# Programmazione di task periodici in ambiente Linux

Corso di
Progetto e Sviluppo di
Sistemi in Tempo Reale

Marcello Cinque

# Task periodici in ambiente Linux

- Sommario della lezione:
  - Introduzione a Real-Time Posix
  - Astrazioni temporali e Timers
  - Programmazione di task periodici
- Riferimenti
  - L. Abeni. “Periodic Timers in modern OS”
  - https://gitlab.retis.santannapisa.it/l.abeni/ExampleCode/

# POSIX

- Lo standard POSIX (Portable Operating System Interface for UNIX) ha definito l'interfaccia di programmazione per applicazioni in esecuzione su sistemi operativi UNIX.
- La portabilità è quindi a livello di codice sorgente.
- Real Time POSIX (RT-POSIX) è la sua estensione per sistemi real time.

# RT-POSIX

- RT-POSIX è lo standard maggiormente diffuso nell'ambito dei sistemi operativi RT.
- Lo standard specifica le primitive per:
  - Programmazione concorrente
  - Mutua esclusione con priority inheritance
  - Sincronizzazione con condition variables
  - Code di messaggi priorizzate per la comunicazione inter-task
- RT-POSIX specifica anche servizi per garantire un prevedibile comportamento temporale del sistema operativo

# Clock e timer

- **Clock**: astrazione che modella un entità che restituisce il tempo corrento
  - Clock: che ora è?
  - Conta il tempo passato da un certo riferimento temporale detto “epoca” (per es., microsecondi trascorsi dal 1 gennaio 1970)
- **Timer**: astrazione che modella un entità che genera eventi ad un certo istante (interruzioni, segnali)
  - Timer: svegliami al tempo *t*

# Tipi di timer

- **One-shot**:
  - Un timer programmato (armato) con un tempo di scadenza *relativo* rispetto al tempo corrente o *assoluto* (basato su una base temporale, ad es: secondi e nanosecondi trascorsi dall'Epoca). Una volta scaduto, il timer viene disarmato.
- **Periodico**:
  - Un timer armato con un tempo di scadenza iniziale (relativo o assoluto)  e un intervallo di ripetizione. Quando la scadenza iniziale occorre, il timer viene ri-armato con l'intervallo di ripetizione.

# Leggere l'ora nei sistemi UNIX

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

- L'argomento *tv* è di tipo *struct timeval*

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

- E restituisce il numero di secondi e microsecondi dall'epoca
- La funzione e la struttura sono definite in <sys/time.h>

# Timer in UNIX

- Le API standard Unix forniscono tre timer per processo, collegati a tre clock:
  - **Real-time** (`ITIMER_REAL`): il clock real-time di sistema (wall clock)
  - **Virtual** (`ITIMER_VIRTUAL`): passaggio di tempo virtuale, aggiornato solo quando il processo è in esecuzione in modo utente
  - **Profiling** (`ITIMER_PROF`): passaggio di tempo virtuale più il tempo in cui il kernel sta eseguendo per conto del processo passage

  → Solo un timer real-time per processo!

# Settare un timer

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

- Programma un timer per scadere al tempo specificato in *new_value*, di tipo *struct itimerval*: (tutti definiti in <sys/time.h>)

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

- Il parametro `which` specifica il tipo di clock (es., `ITIMER_REAL`: in questo caso il segnale `SIGALARM` viene inviato al processo quando il timer scade)
- Se il parametron `it_interval` è zero (in entrambi i campi di secondi e microsecondi) il timer è one shot, altrimenti è periodico

# Gestione del tempo in RT-POSIX

- La libreria da usare è <time.h>
- I valori temporali sono gestiti tramite la struttura `timespec`, simile a `timeval`, ma basata sui nanosecondi

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

- Le funzioni per leggere e settare il tempo e i timer sono parte dello standard RT-POSIX
- `itimerspec` da usare invece di `itimerval` per i timer
- Non sono presenti funzioni di utilità per la gestione del tempo, e vanno implementate dall'utente

# Un esempio di funzione di utilità

- Aggiungere `d` microsecondi ad una timespec `t`

```
#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;
}
```

- Funzioni simili possono essere implementate per sottrarre, confrontare due timespec, ecc.

# Leggere e modificare il tempo in RT-POSIX

```
#include <time.h>

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

`clock_id` può essere:

- `CLOCK_REALTIME` rappresenta il real-time clock di sistema, supportato da tutte le implementazioni. Il valore di questo clock può essere cambiato con `clock_settime()`
- `CLOCK_MONOTONIC` rappresenta il tempo di sistema a partire dal boot, non può esseere modificato e non è supportato da tutte le implementazioni
- Se il flag `_POSIX_THREAD_CPUTIME` è definito, `clock_id` può assumere il valore `CLOCK_THREAD_CPUTIME_ID`, un clock speciale che misura il tempo di esecuzione del thread chiamante (incrementato solo quando il thread esegue)

# RT-POSIX Timers

- Un processo può creare e far partire più timer
- Un timer che scade genera un segnale configurabile

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

- `c_id` è il clock da usare (CLOCK_REALTIME o CLOCK_MONOTONIC)
- `e` specifica come notificare il chiamante quando il timer scade
- Se la creazione ha successo, l'ID del timer creato è posto in `t_id`

- NOTA: Essendo questi timer strettamente parte di RT-POSIX, il codice va compilato con il flag `-lrt`

# RT-POSIX Timers

- Un timer può essere settato (armed) con:

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

- flags: TIMER_ABSTIME setta il timer come tempo assoluto, altrimenti relativo al tempo del chiamante
- Il funzionamento è analogo a `setitimer`, ma si usa `itmerspec` al posto di `itimerval`

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

# Funzioni di sleep

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
int usleep(useconds_t usec);
```

- Funzioni standard UNIX basate su secondi o micorsecondi

```
#include <time.h>

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

- Funzione RT-POSIX, basata su nanosecondi
- `*req` è il tempo di sleep is the amount of time required to sleep
- Se il thread si sveglia prima che il timer scade (ad es., per via di un segnale), la funzione ritorna -1 e ***`rem`** conterrà il tempo rimanente

# Funzioni di sleep

```
#include <time.h>

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

- Simile a `nanosleep`, ma
  - Permette di specificare il clock, ad es. `CLOCK_REALTIME`
  - Permette di specificare un tempo assoluto fino a quando sospendere il task, se `flags = TIMER_ABSTIME`
  - `*req` contiene l'intervallo di tempo di sleep o il tempo assoluto di risveglio del thread (a seconda del valore del flag)
  - `*rem` viene usato solo se il tempo di sleep è relative (`flags = 0`)

# Programmazione di task periodici

# Struttura di un generico task periodico

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

- Un task periodico che parte dopo 2 secondi e “cicla” ogni 5 ms
- Come possiamo implementare `start_periodic_timer` e `wait_next_activation`?

# Soluzione 1: sleep per il tempo rimanente

- Quando termina il job, dorme per il tempo rimanente fino alla prossimo release time
  - Leggi il tempo corrente
  - Calcola il delay come = next activation time – current time
  - usleep(delay)

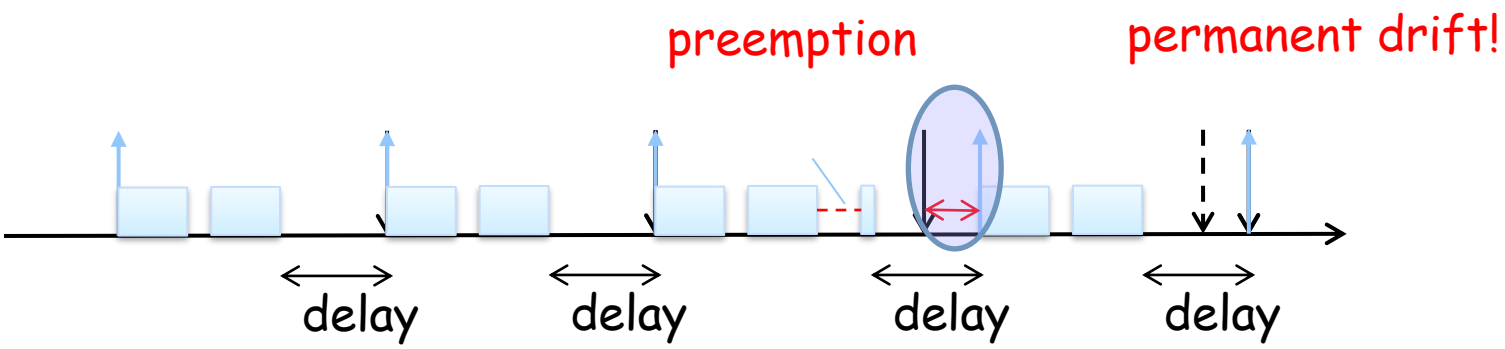
```
static long next_period;
static int period;

void start_periodic_timer(uint64_t offs, int t){
            struct timeval t1;
            gettimeofday(&t1,NULL);
            long now = t1.tv_sec*1000000 +
                        t1.tv_usec;
            next_period = now + offs;
            period = t;
}
```

```
void wait_next_activation(void) {
            struct timeval t1;
            gettimeofday(&t1,NULL);
            long now = t1.tv_sec*1000000 +
                        t1.tv_usec;
            long delay = next_period - now;
            next_period += period;
            usleep(delay);
}
```

# Soluzione 1: non è una buona idea…

- Possono accadere preemption durante `wait_next_activation()` tra la `gettimeofday()` e la `usleep()`, causando un time drift permanente, pericoloso per il task!!



# Soluzione 2: uso dei timer

- Il problema del «relative sleep» può essere risolto utilizzando timer periodici

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

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

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

    signal(SIGALRM, sighand);

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

# Soluzione 2: uso dei timer

- `wait_next_activation` mette in pausa il task
- Il task riceve un segnale `SIGALARM` ogni `period`, dopo il primo offset `offs`,
- Il segnale ha il solo effetto di risvegliare il task tramite un handler vuoto (`sighand`)
- L'handler vuoto può essere evitato usando `sigwait` in `wait_next_activation`, sospendendo il task sul segnale
- PROBLEMi:
  - 1solo  timer per processo!
    - Può essere risolto utilizzando timer RT-POSIX
  - Overhead (e latenza) dovuto alla generazione e handling del segnale
    - Serve una strategia differente
  - I timer possono avere una risoluzione limitata (multipla del tick di sistema, ma non è un problema nei processor moderni)

# Soluzione 2: con timer RT-POSIX

```
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;
        res = timer_create(CLOCK_MONOTONIC, &sigev, &timer);
        if (res < 0) {perror("Timer Create"); exit(-1);}
        return timer_settime(timer, 0 /*TIMER_ABSTIME*/, &t, NULL);
}
```

↑ Evento collegato al timer

# Soluzione 3: clock_nanosleep

- Il problema del «relative sleep» può essere risolto facendo aspettare il task direttamente il release time *assoluto* dell'istanza successiva

```
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;
}
```

  - `timespec r` inizializzata al tempo attuale, ottenuto con la `clock_gettime` a cui si somma l'offset (`offs`) per ottenere la fase

# Soluzione 3: clock_nanosleep

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

- `clock_nanosleep` mette il task in sleep fino al tempo specificato nella variabile `r`
- `r` incrementato di un `period` ogni ciclo

Questa soluzione usa variabili globali, `r` e `period`. Da evitare in generale.