RTLinux HOWTO

Table of Contents

RTLinux HOWTO

Dinil Divakaran

1.1, 2002−08−29

RTLinux Installation and writing realtime programs in Linux

---

1. Introduction

1.1 Purpose

This document aims at getting the novice user up and running with RTLinux in as painless a manner as possible.

1.2 Who should read this HOWTO

This document is meant for all those who wish to know the working of a realtime kernel. For those of you already familiar with module programming, the document wouldn't appear as a difficult one. And for others, you needn't worry; since only the basic concepts of module programming are required, which we would indeed discuss, as and when required.

1.3 Acknowledgement

First of all I would like to thank my advisor, Pramode C. E, for his encouragement and help. Also I express my sincere appreciation to Victor Yodaiken. This document would not have been possible without all the information gathered from different papers contributed by Victor Yodaiken. I am also grateful to Michael Barabanov for his thesis on "A Linux−base Real−Time Operating System".

1.4 Feedback

Any doubt or comment about this document, is always welcome. Please feel free to email me. If there is any mistake in this document, please let me know so that I can correct it in the next revision. Thanks.

1.5 Distribution Policy

---

2. Installing RTLINUX

The first step in the compilation of RTLinux kernel, is to download a pre−patched kernel 2.2.18 (x86 only) 2.2.18 (x86 only) or 2.4.0−test1 (x86, PowerPC, Alpha) into /usr/src/ and untar it. Also put a fresh copy of RTLinux kernel (version 3.0) from www.rtlinux.org in /usr/src/rtlinux/. (We will use $ to represent the prompt).

1. Now, configure the Linux kernel :

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

2. For building the kernel image, type :

```
$ make dep
```

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

3. Next step is to configure LILO. Type in the following lines in the file /etc/lilo.conf

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

   WARNING: replace /dev/hda1 in the above with your root filesystem. The easiest way to find out which filesystem it should be, take a look at the existing entry in your /etc/lilo.conf for "root=".
4. Now, restart your computer and load the RTLinux kernel by typing 'rtl' at the LILO prompt. Then 'cd' to /usr/src/rtlinux/ and configure RTLinux.

   ```
   $ make config
           or
   $ make menuconfig
           or
   $ make xconfig
   ```
5. Finally, for compiling RTLinux

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

The last step will create the directory:

/usr/rtlinux−xx (xx denotes the version)

which contains the default installation directory for RTLinux which is needed to create and compile user programs (that is, it contains the include files, utilities, and documentation). It will also create a symbolic link:

/usr/rtlinux

which points to /usr/rtlinux−xx. In order to maintain future compatibility, please make sure that all of your own RTLinux programs use /usr/rtlinux as its default path.

NOTE : If you change any Linux kernel options, please don't forget to do:

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

3. Why RTLinux

The reasons for the design of RTLinux can be understood by examining the working of the standard Linux kernel. The Linux kernel separates the hardware from the user−level tasks. The kernel uses scheduling algorithms and assigns priority to each task for providing good average performances or throughput. Thus the kernel has the ability to suspend any user−level task, once that task has outrun the time−slice allotted to it by the CPU. This scheduling algorithms along with device drivers, uninterruptible system calls, the use of interrupt disabling and virtual memory operations are sources of unpredictability. That is to say, these sources cause hindrance to the realtime performance of a task.

You might already be familiar with the non−realtime performance, say, when you are listening to the music played using 'mpg123' or any other player. After executing this process for a pre−determined time−slice, the standard Linux kernel could preempt the task and give the CPU to another one (e.g. one that boots up the X server or Netscape). Consequently, the continuity of the music is lost. Thus, in trying to ensure fair distribution of CPU time among all processes, the kernel can prevent other events from occurring.

A realtime kernel should be able to guarantee the timing requirements of the processes under it. The RTLinux kernel accomplishes realtime performances by removing such sources of unpredictability as discussed above. We can consider the RTLinux kernel as sitting between the standard Linux kernel and the hardware. The Linux kernel sees the realtime layer as the actual hardware. Now, the user can both introduce and set priorities to each and every task. The user can achieve correct timing for the processes by deciding on the scheduling algorithms, priorities, frequency of execution etc. The RTLinux kernel assigns lowest priority to the standard Linux kernel. Thus the user−task will be executed in realtime.

The actual realtime performance is obtained by intercepting all hardware interrupts. Only for those interrupts that are related to the RTLinux, the appropriate interrupt service routine is run. All other interrupts are held and passed to the Linux kernel as software interrupts when the RTLinux kernel is idle and then the standard Linux kernel runs. The RTLinux executive is itself nonpreemptible.

Realtime tasks are privileged (that is, they have direct access to hardware), and they do not use virtual memory. Realtime tasks are written as special Linux modules that can be dynamically loaded into memory. The initialization code for a realtime tasks initializes the realtime task structure and informs RTLinux kernel of its deadline, period, and release−time constraints.

RTLinux co−exists along with the Linux kernel since it leaves the Linux kernel untouched. Via a set of relatively simple modifications, it manages to convert the existing Linux kernel into a hard realtime environment without hindering future Linux development.

---

4. Writing RTLinux Programs

4.1 Introduction to writing modules

So what are modules? A Linux module is nothing but an object file, usually created with the −c flag argument to gcc. The module itself is created by compiling an ordinary C language file without the main() function. Instead there will be a pair of init_module/cleanup_module functions:

- The init_module() which is called when the module is inserted into the kernel. It should return 0 on success and a negative value on failure.
- The cleanup_module() which is called just before the module is removed.

Typically, init_module() either registers a handler for something with the kernel, or it replaces one of the kernel function with its own code (usually code to do something and then call the original function). The cleanup_module() function is supposed to undo whatever init_module() did, so the module can be unloaded safely.

For example, if you have written a C file called module.c (with init_module() and cleanup_module() replacing the main() function), the code can be converted into a module by typing :

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

This command creates a module file named module.o, which can now be inserted into the kernel by using the 'insmod' command :

```
$ insmod module.o
```

Similarly, for removing the module, you can use the 'rmmod' command :

```
$ rmmod module
```

4.2 Creating RTLinux Threads

A realtime application is usually composed of several ``threads'' of execution. Threads are light−weight processes which share a common address space. In RTLinux, all threads share the Linux kernel address space. The advantage of using threads is that switching between threads is quite inexpensive when compared with context switch. We can have complete control over the execution of a thread by using different functions as will be shown in the examples following.

4.3 An example program

The best way to understand the working of a thread is to trace a realtime program. For example, the program shown below will execute once every second, and during each iteration it will print 'Hello World'.

The Program code (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);
}
```

So, let us start with the init_module(). The init_module() invokes pthread_create(). This is for creating a new thread that executes concurrently with the calling thread. *This function must only be called from the Linux kernel thread (i.e., using init_module()).*

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

The new thread created is of type pthread_t, defined in the header pthread.h. This thread executes the function thread_code(), passing it *arg* as its argument. The *attr* argument specifies thread attributes to be applied to the new thread. If *attr* is NULL, default attributes are used.

So here, thread_code() is invoked with no argument. thread_code has three components − initialization, run−time and termination.

In the initialization phase, is the call to pthread_make_periodic_np().

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

pthread_make_periodic_np marks the *thread* as ready for execution. The thread will start its execution at *start_time* and will run at intervals specified by *period* given in nanoseconds.

gethrtime returns the time in nanoseconds since the system bootup.

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

This time is never reset or adjusted. gethrtime always gives monotonically increasing values. hrtime_t is a 64−bit signed integer.

By calling the function pthread_make_periodic_np(), the thread tells the scheduler to periodically execute this thread at a frequency of 1 Hz. This marks the end of the initialization section for the thread.

The while() loop begins with a call to the function pthread_wait_np(), which suspends execution of the currently running realtime thread until the start of the next period. The thread was previously marked for execution using pthread_make_periodic_np. Once the thread is called again, it executes the rest of the contents inside the while loop, until it encounters another call to pthread_wait_np().

Because we haven't included any way to exit the loop, this thread will continue to execute forever at a rate of 1Hz. The only way to stop the program is by removing it from the kernel with the rmmod command. This invokes the cleanup_module(), which calls pthread_delete_np() to cancel the thread and deallocate its resources.

5. Compiling and Executing

In order to execute the program, hello.c, (after booting rtlinux, ofcourse) you must do the following :

1. Compile the source code and create a module using the GCC compiler. To simplify things, however, it is better to create a Makefile. Then you only need to type 'make' to compile the code. Makefile can

be created by typing in the following lines in a file named 'Makefile'.

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

2. Locate and copy the rtl.mk file into the same directory as your hello.c and Makefile. The rtl.mk file is an include file which contains all the flags needed to compile the code. You can copy it from the RTLinux source tree and place it alongside of the hello.c file.
3. For compiling the code, use the command 'make'.

   ```
   $ make
   ```
4. The resulting object binary must be inserted into the kernel, where it will be executed by RTLinux. Use the command 'rtlinux' (you need to be the 'root' to do so).

   ```
   $ rtlinux start hello
   ```

You should now be able to see your hello.o program printing its message every second. Depending on the configuration of your machine, you should either be able to see it directly in your console, or by typing:

```
$ dmesg
```

To stop the program, you need to remove it from the kernel. To do so, type:

```
$ rtlinux stop hello
```

Alternate ways for inserting and removing the module is to use *insmod* and *rmmod* respectively.

Here, we have made our example program too simple. Contrary to what we have seen, there may be multiple threads in a program. Priority can be set at thread creation or modified later. Also, we can select the scheduling algorithm that should be used. In fact, we can write our own scheduling algorithms!

In our example, we can set priority of the thread as 1, and select FIFO scheduling by inserting the following lines in the beginning of the function, thread_code() :

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

6. Inter−Process Communication

The example program that we have seen above is what is known as a realtime process. Not every part of a application program need be written in realtime. It is found that only that part of a program which requires precise time restrictions should be written as a realtime process. Others can be written and executed in user space. User spaces processes are often easier to write, execute and debug than realtime threads. But then, there should be a way to communicate between user space Linux processes and realtime thread.

There are several ways for inter−process communication. We will discuss the most important and common way of communication − the realtime FIFO.

6.1 Realtime FIFO

Realtime FIFOs are unidirectional queues (First In First Out). So at one end a process writes data into the FIFO, and from the other end of the FIFO, information is read by another process. Usually, one of these processes is the realtime thread and the other is a user space process.

The Realtime FIFOs are actually character devices (/dev/rtf*) with a major number of 150. Realtime threads uses integers to refer to each FIFO (for example − 2 for /dev/rtf2). There is a limit to the number of FIFOs. There are functions such as rtf_create(), rtf_destroy(), rtf_get(), rtf_put() etc for handling the FIFOs.

On the other hand, the Linux user process sees the realtime FIFOs as normal character devices. Therefore the functions such as open(), close(), read() and write() can be used on these devices.

6.2 Application Using FIFO

First, let us consider a simple C program (filename pcaudio.c) to play music (of just two tones) through the PC speaker. For the time being, let us assume that for playing the note, we need only write to the character device /dev/rtf3. (Later, we will see a realtime time process that reads from this FIFO (/dev/rtf3) and sends to the PC speaker)

```
#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);
        }
}
```

Now, if the above shown program (pcaudio.c) is compiled and run, it should create regular sound patters corresponding to a square wave. But before that we need a module for reading from '/dev/rtf3' and sending the corresponding data to the PC speaker. This realtime program can be found at the rtlinux source tree (/usr/src/rtlinux/examples/sound/) . Insert the module sound.o using the command 'insmod'.

Since we have inserted a module for reading from the device, we can now execute our program (compile using 'gcc' and then execute the corresponding 'a.out'. So the process produces somewhat regular tones, when there is no other (time consuming) process in the system. But, when the X server is started in another console, there comes more prolonged silence in the tone. Finally, when a 'find' command (for a file in /usr directory) is executed, the sound pattern is completely distorted. The reason behind this is that, we are writing the data onto the FIFO in non−realtime.

We will, now, see how to run this process in realtime, so that the sound is produced without any kind of disturbance. First, we will convert the above program into a realtime program. (Filename 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);
}
```

If not already done, 'plug in' the module sound.o into the kernel. Compile the above program by writing a Makefile for it (as said earlier), thus producing the module 'rtaudio.o'. Before inserting this module, one more thing. Note that the above program runs into infinite loop. Since, we have not included code for the thread to sleep or stop, the thread will not cease its execution. In short, your PC speaker will go on producing the tone, and you will have to restart your computer for doing anything else.

So, let us change the code a little bit (only in the function sound_thread()) by asking the thread itself to make the delay between tones.

```
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;

}
```

This time we can stop the process by just removing the module by using the 'rmmod' command.

Here, we have seen how realtime FIFOs can be used for inter−process communication. Also the real need for RTLinux can be understood from the above example.

7. What next

This document has gone through the basics of programming in RTLinux. Once you have understood the basic concept it is not difficult to make steps by your own. So you can go through all other examples available along with the rtlinux source. Then you should be able to write modules and test them. For more information regarding module programming, you can refer to 'Linux Kernel Module Programming Guide' by *Ori Pomerantz*.