

LEZIONE ESERCITATIVA

(Continuo della lezione su Task periodici)

RICORDA:

→ USATO: ha "usleep" di UNIX

LA SOLUZIONE 1: SLEEP (per il tempo rimanente)

Questa non è una buona idea a causa del RELATIVE SLEEP

POICHÉ: il task potrebbe dormire per a lungo o molto meno
rispetto al tempo prescelto

→ TRABONI: vuol dire che so una
scorsa posizione

SORITA: Ad una preemption
che può accadere durante la
"WAKE-next-Activation"

• SOLUZIONE 2: L'USO DEI TIMER

AVVARENTE: Deve considerare
entrambi gli standard
(UNIX e POSIX)

→ RAPPRESENTA: ha soluzioni
migliore da considerare
POICHÉ: Posso risegnalare il processo
in ogni momento, qualunque
sia il suo stato

✓ Vedi File: Periodic Task 2a e 2b

a) **TIMER** **UNIX** → BisOGNA opportunamente **modificare** le funzioni

- ha "Start Periodic Timer"

```
#define wait_next_activation pause
static void sighand(int s) {

void start_periodic_timer(uint64_t off, int period) {
    struct itimerval t;

    t.it_value.tv_sec = off / 1000000;
    t.it_value.tv_usec = off % 1000000;
    t.it_interval.tv_sec = period / 1000000;
    t.it_interval.tv_usec = period % 1000000;

    signal(SIGALRM, sighand);

    setitimer(ITIMER_REAL, &t, NULL);
}
```

Nota: Se vuoi anche impostare quote timer vuoi solo usare

ATTENZIONE: Ha questa non è l'unica che viene modificata

SOPRATTUTTO: ha funzione **void next Activation** che "diventa" la **funzione pause**

Allo Scattare del Timer: Viene generato un **SEGNALE**
(di tipo **SIGALARM**)

È NECESSARIO: In quanto richiede un **lavoro**
che va a **risvegliare il Task**

TUTTAVIA: Il Task potrebbe essere
risvegliato da **qualsiasi segnale**
in avvio
(causando problemi di tempo)

TUTTAVIA: Causa un **overhead**
Dovuto al meccanismo delle
interruzioni generato dalla
gestione del segnale

Nota: Così fanno anche
non usare l'handle
nudo

PER QUESTO: Usò una funzione apposita
"sigwait"

Mette il Task in attesa di uno specifico segnale

ATTENZIONE: Devo anche usare una serie di
funzioni per impostare il segnale etc...

Un PROBLEMA:

Posso avere un unico Timer PT per ogni Task

Può essere molto problematico
da usare se ho molti
Thread

CAUSATO: dalle caratteristiche
dei Timer UNIX

b) **TIMER POSIX** \Rightarrow Vedi Periodic Task 2c

PERMETTONO di assegnare più Timer PT ad ogni processo
(ma sono molto più complessi da usare)

ha funzione Task Periodic Timer diversa infatti

```
int start_periodic_timer(uint64_t offs, int period){  
    struct itimerspec t;    struct sigevent sigev;  
    timer_t timer;    const int signal = SIGALRM;  
    int res;  
  
    t.it_value.tv_sec      = offs      / 1000000;  
    t.it_value.tv_nsec     = (offs % 1000000) * 1000;  
    t.it_interval.tv_sec   = period    / 1000000;  
    t.it_interval.tv_nsec = (period % 1000000) * 1000;  
    sigemptyset(&sigset); sigaddset(&sigset, signal);  
    sigprocmask(SIG_BLOCK, &sigset, NULL);  
  
    memset(&sigev, 0, sizeof(struct sigevent));  
    sigev.sigev_notify = SIGEV_SIGNAL;  
    sigev.sigev_signo = signal;           ↑ Evento collegato al timer  
    res = timer_create(CLOCK_MONOTONIC, &sigev, &timer);  
    if (res < 0) {perror("Timer Create"); exit(-1);}  
    return timer_settime(timer, 0 /*TIMER_ABSTIME*/, &t, NULL);  
}
```

QUINDI: Molto più complessa
rispetto ai casi precedenti

Norosante La Complessità le funzioni mettono a disposizione una serie di strumenti molto utili



Ad ESEMPIO: ha possibilità di impostare un segnale



Inoltre: Ad ogni timer posso assegnare un segnale differente

SOPRANNO: Vantaggio importante e che sono fortunati
(Proprio grazie all'uso della libreria Standard)

Il Clock Monotonic: È quello migliore da usare con questa tipologia di timer

Poiché: risulta INCORRIGIBILE ed ha la stessa risoluzione del clock del sistema

• SOLUZIONE 3: SLEEP ASSOLUTO

Così il task viene sospeso fino al tempo assoluto della successiva istruzione



QUESTA: è la soluzione migliore da usare ed quella più semplice

Un **ESEMPIO** è la **Clock_Nanosleep**

→ **QUESTA:** fa sempre parte dello standard **Posix**

In Alcuni Casi è anche più precisa dei Timer

→ **RISPOSTA A QUESTI**

Nota: Anche considerando quello doendo alla gestione delle int.

• **Hanno meno overhead**

• Non devono essere reimpostati ogni volta

• Sono anche molto più precisi

Considero le Funzioni:

• **Start Periodic Timer**

```
static struct timespec r;
static int period;

void start_periodic_timer(uint64_t offs, int t) {
    clock_gettime(CLOCK_REALTIME, &r);
    timespec_add_us(&r, offs);
    period = t;
}
```

Nella Quale: Vedo ad inserire le strutture che alloranno

Nota: Questo valore è il **riferimento assoluto** rispetto a cui vengono calcolati i **risegni successivi**

• Wait Next Activation

```
void wait_next_activation(void) {
    clock_nanosleep(CLOCK_REALTIME,
                     TIMER_ABSTIME, &r, NULL);
    timespec_add_us(&r, period);
}
```

Si limita ad eseguire una sleep ed andare ad aggiornare il valore per il successivo risveglio

UNA NOTA: In questo caso, si usa delle variabili globali

ANCHE SE: non sono la giusta scelta da fare soprattutto quando uso dev thread

INTRODUZIONE

PREEHPT - RT

→ Nota: Non è l'unico approccio per usare, ma si hanno anche altre soluzioni come RTI

La PROBLEMATICA di Linux: È un SO Preemptive

Perciò: Posso conoscere la latenza dello scheduling, ma bisogna vedere se è sempre possibile

Esempio: I task sono Preemptible, ma il Kernel no, se arriva un'interruzione

→ Vene serveba subito poiché è interrompibile ma dopo **può** **proseguire con il suo compito**

Quindi non posso fornire il tempo di calcolo residuo

- Inoltre potranno usare un meccanismo che **disattiva le interruzioni**

uso di Spinlock, lock con attesa attiva, il thread del kernel resta attivo fino a che la risorsa non si libera

QUESTO: Vene fatto per **motivazioni di prestazione** rispetto ad uno sleep lock, infatti il codice è più veloce

Pero' le sezioni critiche protette da spinlock, devono anche disabili
fare le interruzioni

→ ALTRIMENTI Polley avverte la lock
soprattutto se lo task con diverse
priorità che agiscono su una
risorsa

OBIETTIVO: Ponere il kernel fully preemptible in un弘扬
vengono usati 2 approcci

• Dual Kernel, che prevede l'aggregazione di un ulteriore kernel
piccolo → CHE: gestisce e filtra le interruzioni

Quando i task RT eseguono sull'executive, mentre quelli non RT,
sul kernel normale, possiamo evitare le soluzioni

ho SVANTAGGIO: I processi RT non possono usare direttamente
le system call, bisogna sempre affidarsi ad
un processo sul kernel normale

Ed inoltre esegue sempre a livello kernel

Risulta molto efficiente e prevedibile

Linux vede solo come un processo che ha una priorità più bassa
quando esegue solo quando non lo task RT

Tra questi meccanismi si usa: RTAI

che vengono anche usati: RT Linux e XENOMAI

RICORDA: Un mondo non può usare bene l'altro, bisogna usare meccanismi appositi

RTAI, fa anche in modo che le istruzioni che disabilitano le interruzioni, sono fatte sempre dal kernel minore che non manola le interruzioni del kernel maggiore

→ Quindi: bisogna comunque modificare il kernel

Non è manutenuta direttamente all'hardware
ma passa prima per il kernel minore

L'unico componente di linux che parla è HAL - Hardware Abstraction Layer

→ Si interagisce con l'hardware, dunque ne esistono numerose per ogni architettura

È quello modello della
Padan di RT

RTAI, permette task RT di livello utente

Venne aggiunto un nuovo modo, per i task RT che vengono sovrapposti
implementa anche uno scheduler apposito

PROBLEMI:

Non uso direttamente i servizi del kernel

Bürokrat & Problematiker

In Preemptible RT, le int. vengono eseguite dopo gli scheduling RT

Nella compilazione del Kernel, posso scegliere quali meccanismi di preemption usare

Preemptible RT, rimuove gli spinlock, per permettere di interrompere il Kernel in ogni momento