

# LEZ ESERCITATIVA 1

- Installazione della macchina virtuale, con comando di base

**ATTENZIONE** A causa della Virtualizzazione, non è sempre possibile conoscere le tempi d'esecuzione precise

→ Infatti: Il sistema host può sempre interrompere la macchina virtuale

Ottengo prestazioni migliori facendo eseguire direttamente sull'hardware

## ALCUNE PRIMITIVE FONDAMENTALI



La maggior parte appartengono ad uno standard  
**POSIX**

**Nota:** C'è viene usato per una programmazione funzionale

Molto usato in sistemi embedded e microcontrollo



→ **QUESTO:** È una libreria (insieme di funzioni) che definisce l'interfaccia di programmazione in UNIX



**PERMETTE:** una maggiore portabilità del codice, tra sistemi differenti

**Nota:** È uno degli aspetti che Linux riferisce da questo sistema

**Lo STANDARD** Specifica le primitive per

- Programmazione concorrente
- Mutua esclusione
- Sincronizzazione con C.V
- Code di messaggi

# RT-POSIX

→ **QUESTO:** È un'estensione dello standard **POSIX** usato nei sistemi Real-Time.

**SPECIFICA**, oltre le primitive precedenti, anche **servizi** che permettono di garantire una **prevedibilità Temporale** del sistema.

## LA GESTIONE DEL TEMPO

Risulta un **mecanismo fondamentale** per questi sistemi, **per consentire** una **corretta temporificazione**.

**In Questi Sistemi:** Si hanno 2 astrazioni del tempo

- **Clock** → Conta del tempo rispetto ad un **riferimento** → **CHIAMATO: EPOCA**

**Nota:** Può avere una **risoluzione diversa**, in base al sistema  
(Ma si possono anche avere **clock differenti**)

- **TIMER** → È un'astrazione che **genera un evento**



**QUESTI:** Possono essere di 2 tipologie

- **One-Shot**: È disattivato alla scadenza.

**ATTENZIONE**: un'altra distinzione è data da come valgono

programmato

• ASSOLUTO: Rispetto all'epoca

• RELATIVO: Rispetto al **tempo corrente**

• PERIODICO: Scatta continuamente dopo ogni intervallo con un certo periodo  
NOTA: Il timer viene armato specificandone anche un periodo

## LEGGERE L'ORA IN UNIX (NOTA: Devore inserire: <sys/time.h>)

Venne usata la **Primitiva**

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Il **Primo Argomento**: È un parametro **ingresso-uscita**

È una **struttura composta da**

NOTA: Può essere un parametro nullo

```
struct timeval {  
    time_t tv_sec; /* secondi */  
    suseconds_t tv_usec; /* microsecondi */  
};
```

Quindi: Va a restituire **secondi e microsecondi dell'epoca**  
(Eventualmente da **considerare approssimativamente**)

## TIMER IN UNIX

NOTA: E' POSSIBILE avere 3 timer per ogni processo,  
NOTA: Questi vengono **allegati** a **clock diversi**

• **REAL TIME** → Il clock Real-Time del sistema  
chiamato **Wall clock**

OSSIMENTE: In base al  
clock salto, cambia

L'istante in cui scatta il timer

• VIRTUAL  $\Rightarrow$  Collega solo quando il processo è in esecuzione

• PROFILING  $\Rightarrow$  Tiene conto del Tempo di esecuzione + Tempo speso dal SO per quel processo

Nota Bene: Gli ultimi due sono importanti per eseguire una stima del tempo di esecuzione

ATTENZIONE: Ogni Processo ha Un Unico TIMER REAL TIME

Quindi: I thread, dello stesso processo, se lo condividono

ATTENZIONE: Questo può portare a numerose problematiche SPECIALMENTE nella loro gestione e impostazioni

IMPOSTARE UN TIMER

Uso: la Primitiva affissa

```
int setitimer(int which,  
             const struct itimerval *new_value,  
             struct itimerval *old_value);
```

QUESTA: Programma un timer che scade al tempo indicato in "new\_Value" di Xipologa

Può essere usato per Riprogrammare

relo

```
struct itimerval {  
    struct timeval it_interval; /* intervallo per timer periodico*/  
    struct timeval it_value; /* tempo fino alla prossima scadenza*/  
};
```

• Se interval è zero, allora è un timer One-Shot

Nota: "which" esprime la Xipologa del timer

## Allo Scattare:

Venne generato un **affondo segnale**



Nel (ASO) di Ximek real-time, si genera il segnale **SIGALARM**



**Nota:** È Parola di un meccanismo di gestione di Linux

**Dove Essere Gestito**, Ximek ha un affondo timer, cioè una **programmazione asincrona**

## GESTIONE DEL TEMPO / RT-POSIX



Vengono usate delle librerie differenti: `<Xim.h>`  
con una precisione maggiore => (ad es: nanosecondi)

**Uso:** una struttura chiamata **TIMESPEC**

```
struct timespec {  
    time_t tv_sec; // secondi  
    long tv_nsec; // nanosecondi  
}
```

**Nota:** È molto simile alla struttura `timesval` UNIX  
(con una precisione differente)

**TUTTAVIA:** Non presenti funzionalità  
di utilità per la gestione  
del tempo

Quindi: Dove implementare approssimativamente

Ad ESEMPIO: La Funzione che aggiunge un certo numero di milisecondi

```
#define NSEC_PER_SEC 1000000000ULL
void timespec_add_us(struct timespec *t, uint64_t d) {
    d *= 1000;
    t->tv_nsec += d;
    t->tv_sec += t->tv_nsec / NSEC_PER_SEC;
    t->tv_nsec %= NSEC_PER_SEC;
}
```

# LEGGERE ED IMPOSTARE IL TEMPO

RT-Posix mette a disposizione **funzioni a sostituzione** **Considerate**

```
#include <time.h>
```

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

**Inoltre:** vengono diverse tipologie di clock, indicate dal parametro **CLOCK-ID**

**LE TIPOLOGIE DI CLOCK** disponibili sono:

- **Clock-Realtime** (il cui valore può essere cambiato)

- **Clock-Monotonic** → Tempo che viene trascurato da Boot  
**Nota Bene:** Questo non può essere modificato (può solo crescere, per questo è monotonic)

## TIMER RT-POSIX

→ Mette a disposizione un meccanismo più sofisticato ma anche più complesso da usare

**OCCORRE: CREALO ed ARMARLO**

**QUESTO:** far sì che si possano avere più timer per ogni processo

→ **INFATI:** ogni timer creato avrà id differente

## • CREAZIONE

Venne usata

```
int timer_create(clockid_t c_id,  
                 struct sigevent *e, timer_t *t_id)
```



| CUI PARAMETRI involvono:

- **c\_id** ➔ Tipologia di clock da usare

- **e** ➔ molta come **Notificare il chiamante**

- **t\_id** ➔ È un parametro ingresso-uscita che contiene l'<sup>18</sup> identificativo del timer create

ATTENZIONE: È una differenza importante con lo Standard UNIX

Poiché: Posso scegliere quale segnale andare a generare

↳ Ricorda: In UNIX, si generava solo un segnale di tipo **SIGALARM**

## • ARMARE & USO:

```
int timer_settime(timer_t timerid, int flags,  
                  const struct itimerspec *v, struct itimerspec *ov)
```



ATTENZIONE: Anche se è simile a "setitimer" (di UNIX),  
PREVEDE: "nuovi" parametri

- **timerid** ➔ Per indicare quale armare

Poiché: Ogni processo può averne anche più di uno

- **flags** ➔ indica se il timer è **assoluto o relativo**

Ma Non Solo: Anche la struttura usata per il timer è differente

```
struct itimerspec {  
    struct timespec it_interval; /* intervallo */  
    struct timespec it_value;   /* scadenza */  
};
```

FUNZIONI SLEEP ↗ Sono funzioni che fanno addormentare il processo

QUESTE: possono combinare in base alla libreria che andiamo a utilizzare ↗ CAMBIA anche la complessità della funzione che si considera

• In UNIX ↗ È Più semplice

PONTE: molto solo i secondi per un obie dormire o anche i microsecondi

```
#include <unistd.h>  
  
unsigned int sleep(unsigned int seconds);  
int usleep(useconds_t usec);
```

• In RT-POSIX → È una funzione più complessa, con una precisione delle nanosecondi

**INFATTI:** Prende in ingresso un altro numero di parametri fondamentali

```
#include <time.h>
```

```
int nanosleep(const struct timespec *req, struct timespec *rem);
```

• **req** → indica quanto tempo devo dormire

• **rem** → indica il tempo rimasto, nel caso in cui dovesse svegliarmi prima della scadenza

In QUESTO CASO: ha funzione restitutiva "-1"

Oltre QUESTA: È prevista anche un'altra funzione

```
#include <time.h>
```

```
int clock_nanosleep(clockid_t clock_id, int flags,  
const struct timespec *req, struct timespec *rem);
```

**PERMETTE:** Si dormire fin dagli un altro tempo nel futuro

I PARAMETRI sono importanti per

a) indicare un dock a cui riferirsi

b) indicare se è un tempo assoluto (flags)

**Quindi:** req o contiene l'intervallo di tempo o il tempo assoluto

**NOTA:** E quindi sembra ad un timer ma più semplice

**Poiché:** non occorre gestire segnali

**PROGRAMMARE UN TASK PERIODICO**

RICORDA: Un Task ha **parte** ad un certo istante e che si **ripete ugualmente** dopo un certo periodo

Lo considero come un programma, che ha una struttura, composta da

**CICLO + WAIT-NEXT-ACTIVATION + JOB**

```
int main() {
    start_periodic_timer(2000000, 5000);
    while(1) {
        wait_next_activation();
        job_body();
    }
    return 0;
}
```

**Un PROBLEMA:** Come implementare le due funzioni "Start" e "Wait"

**SOLUZIONE 1: SLEEP**, per il tempo rimanente

La funzione **start** viene usata per inizializzare le variabili

**NON È UNA BUONA IDEA:** Si potrebbe creare un **accumulo di ritardo** a causa di un eventuale **preemption**

**2. DURQUE:** non è bene usare operazioni di calcolo relativo che potrebbe portare a questi problemi

**MOLTO:** L'accumulo di ritardo potrebbe generare errori anche nell'esecuzione

**3. GENERANDO:** Un output errato (offre un bloccaggio del Task)

RICORDA: I task vengono sempre schedolati con una **proroga 19** e  
con lo schedule **CFS** ✓

(cioè: Una **proroga media**)