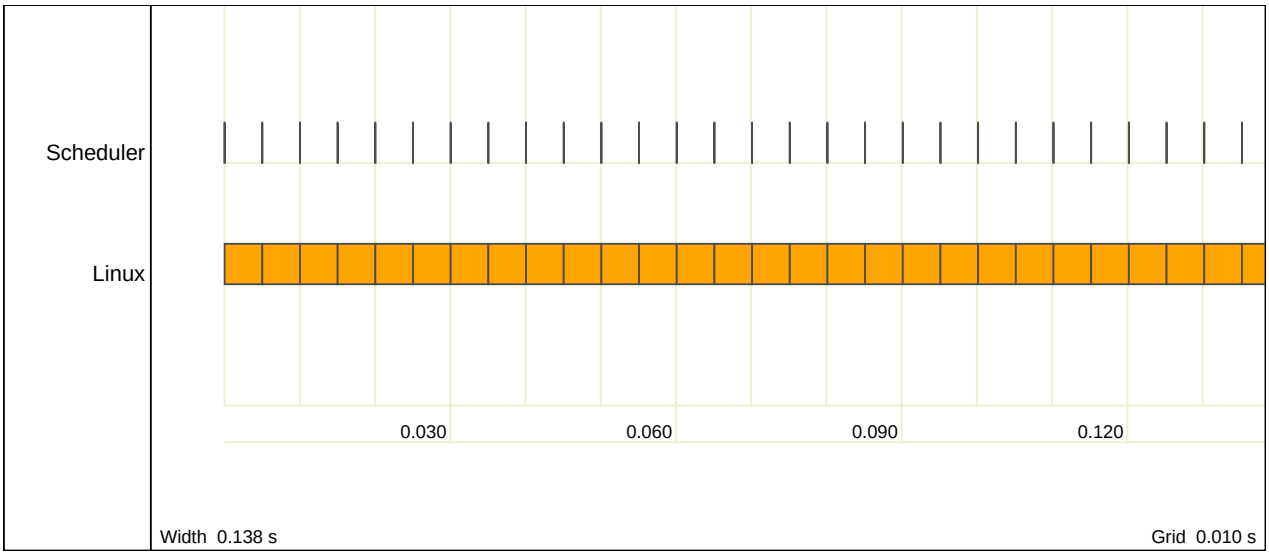

Figura 4.3: Invocazioni dello scheduler nel caso non sia presente alcun task real-time.

tasktest.o, fifotest.o e lo scheduler modificato (con l'aggiunta delle operazioni di log) *rtl_sched2.o* si è ottenuto mediante *printevevent* il file per *kiwi* visualizzato in figura 4.3.

Per rendere il grafico esteticamente migliore con un editor di testo si sono aggiunte le righe

```
DECIMAL_DIGITS 3
LINE_NAME 0 Scheduler
LINE_NAME 1 Linux
```

e si è eseguita una sostituzione automatica delle stringe "EXEC-B 20", "EXEC-B 21" "EXEC-E 20" ed "EXEC-E 21" rispettivamente con "EXEC-B 0", "EXEC-B 1" "EXEC-E 0" ed "EXEC-E 1" per portare le due tracce nella parte superiore del disegno. Nella figura appena citata si può osservare come effettivamente lo scheduler venga invocato con una cadenza di circa 5 ms. Aumentando lo zoom è possibile andare ad osservare per quanto tempo lo scheduler entra in esecuzione, ed un esempio di ciò che si può vedere (in questo caso, si è portato *DECIMAL_DIGITS* a 9) è riportato in figura 4.4.

Con i comandi

grep "EXEC-B 0" scheduler.txt | cut -d' ' -f 1 >inizi.txt

e

grep "EXEC-E 0" scheduler.txt | cut -d' ' -f 1 >terminazioni.txt

si sono poi salvati in due file distinti gli istanti relativi e all'uscita dallo scheduler. Questo ha consentito di importare tali valori in *octave*, un programma open source simile a *MATLAB,* e determinare rapidamente le caratteristiche del tempo tra due invocazioni dello scheduler e del suo tempo di esecuzione. In particolare sui primi 4800 campioni raccolti (in realtà su 4801 per il calcolo dell'ultimo tempo tra due invocazioni successivi) si sono determinati i seguenti tempi, espressi in nanosecondi

	min	t. medio	max
tempo tra due invocazioni successive	5007104	5009226	5019424
tempo di esecuzione delle scheduler	3904	4968	11152

Sempre con *octave* è stato possibile generare il grafico di figura 4.5 che illustra la distribuzione dei tempi di esecuzione e quello di figura 4.6 che mostra l'andamento della distribuzione dei tempi trascorsi tra due invocazioni successive dello scheduler.

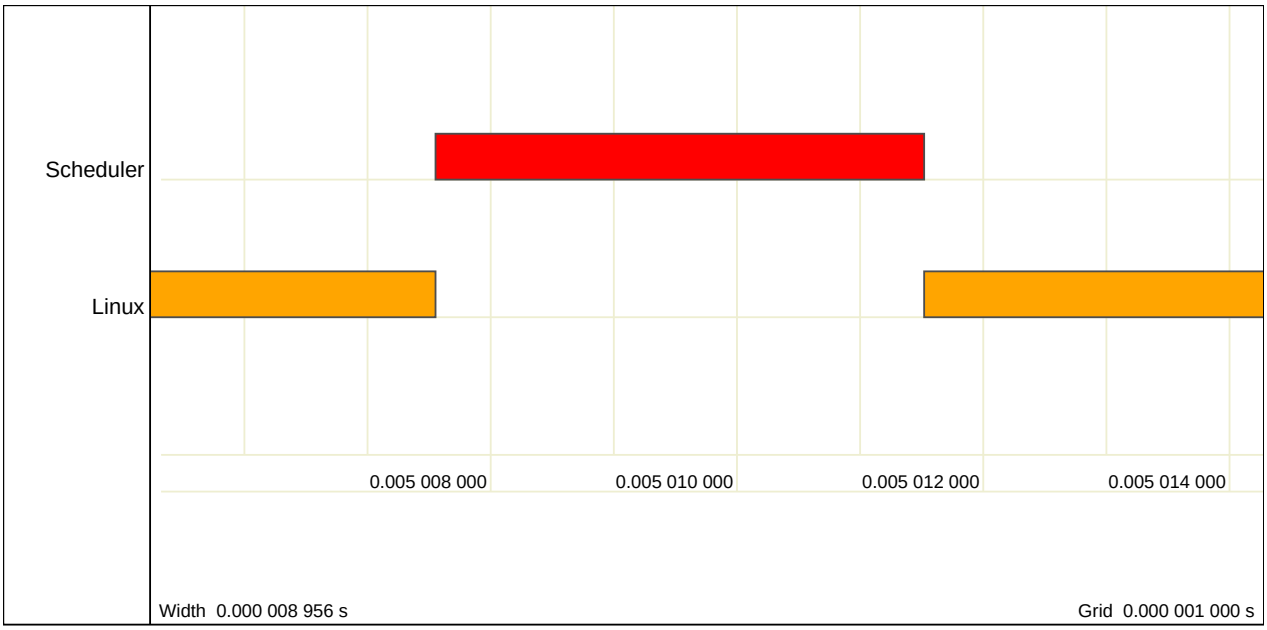

Figura 4.4: Zoom che permette di osservare l'entrata in esecuzione e la terminazione dello scheduler.

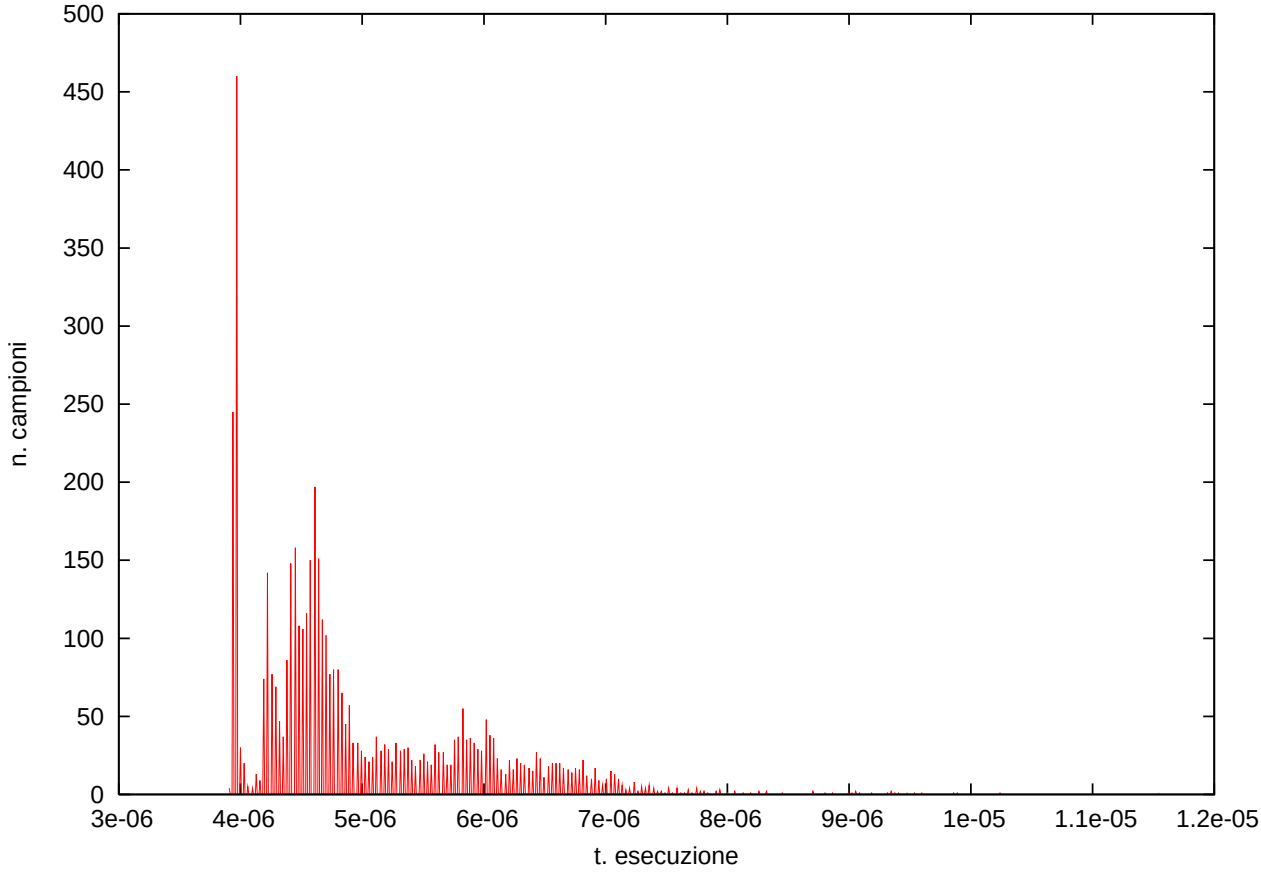

Figura 4.5: Andamento della distribuzione dei tempi di esecuzione. Il grafico è stato ottenuto mediante il comando *hist* con il valore 1000 come numero di bin; ciò corrisponde a suddividere in 1000 parti di uguale lunghezza l'intervallo su cui spaziano i tempi di esecuzione ed a contare il numero di campioni che cade in ciascun intervallo.

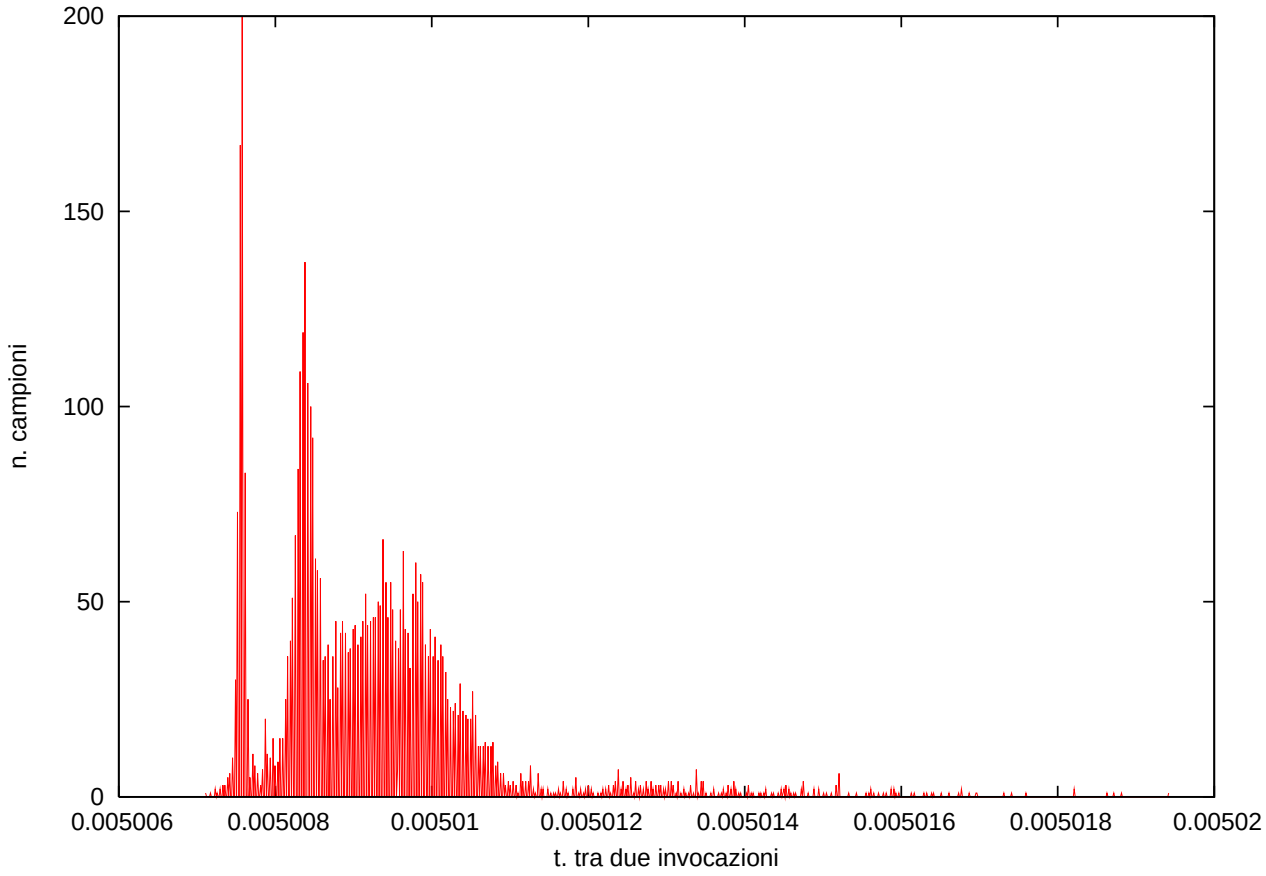

Figura 4.6: Andamento della distribuzione dei tempi intercorsi tra due invocazioni dello scheduler, ottenuto con impostazioni analoghe a quelle della figura precedente.

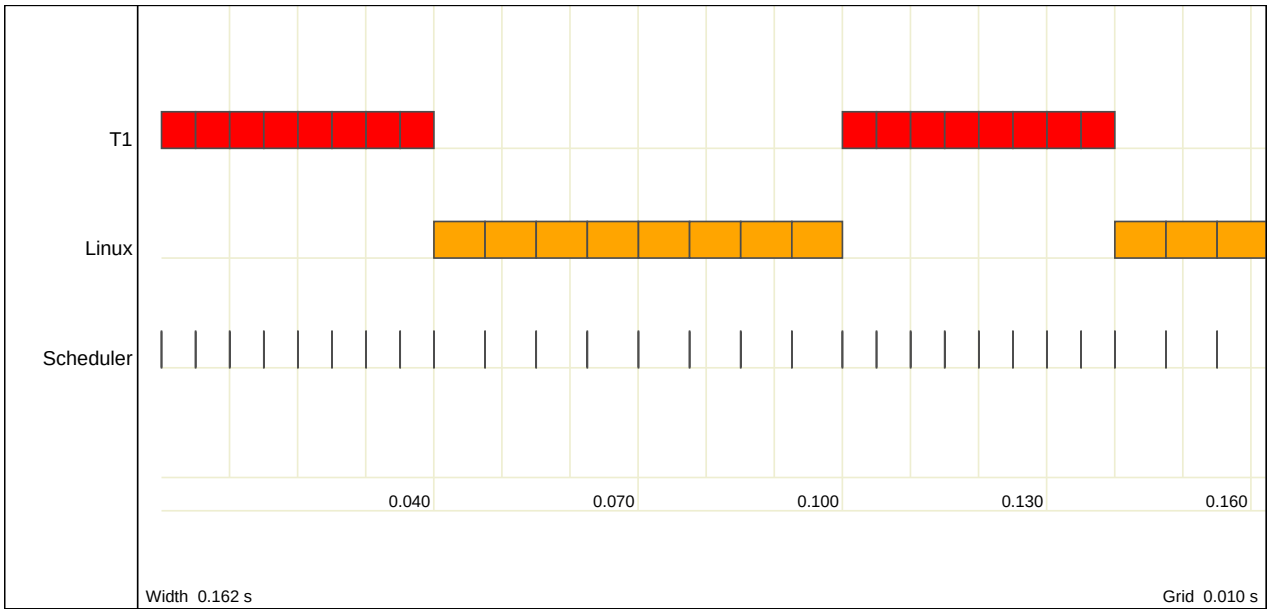

Figura 4.7: Andamento temporale del task $T_1 = (100, 40)$.

Mediante l'utilizzo dell'interfaccia realizzata *sendcmd* si è poi creato un task T_1 a fase nulla di periodo 100 ms e con tempo di esecuzione 40 ms (e priorità 1), ossia il task $T_1 = (100, 40)$ secondo la notazione usata in [1], qualora si assuma che l'unità temporale sia il millisecondo. L'andamento temporale ottenuto è quello di figura 4.7, e si può osservare come Linux venga interrotto circa ogni 7.5 ms, confermando le osservazioni fatte nell'analisi dello scheduler. Infatti visto che quando si manda in esecuzione Linux la funzione *find_preemptor* ritorna T_1 e, visto che la distanza dal tempo di rilascio del job di T_1 è sempre superiore (tranne che per l'ultima invocazione prima del rilascio dei job di T_1) a 7.5ms, il controllo presente nella funzione d'impostazione del timeout del timer pone sempre l'interrupt successivo *MAX_LATCH_ONESHOT*=7.5 ms dopo. Al contrario, quando si manda in esecuzione T_1 non vi è alcun task che potrebbe operare preemption su di esso quindi il tempo tra due invocazioni successive dello scheduler è di circa 5 millisecondi.

4.2.4.2 Task alla stessa di priorità

Un'altro dei test che si può effettuare consiste nell'avviare due task alla stessa priorità e verificare che il comportamento sia quello previsto. Ad esempio se si crea prima il task $T_1 = (100, 40)$ e poi il task $T_2 = (\Phi = 20, 100, 20)$, si ottiene, assumendo la stessa unità temporale precedentemente utilizzata (1 ms), l'andamento riportato in figura 4.8. Come si può notare il secondo task al momento del rilascio dei suoi job effettua preemption sul primo, perché, come spiegato nella sezione di analisi dello scheduler, è inserito in testa nella lista dei task. Viceversa se si aggiungono i task in ordine inverso, ossia $T_1 = (20, 100, 20)$ e $T_2 = (100, 40)$, la preemption non avviene, come conferma la figura 4.9. Per un motivo puramente estetico anche in questo caso si sono elaborati i file prodotti da *printevent*, rimuovendo le informazioni relative allo scheduler ed a Linux con il comando

cat nomefile | grep -v "EXEC-[BE] 2[01]" >nomefile2

4.2.4.3 Eliminazione di job

Una delle peculiarità di RTLinux è certamente quella, quando si calcola il tempo di rilascio del prossimo job, di ignorare i job che hanno un tempo di rilascio nel passato, ed il modo in cui viene gestito il rilascio di job, mediante il flag *RTL_THREAD_TIMERARMED* e la memorizzazione del

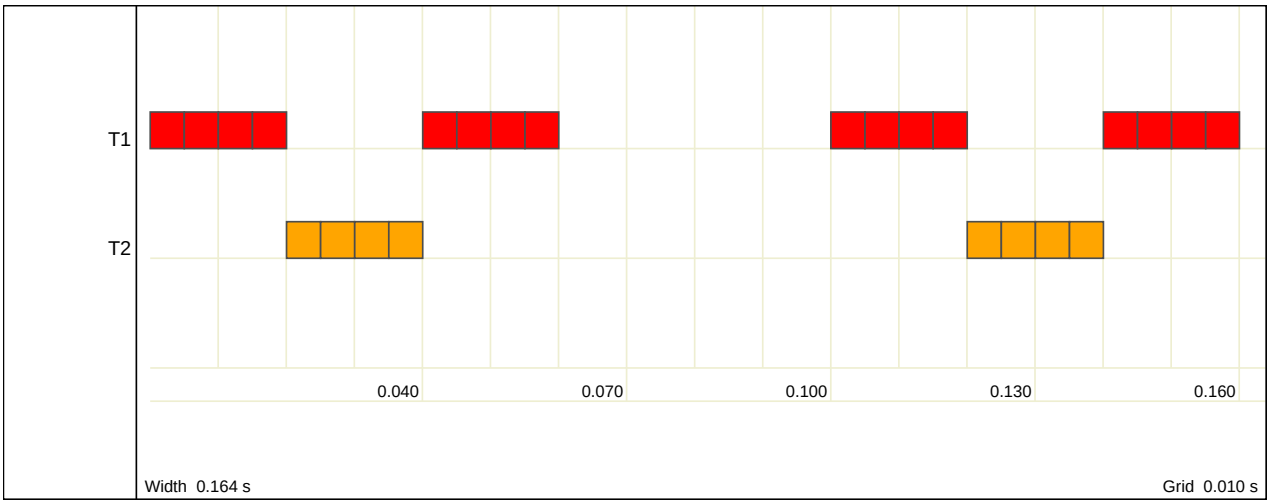

Figura 4.8: Andamento temporale dei task $T_1 = (100, 40)$ e $T_2 = (20, 100, 20)$ aventi la stessa priorità.

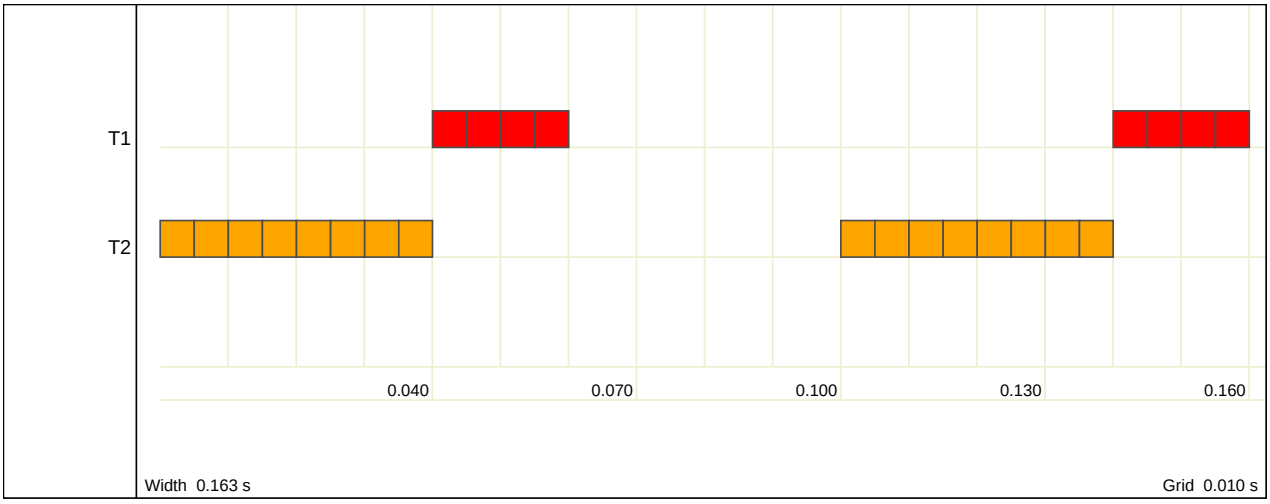

Figura 4.9: Andamento temporale dei task $T_1 = (20, 100, 20)$ e $T_2 = (100, 40)$ aventi la stessa priorità.

tempo di rilascio del prossimo job, può non essere del tutto intuitivo. Un esempio può chiarire questo fatto. Si supponga di avere due task periodici $T_1 = (\Phi = 3, 24, 6)$ e $T_2 = (4, 2, D = 9)$ con T_1 avente priorità maggiore di T_2. I risultati della simulazione ottenuta utilizzando le applicazioni sviluppate e supponendo che un unità di tempo sia pari a 100 ms sono riportati in figura 4.10. I risultati confermano l'analisi dello scheduler infatti:

- Al tempo 0, poiché il flag *RTL_THREAD_TIMERARMED* risulta attivo (la creazione dei task prevede una chiamata *pthread_make_periodic_np*), viene rilasciato il primo job di T_2, $J_{2,1}$ usando la notazione di [1], viene impostato il tempo di rilascio del prossimo job a 4 ed il flag viene posto a zero.
- Al tempo 2 il job termina e viene invocata *pthread_wait_np,* che ha come effetto quello di porre a 1 il flag *RTL_THREAD_TIMERARMED*
- Al tempo 3 entra in esecuzione il primo job di T_1, $J_{1,1}$
- Al tempo 4 viene rilasciato $J_{2,2}$, il tempo di rilascio del prossimo job viene portato a $4 + 4 = 8$ e si porta a zero *RTL_THREAD_TIMERARMED*
- Al tempo 8 teoricamente viene rilasciato $J_{2,3}$, ma in RTLinux, essendo *RTL_THREAD_-TIMERARMED* disattivato, non succede nulla
- Al tempo 9 $J_{1,1}$ termina (attivando il suo flag *RTL_THREAD_TIMERARMED*, ma ai fini dell'analisi ciò che interessa è solo lo stato del flag di T_2) e $J_{2,2}$ viene avviato
- Al tempo 11 $J_{2,2}$ termina ed invoca *pthread_wait_np,* con conseguente attivazione di *RTL_TH-READ_TIMERARMED*; all'entrata nello scheduler il tempo è superiore ad 8, tempo di rilascio di $J_{2,3}$, quindi si avvia questo job, si porta il tempo di rilascio del prossimo job a 12 e si effettua il reset del flag *RTL_THREAD_TIMERARMED.*
- Al tempo 12 teoricamente si ha il rilascio di $J_{2,4}$ ma a causa della disattivazione di *RTL_THREAD_TIMERARMED* non succede nulla
- Al tempo 13 $J_{2,3}$ finisce la sua esecuzione ed invoca *pthread_wait_np* attivando così *RTL_TH-READ_TIMERARMED*; questo permette, visto che il tempo di rilascio del prossimo job era stato posto a 12, di avviare il relativo job $J_{2,4}$, di portare la variabile *resume_time* (tempo di rilascio del prossimo job) a 16 e di disattivare nuovamente *RTL_THREAD_TIMERARMED*
- Al tempo 15 anche $J_{2,4}$ termina, con conseguente riattivazione del flag
- Al tempo 16 $J_{2,5}$ inizia la sua esecuzione e tutto torna a procedere normalmente.

Si supponga ora di allungare ad 8 unità il tempo di esecuzione di T_1. Utilizzando il normale modello a task periodici ed una schedulazione a priorità fissa, con T_1 più prioritario di T_2, la situazione, adottando sempre 100 ms come unità temporale, sarebbe quella presentata in figura 4.11, dove i job di T_2 rilasciati ai tempi 4 e 8, durante l'esecuzione del job di T_1 si accumulano, andando a ritardare l'esecuzione dei job successivi, rilasciati agli istanti 12 e 16. Se si eseguono tali task su RTLinux si ottiene l'andamento riportato in figura 4.12. Un'analisi simile alla precedente si rivela in questo caso molto più interessante:

- Per i tempi 0,2,3,4, 8 la situazione è analoga al caso già esaminato.
- Al tempo 11 il task T_1 termina la sua esecuzione, ed il job $J_{2,2}$ inizia la sua esecuzione.
- Al tempo 13 $J_{2,2}$ termina ed invocando *pthread_wait_np* riattiva *RTL_THREAD_TIMERARMED.* All'entrata nello scheduler viene rilasciato il job $J_{2,3}$ (visto che il tempo corrente è maggiore del suo tempo di rilascio teorico) ed il tempo di rilascio del prossimo job viene portato a 12. Essendo tale valore, corrispondente al rilascio del teorico $J_{2,4}$, nel passato rispetto al momento

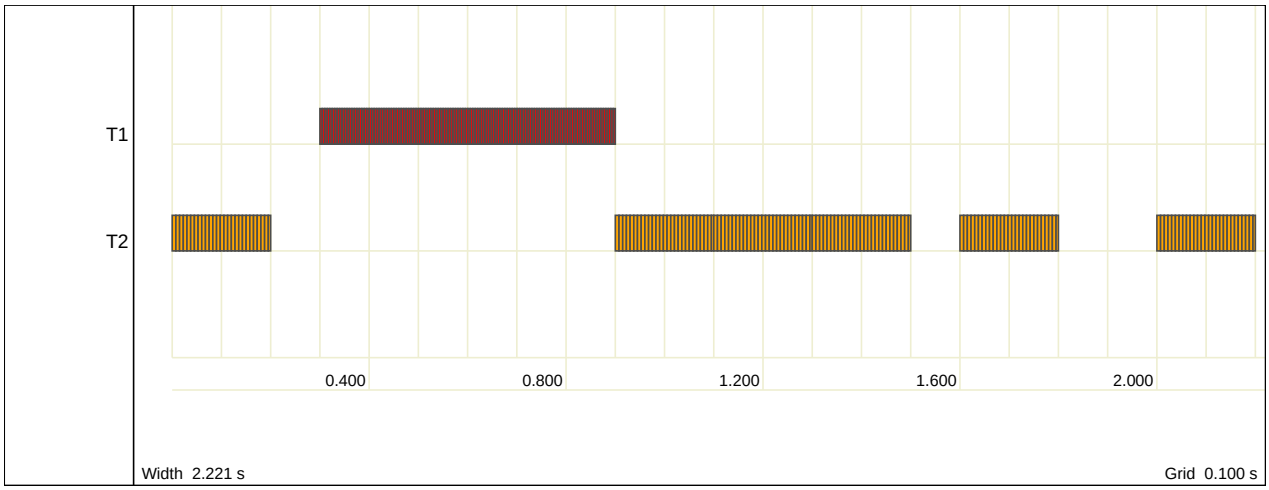

Figura 4.10: Andamento temporale dei task $T_1 = (\Phi = 3, 24, 6)$ e $T_2 = (0, 4, 2, 9)$ in RTLinux.

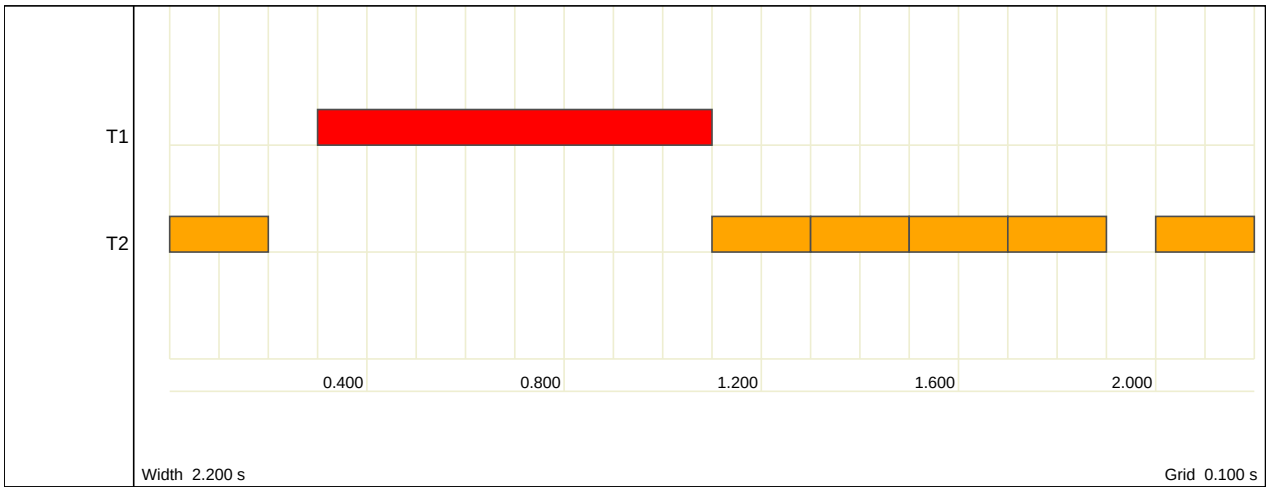

Figura 4.11: Andamento temporale dei task $T_1 = (\Phi = 3, 24, 8)$ e $T_2 = (0, 4, 2, 9)$ nel caso del modello a task periodici. Il disegno è stato prodotto scrivendo manualmente un file per *kiwi* quindi non presenta le varie interruzioni dovute agli interventi dello scheduler.

corrente si aggiunge un nuovo periodo ed il tempo di rilascio viene portato a 16; questo è il tempo di rilascio di ciò che convenzionalmente sarebbe stato $J_{2,5}$ e l'operazione compiuta può quindi concettualmente essere vista come un'eliminazione di $J_{2,4}$. Infine, come di consueto, *RTL_THREAD_TIMERARMED* viene resettato.

- Al tempo 15 $J_{2,3}$ termina la sua esecuzione, riattivando *RTL_THREAD_TIMERARMED*.
- Al tempo 16 $J_{2,5}$ viene rilasciato e l'esecuzione riprende normalmente.

4.2.4.4 Esercizio 6.16 del libro di testo

Il primo punto dell'esercizio chiede se i task periodici {(7,10,1,10), (12,2), (25,9)} di utilizzazione totale 0.63 siano schedulabili dall'algoritmo rate-monotonic (in realtà l'utilizzazione massima è leggermente inferiore, e pari a $\frac{1}{10} + \frac{2}{12} + \frac{9}{25} = \frac{47}{75} = 0.62\bar{6}$). Il teorema 6.11 a pagina 146 di [1] afferma che n task periodici indipendenti, preemptable e con deadline relative uguali ai periodi sono schedulabili dall'algoritmo RM se la loro utilizzazione totale U è inferiore a

$$U_{RM}(n) = n(2^{\frac{1}{n}} - 1)$$

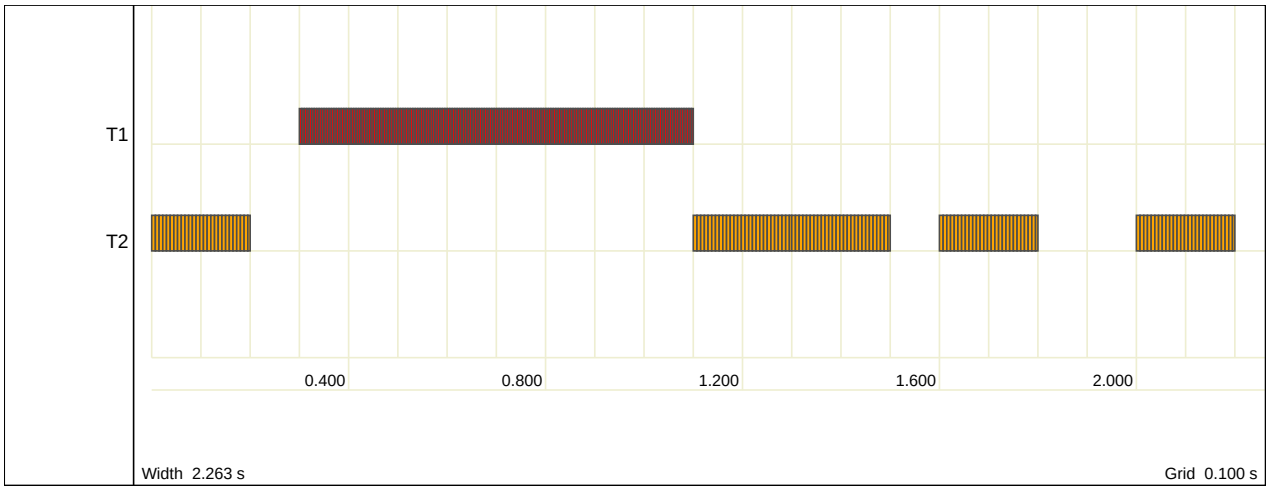

Figura 4.12: Andamento temporale dei task $T_1 = (\Phi = 3, 24, 8)$ e $T_2 = (0, 4, 2, 9)$ in RTLinux.

Questo teorema costituisce una condizione sufficiente per dare risposta affermativa alla domanda, visto che $U_{RM}(3) = 3(\sqrt[3]{2} - 1) \simeq 0.78$. Ci si potrebbe chiedere se il fatto che la fase del primo task sia non nulla costituisca un problema, ma il teorema non riporta alcuna ipotesi sulle fasi dato che la dimostrazione viene fatta nel caso pessimo, corrispondente a task in fase per il teorema 6.5. L'andamento temporale dei task, simulato con l'applicazione sviluppata, è riportato in figura 4.13.

Il secondo punto chiede se i task {(7,10,1,10), (12,6), (25,9)} di utilizzazione totale 0.96 siano schedulabili da RM. In questo caso l'ipotesi del teorema precedente non è soddisfatta e risulta quindi necessario ricorrere ad altri metodi, ad esempio la time demand analisys. Questo algoritmo determina il massimo tempo di risposta dei job di ciascun task nel caso peggiore, costituito da task tutti in fase, come afferma il già citato teorema 6.5. Quindi nel caso le fasi non siano tutte nulle la time demand analisys fornisce una condizione sufficiente ma non necessaria; si noti comunque che se il jitter del tempo di rilascio dei job non è trascurabile la condizione diventa anche necessaria perché non si può escludere che avvenga il caso considerato dalla time demand analysis in cui ogni job viene rilasciato nello stesso momento di tutti quelli a priorità maggiore.

Dato che le deadline relative sono (inferiori o) uguali ai periodi è sufficiente considerare il primo job di ogni task rilasciato in un istante critico. Per il task a priorità più alta non vi sono ovviamente problemi di schedulabilità e

$$w_{1,1} = e_1 = 1$$

Per verificare la schedulabilità di T_2 è necessario controllare se il valore della time demand function

$$w_{2,1}(t) = e_2 + \left\lceil \frac{t}{p_1} \right\rceil e_1$$

risulta inferiore o uguale a t prima della deadline di $J_{2,1}$ in uno degli istanti di rilascio (ad eccezione dell'istante critico) dei job di task a priorità maggiore o uguale, ossia negli istanti $t = jp_k$ con k pari a 1 o 2 e j numero intero che varia da 1 a $\left\lfloor \frac{D_2}{p_k} \right\rfloor$. Analizzando il primo caso, $k = 1$ e $j = 1$, si ottiene subito conferma della schedulabilità di T_2, infatti $t = 10$ e

$$w_{2,1}(t) = e_2 + \left\lceil \frac{t}{p_1} \right\rceil e_1 = 6 + \left\lceil \frac{10}{10} \right\rceil 1 = 7 \leq 10$$

L'analisi della schedulabilità di T_3, analogamente al caso precedente, prevede la verifica dei valori della funzione

$$w_{3,1}(t) = e_3 + \left\lceil \frac{t}{p_1} \right\rceil e_1 + \left\lceil \frac{t}{p_2} \right\rceil e_2$$

negli istanti di tempo $t = jp_k$ con k variabile (sugli interi) tra 1 e 3 e con j numero intero che varia da 1 a $\left\lfloor \frac{D_3}{p_k} \right\rfloor$.

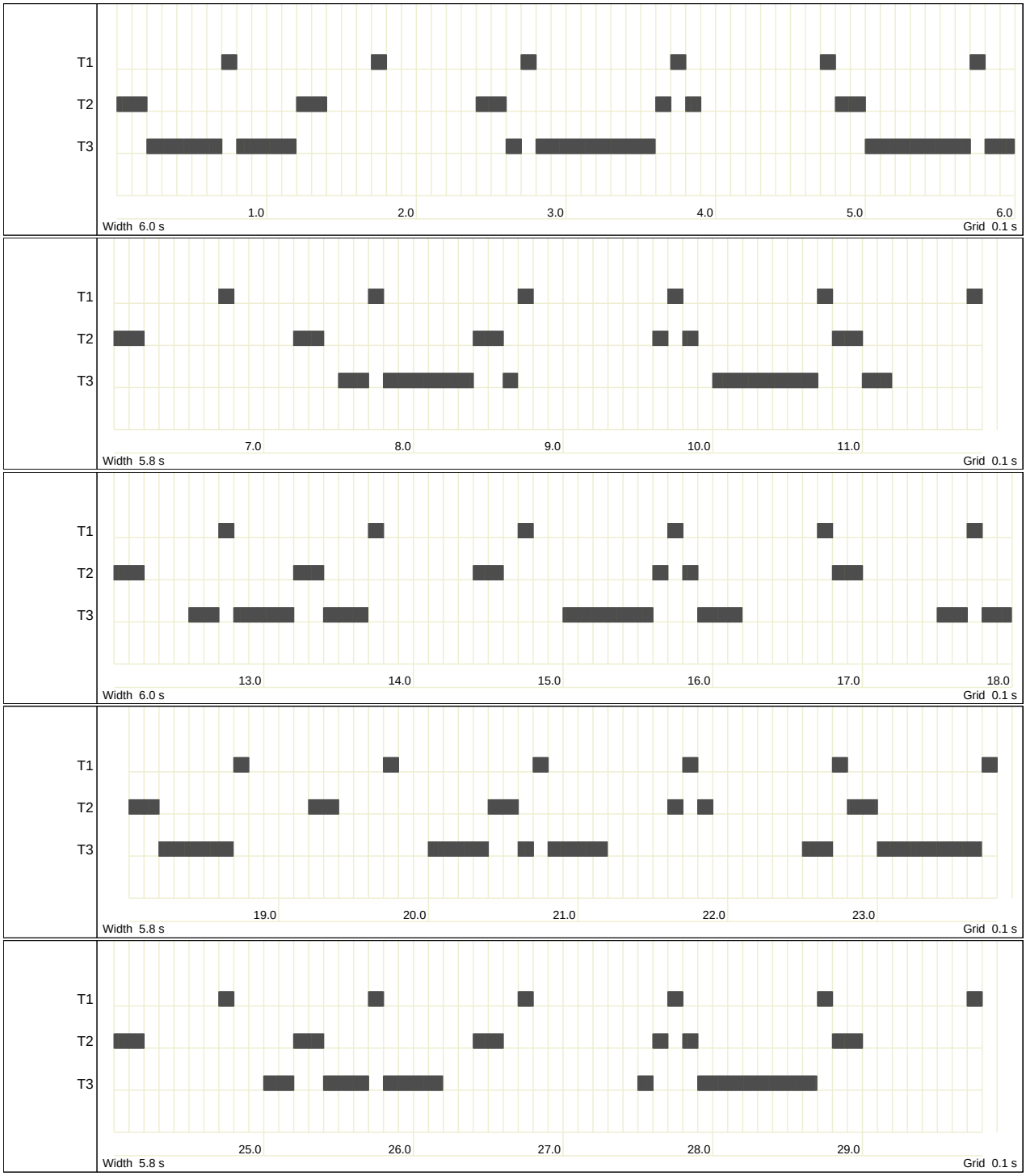

Figura 4.13: Primo iperperiodo della schedulazione dei task $T_1 = (7, 10, 1, 10)$, $T_2 = (12, 2)$ e $T_3 = (25, 9)$ secondo RM assumendo 100 ms come unità di misura. Nonostante la fase sia non nulla la situazione si ripresenta identica nel secondo iperperiodo visto che non vi sono job che terminano oltre la fine del primo iperperiodo.

Per $k = 1$ j può variare da 1 a $\left\lfloor \frac{D_3}{p_1} \right\rfloor = \lfloor \frac{25}{10} \rfloor = 2$. Quando $j = 1$ si ha $t = 10$ e

$$w_{3,1}(t) = e_3 + \left\lceil \frac{t}{p_1} \right\rceil e_1 + \left\lceil \frac{t}{p_2} \right\rceil e_2 = 9 + \left\lceil \frac{10}{10} \right\rceil 1 + \left\lceil \frac{10}{12} \right\rceil 6 = 16 > 10$$

mentre per $j = 2$ si ha $t = 20$ e

$$w_{3,1}(t) = e_3 + \left\lceil \frac{t}{p_1} \right\rceil e_1 + \left\lceil \frac{t}{p_2} \right\rceil e_2 = 9 + \left\lceil \frac{20}{10} \right\rceil 1 + \left\lceil \frac{20}{12} \right\rceil 6 = 23 > 20$$

.

Anche per $k = 2$ j può variare da 1 a $\left\lfloor \frac{D_3}{p_2} \right\rfloor = \lfloor \frac{25}{12} \rfloor = 2$. Per $j = 1$ si ha $t = 12$ e

$$w_{3,1}(t) = e_3 + \left\lceil \frac{t}{p_1} \right\rceil e_1 + \left\lceil \frac{t}{p_2} \right\rceil e_2 = 9 + \left\lceil \frac{12}{10} \right\rceil 1 + \left\lceil \frac{12}{12} \right\rceil 6 = 17 > 12$$

Analizzando il caso $k = 2$ e $j = 2$, ossia $t = 24$, è invece possibile asserire che anche T_3 risulta schedulabile visto che

$$w_{3,1}(t) = e_3 + \left\lceil \frac{t}{p_1} \right\rceil e_1 + \left\lceil \frac{t}{p_2} \right\rceil e_2 = 9 + \left\lceil \frac{24}{10} \right\rceil 1 + \left\lceil \frac{24}{12} \right\rceil 6 = 24 \leq 24$$

L'andamento delle tre funzioni $w_1(t)$, $w_2(t)$ e $w_3(t)$ è riportato in figura 4.14

dove è possibile verificare come ciascuna funzione intersechi la bisettrice prima o alla relativa deadline. La rappresentazione dell'andamento temporale dei tre task ottenuta mediante l'utilizzo dell'applicazione sviluppata è riportata in figura 4.15. Per migliorare la comprensibilità sono state aggiunte le indicazioni relative al tempo di rilascio di ciascun job (coincidenti con le deadline del job precedente, se non si tratta del primo job) mediante delle frecce blu, accanto alle quali viene riportato il numero d'ordine del job; con dei triangolini verdi si è invece indicato l'istante di completamento di ciascun job. Ciò è stato fatto utilizzando il seguente script bash

```
echo >add.txt
n=1
for r1 in ‘seq 0.7 1 60‘;
do echo $r1 ARROWUP 0 $n>>add.txt;let n=$n+1;done
n=1
for r2 in ‘seq 0 1.2 60‘;
do echo $r2 ARROWUP 1  $n>>add.txt;let n=$n+1;done
n=1
for r3 in ‘seq 0 2.5 60‘;
do echo $r3 ARROWUP 2 $n>>add.txt;let n=$n+1;done
n=1
grep "Terminazione␣job" nomefilein -A 2|grep EXEC | sed s/EXEC-E/STOP/>>add
   .txt
cat add.txt nomefilein | sort -g>nomefileout
```

Il terzo punto dell'esercizio chiede di determinare quale sia la minima riduzione del tempo e_2 necessaria a consentire la schedulabilità dei tre task qualora il primo ed il terzo task vengano modificati in $T_1 = (7, 10, 2, 10)$ e $T_3 = (25, 9, 20)$. Una stima per difetto del tempo di esecuzione (ossia una stima per eccesso della rispettiva riduzione) non può essere ottenuta mediante $e_2 = p_2(U_{RM}(3) - U_1 - U_3)$ perché l'ipotesi di deadline relative pari ai periodi non è verificata per T_3. Risulta tuttavia possibile fornire una tale stima impiegando ancora una volta la time demand analisys.

La schedulabilità di $J_{1,1}$ non comporta alcun vincolo, visto che ancora una volta il suo tempo di risposta è pari al suo tempo di esecuzione. Per quanto riguarda $J_{2,1}$ (anche il questo caso l'avere deadline relative inferiori ai periodi consente di analizzare solo il primo job di ogni task) deve valere $w_{2,1}(t) = e_2 + \left\lceil \frac{t}{p_1} \right\rceil e_1 \leq t$ per almeno uno dei tempi $t = jp_k$, con j e k soddisfacenti le condizioni viste prima. In particolare sia per $k = 1$ che per $k = 2$ j può valere solo 1 visto che, rispettivamente,

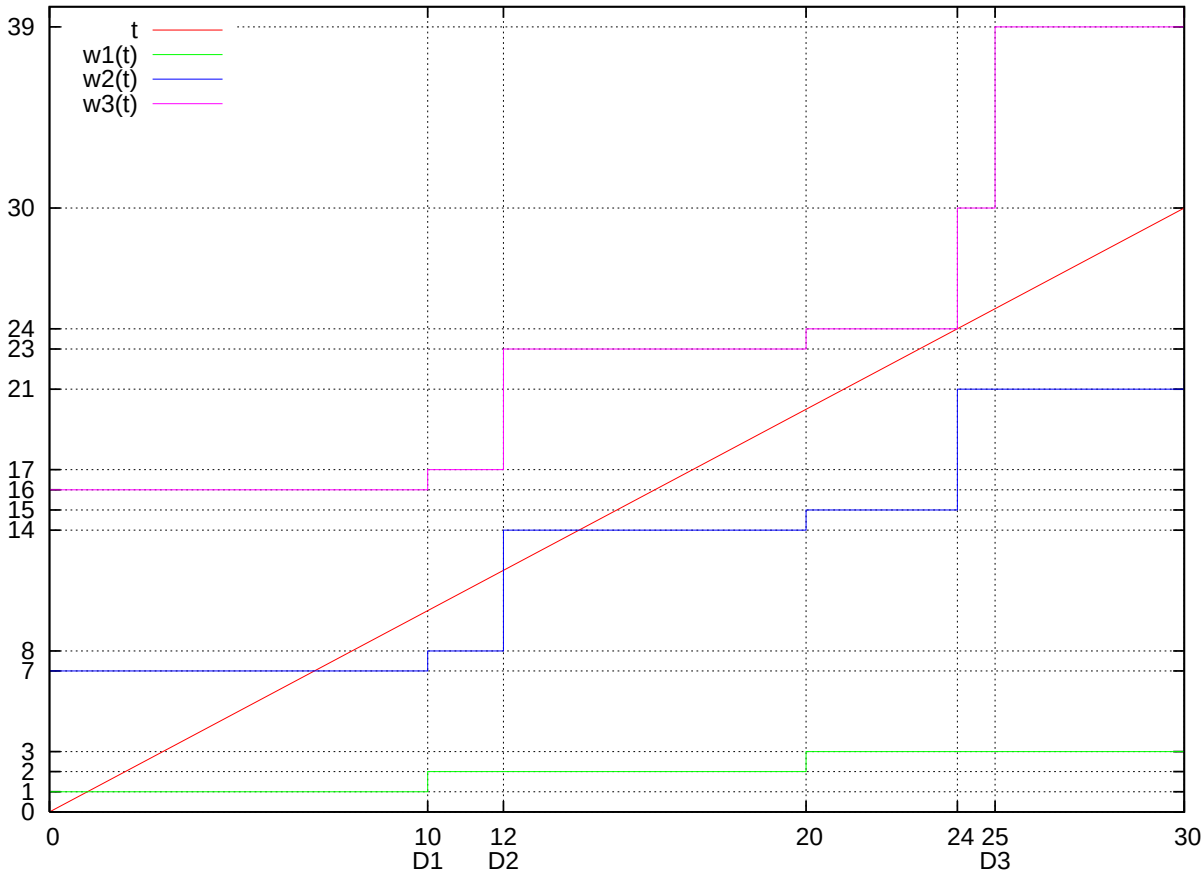

Figura 4.14: Andamento delle funzioni $w_1(t)$, $w_2(t)$ e $w_3(t)$.

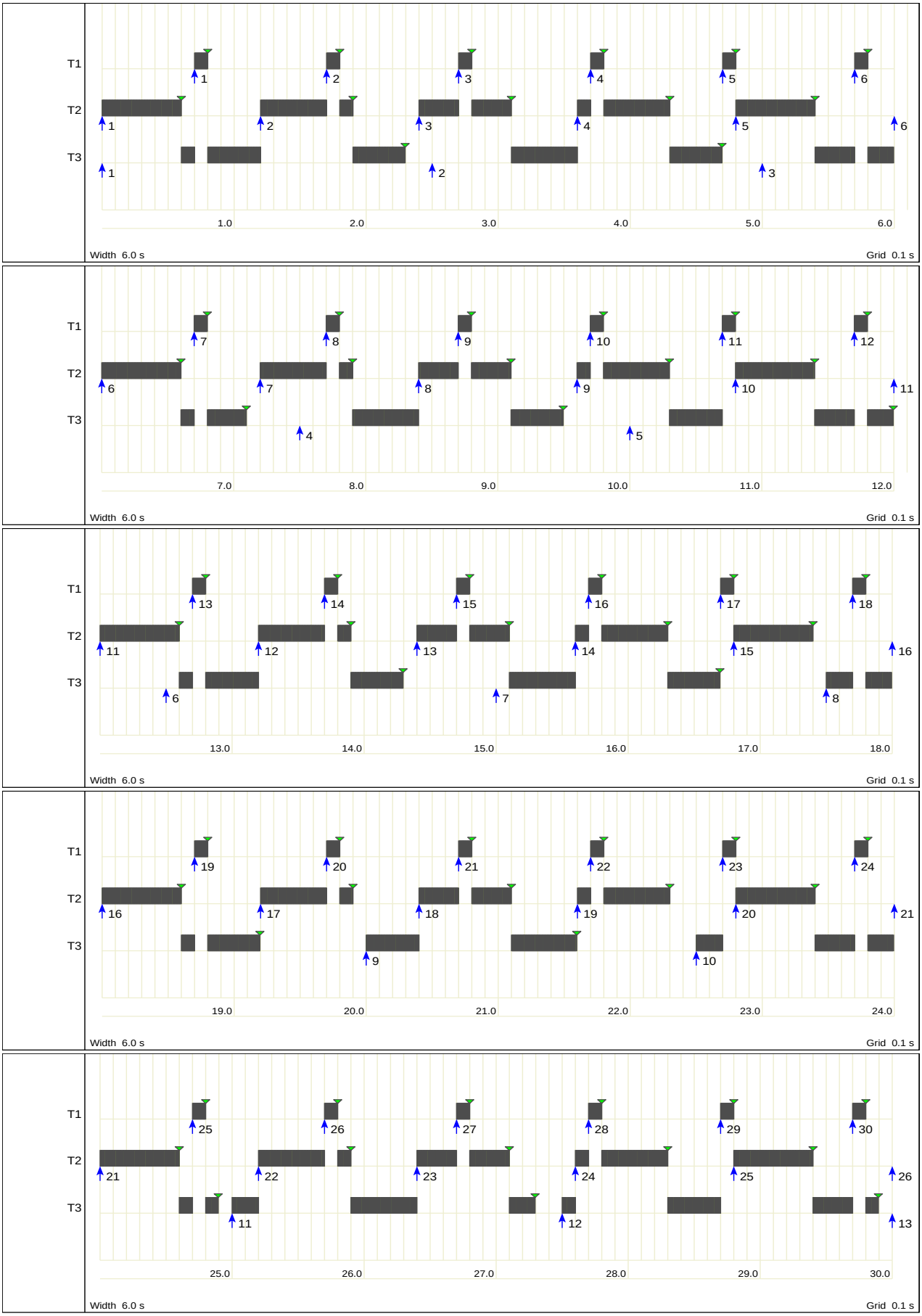

Figura 4.15: Andamento temporale dei task $T_1 = (\Phi = 7, 10, 1)$, $T_2 = (12, 6)$ e $T_3 = (25, 9)$ schedulati secondo RM (con unità temporale pari a 100 ms). Anche in questo caso non è necessario considerare il secondo iperperiodo perché tutti i job del primo terminano prima dell'inizio del secondo.

$\left\lfloor \frac{D_2}{p_1} \right\rfloor = \lfloor \frac{12}{10} \rfloor = 1$ e $\left\lfloor \frac{D_2}{p_2} \right\rfloor = \lfloor \frac{12}{12} \rfloor = 1$; si deve dunque avere

$w_{2,1}(10) = e_2 + \left\lceil \frac{10}{p_1} \right\rceil e_1 \leq 10$ o $w_{2,1}(12) = e_2 + \left\lceil \frac{12}{p_1} \right\rceil e_1 \leq 12$

sostituendo i valori

$w_{2,1}(12) = e_2 + \lceil \frac{10}{10} \rceil 2 \leq 10$ o $w_{2,1}(12) = e_2 + \lceil \frac{12}{10} \rceil 2 \leq 12$

quindi

$$e_2 \leq 8$$

I tempi da considerare per la schedulabilità di $J_{3,1}$ sono per $k = 1$ i tempi jp_1 con j che può variare da 1 a $\left\lfloor \frac{D_3}{p_1} \right\rfloor = \lfloor \frac{20}{10} \rfloor = 2$. Quando $k = 2$ j può invece valere solo 1 dato che $\left\lfloor \frac{D_3}{p_2} \right\rfloor = \lfloor \frac{20}{12} \rfloor = 1$. Poiché la deadline relativa di T_3 è inferiore al periodo, considerando il caso $k = 3$ si ottiene che il massimo valore assumibile da j è $\left\lfloor \frac{D_3}{p_3} \right\rfloor = \lfloor \frac{20}{25} \rfloor = 0$, ed infatti il rilascio del secondo job secondo di T_3 avviene in un istante posteriore all'intervallo di interesse; per considerare casi in cui la bisettrice interseca la funzione in un istante

- precedente o pari alla deadline
- successivo ai tempi di rilascio di tutti i job precedenti tale deadline

è necessario aggiungere al controllo l'istante della deadline. La figura 4.16 fornisce la rappresentazione grafica di tale situazione. In questo caso, comunque, la deadline, pari a 20, coincide con il rilascio del terzo job di T_1. In sintesi i tempi da considerare sono $1p_1 = 10$, $1p_2 = 12$ e $2p_1 = 20$, quindi per garantire la schedulabilità di T_3 si deve imporre

$w_{3,1}(10) = e_3 + \left\lceil \frac{10}{p_1} \right\rceil e_1 + \left\lceil \frac{10}{p_2} \right\rceil e_2 \leq 10$ o $w_{3,1}(12) = e_3 + \left\lceil \frac{12}{p_1} \right\rceil e_1 + \left\lceil \frac{12}{p_2} \right\rceil e_2 \leq 12$ o

$w_{3,1}(20) = e_3 + \left\lceil \frac{20}{p_1} \right\rceil e_1 + \left\lceil \frac{20}{p_2} \right\rceil e_2 \leq 20$

ossia

$w_{3,1}(10) = 9 + \lceil \frac{10}{10} \rceil 2 + \lceil \frac{10}{12} \rceil e_2 \leq 10$ o $w_{3,1}(12) = 9 + \lceil \frac{12}{10} \rceil 2 + \lceil \frac{12}{12} \rceil e_2 \leq 12$ o

$w_{3,1}(20) = 9 + \lceil \frac{20}{10} \rceil 2 + \lceil \frac{20}{12} \rceil e_2 \leq 20$

svolgendo i calcoli

$e_2 \leq -1$ o $e_2 \leq -1$ o $e_2 \leq 3.5$

riassumibili nell'unica condizione

$$e_2 \leq 3.5$$

Condizione sufficiente per la schedulabilità dei tre task è quindi $e_2 \leq 3.5$ ($e_2 \leq 8$ assicura la schedulabilità di T_2 e $e_2 \leq 3.5$ assicura quella di T_3). Analizzando l'andamento temporale dei tre task $T_1 = (7, 10, 2, 10)$, $T_2 = (12, 3.5)$ e $T_3 = (25, 9, 20)$, riportato in figura 4.17 (le frecce rosse indicano le deadline di T_3), si nota come, anche in caso di fase di T_1 non nulla, il primo job di T_3 termina esattamente alla sua deadline; ogni ulteriore incremento di e_2 comporterebbe il non rispetto di tale deadline, quindi la condizione $e_2 \leq 3.5$ oltre che sufficiente è necessaria, e si può quindi affermare che la minima riduzione del tempo di esecuzione del secondo task è pari a 6-3.5=2.5 unità di tempo.

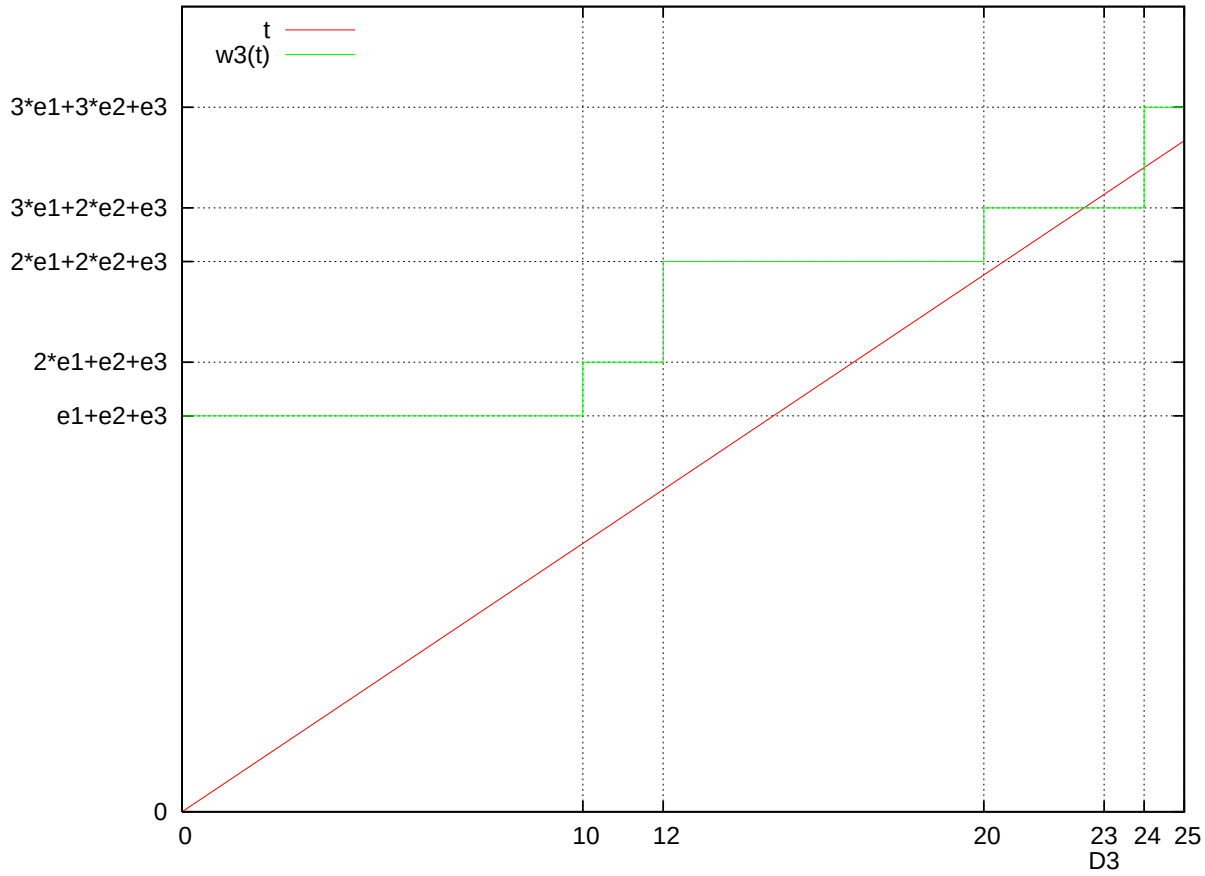

Figura 4.16: Andamento della funzione $w_{3,1}(t)$. Si supponga che la deadline relativa D_3 sia pari a 23. In questo caso considerare solo i tempi jp_k porta al risultato errato che il sistema risulti non schedulabile.

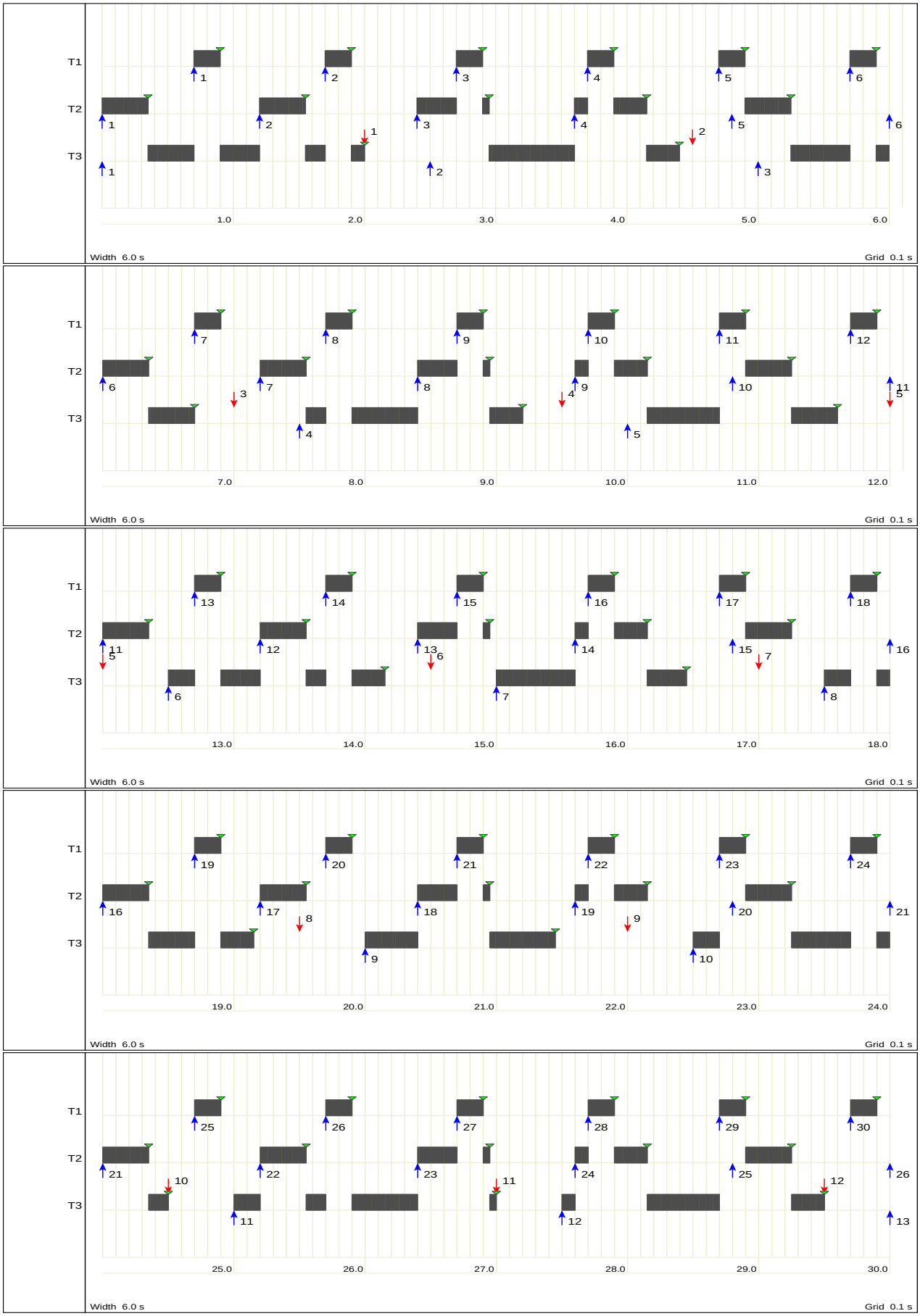

Figura 4.17: Andamento temporale dei task $T_1 = (7, 10, 2, 10)$, $T_2 = (12, 3.5)$ e $T_3 = (25, 9, 20)$ nel primo iperperiodo (unità temporale 100 ms). Ancora una volta è sufficiente tracciare l'andamento di un solo iperperiodo.

4.2.4.5 Verifica della non ottimalità di DM ed RM nel caso $D > p$

Come visto a lezione quando le deadline relative sono minori o uguali ai periodi se un task è schedulabile da un algoritmo a priorità fissa allora può essere schedulato dall'algoritmo Deadline-Monotonic. Quando tale ipotesi non è valida, la tesi non è più vera, ed infatti è possibile costruire dei casi in cui un algoritmo a priorità fissa riesce a schedulare un insieme di task ma DM non garantisce il rispetto delle deadline. Si considerino per semplicità due soli task, T_1 e T_2 e si supponga che siano in fase. Si scelgano e_1, e_2, p_1 e p_2 in modo da soddisfare la disuguaglianza

$$\frac{e_1}{p_1} + \frac{e_2}{p_2} \leq 1$$

Sia

- s_1 il massimo tempo di risposta dei job di T_1 nel caso T_2 sia più prioritario
- s_2 il massimo tempo di risposta dei job di T_2 nel caso T_1 sia più prioritario

Si supponga che s_1 ed s_2 siano diversi, ed in particolare, senza perdita di generalità, che valga $s_1 < s_2$. Risulta allora possibile scegliere D_1e D_2 tali che

$$s_1 \leq D_1 < D_2 < s_2$$

DM conferisce priorità maggiore a T_1 e sotto queste ipotesi il massimo tempo di risposta di T_2 è pari ad s_2, ossia l'insieme di task non risulta schedulabile da DM perché si è scelto $D_2 < s_2$. Viceversa, assumendo T_2 come task più prioritario, la deadline per T_1 verrebbe rispettata, vista la condizione imposta $s_1 \leq D_1$ e, banalmente, la deadline di T_2 sarebbe rispettata in quanto il tempo di risposta risulterebbe pari al tempo di esecuzione.

Una condizione necessaria perché s_1 ed s_2 siano diversi è che valga

$$e_1 + e_2 \geq min\{p_1, p_2\}$$

altrimenti si ha $s_1 = s_2 = e_1 + e_2$, infatti

- considerando un istante critico, esistente sotto le ipotesi di task in fase, $s_1 \geq e_1 + e_2$ e $s_2 \geq e_1 + e_2$;
- visto che i job di entrambi i task rilasciati in un istante critico terminano prima che venga rilasciato un'altro job $s_1 \leq e_1 + e_2$ e $s_2 \leq e_1 + e_2$;

Si noti che la condizione opposta, $e_1 + e_2 \leq min\{p_1, p_2\}$, dovrebbe essere soddisfatta nel caso si volessero aggiungere la condizioni

$$\begin{cases} D_1 \leq p_1 \\ D_2 \leq p_2 \end{cases}$$

ciò dimostra, com'è ovvio che avvenga, come sia impossibile costruire casi di questo tipo sotto le ipotesi del teorema di ottimalità di DM.

Un esempio numerico può essere il seguente. Si prendano i valori $p_1 = 2$, $e_1 = 1$, $p_2 = 7$ ed $e_2 = 3$ che verificano la disuguaglianze $\frac{e_1}{p_1} + \frac{e_2}{p_2} \leq 1$ e $e_1 + e_2 \geq min\{p_1, p_2\}$. È possibile effettuare la time demand analysis per determinare i valori di s_1 ed s_2. Supponendo T_1 più prioritario di T_2,

$t^0 = e_2 = 3$

$t^1 = e_2 + \left\lceil \frac{t^0}{p_1} \right\rceil e_1 = 3 + \lceil \frac{3}{2} \rceil 1 = 5$

$t^2 = e_2 + \left\lceil \frac{t^1}{p_1} \right\rceil e_1 = 3 + \lceil \frac{5}{2} \rceil 1 = 6$

$$t^3 = e_2 + \left\lceil \frac{t^2}{p_1} \right\rceil e_1 = 3 + \lceil \tfrac{6}{2} \rceil 1 = 6$$

Il tempo di risposta del primo job di T_1 in un istante critico è quindi pari a 6, e visto che tale valore è minore del periodo p_2, questo è anche il massimo tempo di risposta s_2. Scambiando le priorità un job del task T_1 rilasciato in un istante critico deve attendere l'esecuzione di almeno un job di T_2, quindi il tempo di risposta vale sicuramente più di $e_1 + e_2 = 4 > p_1 = 2$. E' quindi necessario calcolare la lunghezza del critical busy interval

$$t^0 = e_1 + e_2 = 4$$

$$t^1 = \left\lceil \frac{t^0}{p_1} \right\rceil e_1 + \left\lceil \frac{t^0}{p_2} \right\rceil e_2 = \lceil \tfrac{4}{2} \rceil 1 + \lceil \tfrac{4}{7} \rceil 3 = 5$$

$$t^2 = \left\lceil \frac{t^1}{p_1} \right\rceil e_1 + \left\lceil \frac{t^1}{p_2} \right\rceil e_2 = \lceil \tfrac{5}{2} \rceil 1 + \lceil \tfrac{5}{7} \rceil 3 = 6$$

$$t^3 = \left\lceil \frac{t^2}{p_1} \right\rceil e_1 + \left\lceil \frac{t^2}{p_2} \right\rceil e_2 = \lceil \tfrac{6}{2} \rceil 1 + \lceil \tfrac{6}{7} \rceil 3 = 6$$

Si ricava così il numero di job di T_2 di cui bisogna calcolare il tempo di risposta, pari a $\left\lceil \frac{t^3}{p_1} \right\rceil = 3$. Per $J_{1,1}$ si ha

$$t^0 = e_1 = 1$$

$$t^1 = e_1 + \left\lceil \frac{t^0}{p_2} \right\rceil e_2 = 1 + \lceil \tfrac{1}{7} \rceil 3 = 4$$

$$t^2 = e_1 + \left\lceil \frac{t^1}{p_2} \right\rceil e_2 = 1 + \lceil \tfrac{4}{7} \rceil 3 = 4$$

quindi $w_{1,1} = t_{c1,1} = t_2 = 4$. Per $J_{1,2}$

$$t^0 = t_{c1,1} + e_1 = 5$$

$$t^1 = 2e_1 + \left\lceil \frac{t^0}{p_2} \right\rceil e_2 = 2 + \lceil \tfrac{5}{7} \rceil 3 = 5$$

perciò il tempo di completamento è $t_{c1,2} = t^1 = 5$ ed il tempo di risposta è $w_{1,2} = t_{c1,2} - p_1 = 3$. Infine per $J_{1,3}$ i calcoli sono

$$t^0 = t_{c1,2} + e_1 = 6$$

$$t^1 = 3e_1 + \left\lceil \frac{t^0}{p_2} \right\rceil e_2 = 3 + \lceil \tfrac{6}{7} \rceil 3 = 6$$

da cui è possibile determinare $t_{c1,3} = t^1 = 6$ e $w_{1,2} = t_{c1,2} - 2p_1 = 2$. Il massimo tempo si risposta s_1 è quindi pari a $w_{1,1} = 4$. Risulta ora sufficiente scegliere D_1 e D_2 in modo da soddisfare

$$4 = s_1 \leq D_1 < D_2 < s_2 = 6$$

ad esempio si può imporre $D_1 = 4$ e $D_2 = 5$. Assumendo che l'unità temporale sia pari a 100 ms si sono generate, mediante l'applicazione sviluppata, le figure 4.18 e 4.19 che illustrano l'andamento dei due task schedulati, rispettivamente, mediante le priorità fornite da DM e secondo le priorità inverse.

Un'altro teorema afferma che se un insieme di task con deadline relative uguali ai periodi è schedulabile con un algoritmo a priorità fissa allora è schedulabile dall'algoritmo Rate-Monotonic. Costruire casi non schedulabili da RM in cui le deadline relative siano minori dei periodi ed esista una schedulazione a priorità fissa è semplice. Si supponga sempre di avere due task, T_1 e T_2, e che valga $p_1 < p_2$. Si imponga

$$e_1 + e_2 < p_1$$

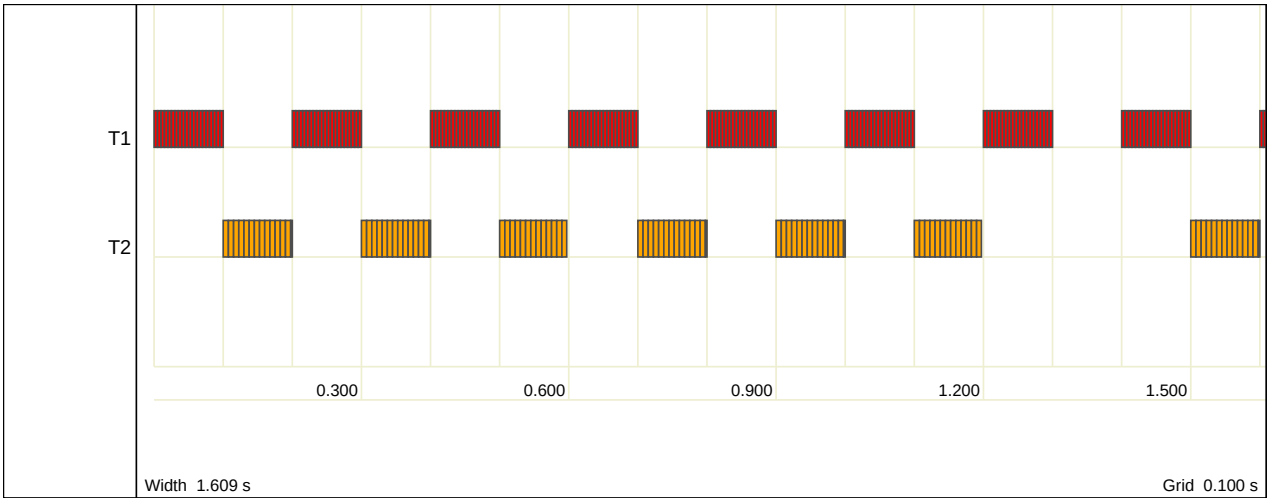

Figura 4.18: Schedulazione dei task $T_1 = (2, 1, D = 4)$ e $T_2 = (7, 3, D = 5)$ quando la priorità di T_1 è maggiore di quella di T_2.

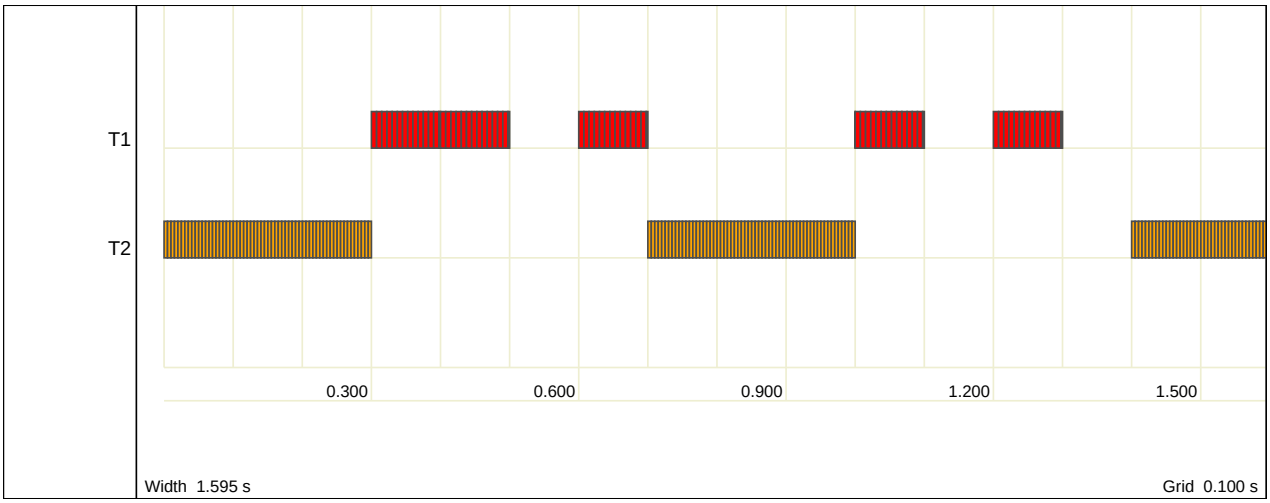

Figura 4.19: Schedulazione dei task $T_1 = (2, 1, D = 4)$ e $T_2 = (7, 3, D = 5)$ quando la priorità di T_2 è maggiore di quella di T_1.

(questa condizione assicura anche che $\frac{e_1}{p_1} + \frac{e_2}{p_2} \leq \frac{e_1}{p_1} + \frac{e_2}{p_1} < 1$). Scegliendo D_1 e D_2 in modo che

$$e_2 \leq D_2 < e_1 + e_2 \leq D_1 < p_1$$

si ottiene un caso schedulabile da un algoritmo a priorità fissa che dà priorità maggiore a T_2 ma non da RM.

Se si vogliono invece costruire casi con deadline relative maggiori dei periodi schedulabili da un algoritmo a priorità fissa ma non da RM si prendano e_1, e_2, p_1 e p_2 in modo che

$$\begin{cases} \frac{e_1}{p_1} + \frac{e_2}{p_2} \leq 1 \\ e_1 + e_2 > p_1 \\ e_2 + \left\lceil \frac{p_2}{p_1} \right\rceil e_1 > p_2 \end{cases}$$

Si calcolino come fatto prima (mediante la time demand analysis)

- s_1 il massimo tempo di risposta dei job di T_1 nel caso T_2 sia più prioritario
- s_2 il massimo tempo di risposta dei job di T_2 nel caso T_1 sia più prioritario

Grazie alle condizioni imposte vale $s_1 \geq e_1 + e_2 > p_1$ ed $s_2 \geq e_2 + \left\lceil \frac{p_2}{p_1} \right\rceil e_1 > p_2$ ed quindi è possibile scegliere $D_1 \geq s_1 > p_1$ e $p_2 < D_2 < s_2$ in modo che i task risultino schedulabili solo quando si assumano priorità inverse rispetto a quelle fornite da RM.

4.3 Non-preemptability

Nella sezione 4.2.4.4 si è visto l'esercizio 6.16 di [1] e si è notato che per il task a priorità più alta il tempo di risposta è sempre pari al tempo di esecuzione. Ciò non è più vero se i task a priorità inferiore hanno sezioni non interrompibili. Si supponga ad esempio che il task T_2 del secondo punto dell'esercizio (avente task $T_1 = (7, 10, 1, 10)$, $T_2 = (12, 6)$, $T_3 = (25, 9)$) sia totalmente non interrompibile, in tal caso la time demand function di T_1 diviene

$$w_1(t) = e_1 + b_1$$

con b_1 blocking time del primo task, in questo caso dato dal solo tempo di porzioni non-preemptable dei task a priorità inferiore $b_1(np) = e_2 = 6$. Il massimo tempo di risposta del task più prioritario passa quindi da $w_1 = e_1 = 1$ a $w_1 = e_1 + e_2 = 7$, ma dato che la deadline relativa D_1 è 10 il sistema di task rimane schedulabile secondo RM.

Si noti che, similmente a quanto visto prima, questo costituisce un caso pessimo che si può non verificare per dei particolari valori di fase (assumendo variazioni dei tempi di interrilascio trascurabili), ed infatti in questo caso la condizione non è verificabile. È comunque possibile calcolare quale sarà il massimo ritardo nell'esecuzione dei job di J_1 causati da T_2. I job del primo task vengono rilasciati agli istanti $7 + 10a$ mentre quelli del secondo agli istanti $12b$ con a e b numeri interi positivi o nulli. Risulta facile verificare che un job del primo task ed uno del secondo non potranno mai essere rilasciati assieme perché si dovrebbe avere $7 + 10a = 12b$ ossia $7 = 12b - 10a$ ma il primo membro è dispari ed il secondo è necessariamente pari. Visto che il task T_2 può essere ritardato solo da T_1 i suoi job inizieranno la loro esecuzione sempre negli istanti di rilascio. Per determinare in quali istanti all'interno del periodo di T_2 vengono rilasciati i job di T_1 è possibile calcolare i valori assumibili dalla formula $(7 + 10a)\%12$ dove con % si indica l'operazione di modulo. Ciò corrisponde, in termini formali[2], a determinare tutti gli elementi del gruppo ciclico generato dall'elemento [10] in $\mathbb{Z}/12\mathbb{Z}$ ed a sommarvi [7] sempre in $\mathbb{Z}/12\mathbb{Z}$. I valori che si ottengono sono 7, 5, 3, 1, 11 e 9. Il minimo di questi valori, 1, determina il caso pessimo: J_1 viene rilasciato un'unità temporale dopo l'inizio di T_2, ossia T_2 può ritardare l'esecuzione di un job di T_1 per un tempo massimo pari a $e_2 - 1 = 5$ unità temporali. Il seguente programma permette di simulare i tre task. Si noti come si faccia uso

[2] La notazione è quella utilizzata da [2].

quasi esclusivamente delle routine fornite da *tasktest.o*. L'unica funzione definita, *notpreemptable*, è sostanzialmente analoga a *dummy_f* tranne per il fatto che le interruzioni vengono disabilitate all'inizio di ogni job e riabilitate alla fine.

```
#include <linux/module.h>
#include <rtl_mutex.h>
#include "event.h"
#include "taskdata.h"
#include "tasktest.h"

MODULE_LICENSE("GPL");

#define UNIT 100*mSEC

void notpreemptable(void *args){
  int task=(int)args;
  rtl_irqstate_t flags;
  log_now(TASK_START,task);
  while (1){
    pthread_wait_np();
    log_now(JOB_START,task);
    rtl_no_interrupts(flags);
    use_cpu(task);
    rtl_restore_interrupts(flags);
    log_now(JOB_END,task);
  }
}

int init_module(void) {
  struct task_data task[3];
  task[0].phi=7*UNIT;
  task[0].p=10*UNIT;
  task[0].e=1*UNIT;
  task[0].D=10*UNIT;
  task[0].priority=2;

  task[1].phi=0;
  task[1].p=12*UNIT;
  task[1].e=6*UNIT;
  task[1].D=12*UNIT;
  task[1].priority=1;

  task[2].phi=0;
  task[2].p=25*UNIT;
  task[2].e=9*UNIT;
  task[2].D=25*UNIT;
  task[2].priority=0;
  int i;
  for (i=0;i<3;i++) add_task(&(task[i]));

  set_fun(1,&notpreemptable);

  start_tasks(gethrtime()+2*SEC);

  return 0;
}

void cleanup_module(void) {
  stop_tasks();
}
```

La figura 4.20 ottenuta come output di *printevent* conferma le previsioni fatte. Si può però notare che l'esecuzione del task T_2 riprende per un tempo molto breve dopo l'esecuzione dei job di T_1. Ciò trova spiegazione nel fatto che all'invocazione di *rtl_restore_interrupts(flags)* le interruzioni hardware vengono riabilitate e l'interrupt pendente relativo al timer causa l'esecuzione immediata dello scheduler; il job di T_1, rilasciato nel frattempo, viene quindi mandato in esecuzione prima che all'interno della funzione *notpreemptable* si giunga all'istruzione di attesa del prossimo job *pthread_wait_np(),* che viene invocata non appena si torna ad eseguire dopo la preemption operata da T_1.

4.4 Mutex

Basandosi sulle funzioni offerte da *tasktest.o* e dal sistema di registrazione degli eventi, risulta molto veloce implementare anche task che utilizzino mutex e verificarne il comportamento. Il programma riportato in seguito simula la situazione presentata nell'esempio di figura 8.8 a pagina 288 di [1], con alcune piccole variazioni:

- si è chiamata A la risorsa *Shaded* e B la risorsa *Black,*
- si è posto a 4.8 il tempo di rilascio di J_2 per evidenziare la non preemption nel caso di utilizzo di protocollo ceiling-priority, come fatto a pagina 302 del libro,
- per semplificare il codice si è portato a 2 il tempo di utilizzo della risorsa B da parte di J_4.

In ultima analisi, quindi, se ciascun job disponesse del processore e di tutte le risorse

- J_1, rilasciato al tempo 7, eseguirebbe per un'unità di tempo, userebbe la risorsa A per un'altra unità, la cederebbe e proseguirebbe per un'altra unità di tempo.
- J_2, rilasciato al tempo 4.8, eseguirebbe per un'unità di tempo, userebbe la risorsa B per un'unità, la cederebbe e proseguirebbe l'esecuzione per un'ulteriore unità di tempo.
- J_3, rilasciato al tempo 4, eseguirebbe per due unità di tempo.
- J_4, rilasciato al tempo 2, eseguirebbe per un'unità di tempo, effettuerebbe un lock sulla risorsa A, proseguirebbe per un'unità di tempo, utilizzerebbe la risorsa B per due unità di tempo, continuerebbe l'impiego di A per un ulteriore unità, la rilascerebbe e dopo un'ultima unità di tempo terminerebbe la sua esecuzione (il suo utilizzo delle risorse è cioè $[A; 4[B; 2]]$ secondo la notazione di[1]).
- J_5, rilasciato all'istante 0, eseguirebbe per un'unità di tempo, impiegherebbe la risorsa B per 4 unità e poi eseguirebbe senza utilizzare risorse per un'unità di tempo.

```
#include <linux/module.h>
#include <rtl_mutex.h>
#include "event.h"
#include "taskdata.h"
#include "tasktest.h"

MODULE_LICENSE("GPL");

#define UNIT 100*mSEC

#define MUTEXPROTOCOL PTHREAD_PRIO_NONE
//#define MUTEXPROTOCOL PTHREAD_PRIO_PROTECT

#define TASK1ID 0
#define TASK2ID 1
```

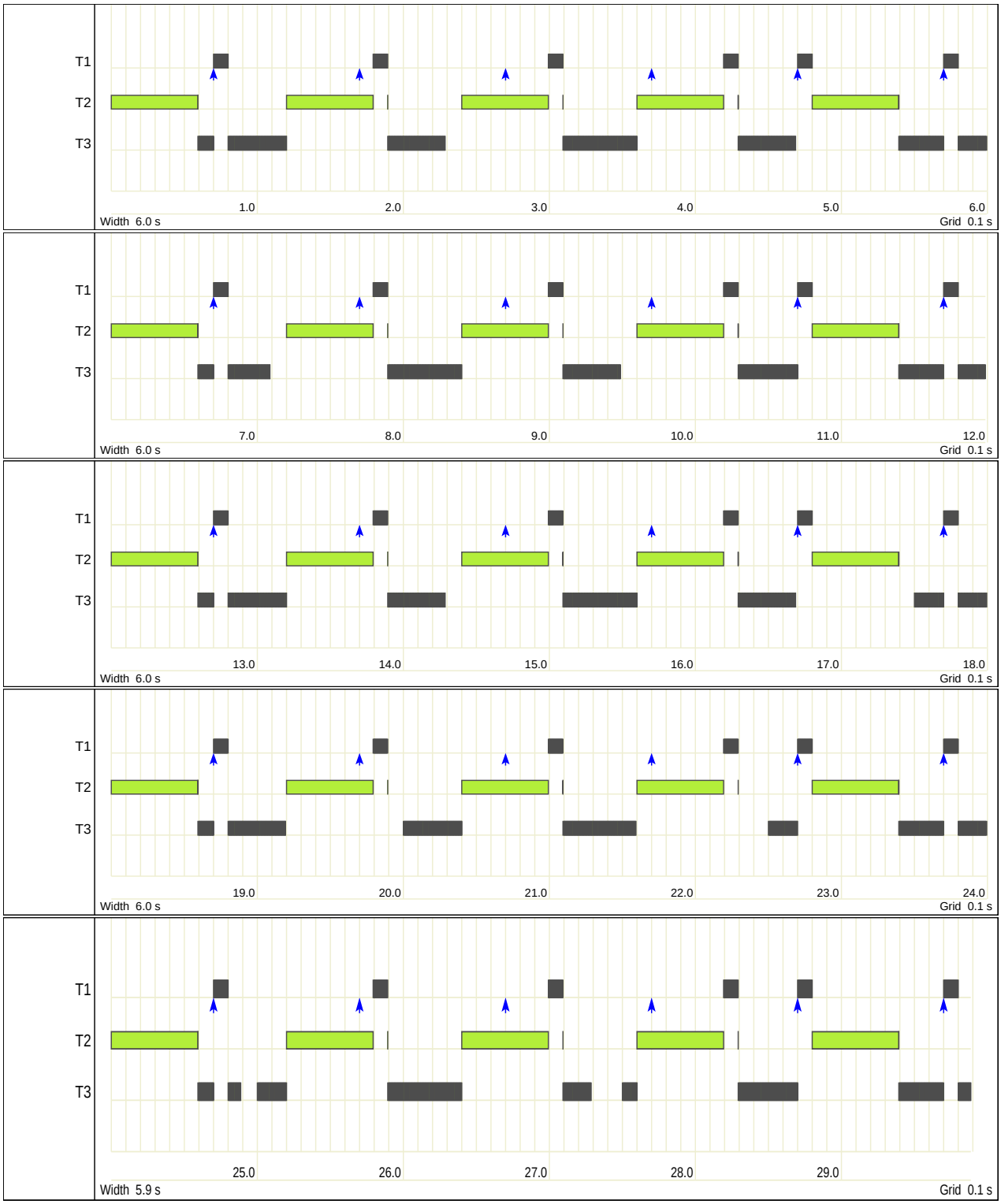

Figura 4.20: Andamento temporale dei task $T_1 = (7, 10, 1, 10)$, $T_2 = (12, 6)$ e $T_3 = (25, 9)$ nel primo iperperiodo (unità temporale 100 ms) supponendo che i job di T_2 siano totalmente non preemptable. I job di T_2 eseguono a partire dal loro rilascio, mentre il rilascio dei job di T_1 è indicato dalle frecce blu.

```
#define TASK3ID 2
#define TASK4ID 3
#define TASK5ID 4

#define AMUTEXID 0
#define BMUTEXID 1

pthread_mutex_t mutexA,mutexB;


void f_task1(void *arg){
  pthread_wait_np();
  log_now(JOB_START,TASK1ID);
  use_cpu(TASK1ID);
  log_now_full(MUTEX_LOCK,TASK1ID,AMUTEXID);
  if(!pthread_mutex_lock(&mutexA)){
    log_now_full(MUTEX_GOT,TASK1ID,AMUTEXID);
    use_cpu(TASK1ID);
    pthread_mutex_unlock(&mutexA);
    log_now_full(MUTEX_UNLOCK,TASK1ID,AMUTEXID);
    use_cpu(TASK1ID);
  }
  log_now(JOB_END,TASK1ID);
}

void f_task2(void *arg){
  pthread_wait_np();
  log_now(JOB_START,TASK2ID);
  use_cpu(TASK2ID);
  log_now_full(MUTEX_LOCK,TASK2ID,BMUTEXID);
  if (!pthread_mutex_lock(&mutexB)){
    log_now_full(MUTEX_GOT,TASK2ID,BMUTEXID);
    use_cpu(TASK2ID);
    pthread_mutex_unlock(&mutexB);
    log_now_full(MUTEX_UNLOCK,TASK2ID,BMUTEXID);
    use_cpu(TASK2ID);
  }
  log_now(JOB_END,TASK2ID);
}


void f_task3(void *arg){
  pthread_wait_np();
  log_now(JOB_START,TASK3ID);
  use_cpu(TASK3ID);
  use_cpu(TASK3ID);
  log_now(JOB_END,TASK3ID);
}

void f_task4(void *arg){
  pthread_wait_np();
  log_now(JOB_START,TASK4ID);
  use_cpu(TASK4ID);
  log_now_full(MUTEX_LOCK,TASK4ID,AMUTEXID);
  if (!pthread_mutex_lock(&mutexA)){
    log_now_full(MUTEX_GOT,TASK4ID,AMUTEXID);
    use_cpu(TASK4ID);
    log_now_full(MUTEX_LOCK,TASK4ID,BMUTEXID);
    if (!pthread_mutex_lock(&mutexB)){
      log_now_full(MUTEX_GOT,TASK4ID,BMUTEXID);
```

```
      use_cpu(TASK4ID);
      use_cpu(TASK4ID);
      pthread_mutex_unlock(&mutexB);
      log_now_full(MUTEX_UNLOCK,TASK4ID,BMUTEXID);
      use_cpu(TASK4ID);
      pthread_mutex_unlock(&mutexA);
      log_now_full(MUTEX_UNLOCK,TASK4ID,AMUTEXID);
      use_cpu(TASK4ID);
    }//mutexB
  }//mutexA
  log_now(JOB_END,TASK4ID);
}

void f_task5(void *arg){
  pthread_wait_np();
  log_now(JOB_START,TASK5ID);
  use_cpu(TASK5ID);
  log_now_full(MUTEX_LOCK,TASK5ID,BMUTEXID);
  if (!pthread_mutex_lock(&mutexB)){
    log_now_full(MUTEX_GOT,TASK5ID,BMUTEXID);
    use_cpu(TASK5ID);
    use_cpu(TASK5ID);
    use_cpu(TASK5ID);
    use_cpu(TASK5ID);
    pthread_mutex_unlock(&mutexB);
    log_now_full(MUTEX_UNLOCK,TASK5ID,BMUTEXID);
    use_cpu(TASK5ID);
  }
  log_now(JOB_END,TASK5ID);
}

struct task_data task[5];

int init_module(void) {
  pthread_mutexattr_t attrA,attrB;
  pthread_mutexattr_init(&attrA);
  pthread_mutexattr_init(&attrB);
  pthread_mutexattr_setprotocol(&attrA,MUTEXPROTOCOL);
  pthread_mutexattr_setprotocol(&attrB,MUTEXPROTOCOL);

  task[TASK1ID].phi=7*UNIT;
  task[TASK2ID].phi=4*UNIT+UNIT*8/10;
  task[TASK3ID].phi=4*UNIT;
  task[TASK4ID].phi=2*UNIT;
  task[TASK5ID].phi=0*UNIT;
  int i;
  for (i=0;i<5;i++) {
    task[i].p=0;
    task[i].e=UNIT;
    task[i].D=0;
    task[i].priority=5-i;
  }
  pthread_mutexattr_setprioceiling(&attrA,task[TASK1ID].priority);
  pthread_mutexattr_setprioceiling(&attrB,task[TASK2ID].priority);
  pthread_mutex_init (&mutexA, &attrA);
  pthread_mutex_init (&mutexB, &attrB);
  for (i=0;i<5;i++) add_task(&(task[i]));
  set_fun(TASK1ID,&f_task1);
  set_fun(TASK2ID,&f_task2);
  set_fun(TASK3ID,&f_task3);
```

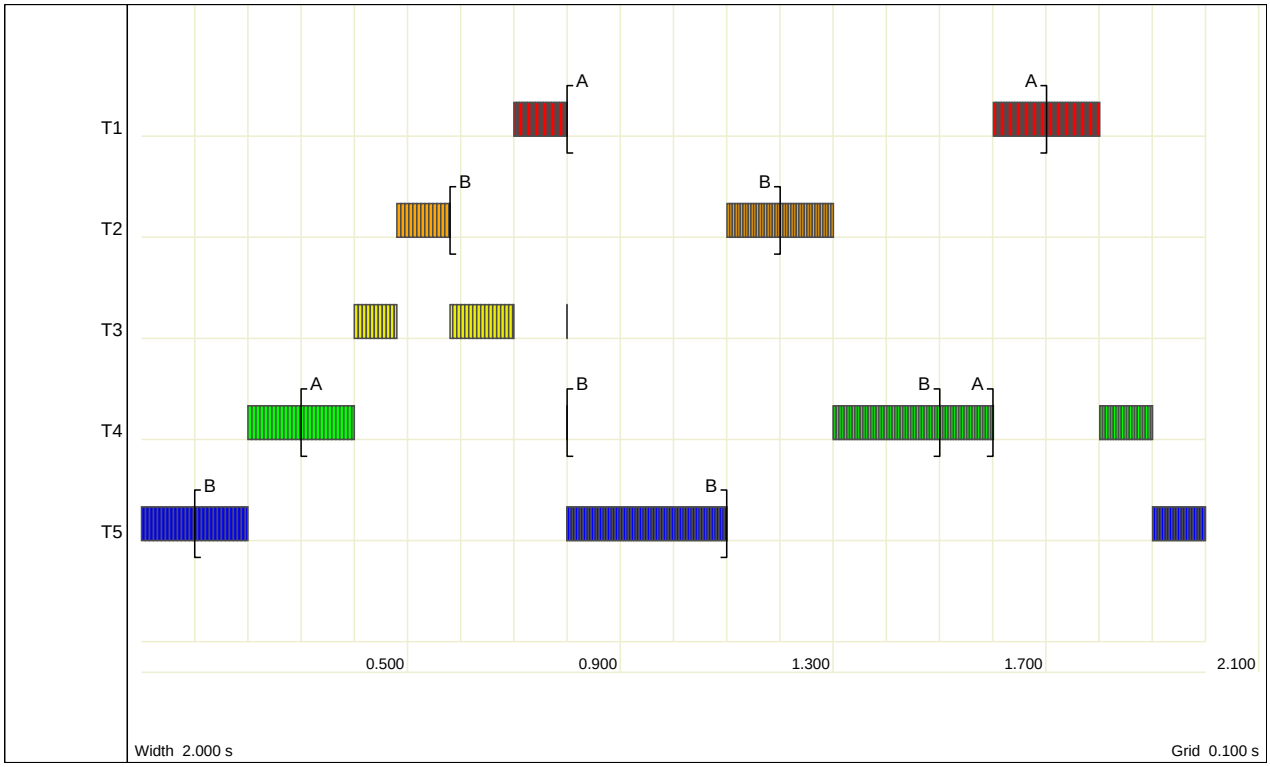

Figura 4.21: Andamento nel caso di protocollo *PTHREAD_PRIO_NONE*. L'esecuzione di un lock è indicata dall'apertura di una parentesi quadra avente come apice la risorsa che si intende acquisire, e, analogamente, il rilascio di una risorsa è indicato dalla chiusura della parentesi.

```
  set_fun(TASK4ID,&f_task4);
  set_fun(TASK5ID,&f_task5);
  start_tasks(gethrtime()+2*SEC);
  return 0;
}

void cleanup_module(void) {
  stop_tasks();
}
```

Come si può notare nel codice vi sono due direttive per definire la costante *MUTEXPROTOCOL*, usata per impostare il protocollo di entrambe le risorse

```
#define MUTEXPROTOCOL PTHREAD_PRIO_NONE
//#define MUTEXPROTOCOL PTHREAD_PRIO_PROTECT
```

Commentando l'una o l'altra si è potuto registrare il comportamento del sistema nel caso le risorse vengano assegnate non appena sono disponibili (protocollo *PTHREAD_PRIO_NONE*) e nel caso si utilizzi l'implementazione di RTLinux del protocollo ceiling-priority (protocollo *PTHREAD_PRIO_PROTECT*). Nel programma a queste direttive ne seguono altre, inserite con l'unico scopo di aumentare la leggibilità del codice. Sono state poi definite delle funzioni, che vengono impostate, mediante *set_fun*, come corpo dei vari thread creati da *start_tasks*. Nelle funzioni si possono notare delle *if* che fanno in modo che l'esecuzione venga interrotta qualora una risorsa non venga concessa per violazione del protocollo. Utilizzando il protocollo *PTHREAD_PRIO_NONE* si ha la situazione presentata in figura 4.21 , dove, come indica il codice, l'unità di tempo è stata assunta pari a 100 millisecondi:

- All'istante 0 J_5 entra in esecuzione.
- All'istante 1 J_5 effettua un lock su B ed acquisisce la risorsa.

- All'istante 2 J_5 subisce preemption da parte di J_4che viene rilasciato in questo momento.
- All'istante 3 J_4 esegue un lock su A ed acquisisce subito la risorsa.
- All'istante 4 J_3 viene rilasciato ed effettua preemption su J_4.
- Al tempo 4.8, istante di rilascio di J_2, J_3 subisce preemption.
- All'istante 5.8 J_2 tenta un lock su B ma la risorsa è posseduta da J_4 quindi viene bloccato; in altri termini, J_4 effettua un blocco diretto su J_2. J_3 riprende la sua esecuzione e si assiste perciò ad una priority inversion.
- All'istante 7 J_1 nel momento del suo rilascio entra in esecuzione. In teoria in questo stesso istante J_3 dovrebbe aver terminato la sua esecuzione, tuttavia a causa dei tempi non conteggiati da *use_cpu*, come il tempo di invocazione della funzione stessa e delle operazioni necessarie alla distruzione del thread, il job J_3 subisce il realtà preemption, per tornare ad eseguire per un tempo brevissimo (circa 30 microsecondi) al tempo 8. Per tener conto di questi tempi sarebbe possibile ridurre leggermente i tempi di esecuzione. Questo è stato fatto (con una riduzione sempre inferiore all'1%) per la realizzazione di alcune delle figure presentate in precedenza, in particolare il tempo e_3 per le figure 4.13 e 4.15 ed il tempo e_2 per la figura 4.18.
- All'istante 8 J_1 viene bloccato perché richiede A. Come per J_3 anche per J_4 vi è un piccolissimo ritardo (inferiore ai 20 microsecondi) che fa si che l'esecuzione del lock su B avvenga in questo momento anziché al tempo 4. Visto che la risorsa è posseduta da J_5 anche J_4 viene sospeso e J_5, unico job ready, riprende la sua esecuzione.
- All'istante 11 J_5 rilascia B e J_2 può riprendere la sua esecuzione.
- All'istante 12 J_2 rilascia B.
- All'istante 13 J_2 termina la sua esecuzione ed entra perciò in esecuzione J_4, il quale può ora disporre anche di B.
- All'istante 15 J_4 rilascia la risorsa B.
- All'istante 16 J_4 rilascia la risorsa A, J_1 torna ad essere ready ed avendo priorità maggiore degli altri job ready (J_4 e J_5) torna in esecuzione. Si noti che la preemption non è immediata, come conferma la figura 4.22, per la mancanza in *pthread_mutex_unlock* d'invocazioni ad *rtl_schedule* segnalata a pagina 26.
- All'istante 17 J_1 rilascia A.
- All'istante 18 J_1 termina e J_4 torna in esecuzione.
- All'istante 19 J_4 termina e J_5 torna in esecuzione essendo l'unico job che deve ancora terminare la sua esecuzione.
- All'istante 20 J_5 termina.

La figura 4.23 illustra invece l'andamento temporale nel caso si impieghi il protocollo *PTHREAD_PRIO_PROTECT*

- All'istante 0 J_5 entra in esecuzione.
- All'istante 1 J_5 acquisisce la risorsa B ed assume così il priority ceiling di B, Π_B pari alla priorità π_2 di J_2.

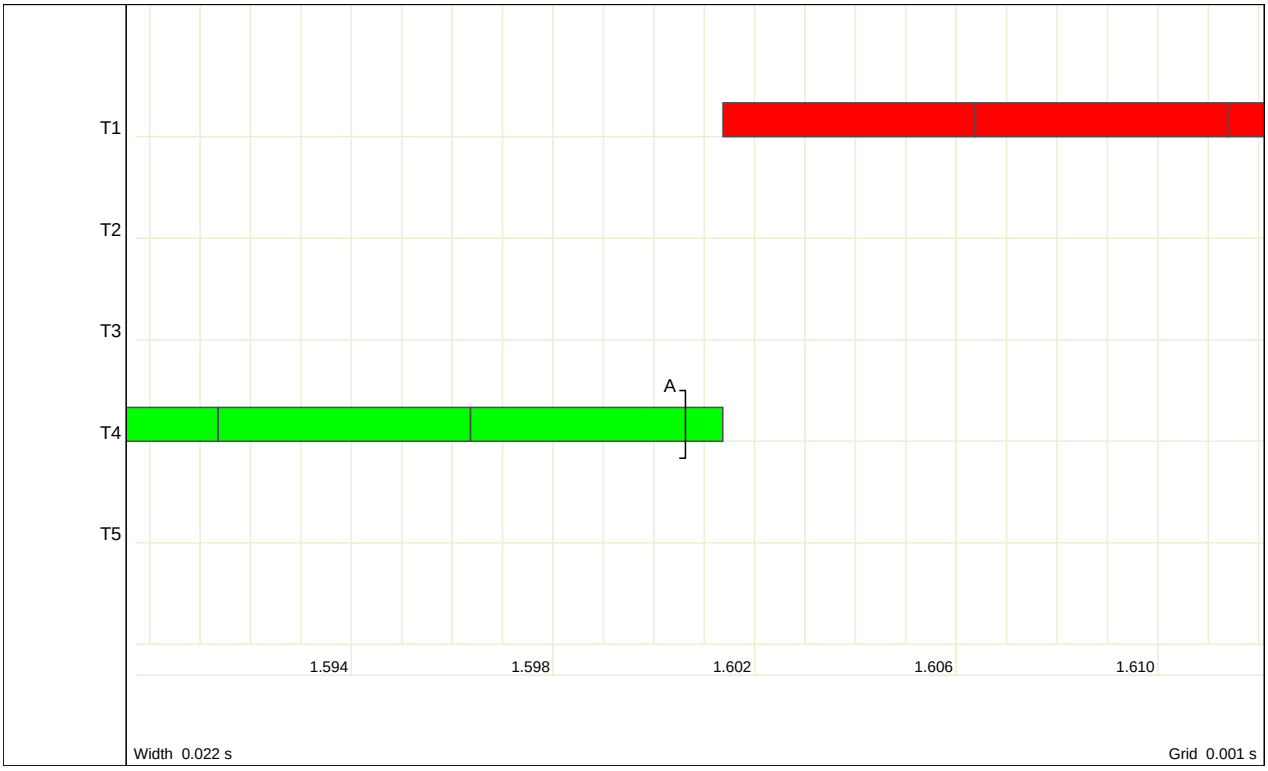

Figura 4.22: Andamento temporale dei task in prossimità dell'istante 16. Risulta possibile notare come il task T_1 torni in esecuzione solo grazie all'invocazione periodica dello scheduler e non per l'invocazione di *pthread_mutex_unlock* su A.

- All'istante 5 J_5 rilascia B. Nel frattempo sono stati rilasciati J_4 (istante 2), J_3 (istante 4) e J_2 (istante 4.8) ma nessuno dei job ha compiuto preemption perché J_5 ha eseguito con priorità π_2. Si noti che, visto che non si è utilizzato l'approccio indicato a pagina 25 (consistente nell'associare ad una risorsa una priorità maggiore della priorità del job a massima priorità che la utilizza), in realtà J_2 non ha effettuato preemption su J_5 solo perché il corrispondente task è stato aggiunto dopo T_2 in fase di creazione. Tra i job ready quello a massima priorità è J_2 e quindi è questo il job che entra in esecuzione.
- All'istante 6 J_2 effettua un lock su B ma non varia la sua priorità perché è proprio J_2 il job che determina il priority ceiling di B.
- All'istante 7 viene rilasciato J_1 che effettua preemption su J_2.
- All'istante 8 J_1 acquisisce A, ed anche in questo caso non si ha alcuna variazione di priorità.
- All'istante 9 J_1 effettua unlock su A.
- All'istante 10 J_1 termina e J_2 riprende la sua esecuzione. Analogamente a quanto visto prima, piccoli ritardi fanno sì che il rilascio di B da parte di J_2 avvenga in questo momento anziché a 7.
- All'istante 11 J_2 termina la sua esecuzione e J_3 può partire.
- All'istante 13 J_3 termina e viene avviata l'esecuzione di J_4.
- All'istante 14 J_4 acquisisce A e la sua priorità viene perciò innalzata a $\Pi_A = \pi_1$.
- All'istante 15 J_4 tenta di acquisire B. Come spiegato a causa del fatto che RTLinux non distingue tra priorità assegnata e priorità corrente di un job questa acquisizione viene

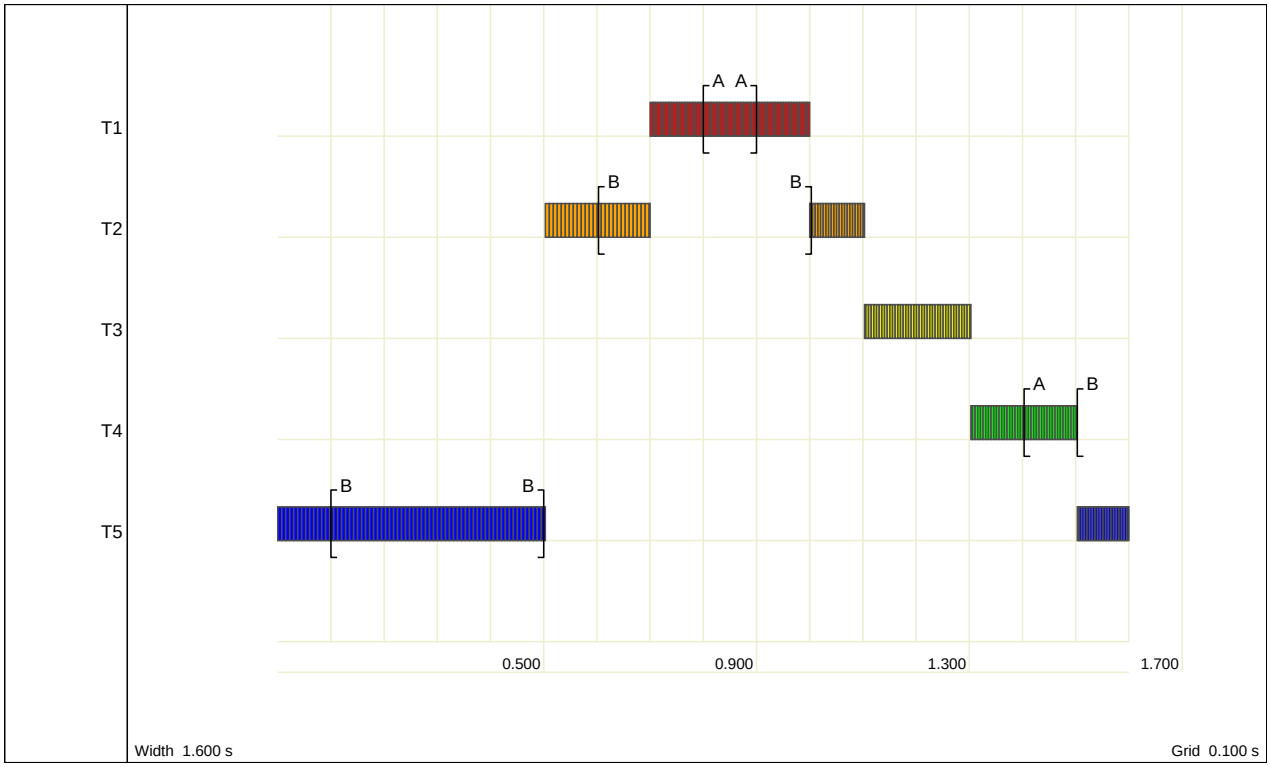

Figura 4.23: Andamento nel caso di protocollo *PTHREAD_PRIO_PROTECT*.

interpretata come violazione del protocollo ed il task viene terminato; J_5 rimane l'unico task attivo e riprende la sua esecuzione.

- All'istante 16 anche J_5 termina la sua esecuzione.

Capitolo 5

Considerazioni conclusive

RTlinux versione 3.1 si è rivelato stabilissimo e, almeno per quanto si è potuto osservare analizzando il codice e implementando le applicazioni presentate, privo di bug. Il fatto di essere totalmente compatibile con Linux e di poter sviluppare codice real time sulla stessa macchina su cui lo si va ad eseguire risulta molto comodo e riduce sicuramente i tempi di sviluppo. Comunque, visto che le applicazioni girano, almeno in parte, in spazio kernel, se non altro nelle prime fasi di debug può risultare utile utilizzare un emulatore, per evitare di dover riavviare la macchina fisica qualora si siano effettuati errori comportanti il blocco del sistema. L'utilizzo di emulatori open source, come QEMU, o commerciali, come VMware, che emulino la rete permette di sviluppare il codice sul computer fisico e copiarlo sulla macchina virtuale con un semplice *scp* se si installa *ssh* su entrambe le macchine, quella fisica e quella simulata.

In molte fonti viene riportato che il tempo di latenza tra l'arrivo di un interrupt e l'invocazione della routine associata è, con RTLinux, vicina ai limiti dell'hardware; non avendo a disposizione un oscilloscopio non è stato possibile effettuare tale misura, ma sarebbe molto interessante un confronto con altri sistemi operativi come VxWorks, RTAI e Xenomai.

La documentazione su RTLinux è abbastanza scarsa, e rivolta quasi esclusivamente all'installazione e all'utilizzo delle API. Per capirne il funzionamento, almeno per quanto riguarda la versione free, è possibile analizzarne il sorgente, che, seppur quasi privo di commenti, risulta abbastanza ben organizzato, come del resto si sarà potuto constatare leggendo la descrizione delle caratteristiche.

Per quanto riguarda l'aspetto didattico RTLinux non prevede la scelta tra vari scheduler, come succede per alti sistemi operativi real-time come Shark, ed in particolare non prevede EDF, molto interessante dal punto di vista teorico perché ottimo sotto le ipotesi di singolo processore e task indipendenti e totalmente preemptable. Data la struttura modulare di RTLinux non risulta comunque difficile implementare nuovi algoritmi di schedulazione, infatti in [56] è presentato uno scheduler EDF per RTLinux. Analogamente non viene reso disponibile alcun server per i task aperiodici, se si trascura il caso di considerare Linux come un background server, e non esiste alcun algoritmo di accettazione di task sporadici, tuttavia vi sono vari lavori (come [57]), realizzati in genere in ambito accademico, che forniscono implementazioni di server per job aperiodici per RTLinux.

Come ultima cosa si sottolinea il fatto che molti degli elementi di RTLinux, come le code FIFO, si possono ritrovare, ovviamente con qualche differenza, anche in Windows CE, rendendo molto interessante un confronto tra due sistemi operativi sviluppati secondo percorsi fortemente diversi.

Appendice A

Comandi bash

Nella trattazione si sono inclusi comandi e script bash utilizzanti vari programmi normalmente presenti in Linux. Il significato dei comandi utilizzati è il seguente

- *echo stringa* stampa la stringa *stringa* su standard output.
- seq *min incremento max* stampa tutti i numeri (anche reali) a partire da *min* con passo *incremento* minori o uguali a *max.*
- *cat nomefile* stampa il file *nomefile* su standard output.
- *sort nomefile* ordina le righe del file in ordine alfabetico se non si spefica nulla, in ordine numerico (nel caso la prima parte della riga sia un numero) se si specifica l'opzione *-g* (mediante questa opzione vengono ordinati anche numeri nel formato esponenziale *xEy*).
- *tail nomefile* stampa le ultime righe del file *nomefile*. Mediante l'opzione *-n numerorighe* è possibile indicare il numero di righe da stampare. Se al numero viene preposto il segno '+' vengono stampate le righe a partire dalla *numerorighe*-esima.
- *cut -f ncampo -d delimitatore nomefile* stampa l'*ncampo*-esimo campo di ciascuna riga del file *nomefile* considerando il carattere *delimitatore* come separatore tra i campi.
- *grep espressione nomefile* stampa le righe del file *nomefile* contenenti l'espressione regolare *espressione*. Se viene specificata l'opzione *-v* vengono invece stampate le righe che non contengono l'espressione regolare fornita. Con l'opzione *-A numerorighe* si può indicare di stampare anche le *numerorighe* righe successive a ciascuna occorrenza dell'espressione regolare.
- *sed* (stream editor) permette di effettuare molti tipi di modifiche su stream. In particolare *sed s/stringa1/stringa2/ nomefile* sostituisce ogni occorrenza della stringa *stringa1* con la stringa *stringa2* (si possono specificare anche espressioni regolari).

In ciascuno dei programmi appena citati se *nomefile* viene omesso la lettura avviene da standard input. Per fare in modo che lo standard output del comando *comando1* sia lo standard input del comando *comando2* è possibile utilizzare

comando1 | *comando2*

Per redirigere lo standard output sul file *nomefile* è sufficiente scrivere

>nomefile

in tal caso tutti i dati precedentemente presenti in *nomefile* vengono eliminati. Per fare in modo che lo standard output venga aggiunto in coda al file *nomefile* è necessario utilizzare

>>nomefile

La sintassi

comando1 ‘comando2‘

fa sì che l'output di *comando1* venga passato come argomento a *comando2*. Il costrutto

for variabile in lista; do comandi; done

fa in modo che i comandi *comandi* vengano eseguiti tante volte quanti sono gli elementi della lista *lista*. All'interno dei comandi *comandi* è possibile utilizzare la variabile *$variabile* che assume ad ogni iterazione come valore uno degli elementi della lista.

Per valutare espressioni matematiche è possibile utilizzare *let,* ad esempio

let n=$n+1

incrementa il valore della variabile n (se non si specifica *let* nell'istruzione precedente si ottiene la concatenazione ad n della stringa "+1").

Bibliografia

[1] Jane W. S. Liu, Real-Time Systems, Prentice Hall

[2] Niels Lauritzen, Concrete Abstract Algebra, Cambridge University Press

[3] http://www.linuxjournal.com/article/7178

[4] http://www.danielinux.net/projects/scheduler.pdf

[5] http://www.realtimelinuxfoundation.org/

[6] http://www.univ.trieste.it/~mumolo/rtlinux.pdf

[7] http://www.linuxdevices.com/articles/AT7005360270.html

[8] http://www.freego.it/articles/show/53

[9] http://rtportal.upv.es/comparative/rtl_vs_rtai.html

[10] http://www.uclinux.org/

[11] http://www.ittc.ku.edu/kurt/

[12] http://www.mvista.com/products/realtime.html

[13] https://www.rtai.org/

[14] http://www.fsmlabs.com

[15] http://www.fsmlabs.com/openpatentlicense.html

[16] http://www.rtlinuxfree.com/

[17] http://en.wikipedia.org/wiki/RTLinux

[18] http://www.linuxdevices.com/articles/AT3694406595.html

[19] http://www.mnis.fr/ocera_support/rtos/c1450.html

[20] http://www.linuxfocus.org/English/July1998/article56.html

[21] http://www.ueidaq.com/press/publications/realtimelinux/

[22] https://users.cs.jmu.edu/abzugcx/public/Student-Produced-Term-Projects/Operating-Systems-2003-FALL/

[23] http://nferre.free.fr/emlnx/rapport/node9.html

[24] http://www.vmlinux.org/rtl/docs/RTLinux-part1.pdf

[25] http://citeseer.ist.psu.edu/barabanov97linuxbased.html

[26] http://scuola.linux.it/docs/linuxmagazine/zanatta21.html

[27] http://tldp.org/linuxfocus/English/May1998/article44.html

[28] http://www.bertolinux.com/kernel/italiano/KernelAnalysis-HOWTO-6.html

[29] http://www.faqs.org/docs/kernel/x1206.html

[30] http://www.yodaiken.com/notes.html

[31] http://www.slac.stanford.edu/exp/glast/flight/docs/VxWorks/docs/vxworks/ref/clockLib.html

[32] http://www.embedded.com/2001/0104/0104feat4table1.htm

[33] http://en.wikipedia.org/wiki/RDTSC

[34] http://rtlinux.lzu.edu.cn/documents/documentation/man_pages/html/man_page_index.html

[35] http://pwet.fr/man/linux/fonctions_bibliotheques/rtl/

[36] http://www.ece.osu.edu/~cglee/ECE694Z/rtlinux/GettingStarted.pdf

[37] http://www.opengroup.org/onlinepubs/009695399/functions/clock_nanosleep.html

[38] http://www.die.net/doc/linux/man/man2/sigaction.2.html

[39] http://www3.cc.gatech.edu/classes/AY2003/cs6235_spring/project1/doc/html/MAN/sigaction.3.html

[40] http://bcook.cs.georgiasouthern.edu/cs523/critical.htm

[41] http://www.opengroup.org/onlinepubs/007908799/

[42] http://en.wikipedia.org/wiki/Debian#Debian_package_life_cycle

[43] http://en.wikipedia.org/wiki/Ncurses

[44] http://tldp.org/HOWTO/NCURSES-Programming-HOWTO/

[45] http://www.gnu.org/software/grub/manual/grub.html

[46] http://www.dirac.org/linux/system.map/

[47] http://download.nvidia.com/XFree86/Linux-x86/1.0-8174/README/32bit_html/chapter-05.html

[48] http://www.linux-mips.org/wiki/Modules

[49] http://www.lirc.org/html/install.html

[50] http://bama.ua.edu/~dunna001/journeyman/html/c241.htm

[51] http://marc2.theaimsgroup.com/?l=linux-rt&m=111625593625662&w=2

[52] http://rtportal.upv.es/apps/kiwi/

[53] http://rtportal.upv.es/apps/kiwi/kiwi-1.0/docs/user-guide.html

[54] http://fabrice.bellard.free.fr/qemu/

[55] http://www.vmware.com/

[56] http://rtportal.upv.es/apps/edf-sched/index.shtml

[57] http://citeseer.ist.psu.edu/490264.html

[58] http://shark.sssup.it/