RTAI IPC

Andrea Sambi

Inter-Process Communication

- Inter-Process Communication (IPC) indica la possibilità di scambiare informazioni tra processi attivi
- Per rendere lo sviluppo delle applicazioni il più flessibile possibile è necessario fornire diversi sistemi di comunicazione tra processi
- I meccanismi forniti da Linux per la comunicazione tra processi non sono disponibili in spazio kernel
- Le system calls di Linux per la comunicazione tra processi non considerano le problematiche dei sistemi real-time: assicurare determinismo nell'esecuzione evitando bloccaggi inutili, rispettare i vincoli imposti dalle priorità dei processi

RTAI IPC

RTAI realizza propri meccanismi di interazione tra processi

- FIFO
- Shared Memory
- Message Passing e RPC
- Mailbox

- Permettono la comunicazione sia tra processi real-time sia tra processi Linux standard
- I meccanismi di comunicazione sono realizzati come moduli kernel indipendenti: devono essere caricati in memoria solo al momento in cui vi sono processi che ne fanno uso
- La disponibilità del codice sorgente consente la personalizzazione dei meccanismi

RTAI FIFO

- È il primo meccanismo di comunicazione di cui RTAI fornisce una versione specializzata per i processi real-time
- È un buffer di lettura/scrittura unidirezionale non bloccante che permette il trasferimento asincrono di dati
- I dati sono letti in maniera sequenziale dalla ipotetica coda in cui sono inseriti al momento della scrittura: nessun dato può essere sovrascritto
- Corrisponde ad un file del tipo /dev/rtfn e deve essere acceduto dai processi real-time tramite le API di RTAI. I processi standard vi accedono con le primitive previste da Linux per l'accesso ai file
- La FIFO non ha conoscenza del significato e della lunghezza dei dati che sta trasferendo: la semantica del trasferimento è a carico dei processi utilizzatori

RTAI FIFO

- `int rtf_create(unsigned int fifo, int size);`
  `int rtf_open_sized(const char *device, int permission, int size);`
  - Creano una RTAI FIFO di dimensione <size> rispettivamente da processi real-time e da processi standard. Restituiscono l'identificatore della FIFO
- `int rtf_destroy(unsigned int fifo);`
- `int rtf_put(unsigned int fifo, void *buf, int count);`
  - Scrive un blocco di byte di dimensione <count> presenti nel buffer puntato da <buf> sulla FIFO <fifo>
  - Nei processi standard si usa la primitiva `write` di Linux
- `int rtf_get (unsigned int fifo, void *buf, int count);`
  - Legge dalla FIFO <fifo> un blocco di byte di dimensione <count> che inserisce nel buffer puntato da <buf>
  - Nei processi standard si usa la primitiva `read` di Linux

RTAI FIFO

- `int rtf_create_handler(unsigned int fifo, int (*handler)(unsigned int fifo));`
  - Definisce una funzione che gestisce qualsiasi cambiamento avviene sulla FIFO. Permette di evitare interrogazioni a polling sulla FIFO e di reagire automaticamente alle operazioni effettuate da altri processi su di essa. Alla funzione <handler> è passato l'identificativo della FIFO che l'ha scatenata
- Un utilizzo tipico delle FIFO consiste nel passare ad un processo Linux dati acquisiti da un processo RTAI

Esempio: RTAI FIFO

Un processo real-time attiva lo speaker ad una frequenza fissa.
Da processo utente è possibile inviare comandi al processo RTAI per accendere/spegnere l'audio o modificarne la frequenza.

```
int init_module(void) {
  ...
  rtf_create(RTF_COMMAND, RTF_SIZE);
  rtf_reset(RTF_COMMAND);
  rtf_create(RTF_STATUS, RTF_SIZE);
  rtf_reset(RTF_STATUS);
  rtf_create_handler(RTF_COMMAND, fifo_handler);
  rt_task_init(&rt_task, task_code, 0,1024,RT_LOWEST_PRIORITY, 0,0);
  ...
  task_period = nano2count(1e7); /* start at 100 Hz */
  rt_task_make_periodic(&rt_task, rt_get_time(), task_period);
  ...
}

void cleanup_module(void) {
  rt_task_delete(&rt_task);
  rtf_destroy(RTF_COMMAND);
  ...
}
```

FIFO di ricezione dei comandi.

FIFO di invio dello stato: fa l'eco dei comandi ricevuti.

Esempio: RTAI FIFO

```
static void task_code(int arg){
  ...
  while(1) {
    /* emette un suono sullo speaker */
    rt_task_wait_period();
  }
}

static int fifo_handler(unsigned int fifo) {
  ...
  rtf_get(RTF_COMMAND, &command, sizeof(command));
  if (command.command == SOUND_ON) {
    enable_sound = 1;
  } else if (command.command == SOUND_OFF) {
    enable_sound = 0;
  } else if (command.command == SOUND_FREQ) {
    freq = command.freq > 0 ? command.freq : 1;
    task_period = nano2count(1e9 / freq);
    rt_task_make_periodic(&rt_task, rt_get_time(), task_period);
  }
  ...
}
```

Cambia la frequenza con la quale gira il processo che emette il suono.

Esempio: RTAI FIFO

Processo utente Linux

```
int main() {
  ...
  command_fd = open(RTF_COMMAND_USR, O_WRONLY);
  status_fd = open(RTF_STATUS_USR, O_RDONLY);
  ...
  while(...) {
    fgets(buffer, BUFFERLEN, stdin);
    if (!strncmp(buffer, "on", 2)) {
      command.command = SOUND_ON;
    } else if (! strncmp(buffer, "off", 3)) {
      command.command = SOUND_OFF;
    } else if (1 == sscanf(buffer, "%d", &freq)) {
      command.command = SOUND_FREQ;
      command.freq = freq;
    }
    ...
    write(command_fd, &command, sizeof(command));
    ...
  }
}
```

Attende l'inserimento dei comandi e li invia al processo real-time.

RTAI Shared Memory

- Si definisce un blocco di memoria che può essere letto e scritto da qualsiasi processo attivo nel sistema
- Prevede comunicazione asincrona tra processi
- Rispetto al meccanismo FIFO permette di trasmettere grandi quantità di dati in un unico passaggio
- Si accede alla memoria direttamente tramite un puntatore al primo indirizzo allocato
- I dati non sono accodati ma sovrascritti: occorre garantire accesso in mutua esclusione per mantenere l'integrità dei dati
- Il meccanismo di memoria condivisa viene realizzato tramite un file (/dev/rtai_shm)

RTAI Shared Memory

- `void *rtai_malloc_adr(void *adr, unsigned long name, int size);`
  `void *rtai_malloc(unsigned long name, int size);`
  `void *rtai_kmalloc(unsigned long name, int size);`
  - Allocano un blocco di memoria per essere condiviso tra processi real-time e process standard
  - `rtai_kmalloc` è utilizzabile da moduli kernel, mentre `rtai_malloc` da processi Linux
  - Le funzioni restituiscono il puntatore all'indirizzo iniziale della memoria allocata. L'allocazione di un blocco di memoria viene effettuata solo una volta dal primo processo che lo richiede

RTAI Shared Memory

- `void rtai_free(unsigned long name, void *adr);`
  `void rtai_kfree(void *adr);`
  - Libera un'area di memoria condivisa allocata in precedenza
  - `rtai_free` è utilizzabile dai processi Linux standard, mentre `rtai_kfree` è la corrispondente funzione per i moduli kernel
  - La memoria viene effettivamente liberata solo quando tutti i processi che la condividevano ne hanno chiesto la deallocazione

RTAI Message e RPC

- È un semplice meccanismo di scambio messaggi punto-a-punto: mittente e destinatario devono conoscersi
- Nella sua versione base la dimensione dei messaggi è limitata a 4 byte
- Le code in cui sono depositati i messaggi in attesa di essere consumati sono ordinate secondo la priorità del messaggio (del mittente)
- L'invio di messaggi blocca per il mittente fino a quando il messaggio viene ricevuto. Nel caso di RPC il mittente attende fino al ricevimento della risposta del destinatario

RTAI Message e RPC

- `RT_TASK* rt_send(RT_TASK* task, unsigned int msg);`
  - Invia il messaggio `msg` al processo `task`
- `RT_TASK* rt_receive(RT_TASK* task, unsigned int *msg);`
  - Riceve il primo messaggio disponibile inviatogli dal processo `task`. In caso non ve ne siano il processo chiamante si blocca in attesa.
  - Se `task` vale 0 il processo chiamante accetta il messaggio più prioritario inviatogli da qualsiasi processo

RTAI Message e RPC

- `RT_TASK *rt_rpc(RT_TASK *task, unsigned int msg, unsigned int *reply);`
  - Fa una richiesta di procedura remota: invia il messaggio `msg` al processo `task`, attende una risposta da depositare all'indirizzo `reply`
  - Attende fino al ricevimento della risposta
- `RT_TASK *rt_return(RT_TASK *task, unsigned int result);`
  - Invia `result` in risposta alla richiesta RPC effettuata dal processo `task`

Esempio: RTAI Shared Memory e Message

Un processo modifica un'area dati condivisa, invia un messaggio ad un processo prioritario in attesa che, svegliatosi, sovrascrive la memoria.

```
#define SHM_KEY         101
#define N_CHAR          20
...

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

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: RTAI Shared Memory e Message

```
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 READ\n");
  for (i = 0; i < N_CHAR; i++) {
    rtai_print_to_screen("%c", shm_ptr[i]);
  }
  rtai_print_to_screen("TASK H END\n");
}
```

Attesa bloccante ("taskH" si sospende) di un messaggio dal processo "taskL".

Accesso diretto alla RAM. Non vi è alcuna protezione dell'area di memoria condivisa.

Esempio: RTAI Shared Memory e Message

```
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';
  }
  rtai_print_to_screen("TASK L READ\n");
  for (i = 0; i < N_CHAR; i++) {
    rtai_print_to_screen("%c", shm_ptr[i]);
  }
  rt_send(&taskH, taskH_alert);
  rtai_print_to_screen("TASK L END\n");
}
```

Accesso diretto alla RAM.

Invio di un dato al processo "taskH".

Inizio:	00000 00000 00000 00000
Fine accesso alla memoria "taskL":	LLLLL LLLLL LLLLL LLLLL
Fine accesso alla memoria "taskH":	HHHHH HHHHH HHHHH HHHHH

RTAI Mailbox

- È il meccanismo IPC più flessibile che RTAI mette a disposizione
- Permette ai processi di inviare messaggi che sono automaticamente memorizzati ordinati per priorità e letti al momento del bisogno
- I messaggi possono avere dimensione qualsiasi: messaggi troppo lunghi per una mailbox possono essere spediti anche parzialmente
- Più produttori e più consumatori di messaggi possono essere connessi ad una mailbox
- Rispetto al meccanismo FIFO permette una comunicazione simmetrica tra processi

RTAI Mailbox

- `int rt_typed_mbx_init(MBX *mbx, int size, int qtype);`
  - Inizializza una mailbox puntata da `mbx` di dimensione `size`
  - `qtype` specifica l'ordinamento della coda: per priorità o Fifo
- `int rt_mbx_init(MBX* mbx, int size);`
  - Inizializza una mailbox con la coda dei messaggi ordinata per priorità
- `int rt_mbx_delete(MBX* mbx);`
- `int rt_mbx_send(MBX* mbx, void* msg, int msg_size);`
  - Invia un messaggio `msg` alla mailbox `mbx`. Il processo mittente è bloccato fino alla completa trasmissione del messaggio
- `int rt_mbx_receive(MBX* mbx, void* msg, int msg_size);`
  - Riceve un messaggio di dimensione `msg_size` dalla mailbox. Il processo ricevente è bloccato fino al termine della ricezione
- Esistono primitive di send e receive con semantica diversa: con timeout oppure non bloccanti

RTAI IPC

- I meccanismi IPC non sono caratteristiche speciali che appartengono solo ai sistemi real-time
- Ogni sistema operativo fornisce meccanismi di comunicazione tra processi: i meccanismi IPC sono fondamentali per lo sviluppo di software in maniera veloce e sicura
- I processi real-time non possono utilizzare i meccanismi IPC forniti da Linux
  - In spazio kernel la maggior parte delle primitive Linux per la comunicazione non sono disponibili
  - Processi RT necessitano di primitive "veloci" (devono introdurre poco overhead)
  - Sono preferibili primitive di comunicazione non bloccanti per evitare di introdurre indeterminismo nell'esecuzione dei processi
  - Può essere necessario mantenere code di messaggi ordinate per priorità

RTAI IPC: usi tipici

- Trasferimento di dati acquisiti in tempo reale. Elaborazione a carico di processi standard.
- Richieste di servizio e sincronizzazione sui risultati.
- Comunicazione di informazioni necessarie ai processi per una corretta esecuzione.

RTAI FIFO

Stream di dati ordinati.
Informazioni di log.
Non adatto a trasferire blocchi di grandi dimensioni.

Messaggi di errore.
Trasferimento informazioni di configurazione.

RTAI IPC: usi tipici

- Trasferimento di dati acquisiti in tempo reale. Elaborazione a carico di processi standard.
- Richieste di servizio e sincronizzazione sui risultati.
- Comunicazione di informazioni necessarie ai processi per una corretta esecuzione.

RTAI Shared Memory

Grandi strutture dati condivise da due o più processi RT. Pochi cambiamenti nei dati delle strutture.

Nessuna necessità di mantenere traccia delle modifiche. I dati sono sovrascritti.

RTAI IPC: usi tipici

- Trasferimento di dati acquisiti in tempo reale. Elaborazione a carico di processi standard.
- Richieste di servizio e sincronizzazione sui risultati.
- Comunicazione di informazioni necessarie ai processi per una corretta esecuzione.

RTAI Message e RPC

Semantica RPC: necessità dei risultati forniti da un altro processo. Ne forza l'esecuzione anche se è a priorità minore.

RTAI IPC: usi tipici

- Trasferimento di dati acquisiti in tempo reale. Elaborazione a carico di processi standard.
- Richieste di servizio e sincronizzazione sui risultati.
- Comunicazione di informazioni necessarie ai processi per una corretta esecuzione.

RTAI Mailbox

Trasferimento dati di maggiori dimensioni rispetto al meccanismo FIFO, possibilità di avere comunicazione bidirezionale e di avere messaggi con priorità (concetto di urgenza di un messaggio).

Persistenza dei messaggi (no sovrascritture) ed importanza nel loro ordine di arrivo.

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