# RTAI e scheduling

Andrea Sambi

# Scheduling Linux

Due politiche di scheduling priority-driven possibili.

**FIFO – priorità uguali**

```
Processo 1 iniziato
Processo 1: iterazione 1
Processo 1: iterazione 2
Processo 1: iterazione 3
Processo 1: iterazione 4
...
Processo 1 terminato
Processo 2 iniziato
Processo 2: iterazione 1
Processo 2: iterazione 2
Processo 2: iterazione 3
...
Processo 2 terminato
Processo 3 iniziato
Processo 3: iterazione 1
Processo 3: iterazione 2
...
Processo 3 terminato
```

**FIFO – prio. diverse**

```
Processo 3 iniziato
Processo 3: iterazione 1
Processo 3: iterazione 2
Processo 3: iterazione 3
Processo 3: iterazione 4
...
Processo 3 terminato
Processo 2 iniziato
Processo 2: iterazione 1
Processo 2: iterazione 2
Processo 2: iterazione 3
...
Processo 2 terminato
Processo 1 iniziato
Processo 1: iterazione 1
Processo 1: iterazione 2
...
Processo 1 terminato
```

**RR – priorità uguali**

```
Processo 1 iniziato
Processo 2 iniziato
Processo 3 iniziato
Processo 1: iterazione 1
Processo 1: iterazione 2
Processo 1: iterazione 3
Processo 2: iterazione 1
Processo 2: iterazione 2
Processo 3: iterazione 1
Processo 3: iterazione 2
Processo 1: iterazione 4
Processo 1: iterazione 5
...
Processo 1 terminato
Processo 2: iterazione 30
Processo 2 terminato
Processo 3: iterazione 30
Processo 3 terminato
```

# Scheduling Linux

- `int sched_setscheduler(<pid>, <policy>, struct sched_param);`
  - Setta l'algoritmo di scheduling e la priorità statica per un processo
  - pid: Process Id del processo di cui modificare la politica di scheduling (0 per il processo chiamante)
  - policy: SCHED_FIFO | SCHED_RR |SCHED_OTHER
  - struct sched_param { int sched_priority; };
- `int sched_getscheduler(<pid>);`
  - Restituisce l'algoritmo di scheduling di un processo
- `int sched_setparam(<pid>, struct sched_param);`
  - Setta la priorità statica di un processo
- `int sched_getparam(<pid>, struct sched_param);`
  - Ricava la priorità statica di un processo
- `int sched_get_priority_min(<policy>);`
  - Restituisce il minimo valore di priorità possibile per un algoritmo di scheduling
- `int sched_get_priority_max(<policy>);`
  - Restituisce il massimo valore di priorità possibile per un algoritmo di scheduling
- `int sched_yield();`
  - Fa assumere al processo chiamante lo stato READY

# RTAI kernel module

- I processi real-time gestiti da RTAI sono entità presenti in spazio kernel
- I processi RTAI non possono comunicare con i processi Linux standard
- Non è possibile utilizzare system calls definite per lo spazio utente
- Un'applicazione RTAI è realizzata come modulo kernel che viene caricato all'inizio dell'esecuzione
- Ogni modulo kernel contiene una funzione di inizializzazione `init_module()` che si occupa di creare i processi RTAI e tutte le strutture necessarie
- Al termine dell'applicazione viene eseguita la funzione `cleanup_module()` che si occupa di liberare la memoria dagli oggetti creati durante l'esecuzione

# Alcune API RTAI

- `int rt_task_init(RT_TASK *task, <funzione>, <data>, <stack_size>, <priorità>, 0, 0);`
  - Crea un processo real-time `task` di priorità <priorità>. <funzione> indica l'entry point della funzione da cui `task` inizia l'esecuzione
  - RT_TASK è la struttura dati che identifica un processo RTAI
- `int rt_task_delete(RT_TASK);`
  - Elimina un processo RTAI creato con `rt_task_init(…)`
- `int rt_task_make_periodic(RT_TASK, <start_time>, <period>);`
  - Rende un processo RTAI periodico ed avvia la prima esecuzione all'istante <start_time>
- `void rt_task_wait_period();`
  - Sospende il processo fino all'inizio del periodo successivo
- `void rt_set_oneshot_mode();`
  - Imposta lo scheduler in One-shot mode
- `void rt_set_periodic_mode();`
  - Imposta lo scheduler in Periodic mode

# Alcune API RTAI

- `RTIME start_rt_timer(<period>);`
  - Fa partire il timer con risoluzione <period>. In modalità one-shot <period> non ha importanza: la risoluzione del timer è quella del clock di RTAI
- `void stop_rt_timer();`
- `RTIME nano2count(<nanoseconds>);`
  - Converte il tempo da ns a numero di ticks del clock interno
- `RTIME rt_get_time_ns();`
  - Restituisce il tempo in ns da quando è stato fatto partire il timer
- `void rt_busy_sleep(<nanoseconds>);`
  - Ritarda l'esecuzione del processo senza perdere l'uso della CPU: simula l'utilizzo della CPU per <nanoseconds> ns
- `int rtai_print_to_screen(…);`
  - Funzione di visualizzazione da utilizzare in modalità hard real-time

# RTAI LXRT

- Permette di eseguire processi real-time in user mode
- I processi eseguono in modalità protetta senza poter accedere liberamente alla memoria
- Permette di avere in esecuzione processi real-time e processi Linux standard. È possibile comunicare tra questi tipi di processi diversi
- Garantisce maggiore portabilità ai processi real-time che non sono dipendenti dalla versione del kernel in uso (al contrario dei moduli kernel)

# RTAI LXRT

- Un processo RTAI LXRT nasce come un processo Linux standard
- Fino a quando il processo lavora in modalità non real-time è possibile utilizzare system calls di Linux
- Viene creato un agente real-time in spazio kernel che assisterà il processo nell'esecuzione di servizi in tempo reale
- Passando alla modalità di funzionamento real-time vengono disabilitate le interruzioni per Linux e vengono serviti gli agenti relativi ai processi in modalità real-time dallo scheduler RTAI
- Una chiamata a system call di Linux durante il funzionamento real-time di un processo provoca la "rottura" della modalità real-time ed il suo ritorno allo stato di processo Linux standard

# RTAI LXRT

Cosa deve fare un processo LXRT per funzionare correttamente?

- Settare la sua politica di scheduling a Fifo
  - Permette di avere una modalità di funzionamento "soft real-time" anche mentre lavora come processo Linux standard: migliori performance
  - È possibile utilizzare le system calls native di Linux
- Creare il collegamento con un agente real-time
  - L'agente creato in spazio kernel gestirà le system calls rese disponibili da RTAI
- Prima di entrare in modalità hard real-time occorre disabilitare la paginazione delle memoria RAM
  - Il funzionamento real-time non può essere interrotto da accessi su disco in maniera non deterministica
  - Codice e dati del processo sono mantenuti in RAM

# RTAI LXRT

- Attivare la modalità di funzionamento hard real-time
  - Il processo viene tolto dalla running-queue di Linux e viene schedulato dallo scheduler RTAI
  - I processi Linux vengono eseguiti solo se non c'è nessun processo schedulato dallo scheduler RTAI in esecuzione
- Esecuzione di codice real-time
- Ritorno alla modalità non real-time (o soft real-time)
  - Il processo è riportato nella coda dei processi pronti dello scheduler Linux
- Prima di terminare l'esecuzione occorre eliminare l'agente RTAI relativo al processo in esecuzione

# API RTAI LXRT

```
#include <rtai_lxrt.h>
```

- `void rt_allow_nonroot_hrt();`
  - Consente di effettuare operazioni normalmente permesse solo ad utenti con privilegi di root
- `RT_TASK* rt_task_init(<name>, <priorità>, 0, 0);`
  - Crea un agente real-time relativo al processo chiamante. La priorità definita dalla funzione riguarda l'esecuzione gestita dallo scheduler RTAI
- `void* rt_get_adr(<name>);`
  - Restituisce l'indirizzo dell'oggetto di nome <name>
- `void rt_make_hard_real_time();`
  - Forza l'esecuzione del processo in modalità hard real-time permettendo full-preemption del sistema
- `void rt_make_soft_real_time();`
  - Ritorna alla modalità non real-time restituendo il controllo a scheduler e kernel di Linux

# Esempio: un modulo kernel RTAI

```
#include <rtai.h>
#include <rtai_sched.h>

#define PERIOD_NS 1000000

RT_TASK task;

void task_execution(int);

int init_module(void) {
  RTIME now;
  rtai_print_to_screen("INIT MODULE\n");
  start_rt_timer(nano2count(PERIOD_NS));
  rt_task_init(&task, task_execution, 1, 10000, 1, 0, 0);
  now = rt_get_time();
  rt_task_make_periodic(&task, now + nano2count(PERIOD_NS),
                        nano2count(PERIOD_NS));
  return 0;
}
```

# Esempio: un modulo kernel RTAI

```
void task_execution(int task_number) {
  int i;
  /* Inizio codice real-time */
  rtai_print_to_screen("Inizio task %d\n", task_number);
  for(i = 1; i <= 30; i++) {
    rt_task_wait_period();
    rtai_print_to_screen("Inizio job %d\n", i);
    rt_busy_sleep(1000);
    rtai_print_to_screen(“Fine job %d\n", i);
  }
  rtai_print_to_screen("Fine task %d\n", task_number);
  /* Fine codice real-time */
}

void cleanup_module(void) {
  stop_rt_timer();
  rt_task_delete(&task);
  rtai_print_to_screen("FINE\n");
}
```

# Esempio: un processo LXRT

```
#include <unistd.h>
#include <sched.h>
#include <rtai_lxrt.h>
#include <sys/mman.h>

#define PERIOD_MS 1
#define MS_IN_NS 1000000
#define PERIOD_NS PERIOD_MS*MS_IN_NS
#define SEC_IN_NS 1000000000

int main (void) {
  RT_TASK *task;
  struct sched_param priorita;
  int i;

  rt_allow_nonroot_hrt();
  rtai_print_to_screen("INIZIO LXRT\n");
  priorita.sched_priority = sched_get_priority_max(SCHED_FIFO);
  sched_setscheduler(0, SCHED_FIFO, &priorita);
```

# Esempio: un processo LXRT

```
  task = rt_task_init(nam2num("Name"), 1, 0, 0);
  mlockall(MCL_CURRENT | MCL_FUTURE);
  start_rt_timer(nano2count(PERIOD_NS));
  rt_make_hard_real_time();
  rt_task_make_periodic(task, now + nano2count(PERIOD_NS),
                        nano2count(PERIOD_NS));
  /* Inizio codice real-time */
  for(i = 1; i <= 30; i++) {
    rt_task_wait_period();
    rtai_print_to_screen("Inizio job %d\n", i);
    rt_busy_sleep(SEC_IN_NS);
    rtai_print_to_screen("Fine job %d\n", i);
  }
  /* Fine codice real-time */

  rt_make_soft_real_time();
  stop_rt_timer();
  munlockall();
  rt_task_delete(task);

  return 0;
}
```

# Scheduling RTAI

- Due politiche di scheduling native di RTAI
  - FIFO: il processo alla massima priorità esegue fino alla terminazione
  - Round Robin: c'è un limite al tempo di esecuzione di un processo, dopodiché cede il controllo della CPU ad un processo al suo stesso livello di priorità
- Scheduling priority-driven in cui le priorità sono assegnate manualmente ai processi

Come è possibile implementare algoritmi di scheduling più avanzati?

# Scheduling RTAI: RM

- Rate Monotonic
  - È di facile realizzazione a partire dallo scheduling FIFO
  - Occorre determinare le priorità dei processi in funzione della loro frequenza di attivazione
  - Se non si vuole implementare l'intero meccanismo manualmente RTAI mette a disposizione una primitiva che determina la priorità dei processi real-time presenti nel sistema in funzione del periodo impostato con `rt_task_make_periodic(…)`
- `void rt_spv_RMS(<cpu>);`
  - Setta la priorità dei processi RTAI che eseguono sul processore <cpu> in maniera inversamente proporzionale al loro periodo di attivazione

# Esempio: scheduling RM

- Creare i processi real-time definendone le priorità (anche se saranno sovrascritte)
  - `rt_task_init(&taskHighFrequency, task_execution, “HF”, 10000, LOW_PR, 0, 0);`
  - `rt_task_init(&taskLowFrequency, task_execution, “LF”, 10000, HIGH_PR, 0, 0);`
- Rendere i processi periodici
  - `rt_task_make_periodic(&taskHighFrequency, now + …, SHORT_PERIOD_COUNT);`
  - `rt_task_make_periodic(&taskLowFrequency, now + …, LONG_PERIOD_COUNT);`
- Settare le priorità in modo inversamente proporzionale alla durata del periodo
  - `rt_spv_RMS(0);`

# Esempio: scheduling RM con LXRT

- Creare un processo figlio
  - `fork();`
- Creare un agente real-time per ogni processo
  - `rt_task_init(nam2num("TaskHF"), LOW_PR, 0, 0);`
  - `rt_task_init(nam2num("TaskLF"), HIGH_PR, 0, 0);`
- Entrare in modalità real-time
  - `rt_make_hard_real_time();`
- Rendere i processi periodici
  - `rt_task_make_periodic(taskHighFrequency, now + …, SHORT_PERIOD_COUNT);`
  - `rt_task_make_periodic(taskLowFrequency, now + …, LONG_PERIOD_COUNT);`
- L'ultimo processo che setta la sua periodicità deve impostare le priorità coerentemente con la politica RM
  - `rt_spv_RMS(0);`

# Esempio: RM

**RATE MONOTONIC SCHEDULING**

```
Inizio Task HF
Fine Task HF
Inizio Task LF
Fine Task LF
Inizio Task HF
Fine Task HF
Inizio Task HF
Fine Task HF
Inizio Task LF
Inizio Task HF
Fine Task HF
Fine Task LF
Inizio Task HF
Fine Task HF
Inizio Task LF
Fine Task LF
Inizio Task HF
Fine Task HF
```

Il processo ad alta frequenza di esecuzione è in grado di fare preemption su Task LF anche se era stato creato con una priorità inferiore. rt_spv_RMS aggiusta le priorità automaticamente.

# Scheduling RTAI: EDF

- Earliest Deadline First
  - Si ottiene a partire dall'algoritmo di scheduling FIFO
  - Occorre eseguire il processo pronto che deve terminare prima
  - RTAI semplifica l'uso di questa politica di scheduling rendendo disponibile una funzione che rende possibile l'esecuzione del processo con deadline più imminente
- `void rt_task_set_resume_end_times(<resume>, <end>);`
  - Imposta gli istanti assoluti di resume e deadline di un processo. Esso viene deschedulato ed al risveglio è inserito nella coda dello scheduler in posizione dipendente dall'istante assoluto entro cui deve terminare
  - Non è specificata alcuna periodicità, ma è possibile definire un istante di resume relativo a quello precedente e deadline relativa al resume ottenuto indicando valori negativi di <resume> e <end>

# Esempio: scheduling EDF

- Creare i processi real-time definendone le priorità (anche se saranno inutili)
  - `rt_task_init(&task1, task_execution, "1", 10000, LOW_PR, 0, 0);`
  - `rt_task_init(&task2, task_execution, "2", 10000, HIGH_PR, 0, 0);`
- Rendere i processi periodici (anche se il periodo impostato sarà inutile)
  - `rt_task_make_periodic(&task1, now + …, PERIOD_1);`
  - `rt_task_make_periodic(&task2, now + …, PERIOD_2);`
- Settare gli istanti di attivazione e di deadline per ogni processo
  - `rt_task_set_resume_end_times(now + …, -REL_DEADL_1);`
  - `rt_task_set_resume_end_times(now + …, -REL_DEADL_2);`
- Un processo al termine di ogni suo job deve risettare i propri istanti di attivazione e di deadline
  - `rt_task_set_resume_end_times(-PERIOD_n,-REL_DEADL_n);`

# Esempio: scheduling EDF con LXRT

- Creare un processo figlio
  - `fork();`
- Creare un agente real-time per ogni processo
  - `rt_task_init(nam2num("Task1"), LOW_PR, 0, 0);`
  - `rt_task_init(nam2num("Task2"), HIGH_PR, 0, 0);`
- Entrare in modalità real-time
  - `rt_make_hard_real_time();`
- Rendere i processi periodici
  - `rt_task_make_periodic(task1, now + …, PERIOD_1);`
  - `rt_task_make_periodic(task2, now + …, PERIOD_2);`
- Settare gli istanti di attivazione e di deadline per ogni processo
  - `rt_task_set_resume_end_times(now + …, -REL_DEADL_1);`
  - `rt_task_set_resume_end_times(now + …, -REL_DEADL_2);`
- Un processo al termine di ogni suo job deve risettare i propri istanti di attivazione e di deadline
  - `rt_task_set_resume_end_times(-PERIOD_n,-REL_DEADL_n);`

# Esempio: EDF

**EARLIEST DEADLINE FIRST SCHEDULING**

```
Inizio Task 1
Fine Task 1
Inizio Task 2
Inizio Task 1
Fine Task 1
Inizio Task 1
Fine Task 1
Fine Task 2
Inizio Task 1
Fine Task 1
Inizio Task 2
Inizio Task 1
Fine Task 1
Inizio Task 1
Fine Task 1
Fine Task 2
```

Task 1 definito come LOW_PR, periodo 1 ms

Task 2 definito come HIGH_PR, periodo 4 ms

Deadline relative = Periodo

Il processo Task 1 fa preemption su Task 2 anche se ha priorità statica inferiore. rt_task_set_resume_end_times ordina un processo nella coda dello scheduler in base alla sua deadline assoluta.

# Considerazioni

Come viene gestita la situazione di missed deadline?

- Non si ha nessuna segnalazione di fault (politica As Soon As Possible Execution)
- Se il processo continua ad essere prioritario prosegue nell'esecuzione
- Politiche di rilevazione e di reazione ad un evento missed deadline sono lasciate al programmatore
  - Skip Next Execution Policy: nei processi periodici viene annullata l'esecuzione nel periodo successivo a quello in cui è avvenuta la situazione di overrun

# Documentazione

- RTAI 3.3 User Manual rev. 0.2
- DIAPM RTAI Programming Guide 1.0
- DIAPM RTAI. A hard real-time support for Linux
- www.rtai.org, www.rts.uni-hannover.de/rtai/lxr/source, www.rtai.dk