Real-Time Linux

Michael Barabanov Victor Yodaiken

March 3, 1996

1 Introduction

If you wanted to control a camera or a robot or a scientific instrument from a PC, it would be natural to think of using Linux so that you could take advantage of the development environment, X-windows, and all the networking support. But, Linux cannot reliably run these kinds of *hard real-time* applications. A simple experiment will illustrate the problem. Take a speaker and hook it up to one of the pins from the parallel port. Then run a program that toggles the pin. If your program is the only one running, the speaker will produce a nice somewhat steady tone. Not completely steady, but not bad. When Linux updates the file system every couple of seconds, you might notice a small change in tone. If you move the mouse over a couple of windows the tone becomes irregular. If you start netscape in one of the windows, you will hear intervals of silence as your program waits for higher priority processes to run.

The problem is that Linux, like most general purpose operating systems, is designed to optimize *average* performance and to try to give every process a fair share of compute time. This is great for general purpose computing, but for real-time programming precise timing and predictable performance is more important than average performance. For example, if a camera fills a buffer every millisecond, then a momentary delay in the process reading that buffer may cause data loss. If a stepper motor in a lithography machine must be turned on and off in precise intervals in order to minimize vibration and to move a wafer into position at the correct time — a momentary delay may cause an unrecoverable failure. And consider what might happen if the task that causes an emergency shutdown of a chemistry experiment must wait to run until Netscape redraws the window.

It turns out that redesigning Linux to provide guaranteed performance would take an enormous amount of work. And taking on such a job would defeat our original purpose. Instead of having an off the shelf general purpose OS, we would have a custom made special purpose OS that would not be riding the wave of the main Linux development effort. So what we did was slip a small, simple, real-time operating system *underneath* Linux. Linux becomes a task that runs only when there is no real-time task to run and we pre-empt Linux whenever a real-time task needs the processor. The changes needed in Linux itself are pretty minimal. Linux is mostly

unaware of the real-time operating system as it goes about its business of running processes, catching interrupts, and controlling devices. But real-time tasks can run to a quite high level of precision. In our test P120 system, we can schedule tasks to run within a precision of about 20 microseconds.

Real-time Linux is a research project with two goals. First, we want a practical non-proprietary tool we can use to control scientific instruments and robots. Our other goal is to use RT-Linux for research in real and non-real-time OS design. We'd like to be able to learn something about how to make operating systems efficient and reliable. For example: even a non-real-time operating system should be able to determine whether it can guarantee timing needed for its I/O devices. And we're interested in what types of scheduling disciplines actually turn out to be most useful for real-time applications. Following this dual purpose, in this paper we will discuss both how to use RT-Linux and how it works.

2 Using RT-Linux 2.0.RT.1

Let us consider an example. Suppose we want to write an application that polls a device for data in real-time and stores this data in a file. The main design philosophy behind RT-Linux is the following:

> Real-time programs should be split into small and simple parts with hard real-time constraints, and the larger parts that do more sophisticated processing

Following this principle, we split our application into two parts. The hard-real-time part will execute as a real-time task and will copy data from the device into a special I/O interface called *real-time fifo*. The main part of the program will execute as an ordinary Linux process. This part will read data from the other end of the real-time fifo and will then display and store the data in a file.

The real time component will be written as a kernel module. Linux allows us to compile and load kernel modules without rebooting the system. Code for a module always starts with a define of `MODULE` and an include of the `module.h` file. After that, we include the real-time header files `rt_sched.h` and `rt_fifo.h` and declare a `RT_TASK` structure .

```
#define MODULE
#include <linux/module.h>

/* always needed for real-time tasks */
#include <linux/rt_sched.h>
#include <linux/rt_fifo.h>
RT_TASK mytask;
```

The real-time task structure will contain pointers to code, data, and scheduling information for the this task. The task structure is defined in the first include file. Currently, RT-Linux has only one, pretty simple, scheduler. In the future the schedulers will also be loadable modules. Currently, the only way for real-time tasks to communicate with Linux processes is through special queues called real-time fifos. Real-time fifos have been designed so that the real-time task can never be blocked when it reads or writes data. Figure 1) illustrates real-time fifos.

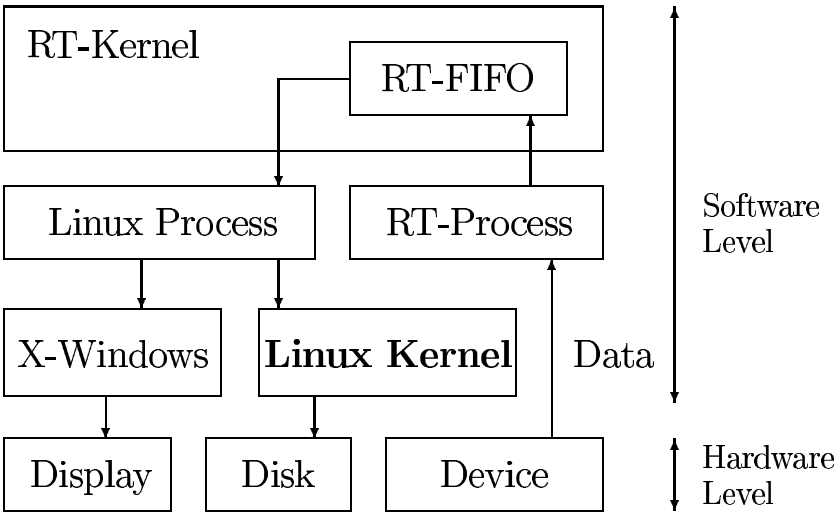

Figure 1: Data Collection Application

The example program will simply loop, reading data from the device, writing the data on a RT-fifo, and waiting for a fixed amount of time.

```
/* this is the main program */
void mainloop(int fifodesc) {
        int data;

        /* in this loop we obtain data from the device and put it */
        /*  into the FIFO number 1 */
        while (1) {
                data = get_data();
                rt_fifo_put(fifodesc, (char *) &data, sizeof(data));
                  /* give up the CPU till the next period */
                rt_task_wait();
        }
```

```
}
```

All modules must contain an initialization routine. The initialization routine for the example real-time task will: record the current time, initialize the real-time task structure, and put the task on the periodic schedule. The `rt_task_init` routine initializes the task structure and arranges for an argument to be passed to the task. In this case, the argument is a fixed descriptor for a real-time fifo. The `rt_make_periodic` routine puts the new task on the periodic scheduling queue. Periodic scheduling means that the task is scheduled to run every so many time units. The alternative is to make the task run only when an interrupt causes it to become active.

```
/* This function is needed in any module. It will be invoked when
/* the module is loaded. */
int init_module(void)
{
        #define RTFIFODESC  1
        RTIME now = rt_get_time();      /* get the current time */

        /* 'rt_task_init' associates a function with the */
        /* RT_TASK structure */
        /* and sets several execution parameters:      */
        /* priority level = 4, stack size = 3000 bytes */
        /* pass 1 to 'mainloop' as an argument */

        rt_task_init(&mytask, mainloop, RTFIFODESC, 3000, 4);

        /* Mark 'mytask' as periodic. */
        /* It could be interrupt-driven as well */
        /* mytask will have period of 25000 time units   */
        /* the first run is in 1000 time units from now  */

        rt_task_make_periodic(&mytask, now + 1000, 25000);
        return 0;
}
```

Linux also requires that any module have a cleanup routine. For a real-time task, we want to make sure that a dead task is no longer scheduled.

```
/* cleanup routine. It is invoked when the module is unloaded. */
void cleanup_module(void)
{
```

```
        /* kill the real-time task */
        rt_task_delete(&mytask);
}
```

That's the end of the module. We also need a program that runs as an ordinary Linux process. In this example, the process will just read data from the fifo and write copy the data to stdout.

```
#include <rt_fifo.h>
#include <stdio.h>

#define RTFIFODESC  1
#define BUFSIZE 10
int buf[BUFSIZE];

int main()
{
        int i;
        int n;

        /* create FIFO number 1 with size of 1000 bytes */
        rt_fifo_create(1, 1000);
        for (n=0; n < 100; n++) {

                /* read the data from the FIFO and print it */
                rt_fifo_read(1, (char *) buf, BUFSIZE * sizeof(int));
                for (i = 0; i < BUFSIZE; i++) {
                        printf("%d ", buf[i]);
                }
                printf("\n");
        }

        /* destroy FIFO number 1 */
        rt_fifo_destroy(1);
        return 0;
}
```

The main program might as well display data on the screen, send it over network, etc. All these activities are assumed to be non-real-time. The FIFO size must be big enough to avoid

overflows. Overflows can be detected, and another FIFO can be used to inform the main program about them.

3 Why Linux can't do real-time and why simple fixes don't

Although Linux has system calls for suspending a process for a given time interval, it does not guarantee that the process will be resumed as soon as this interval has passed. Depending on system load, the process might be scheduled more than a second later. Furthermore, a user process can be preempted at an unpredictable moment and forced to wait for its share of the CPU time. Assigning the highest priorities to critical tasks does not help, partly because of the Linux "fair" time-sharing scheduling algorithm. This algorithm tries to make sure that every user program gets its fair share of computer time. Of course, if we have a real-time task, we want it to get the CPU whenever it needs it, no matter how unfair that may be. Linux virtual memory also contributes to unpredictability. Pages belonging to any user process can be swapped out to disk at any time. Bringing the requested page back to RAM takes an unpredictable amount of time in Linux.

Some of these problems are easily, or semi-easily, fixed. It's possible to create a new class of special Linux processes that are more real-time. We could change the scheduling algorithm so that real-time processes are scheduled round-robin or periodically. We could *lock* a real-time process into memory so that its pages will never be swapped out. In fact, both of these ideas are part of the POSIX.1b-1993 specification which defines standards for "real-time" processes. And POSIX.1b-1993 is being incorporated into Linux. In newer versions of Linux, system calls are already provided for locking user pages in memory, changing the scheduler policy to a priority-based one, and even for a more predictable handling of signals.

POSIX.1b-1993 does not solve all our problems. It's not really intended to solve the kinds of problems we discussed at the beginning of this article. The standard is aimed at so-called *soft* real-time programs. A program that displays a video in a window is a perfect example of a soft real-time task. We want this task to run quickly and quite often in order to get a good quality display, but a few milliseconds here or there won't make much difference. For hard-real time problems, the POSIX standard has several drawbacks:

- Linux processes are *heavyweight* processes that are associated with significant overhead from process switching. Although Linux is relatively fast in switching processes, it can take several hundred microseconds on a fast machine. This would make it impossible to schedule a task to poll a sensor every 200 microseconds.

- Linux follows the standard UNIX technique of making kernel processes non-preemptive. That is, when a process is making a system call (and running in kernel mode) it cannot be forced to give up the processor to another task, no matter how high the priority of the other task. For people who write operating systems, this is wonderful because it makes

a lot of very complicated synchronization problems disappear. For people who want to run real-time programs, however, it is not so wonderful that important processes cannot get scheduled while the kernel works on behalf of even the least important process. in kernel mode, it cannot be rescheduled. For example, if Netscape calls `fork`, the fork will complete before any other process can run.

- Linux disables interrupts in *critical sections* of kernel code. This means a real-time interrupt may be delayed until the current process, no matter how low priority, finishes its critical section. Consider this piece of code

  ```
  line1: temp = qhead;
  line2: qhead = temp->next;
  ```

  Suppose that before the kernel gets to line 1, `qhead` contains the address of a data structure that is the only data structure on the queue and that `qhead->next` contains 0. Now suppose the kernel routine finishes line 1 and computes the value `temp->next` (which is 0) and then is interrupted by an interrupt that causes a new element to be added the the queue. When the interrupt routine finishes `qhead->next` will not be equal to 0 any more, but when the kernel routine continues it will assign the 0 value to `qhead` and so will lose the new element. To prevent these types of errors, Linux kernel makes extensive use of the `cli` command to *clear (disable) interrupts* during these critical sections. The kernel routine in this example would disable interrupts before it began changing the queue and re-enable interrupts only when the operation was complete. This means that sometimes interrupts would be delayed. It's hard to calculate the worst possible delay that can be caused by a critical section. You'd have to carefully examine the code for every driver and much of the rest of the OS as well to even make a good estimate. We've measured delays of as long as 1/2 millisecond. Consider what such a delay will mean to our camera routine.

Changing the Linux kernel to be a preemptable real-time kernel with low interrupt processing latency would require substantial rewriting of the Linux kernel code, almost writing a new one. Real-Time Linux uses a simpler and more efficient solution.

4 How it works

The basic idea is to make Linux run under the control of a real-time kernel (See Figure 2). When there is real-time work to be done, the RT operating system runs one of its tasks. When there is no real-time work to be done, the real-time kernel schedules Linux to run. So Linux is the lowest priority task of the RT-kernel.

The problem with Linux disabling interrupts is solved by simulating the Linux interrupt-related routines in the real-time kernel. For example, whenever Linux kernel invokes `cli()` -

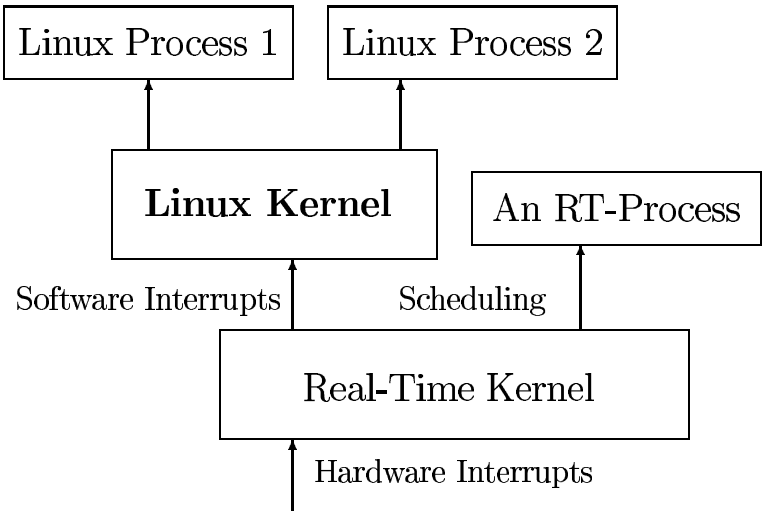

Figure 2: Real-Time Linux

routine that is supposed to disable interrupts, a *software* interrupt flag is reset instead. All interrupts get caught by the RT-kernel and passed to the Linux kernel according to the state of this flag and the interrupt mask. Therefore, the interrupts are always available for the RT-kernel, while still allowing Linux to "disable" them. In the example above, the Linux kernel routine would call `cli()` and this would clear the soft interrupt flag. If an interrupt occurred, the real-time executive would catch it and decide what to do. If the interrupt caused a real-time task to be run, the executive would save the state of Linux and start the real-time task immediately. If the interrupt just needed to be passed along to Linux, the real-time executive would set a flag showing a pending interrupt and then resume Linux execution without running the Linux interrupt handler. When Linux re-enables interrupts, the real-time executive will process all pending interrupts and cause the corresponding Linux handlers to execute.

The real-time kernel is itself non-preemptable, but since its routines are very small and fast, this does not cause big delays. Testing on Pentium 120 shows the maximum scheduling delay to be less than 20 μs.

Real-time tasks run at the kernel privilege level in order to provide direct access to the computer hardware. They have fixed allocations of memory for code and data — because otherwise we would have to allow for unpredictable delays when a task requested new memory or paged in a code page. Real-time tasks cannot use Linux system calls or directly call routines or access ordinary data structures in the Linux kernel, because this would introduce possibilities of inconsistencies. In our example above, the kernel routine changing the queue would invoke `cli` but this would not prevent a real-time task from starting. So we cannot allow the real-time

task to directly access the queue. We do, however, need a way for real-time tasks to exchange data with the kernel and with user tasks. In a data collection application, for example, we might need to send the data collected by an RT-task over the network, or write it locally in a file, while displaying it on the screen.

Real-Time FIFOs are used to pass information between real-time processes and ordinary Linux processes. RT-FIFOs are, like real-time tasks, never paged out. This eliminates the problem of unpredictable delays due to paging. And real-time fifos are designed to never block the real-time task.

Finally, there is the question of how the real-time kernel keeps track of the real-time. When implementing schedulers for real-time systems, there is usually a tradeoff between the rate of clock interrupts and *task release jitter*. Typically, sleeping tasks are resumed during the execution of the periodic clock interrupt handler. A comparatively low clock interrupt rate does not impose much overhead, but at the same time causes tasks to be resumed either prematurely or too late. In real-time Linux this problem is obviated by using a high-granularity one-shot timer in addition to standard periodic clock interrupts. Tasks are resumed in the timer interrupt handler precisely when needed.

5 What's next

The current version of RT-Linux is available by anonymous ftp from `luz.cs.nmt.edu`. Information on RT-Linux can be found on the web at `http://luz.cs.nmt.edu/~rtlinux`. The system is in active development, so it's not at production level of stability, but its pretty reliable. We are developing some applications as well and these will also be on the web site. Also we are asking for people who use the system to make their applications available on the website as well.