RTAI Semaphore

Andrea Sambi

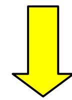
Risorse Condivise

- In ogni sistema le caratteristiche delle applicazioni e le limitate risorse a disposizione inducono la presenza di risorse condivise
- Diversi processi che eseguono in maniera concorrente possono richiedere l'uso delle stesse risorse:
  - perché vi è un numero di risorse inferiore alle necessità (es. una sola stampante per più processi)
  - perché è necessario utilizzare la stessa istanza di una risorsa (es. un unico file aggiornato da più processi)
- L'uso delle risorse condivise può essere libero da qualsiasi vincolo di concorrenza oppure richiedere il controllo degli accessi per soddisfare il vincolo di mutua esclusione

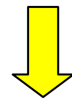
Mutua esclusione

Accesso in mutua esclusione

- Mantenere consistente lo stato delle risorse
- Mantenere coerente il flusso dei processi



Occorrono meccanismi di sincronizzazione.



Soluzione classica: semafori

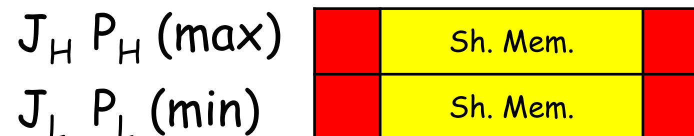
- I processi che cercano di accedere ad una risorsa occupata sono inevitabilmente bloccati

Esempio: nessun controllo degli accessi

Due processi a priorità diversa accedono ad un'area di memoria condivisa (20 caratteri inizializzati a '0').

- Il processo L a bassa priorità sovrascrive tutti i caratteri con la lettera 'L'.
- Il processo H ad alta priorità sostituisce i caratteri 'L' con il carattere 'H'.

Si richiede che un processo che inizi la scrittura sulla memoria condivisa lo faccia in maniera coerente: al termine del processo L tutti i caratteri in RAM devono essere 'L', al termine del processo H non ci deve più essere alcun carattere 'L'.



Esempio: nessun controllo degli accessi

Soluzione senza l'uso di semafori, né di altri meccanismi di sincronizzazione nell'accesso alla memoria condivisa.

```
#define SHM_KEY        101
#define N_CHAR         20
#define HIGH_PRIO      1
#define LOW_PRIO       2

RT_TASK taskH, taskL;
static char * shm_ptr = 0; /* pointer to shared memory */

static void high_prio_code(int arg);
static void low_prio_code(int arg);

int init_module(void) {
  ...
  shm_ptr = rtai_kmalloc(SHM_KEY, N_CHAR * sizeof(char));
  for (i = 0; i < N_CHAR; i++) {
    shm_ptr[i] = '0';
  }
  rt_task_init(&taskH, high_prio_code, 0, 10000, HIGH_PRIO, 0, 0);
  rt_task_init(&taskL, low_prio_code, 0, 10000, LOW_PRIO, 0, 0);
  ...
}
```

Allocazione ed inizializzazione dell'area di memoria condivisa.

Esempio: nessun controllo degli accessi

```
void cleanup_module(void) {
  rt_task_delete(&taskH);
  rt_task_delete(&taskL);
  rtai_kfree(SHM_KEY);
  stop_rt_timer();
  ...
}

static void high_prio_code(int arg) {
  int i, taskL_alert;
  rt_receive(&taskL, &taskL_alert);
  rtai_print_to_screen("TASK H BEGIN\n");
  rtai_print_to_screen("TASK H ACCESSO RAM: write H over L\n");
  for (i = 0; i < N_CHAR; i++) {
    if (shm_ptr[i] == 'L') shm_ptr[i] = 'H';
  }
  rtai_print_to_screen("TASK H END\n");
}
```

Accesso diretto alla RAM.

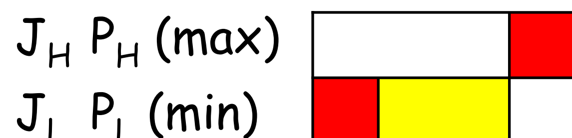
Esempio: nessun controllo degli accessi

```
static void low_prio_code(int arg) {
  int i, taskH_alert = 1;
  rtai_print_to_screen("TASK L BEGIN\n");
  rtai_print_to_screen("TASK L READ\n");
  for (i = 0; i < N_CHAR; i++) {
    rtai_print_to_screen("%c", shm_ptr[i]);
  }
  rtai_print_to_screen("TASK L ACCESSO RAM: write L\n");
  for (i = 0; i < N_CHAR; i++) {
    shm_ptr[i] = 'L';
    if (i == 9) rt_send(&taskH, taskH_alert);
  }
  rtai_print_to_screen("TASK L READ\n");
  for (i = 0; i < N_CHAR; i++) {
    rtai_print_to_screen("%c", shm_ptr[i]);
  }
  rtai_print_to_screen("TASK L END\n");
}
```

Accesso diretto alla RAM.

Esempio: nessun controllo degli accessi

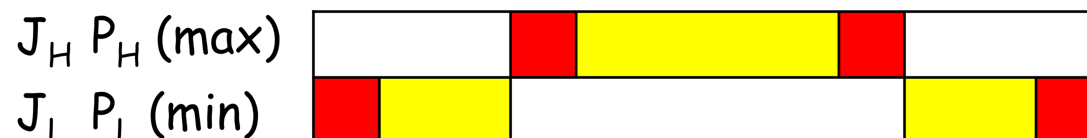
Esempio: J_L accede alla memoria prima dell'attivazione di J_H.



Risultati attesi…

Inizio:	00000 00000 00000 00000
Fine accesso alla memoria J_L:	LLLLL LLLLL LLLLL LLLLL
Termine di J_H (risultato atteso):	HHHHH HHHHH HHHHH HHHHH

…ma…



…Risultato finale: HHHHH HHHHH LLLLL LLLLL

- Le operazioni di scrittura in memoria sono state interrotte da un processo a maggiore priorità
- C'è stata interferenza nel flusso preventivato dei processi
- Il risultato non è stato quello che ci aspettavamo perché non è stato garantito l'accesso in mutua esclusione

Semafori

- I semafori sono un meccanismo di sincronizzazione necessario per qualsiasi S.O.
- Permettono semplice sincronizzazione tra processi per poter lavorare in maniera coerente, ma anche sincronizzazione nell'accesso a risorse condivise
- Un semaforo è realizzato come un contatore:
  - quando un processo richiede l'accesso ad un semaforo se ne decrementa il valore
  - quando un processo rilascia un semaforo se ne incrementa il valore
- Quando il valore del semaforo è 0 i processi che tentano di accedervi vengono bloccati in attesa che il semaforo sia incrementato (che venga rilasciato da qualche processo)
- Un semaforo binario ha valore iniziale 1. È sufficiente un solo processo per azzerare il contatore
  - Posto a protezione di sezioni critiche di accesso a risorse ne garantisce l'uso in mutua esclusione

Semafori

- `void rt_sem_init(SEM* sem, int value);`
  - Inizializza un semaforo `sem` con valore iniziale `value`
- `int rt_sem_delete(SEM* sem);`
- `int rt_sem_wait(SEM* sem);`
  - Richiede che il semaforo sia libero per poter continuare l'esecuzione
  - Se il valore del semaforo è maggiore di 0, lo decrementa di una unità e continua nell'esecuzione
  - Se il valore del semaforo è minore o pari a 0, il processo si blocca in attesa che altri processi ne incrementino il valore
  - Posto all'inizio di sezioni critiche ne permette l'accesso solo se il semaforo è libero
- `int rt_sem_signal(SEM* sem);`
  - Rilascia il semaforo `sem`
  - Incrementa il valore del semaforo di una unità
  - Posto al termine di sezioni critiche libera le risorse utilizzate

Esempio: semaforo binario

La situazione di interferenza nella scrittura in memoria condivisa dell'esempio precedente può essere risolta inserendo un semaforo binario a protezione delle sezioni critiche.

```
...

RT_TASK taskH, taskL;
static char * shm_ptr = 0; /* pointer to shared memory */
SEM sem;

int init_module(void) {
  ...
  rt_sem_init(&sem, 1);
  shm_ptr = rtai_kmalloc(SHM_KEY, N_CHAR * sizeof(char));
  ...
}

void cleanup_module(void) {
  ...
  rtai_kfree(SHM_KEY);
  rt_sem_delete(&sem);
  ...
}
```

Inizializzazione del semaforo binario.

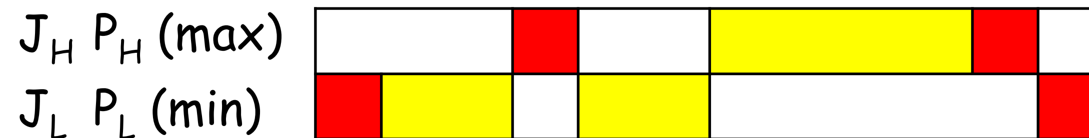
Esempio: semaforo binario

```
static void high_prio_code(int arg) {
 …
  rt_receive(&taskL, &taskL_alert);
  rtai_print_to_screen("TASK H BEGIN\n");
  rt_sem_wait(&sem);
  rtai_print_to_screen("TASK H ACCESSO RAM: write H over L\n");
  for (i = 0; i < N_CHAR; i++) {
    if (shm_ptr[i] == 'L') shm_ptr[i] = 'H';
  }
  rt_sem_signal(&sem);
 …
}

static void low_prio_code(int arg) {
 …
  rt_sem_wait(&sem);
  rtai_print_to_screen("TASK L ACCESSO RAM: write L\n");
  for (i = 0; i < N_CHAR; i++) {
    shm_ptr[i] = 'L';
    if (i == 9) rt_send(&taskH, taskH_alert);
  }
  rt_sem_signal(&sem);
  rtai_print_to_screen("TASK L READ\n");
 …
}
```

Rilascio del semaforo al termine della sezione critica.

Acquisizione del semaforo prima di eseguire la sezione critica.

Esempio: semaforo binario



Risultato finale: HHHHH HHHHH HHHHH HHHHH

- Il processo J_L viene interrotto dal processo più prioritario
- J_H non può entrare in sezione critica perché gli è impedito dal semaforo a protezione della memoria condivisa già in possesso di J_L
- J_L termina la scrittura in RAM prima che J_H sia abilitato ad accedervi: J_H troverà tutte le celle al valore 'L'

Semafori nei sistemi RT

Purtroppo i semafori messi a disposizione dai S.O. non real-time provocano spiacevoli effetti collaterali se utilizzati in sistemi priority-driven.

```
...
#define HIGH_PRIO      1
#define MEDIUM_PRIO    2
#define LOW_PRIO       3

RT_TASK taskH, taskM, taskL;
static char * shm_ptr = 0; /* pointer to shared memory */
SEM sem;

int init_module(void) {
  ...
  rt_sem_init(&sem, 1);
  shm_ptr = rtai_kmalloc(SHM_KEY, N_CHAR * sizeof(char));
  ...
  rt_task_init(&taskH, high_prio_code, 0, 10000, HIGH_PRIO, 0, 0);
  rt_task_init(&taskM, medium_prio_code, 0, 10000, MEDIUM_PRIO,0,0);
  rt_task_init(&taskL, low_prio_code, 0, 10000, LOW_PRIO, 0, 0);
  ...
}
```

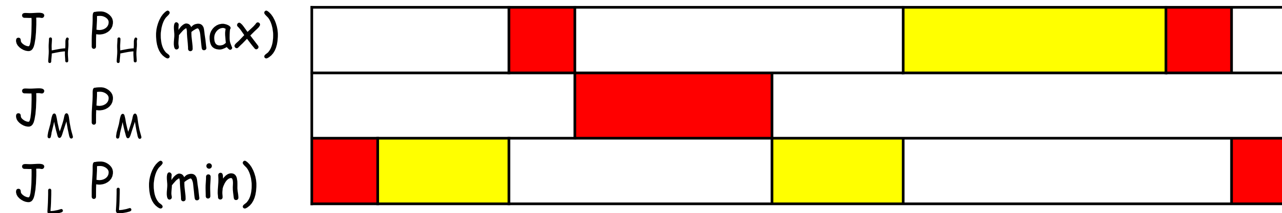
Introduzione di un ulteriore processo a priorità intermedia.

Semafori nei sistemi RT

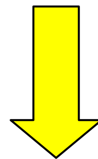
```
static void high_prio_code(int arg) {
 ...
}

static void medium_prio_code(int arg) {
 ...
  rt_receive(&taskL, &taskL_alert);
  rtai_print_to_screen("TASK M BEGIN\n");
  rtai_print_to_screen("TASK M: sto perdendo tempo,blocco taskH\n");
  rtai_print_to_screen("TASK M END\n");
}
static void low_prio_code(int arg) {
 ...
  rt_sem_wait(&sem);
  rtai_print_to_screen("TASK L ACCESSO RAM: write L\n");
  for (i = 0; i < N_CHAR; i++) {
    shm_ptr[i] = 'L';
    if (i == 9) {
      rt_send(&taskH, taskH_alert);
      rt_send(&taskM, task_alert);
  }
  rt_sem_signal(&sem);
  rtai_print_to_screen("TASK L READ\n");
 ...
}
```

Il processo M non fa niente: si inserisce nel flusso di esecuzione durante le scritture in RAM.

Semafori nei sistemi RT



- Il processo J_H si blocca sul semaforo di accesso alla sezione critica
- J_M è il processo pronto a priorità maggiore ed esegue
- J_H a priorità più alta è bloccato sulla risorsa detenuta dal processo a priorità più bassa ed è ritardato dai processi a priorità intermedia



INVERSIONE DI PRIORITÀ INCONTROLLATA

RTAI Semaphore

- Come per altri meccanismi IPC RTAI ha realizzato la propria versione del meccanismo semaforico specifica per processi real-time
- Le primitive fondamentali `wait` e `signal` svolgono le stesse funzioni di quelle fornite da Linux ma hanno alcune proprietà necessarie nei sistemi RT
  - La coda dei processi in attesa su un semaforo può essere ordinata per priorità: la primitiva `signal` sveglia il processo in attesa più prioritario
  - Per evitare il fenomeno dell'inversione di priorità incontrollata le primitive che operano sui semafori realizzano il protocollo d'accesso alle risorse condivise Priority Inheritance

RTAI Semaphore

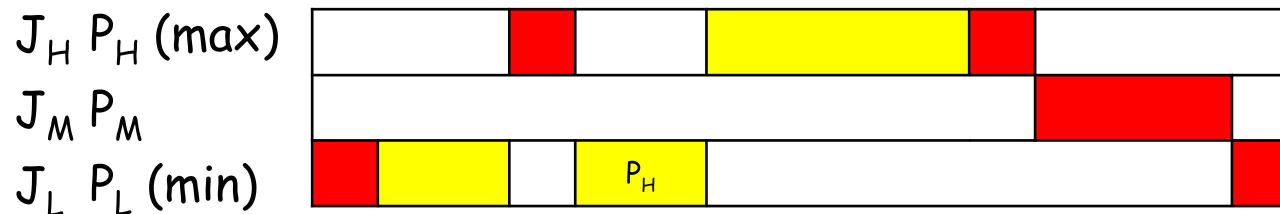
- `void rt_typed_sem_init(SEM* sem, int value, int type);`
- Possono essere creati tre tipi di semafori:
  - `BIN_SEM`: semaforo binario. Stesso funzionamento dei semafori binari Linux
  - `CNT_SEM`: semaforo che può assumere un valore maggiore di 1. Tiene traccia di eventi multipli, permette l'accesso a sezioni critiche a più di un processo per volta. Previsto anche da Linux
  - `RES_SEM`: semaforo specifico per sistemi RT. Implementa l'algoritmo Priority Inheritance: un processo ad alta priorità bloccato da una primitiva `wait` trasferisce la sua priorità al processo che lo sta bloccando

Esempio: ereditarietà della priorità

Un semaforo di tipo RES_SEM lavora con le stesse primitive degli altri tipi di semafori: in base al tipo di semaforo inizializzato il comportamento delle primitive varia.

Occorre solamente definire il semaforo di tipo RES_SEM.

```
int init_module(void) {
  ...
  rt_typed_sem_init(&sem, 1, RES_SEM);
  shm_ptr = rtai_kmalloc(SHM_KEY, N_CHAR * sizeof(char));
  ...
  rt_task_init(&taskH, high_prio_code, 0, 10000, HIGH_PRIO, 0, 0);
  rt_task_init(&taskM, medium_prio_code, 0, 10000, MEDIUM_PRIO,0,0);
  rt_task_init(&taskL, low_prio_code, 0, 10000, LOW_PRIO, 0, 0);
  ...
}
```



Si evitano totalmente inversioni di priorità incontrollate?

RTAI Semaphore

Si evitano totalmente inversioni di priorità incontrollate?

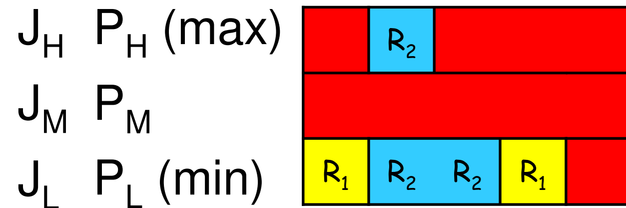
Fino ad RTAI 3.3 NO!

- RTAI 3.3 non implementa completamente il protocollo d'accesso alle risorse condivise Priority Inheritance
- Il protocollo implementato è denominato Adaptive Priority Ceiling
  - L'ereditarietà della priorità è trasferita correttamente dai processi ad alta priorità a quelli a bassa priorità
  - Un processo che ha acquisito contemporaneamente più di una risorsa aggiorna la sua priorità solo quando le ha rilasciate tutte

Da RTAI 3.4 SI'!

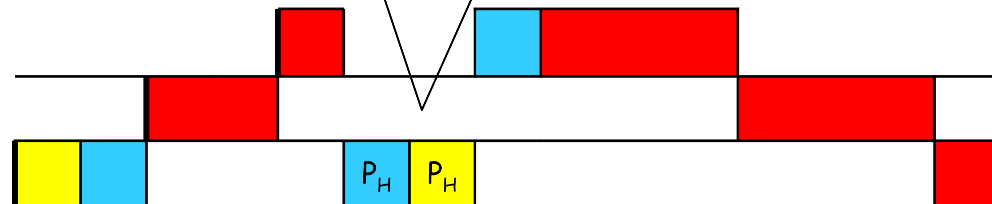
- RTAI 3.4 implementa completamente il protocollo d'accesso alle risorse condivise Priority Inheritance

RTAI Semaphore

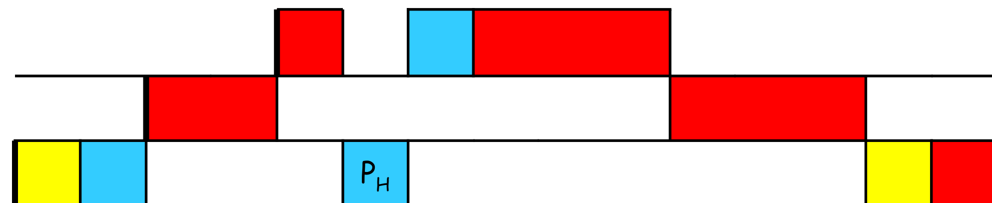


J_L non ritorna alla priorità P_L che gli spetta secondo PI. INVERSIONE DI PRIORITÀ INCONTROLLATA!!

Fino ad RTAI 3.3:
Adaptive Priority Ceiling



Da RTAI 3.4:
Priority Inheritance



Documentazione

- DIAPM RTAI Programming Guide 1.0
- DIAPM RTAI. A hard real-time support for Linux
- Real-Time and Embedded Guide, H. Bruyninckx
- The Real-Time Application Interface, K. Yaghmour
- www.rtai.org, www.rts.uni-hannover.de/rtai/lxr/source, www.rtai.dk