RTLinux HOWTO

Dinil Divakaran `mailto:dinildivakaran@rediffmail.com` 1.1, 2002-08-29

Installazione di RTLinux e come scrivere programmi realtime in Linux. Traduzione a cura di Ettore Benedetti (mantra at elitel.biz)e revisione a cura di Sandro Cardelli (sacarde at tiscali.it).

Indice

1 Introduzione

1.1 Scopo

Questo documento si prefigge di rendere operativo l'utente novizio nella maniera più indolore possibile.

1.2 Chi dovrebbe leggere questo HOWTO

Questo documento è pensato per tutti coloro che vogliono sapere come funziona un kernel realtime. A quelli di voi già familiari con la programmazione dei moduli il documento non sembrerà ostico. Gli altri non si devono preoccupare visto che sono necessarie solo le basi della programmazione dei moduli, che saranno comunque esposte al momento opportuno.

1.3 Ringraziamenti

Prima di tutto vorrei ringraziare il mio advisor, Pramode C. E., per il suo incoraggiamento ed aiuto. Esprimo pure sincero apprezzamento a Victor Yodaiken. Questo documento non sarebbe stato possibile senza tutte le informazioni raccolte sulle svariate pubblicazioni a cui Victor Yodaiken ha contribuito. Sono anche grato a Michael Barabanov per la sua tesi dal titolo Un sistema operativo Realtime basato su Linux.

1.4 Feedback

Qualsiasi dubbio o commento relativo a questo documento è sempre ben accetto. Non abbiate timore di mandarmi un' *email* `mailto:dinildivakaran@rediffmail.com` . Se trovate errori in questo documento segnalatemelo pure, cosicché possa correggere la revisione futura. Grazie.

1.5 Distribution Policy

2 Installare RTLINUX

Il primo passo nella compilazione del kernel RTLinux consiste nello scaricare e decomprimere in /usr/src un kernel *2.2.18* `http://ftp.kernel.org/pub/linux/kernel/v2.2/linux-2.2.18.tar.gz`

pre-patched (solo x86) oppure

2.4.0-test1 `http://ftp.kernel.org/pub/linux/kernel/v2.4/linux-2.4.0-test1.tar.gz`

(x86, PowerPC, Alpha). Mettete inoltre in /usr/src/rtlinux una copia recente del kernel RTLinux (versione 3.0) presa da *www.rtlinux.org* `http://www.rtlinux.org` . (Useremo il carattere $ per rappresentare il prompt).

1. Ora configurate il kernel Linux:

```
$ cd /usr/src/linux
$ make config
        oppure
$ make menuconfig
        oppure
$ make xconfig
```

2. Per costruire l'immagine del kernel digitate:

```
$ make dep
$ make bzImage
$ make modules
$ make modules_install
$ cp arch/i386/boot/bzImage /boot/rtzImage
```

3. Come passo successivo si configura LILO. Digitate le seguenti righe nel file /etc/lilo.conf

```
image=/boot/rtzImage
label=rtl
read-only
root=/dev/hda1
```

   ATTENZIONE: sostituite la voce /dev/hda1 vista sopra col filesystem dove è situata la vostra root directory. La via più semplice per capire quale esso sia consiste nel dare un'occhiata alla voce root= già esistente nel vostro /etc/lilo.conf.

4. Ora riavviate il vostro computer e caricate il kernel RTLinux digitando 'rtl' al prompt di LILO. Quindi spostatevi in /usr/src/rtlinux e configurate RTLinux.

```
$ make config
        oppure
$ make menuconfig
        oppure
$ make xconfig
```

5. Per compilare RTLinux, digitate infine:

```
$ make
$ make devices
$ make install
```

L'ultimo passo creerà la directory:

/usr/rtlinux-xx (xx identifica la versione)

che conterrà la directory predefinita di installazione per RTLinux, necessaria per creare e compilare i programmi d'utente (ovvero essa conterrà gli include files, le utilities e la documentazione). Verrà anche creato un link simbolico:

/usr/rtlinux

che punterà a /usr/rtlinux-xx. Per garantire compatibilità con le versioni future, accertatevi che tutti i vostri programmi per RTLinux usino /usr/rtlinux come percorso predefinito.

NOTA: Se cambiate una qualsiasi delle opzioni del kernel Linux, non dimenticate di fare anche:

```
$ cd /usr/src/rtlinux
$ make clean
$ make
$ make install
```

3 Perché RTLinux

Le motivazioni alla base dello sviluppo di RTLinux possono essere comprese esaminando il modo in cui lavora il kernel standard di Linux. Esso separa l'hardware dai task a livello di utente. Il kernel impiega degli algoritmi di scheduling e assegna ad ogni task una certa priorità in modo tale da fornire in media buone prestazioni o alte velocità di trasmissione dati. A questo scopo, il kernel può sospendere un qualsiasi task a livello utente ogni qualvolta tale task abbia esaurito l'intervallo di tempo assegnatogli. Gli algoritmi di scheduling, assieme ai device driver, alle chiamate di sistema non-interrompibili, alla disabilitazione degli interrupt e alle operazioni di memoria virtuale rendono il kernel meno prevedibile. Si può affermare che essi sono i fattori che impediscono ad un task di operare in realtime.

Potreste avere già familiarità con le prestazioni non-realtime, diciamo, per aver ascoltato musica riprodotta usando 'mpg123' o un qualsiasi altro programma del genere. Dopo aver eseguito tale processo per l'intervallo di tempo pre-determinato, il kernel standard di Linux potrebbe decidere di interromperlo per concedere completamente la CPU ad un altro processo (per es. uno che attiva il server X o Nescape). Come conseguenza, la continuità della musica potrebbe andare perduta. In definitiva, nel tentativo di assicurare a tutti i processi un equo bilanciamento del tempo di CPU, il kernel può impedire che certi eventi vengano gestiti.

Un kernel realtime dovrebbe riuscire a garantire il rispetto dei requisiti temporali del processo sottostante. Il kernel RTLinux raggiunge prestazioni realtime eliminando i fattori di imprevedibilità sopra elencati. Possiamo immaginare il kernel RTLinux come posto fra il kernel standard di Linux e l'hardware. Il kernel di Linux è così portato a scambiare lo strato realtime per l'hardware vero e proprio. Ora l'utente può impostare le priorità di ciascun task, oppure introdurne di nuove. L'utente può ottenere una tempistica corretta per i processi giocando con gli algoritmi di scheduling, le priorità, la frequenza di esecuzione, ecc. Il kernel RTLinux assegna la priorità più bassa al kernel standard di Linux. Col metodo esposto i task di utente saranno così eseguiti in realtime.

Le prestazioni realtime sono ottenute intercettando tutti gli interrupt hardware. Solo per quegli interrupt legati a RTLinux viene subito eseguita la routine di servizio appropriata. Tutti gli altri sono trattenuti e, una volta che il kernel RTLinux sia diventato inattivo, sono passati sotto forma di interrupt software al kernel di Linux. L'eseguibile di RTLinux non è di per sè interrompibile.

I task realtime godono di privilegi (hanno accesso diretto all'hardware) e non fanno uso di memoria virtuale. Essi sono scritti come fossero moduli speciali di Linux che possono essere dinamicamente caricati in memoria. Il codice di inizializzazione dei task realtime imposta un apposita struttura dati e informa il kernel RTLinux dei propri requisiti di durata, vita massima e tempi di rilascio.

RTLinux riesce a co-esistere col kernel Linux perché non gli apporta alcun cambiamento. Attraverso una serie di accorgimenti, esso riesce a convertire il kernel Linux esistente in un ambiente hard realtime senza ostacolare i futuri cambiamenti cui Linux sarà soggetto.

4 Scrivere programmi per RTLinux

4.1 Introduzione alla scrittura di moduli

Ma cosa sono i moduli? Un modulo di Linux non è nient'altro che un file oggetto, di solito creato per mezzo dell'opzione -c di gcc. Di per sè, il modulo è creato compilando un normale file in linguaggio C cui manca la funzione main(). Al suo posto sono presenti un paio di funzioni denominate init_module e cleanup_module:

- init_module() è chiamata quando il modulo è inserito nel kernel. Essa restituisce 0 in caso di successo oppure un valore negativo in caso di problemi.
- cleanup_module() è chiamato appena prima della rimozione del modulo.

Di solito init_module() o registra nel kernel un handler per i motivi più svariati, oppure rimpiazza una funzione del kernel con proprio codice (di solito esso chiamerà a sua volta la funzione originale). La funzione cleanup_module() ci si aspetta rifaccia in senso inverso le azioni compiute da init_module(), cosicché il modulo possa essere rimosso in sicurezza.

Per esempio, se avete scritto un file in C chiamato module.c (con init_module() e cleanup_module() al posto della funzione main()) il codice può essere convertito in un modulo digitando:

```
$ gcc -c {SOME-FLAGS} my_module.c
```

Tale comando crea un modulo chiamato module.o che può essere a questo punto caricato nel kernel per mezzo del comando 'insmod' :

```
$ insmod module.o
```

Nella stessa maniera, per rimuovere il modulo potete impiegare il comando 'rmmod' :

```
$ rmmod module
```

4.2 Creazione di thread in RTLinux

Un' applicazione realtime è di solito composta da svariati 'thread' di esecuzione. I thread sono processi alleggeriti che condividono il medesimo spazio di indirizzamento. In RTLinux, tutti i thread condividono lo spazio di indirizzamento del kernel Linux. Il vantaggio offerto dai thread è che il passaggio dall'uno all'altro è piuttosto efficiente se comparato con un normale passaggio di contesto. Gli esempi che seguono illustrano come il controllo completo dell'esecuzione di un thread possa essere ottenuto usando una serie di funzioni.

4.3 Un programma di esempio

Il miglior modo per capire come lavora un thread è di seguire passo passo un programma realtime. Per esempio, il programma mostrato sotto verrà eseguito una volta al secondo e durante ciascuna iterazione mostrerà il messaggio 'Hello World'.

Il codice sorgente del programma (file - hello.c) :

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

pthread_t thread;

void * thread_code(void)
{
        pthread_make_periodic_np(pthread_self(), gethrtime(), 1000000000);

        while (1)
        {
                pthread_wait_np ();
                rtl_printf("Hello World\n");
        }
```

```
        return 0;
}

int init_module(void)
{
   return pthread_create(&thread, NULL, thread_code, NULL);
}

void cleanup_module(void)
{
   pthread_delete_np(thread);
}
```

Iniziamo da init_module(). Tale funzione richiama pthread_create(). Questi crea un nuovo thread che viene eseguito in contemporanea al thread chiamante. *pthread_create() deve essere chiamata unicamente dal thread nel kernel Linux (per esempio da init_module()).*

```
int  pthread_create(pthread_t  * thread,
                    pthread_attr_t * attr,
                    void * (*thread_code)(void *),
                    void * arg);
```

Il nuovo thread creato è di tipo pthread_t, definito nell'header pthread.h. Tale thread consiste nella funzione thread_code() a cui viene passato *arg* come parametro. Il parametro *attr* serve a specificare quali attributi debbano essere applicati al nuovo thread. Se *attr* è NULL, esso avrà gli attributi predefiniti.

In questo caso thread_code() è chiamato senza parametri. thread_code() è composto da tre spezzoni - inizializzazione, corpo e chiusura.

Nella fase di inizializzazione viene chiamata la funzione pthread_make_periodic_np().

```
int pthread_make_periodic_np(pthread_t thread,
                             hrtime_t start_time,
                             hrtime_t period);
```

pthread_make_periodic_np marca il *thread* come pronto per essere eseguito. Il thread inizierà all'istante *start_time* e sarà invocato a intervalli regolari, di durata pari al valore di *period* , espresso in nanosecondi.

gethrtime restituisce il tempo, sempre in nanosecondi, trascorso dal boot.

```
hrtime_t gethrtime(void);
```

Questo valore non viene mai resettato o modificato. gethrtime ritorna sempre valori monotonicamente crescenti. hrtime_t è un intero con segno a 64 bit.

Attraverso la chiamata della funzione pthread_make_periodic_np(), il thread impone allo scheduler di eseguire periodicamente tale thread ad una frequenza di 1 Hz . Ciò segna la fine della sezione di inizializzazione del thread.

Il ciclo while() inizia con una chiamata alla funzione pthread_wait_np() che sospende l'esecuzione del thread realtime in esecuzione al momento, fino all'inizio del prossimo periodo. Il thread è stato precedentemente marcato per l'esecuzione con pthread_make_periodic_np. Una volta che il thread è richiamato di nuovo, esso esegue il contenuto rimanente del codice dentro il ciclo while() fino a quando non incontri un'altra chiamata a pthread_wait_np().

Visto che non è stata inclusa alcuna via d'uscita dal ciclo, questo thread continuerà ad essere eseguito ad una frequenza di 1Hz. L'unica maniera per interrompere il programma consiste nel rimuoverlo dal kernel col comando rmmod. Questo porta alla chiamata di cleanup_module(), che a sua volta invoca pthread_delete_np() per eliminare il thread e deallocare le sue risorse.

5 Compilazione ed esecuzione

Per poter eseguire il programma hello.c (dopo aver avviato il sistema con rtlinux, ovviamente) dovete seguire i seguenti passi:

1. Compilate il codice sorgente e create un modulo usando il compilatore GCC. Per semplificarvi la vita, comunque, è consigliabile creare un Makefile. In questo modo occorre solo digitare 'make' per compilare il codice.

   Il Makefile può essere creato inserendo le seguenti righe in un file chiamato, appunto, 'Makefile'.

   ```
   include rtl.mk
   all: hello.o
   clean:
       rm -f *.o
   hello.o: hello.c
       $(CC) ${INCLUDE} ${CFLAGS} -c hello.c
   ```

2. Copiate il file rtl.mk nella stessa directory dove si trovano i vostri hello.c e Makefile. rtl.mk è un include file contenente tutti i flag necessari per compilare il codice. Questo file da affiancare al vostro hello.c può essere prelevato dai sorgenti di RTLinux.

3. Per compilare il codice usate il comando 'make'.

   ```
   $ make
   ```

4. Il file oggetto ottenuto deve essere caricato nel kernel, dove verrà eseguito da RTLinux. Per fare ciò usate il comando 'rtlinux' come utente root.

   ```
   $ rtlinux start hello
   ```

Dovreste iniziare a vedere da subito il messaggio di hello.c visualizzato ogni secondo. A seconda di come la vostra macchina è configurata, il messaggio potrebbe apparire o direttamente sulla console oppure eseguendo:

```
$ dmesg
```

Per interrompere il programma occorre rimuoverlo dal kernel. A questo scopo, digitate:

```
$ rtlinux stop hello
```

Un'altra maniera per caricare e rimuovere il modulo consiste nell'usare rispettivamente insmod e rmmod.

L'esempio appena proposto è decisamente semplice. A differenza di ciò che abbiamo appena visto, ci possono essere diversi thread in un programma. Le priorità possono essere impostate al momento della loro creazione ma possono essere anche modificate in un secondo tempo. In aggiunta, possiamo stabilire pure quale algoritmo di scheduling verrà usato. Possiamo addirittura scrivere il nostro algoritmo personale!

Tornando al nostro esempio, potremmo impostare la priorità del thread al valore 1 e scegliere lo scheduling FIFO inserendo le seguenti righe in testa alla funzione thread_code() :

```
        struct sched_param p;
        p . sched_priority = 1;
        pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
```

6 Comunicazione inter-processo

Il programma di esempio visto prima è quel che si dice essere un processo realtime. Comunque non occorre che tutte le parti che compongono un applicativo siano scritte per funzionare in realtime. Di solito, solo quelle sezioni che necessitano di temporizzazioni molto precise sono scritte in tale maniera. Le rimamenti possono essere eseguite in user space. Rispetto ai thread realtime, i processi in user space sono spesso più semplici da scrivere ed eseguire ed è più facile farne il debug.

La comunicazione inter-processo può avvenire in diversi modi. Verrà qui discusso solo il più importante e frequentemente usato: i FIFO realtime.

6.1 FIFO realtime

I FIFO realtime sono code unidirezionali (First In First Out). I dati possono essere scritti da un lato e letti dall'altro da processi differenti. Uno dei due è di solito il thread realtime mentre l'altro si trova in user space.

I FIFO realtime sono semplicemente dei character device (/dev/rtf*) con un numero primario pari a 150. I thread realtime usano numeri interi per riferirsi a ciascun FIFO (per esempio il numero 2 per /dev/rtf2). Si noti che esiste un limite per il numero totale di FIFO. I FIFO devono essere gestiti attraverso apposite funzioni come rtf_create(), rtf_destroy(), rtf_get(), rtf_put() e così via.

Per quanto riguarda invece i processi d'utente, essi vedono i FIFO realtime come normali character device. Di conseguenza possono essere usate le solite funzioni come open(), close(), read() e write().

6.2 Un'applicazione che fa uso di FIFO

Consideriamo in primo luogo un semplice programma in C (di nome pcaudio.c) che riproduce musica (semplicemente due note) per mezzo dell'altoparlante del PC. Per il momento si supponga che per suonare una nota si debba semplicemente scrivere sul character device /dev/rtf3. (In seguito vedremo un processo realtime che manda all'altoparlante ciò che legge da tale FIFO (/dev/rtf3)).

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#define DELAY 30000

void make_tone1(int fd)
{
        static char buf = 0;
        write (fd, &buf, 1);
}

void make_tone2(int fd)
{
```

```
        static char buf = 0xff;
        write (fd, &buf, 1);
}

main()
{
        int i, fd = open ("/dev/rtf3", O_WRONLY);
        while (1)
        {
                for (i=0;i<DELAY;i++);
                make_tone1(fd);
                for (i=0;i<DELAY;i++);
                make_tone2(fd);
        }
}
```

Ora, se il programma sopra esposto (pcaudio.c) è compilato ed eseguito, esso dovrebbe dar luogo ad una serie di suoni corrispondenti ad un'onda quadra. Tuttavia abbiamo prima bisogno di un modulo che legga da '/dev/rtf3' e mandi i dati corrispondenti all'altoparlante del PC. Questo programma realtime può essere trovato fra i sorgenti di rtlinux (/usr/src/rtlinux/examples/sound/). Inserite il modulo sound.o usando il comando 'insmod'.

Visto che abbiamo caricato il modulo che legge dal device, possiamo a questo punto eseguire il nostro programma (compilatelo usando 'gcc' e lanciate il corrispondente 'a.out'). Il processo produrrà dei toni più o meno regolari, almeno nel caso non ci siano altri (impegnativi) processi nel sistema. Tuttavia, se il server X viene fatto partire da un'altra console, si avvertiranno alcune pause nella riproduzione. Peggio ancora, quando un comando 'find' (per un file sotto la directory /usr) viene eseguito, la serie di suoni diverrà completamente distorta. Il motivo di questo fenomeno risiede nel fatto che non stiamo scrivendo dati nel FIFO in realtime.

Verrà illustrato subito come eseguire il processo in realtime, in maniera tale che il suono sia riprodotto senza alcun tipo di disturbo. In primo luogo, convertiamo il programma precedentemente esposto in un programma realtime (nome del file rtaudio.c).

```
#include <rtl.h>
#include <pthread.h>
#include <rtl_fifo.h>
#include <time.h>

#define FIFO_NO 3
#define DELAY 30000
pthread_t thread;

void * sound_thread(int fd)
{
        int i;
        static char buf = 0;
        while (1)
        {
                for(i=0; i<DELAY; i++);
                buf = 0xff;
                rtf_put(FIFO_NO, &buf, 1);
```

```
                for(i=0;i<DELAY;i++);
                buf = 0x0;
                rtf_put(FIFO_NO, &buf, 1);
        }
        return 0;
}

int init_module(void)
{
        return pthread_create(&thread, NULL, sound_thread, NULL);
}

void cleanup_module(void)
{
        pthread_delete_np(thread);
}
```

Se non è già stato fatto prima, caricate il modulo sound.o nel kernel. Compilate il programma visto sopra scrivendo un opportuno Makefile (come descritto in precedenza), quindi ricavatene il modulo 'rtaudio.o'. Prima di inserire questo modulo, un ultima avvertenza. Dovreste notare che il programma esegue un ciclo infinito. Visto che non è stato inserito codice per fermare o interrompere il thread, esso non cesserà mai di operare. In parole povere, l'altoparlante del vostro PC continuerà a produrre il tono indefinitivamente fino a quando non riavvierete il vostro computer.

Cambiamo leggermente il codice (solo la funzione sound_thread()) in modo che il thread stesso gestisca il ritardo fra le note.

```
void * sound_thread(int fd)
{
        static char buf = 0;
        pthread_make_periodic_np (pthread_self(), gethrtime(), 500000000);

        while (1)
        {
                pthread_wait_np();
                buf = (int)buf^0xff;
                rtf_put(FIFO_NO, &buf, 1);
        }
        return 0;

}
```

Questa volta, rimuovendo il modulo col comando 'rmmod' si riesce ad interrompere il processo.

In definitiva abbiamo mostrato come i FIFO realtime possano essere usati per le comunicazioni inter-processo. Inoltre, l'esempio esposto aiuta a comprendere perché sia necessario ricorrere a RTLinux.

7 Passi successivi

In questo documento sono state illustrate le basi della programmazione sotto RTLinux. Una volta afferrati i concetti di base, non dovrebbe essere difficile compiere da soli i passi successivi. Potete esaminare tutti gli esempi che accompagnano i sorgenti di RTLinux. Alla fine dovreste essere in grado di scrivere e testare i vostri moduli. Per ulteriori informazioni riguardanti la programmazione dei moduli potete fare riferimento alla 'Linux Kernel Module Programming Guide' di *Ori Pomerantz* .