Real-Time Linux Kernel Design, Minimization and Optimization

Virginia Tech CS5204 Operating System Project: Jinggang Wang jiwang5@vt.edu

Abstraction:

This paper presented a general discussion about real-time Linux kernel design, minimization and optimization. A specific example of a real-time Linux kernel was discussed in details. Specifically, it covered the following topics: real-time operating system requirements; why the general Linux can not meet those real-time requirements; RTOS implementation approaches; mechanisms used in Real-Time Linux; skeleton code list and explanation; kernel minimization and optimization. The contribution of this paper is that it gave a detailed explanation about the three skeleton kernel programs rtl_sched.c, rt_time.c and rt_fifo.c in a specific embedded real-time Linux: uClinux. More detailed comments were also added to the original source code.

Key words: Linux, RTOS, RTLinux, uClinux, micorkernel, real-time scheduling, IPC.

1. Real-Time Operating System Requirements

A real-time system is a system that performs its functions and responds to external, asynchronous events within some specified time period.[1] There are hard and soft real-time systems. In soft real-time systems, timing requirements are statistically defined. An example can be a video conferencing system: it is desirable that frames are not skipped, but it is acceptable if a frame or two is occasionally missed. In a hard real-time system, the deadlines must be guaranteed. For example, if during a rocket engine test this engine begins to overheat, the shutdown procedure must be completed in time.

Twenty years ago, hard real-time applications were simple and usually placed on dedicated, customized and isolated hardware. However, real-time applications today are getting more and more powerful and yet complicated. They need to control such systems as factory floors connected to supply database, telescopes connected to the Internet, cell phones generating graphic displays, routers and telephone switches.

In order to provide with complex functions, great flexibility as well as strong reliability to real-time applications that will not run over pure hardware any more, a good real-time operating system is required to be embedded into those application facilities.

A real-time operating system (RTOS) is an operating system capable of guaranteeing timing requirements of the processes under its control. While a time-sharing OS like UNIX strives to provide good average performance, for a RTOS, correct timing is the key feature. Throughput is of secondary concern.

In order to deliver the tight worst-case timing performance needed by hard real-time, the RTOS needs to be ***simple, small, predictable***, and ***optimized to minimize the worst-case performance.***

2. Unsuitability of Linux for Hard Real-Time Applications

The following features make Linux infeasible to be used to run hard real-time applications: (1) Unpredictable scheduling – depends on system load; (2) Coarse timer resolution (10 ms); (3) Non-preemptible kernel; (4) Disabling of interrupts used for coarse-grained synchronization; (5) Use of virtual memory; (6) Reordering of requests for efficiency (e.g. for disk I/O). The detailed explanation could be found in [2]. Moreover, Linux processes are heavyweight processes, and it can take several hundred microseconds to finish a context switch, and thus make it impossible to schedule a task to poll a sensor every 100 microseconds.

3. Basic Real-Time Operating System Implementation Approaches

In order to facilitate the implementation of RTOS, POSIX standard has defined RTOS related specifications. POSIX.1b-1993 standard specified some real-time features in UNIX. The standard defined prioritized scheduling, locking of user memory pages in memory, real-time signals, improved IPC and timers, and a number of other features. Linux partially supports the POSIX.1b standard. However, POSIX.1b compatibility only permits certain kinds of soft real-time processing in Linux.

Generally there are three ways to implement a RTOS.

1. ***Microkernal Based Approach***. An example is the QNX[3] which only implements process scheduling, interprocess communication, low-level network communication, and interrupt dispatching. All other services, such as device drivers and file systems, are implemented as user processes. So the kernel is very small (7 KB of code) and fast. Comparing with monolithic kernel, it has some advantages. Debugging user processes is easier than debugging kernel components. If user processes are executed in separate address spaces, memory management errors in different modules are isolated. Another advantage is scalability. A QNX system can be scaled down to 100K to fit in the ROM, or expanded to a full-featured multi-machine development environment. Porting and maintenance is also easier. In addition, a real-time user process can preempt a device driver, which is not the case in monolithic kernels.

However, performance becomes a weak point since microkernel architecture places heavy load on interprocess communication and context switching. Microkernels only provide simple services directly. Therefore, more system calls have to be performed in a microkernel system than in a monolithic one to accomplish the same task. it is most likely for performance reasons that monolithic kernels are still prospering.

2. ***Monolithic Approach***. One example of a monolithic system is VxWorks.[4] VxWorks is a proprietary RTOS geared towards host/target approach. A UNIX host is used for software development and for running non-real-time parts of an applications. The VxWorks kernel called *wind* runs real-time tasks on the target computer. The machines communicate using TCP/IP networking.

In VxWorks, the kernel and tasks run in one address space. This allows task switching to be very fast and eliminates the need for system call traps. A run-time linker allows dynamic loading of both tasks and system modules. This feature makes for scalability. An interactive shell with C-like syntax can be used to examine and modify variables, evaluate expressions, call functions, and perform simple debugging. These features encourage experimentation and make development somewhat easier. However, They also make the system more fragile as errors in one module can easily affect others.

3. ***Decoupled Approach.*** Instead of "making a general purpose operating system kind-of-realtime"(IRIX)[5] and "keeping adding nonreal-time features to a real-time operating system"(VxWorks),[2] the original Real-Time Linux designers took another alternative. RTLinux decouples the real-time and non-real-time systems. This mechanism makes the real-time process simple, fast, predictable and optimized to minimize worst-case

performance. In order to support non-real-time applications at the same time, the general purpose Linux is run as the lowest priority thread. This approach will be discussed in the following part in more details.

4. RTLinux Mechanisms

As stated above, RTLinux decouples the real-time and non-real-time systems. Real-time applications are threads and interrupt handlers. Non-real-time components are put in the Linux thread that has the lowest priority. A patent “virtual interrupt controller” prevents the low priority thread from blocking interrupt aimed at real-time interrupt handlers. Communication between real-time components and the non-real-time Linux thread is designed to make sure the former is never forced to wait for operations of the latter. The application data flow for the whole system is illustrated in the following figure. We will discuss the RTLinux mechanisms in the following subtopics:

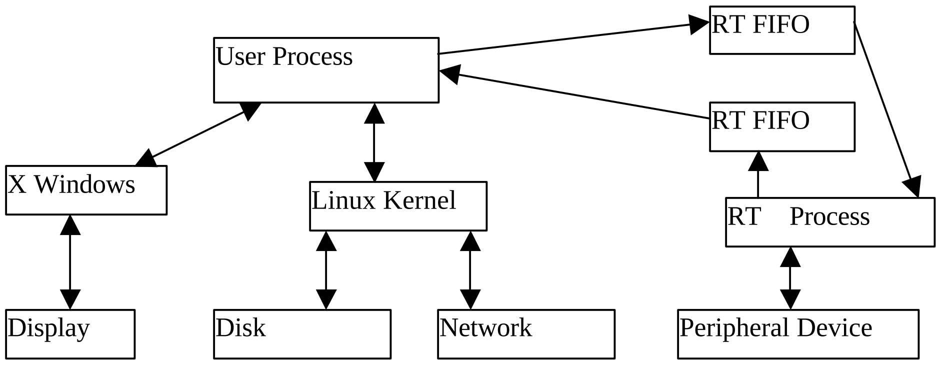

Figure 1: Data flow of a typical application in a RTLinux system[6]

1. Interrupt Emulation

One of the problems with doing hard real-time on a standard Linux system is the fact that the kernel uses disabling interrupts as a means of synchronization. Promiscuous use of disabling and enabling interrupts inflicts unpredictable interrupt dispatch latency.

In Real-Time Linux, this problem is solved by putting a layer of emulation software between the Linux kernel and the interrupt controller hardware. In the Linux source code all occurrences of cli, sti, and iret instructions (iret means return from interrupt) are replaced with emulating macros: S_CLI, S_STI and S_IRET. All hardware interrupts are caught by the emulator.[6]

2: Real-Time Tasks

Real-time tasks are user-defined programs that execute according to a specified schedule. The initial design was to give each real-time task its own address spaces to provide memory protection. This scheme works, but the system performance is not optimal due to inefficiency of TLB invalidation and change of protection level.

One way to improve performance is to run all RT-tasks in one address space. By using the kernel address space, we eliminate the overhead of protection level changes. However, this approach is clearly more fragile: a bug in a real-time task can wipe out the whole system.

Running tasks in the kernel address space has several advantages. Besides eliminating frequent TLB invalidation and protection level changes mentioned above, the approach allows us to refer to functions and objects by names rather than descriptors. For example, real-time tasks are represented as C structures. Each task can be given an arbitrary C identifier that can be used in other tasks. Dynamic linking performed during module loading resolves symbols to addresses, so the access is very efficient.

Task switching is also easier if all tasks run in one address space. Real-Time Linux performs task switching in software because hardware switches are slow on i486 CPUs. A context switch consists of pushing all integer registers on the stack and changing the stack pointer to point to the new task. [6]

3 Scheduling

The main task is to satisfy timing requirements of tasks. There are many ways to express timing constraints and many scheduling policies. No single policy is appropriate for all applications. Real-Time Linux allows users to write their own schedulers. This makes it possible to experiment with different scheduling policies and find the ones that best suit the application. Schedulers can use the interval timer facility described later.

By default, RTLinux provides a priority-based preemptive scheduler in that each task is assigned a unique priority. If several tasks are ready to execute, the task with the highest priority is executed. Whenever a task becomes ready it will immediately preempt the executing task if the current task has a lower priority. Each task is supposed to relinquish the CPU voluntarily.

The scheduler also directly supports periodic tasks. The period and the offset (the starting time) are specified for each of them. An interrupt-driven (sporadic) task can be implemented by defining an interrupt handler that wakes up the needed task.

4. Timing

Precise timing is necessary for the correct operation of the scheduler. Schedulers often require task switching at specific moments of time. Timing inaccuracies cause deviations from the planned schedule, resulting in so-called task release jitter. One reason for low timer resolution typically found in operating systems is the use of periodic clock interrupts. In RTLinux, the author avoids the periodic clock interrupts by using a programmable interval timer to interrupt the CPU only when needed. Specifically, the author puts the timer chip into the interrupt-on-terminal-count mode. Using this mode, an interrupt can be scheduled with approximately 1 microsecond precision. In this scheme the overhead of interrupt processing is minimal while the timer resolution is high. To keep track of the global time, all intervals between interrupts are summed up together. Most modern computers provide a software-readable global time counter.

The timer interface allows the scheduler to obtain the current time and to register functions to be called at particular moments. Periodic interrupts are simulated for Linux. With soft interrupts it is particularly easy to imitate an interrupt request, a bit in the pending interrupts mask is set. On the next soft return from interrupt, or soft sti, the handler will be invoked.

In order to make the interval timer work efficiently, it should not take a long time to reprogram the timer chip. Fortunately, most modern CPUs, e. g., Pentiums, have timers on-chip in addition to the outside timer chip. [6]

5 Interprocess Communications

Since the Linux kernel can be preempted by a real-time task at any moment, no Linux routine can safely be called from real-time tasks. However, some communication mechanism must be present. Simple FIFO buffers are used in RTLinux for moving information between non-real-time processes and real-time processes. RT-FIFO buffers are allocated in the kernel address space. They are referred to by integer numbers. The real-time task interface to RT-FIFOs includes creation, destruction, reading and writing functions. Reads and writes are atomic and do not block, which avoids the priority inversion problem. Linux user processes, on the other hand, see RT-FIFOs as ordinary character devices. Unlike the special system call interface, the character device interface gives the users full power of UNIX API for communication with real-time tasks.[6]

6. RTLinux Kernel Core Code Summary and Explanation

1: Interrupt Emulation Code:

```
S_CLI: movl $0, SFIF
S_STI: sti
       pushfl
       pushl $KERNEL_CS
       pushl $1f
       S_IRET
```

Figure 2: Soft CLI and STI

In S_CLI, the Linux interrupt enable flag SFIF is reset. Whenever an interrupt happens, the emulator checks SFIF. If it is set, the Linux interrupt handler is invoked immediately. Otherwise, the handler is not invoked. Instead, a bit is set in the variable SFREQ that holds the information about all pending interrupts. When Linux re-enables interrupts, the handlers of all pending interrupts are executed. Such simulated interrupts are called as soft interrupts.

Since Linux has no direct control over the interrupt controller, it does not influence processing of real-time interrupts that do not pass through the emulator.

The S_STI macro sets up the stack as if an interrupt is being handled, and then uses S_IRET macro to emulate the return. This works because S_IRET enables soft interrupts just as the hardware iret enables real ones.

The S_IRET macro starts with saving some scratch registers and initializing the data segment register to point to the kernel. The latter is necessary to access global variables. Then the bit mask representing all unmasked pending interrupts is scanned for a set bit. If no pending interrupt was found, the interrupt state variable is set, and a hard return from interrupt is performed. If an interrupt was found, a jump is made to the Linux handler. The handler's S_IRET, in turn, will jump to the next pending interrupt handler, and so on, until no interrupts are pending.

```
S_IRET: push %ds
         pushl %eax
         pushl %edx
        movl $KERNEL_DS ,%edx
        mov %dx,%ds
        cli
        movl SFREQ, %edx
        andl SFMASK, %edx
        bsfl %edx, %eax
        jz 1f
       S_CLI
       sti
       jmp SFIDT(,%eax,4)
```

```
1: movl $1, SFIF
       popl %edx
       popl %eax
       pop %ds
       iret
```

Figure 3: Soft IRET

Scanning and decision taking are done atomically--otherwise, if a new interrupt occurs between them, and the scan has not found any pending interrupts, the invocation of the new interrupt handler will be delayed until the next S_STI or S_IRET. The author used chained jumps instead of subroutine calls because the latter would not fully emulate direct interrupt handling. Linux handlers examine the stack to find out whether it was the user or the kernel code that was interrupted, and make decisions based on it. Therefore, it is important to preserve the stack state.[6]

2: Three Major Kernel Programs: rtl_sched.c, rtl_time.c, **and** *rtl_fifo.c*

As stated in chapter 5, after we determined these two strategies of infrastructure level: interrupt emulation and real-time task construction, scheduling, timing and interprocess communication become 3 major concerns of RTLinux implementaion. The reasons why they are required are given in above chapters and summarized again here: scheduling helps us to run all kinds of real-time tasks in the right schedule and thus meet their deadlines; In order to conduct microsecond-level scheduling, we need to design a more efficient timing program of high resolution; Interprocess communication is important for us to build the communication between a real-time task and a regular Linux task. Therefore, by analyzing these three programs: rtl_sched.c, rtl_time.c and rtl_fifo.c, we can clearly see how a complete RTLinux implementation works.

Following the original authors of RTLinux from New Mexico Institute of Mining and Technology, there are several companies are working on RTLinux now. In order to make the project feasible, I picked a small embedded RTLinux version called uClinux from Lileo company[7]. In uClinux, all real-time components as well as real-time applications are built into the kernel and thus play into effect starting from the system boot. The whole system will run on a small embedded kit: uCsimm[8], which uses a MC68EZ328 chip as the CPU together with some memory chips. The analysis will be divided into the following sub-topics:

How System Install and Start All Real-Time Components?

The function to install the scheduler, timing program,and rt_FIFO are rtl_shedule_init(), rt_time_init() and rtf_init() respectively. In uClinux, the real-time application program starts from the fixed function call rtl_application_init(). In order to use these real-time compeonents, all of these installation functions must be executed when the system boots.

Linux kernel consists of some low-level machine-dependent assembly programs and a group of C programs.[9] But unlike a regular user-mode c program which uses a main() function as the entry function, in Linux kernel, the entry function is start_kernel() in /usr/src/init/main.c file. From the main.c file in uClinux, the following code is found:

```
#ifdef  _RT_
extern void rt_time_init(void);
extern void rt_schedule_init(void);
extern void rt_application_init(void);
#endif
```

```
 asmlinkage void start_kernel(void)
{   ……
    #ifdef _RT_
            rt_time_init();
            rtl_schedule_init();
    #endif
   …
  kernel_thread(init, NULL,0);
  cpu_idle(NULL);
}
static int init(void* unused)
{ ….
    #ifdef _RT_
          rtl_application_init();
     #endif
          setup();      //This system call will finally invoke rtf_init()
    …
}
```

From the above code, it is obvious to see how the kernel installs the real-time sheduler, timing program and application program. The real-time FIFO installation is not so straightforward in that it employs the mechanism of Linux System Call. The function setup() is a System Call, which will be mapped to the System Call assembly routine sys_setup in entry.S file. The assembly routine will finally invoke rtf_init through the following path: →invoke sys_setup() in filesystems.c→device_setup() in genhd.c→chr_dev_init() in mem.c→rtf_init() in rtl_fifo.c

Major Function and Data Flow Chart Blocks about Scheduling and Timing.

In rtl_sched.c, two linked-lists are defined, "rtl_tasks" is for all live real-time tasks which are competing for CPU, and "rtl_zombies" is for all zombie real-time tasks. During its life time, a real-time task could be in the following 5 states: Ready, Delayed, Dormant, Active, and Zombie. The state transition diagram is as follows:

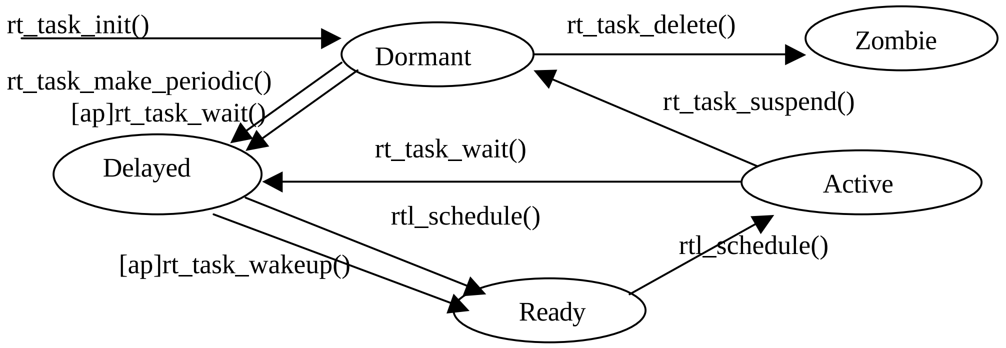

Figure 4: State Transition Diagram of Real-Time Tasks

Typical kernel functions that are called within a real-time application include rt_task_init(), rt_task_make_periodic(), rt_task_wait(), rt_task_delete(), and rt_task_suspend(). A real-time application always starts its execution from the function rtl_application_init(). Within this function, all real-time tasks are initialized by calling rt_task_init() which initializes the task fields such as priorities, states(Dormant), the function to be called as part of the task body, stack size and etc. The application would then call rt_task_make_periodic() to start the periodic tasks by setting the start time and period fields and resetting the timer to interrupt CPU at the specified time to run this task. Within the timer interrupt service routine, rtl_schedule() function is invoked which selects a task that has the highest priority and the earliest start time from those tasks in Delayed or Ready state. Once a task is selected, it executes until it completes its execution for the current period and calls the rt_task_wait() to relinquish the CPU, or it may be preempted if another high priority task is selected by rtl_schedule() through the timer interrupt service routine. If the task were to eventually complete its execution, it will call rt_task_suspend() which changes the task state to Dormant and calls the rtl_schedule() again. If no any task in Delayed or Ready state, rtl_schedule() would deactivate itself and activate the regular Linux scheduler. In this case, user-space programs such as the shell will be able to execute.[10]

The real-time scheduler works in the support of high-resolution timer. The following flowchart illustrates how rtl_sched.c, rtl_time.c and the real-time application program work together.

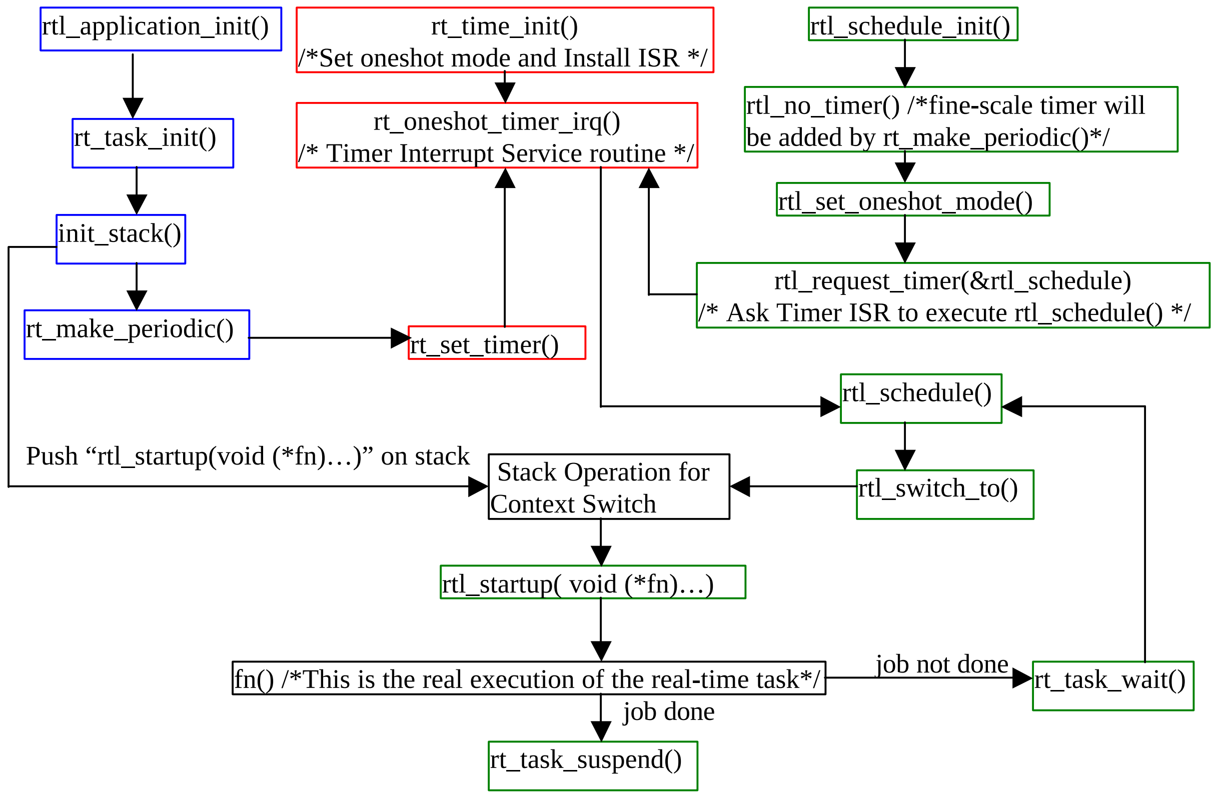
rtl_application_init()
rt_time_init()
/*Set oneshot mode and Install ISR */
rtl_schedule_init()
rt_task_init()
rt_oneshot_timer_irq()
/* Timer Interrupt Service routine */
rtl_no_timer() /*fine-scale timer will be added by rt_make_periodic()*/
rtl_set_oneshot_mode()
init_stack()
rtl_request_timer(&rtl_schedule)
/* Ask Timer ISR to execute rtl_schedule() */
rt_make_periodic()
rt_set_timer()
rtl_schedule()
Push "rtl_startup(void (*fn)…)" on stack
Stack Operation for Context Switch
rtl_switch_to()
rtl_startup(void (*fn)…)
fn() /*This is the real execution of the real-time task*/
job not done
rt_task_wait()
job done
rt_task_suspend()

Figure 5: Data-Flow and Interaction Diagram of rtl_sched.c/rtl_time.c/application

rt_fifo.c: Real-Time FIFO Construction.

The reason why it is necessary to build a real-time FIFO is related to the special architecture of RTLinux itself. According to RTLinux logic, an application is naturally divided into 2 parts: the real-time part could be run as a real-time process and non-real-time part as a regular Linux process. Pipes, FIFOs, and system V IPC system calls related to semaphores, messages queues and shared memory are widely used as interprocess communication methods in Linux. However, two reasons prevent them from being the right choices for RTLinux. First, unlike regular Linux processes, a real-time process is running in kernel space. But Pipes and FIFOs are used for communication between 2 processes in user space, while the communication we need is between a process in kernel space and a process in user space. Second, since Linux has the lowest priority in the whole system, no Linux system call can be trusted as a guarantee to real-time performance. Therefore, a real-time FIFO (rt_FIFO) is needed.

The rt_FIFO needs to meet the following requirements: First, it is in the kernel space; Second, it can be accessed both from kernel space and from user space; Third, access to the rt_FIFO from kernel space should not be blocked, and on the other hand, it may block the access from a process in user space; Fourth, a Linux process in the user space can treat the rt_FIFO as a standard character device so that it can be accessed easily through the standard device interface of open, close, read and write.

Basically a rt_FIFO is nothing but a buffer allocated in the kernel space and associated with a waiting queue due to the blocking feature for non-real-time service side. As in the kernel space, a rt_FIFO can only be created and destroyed by a real-time process through the kernel function of kmalloc(size, GFP_KERNEL) and kfree(buffer) respectively. For the sake of non-blocking, whenever a real-time process accesses a rt_FIFO, the CPU interrupt is disabled so that no other task could have a chance in competing with the same resource.

As illustrated before, when uClinux boots, rtf_init() executes once and it creates a fixed number of empty FIFO arrays. After the real buffer space for a rt_FIFO is allocated by a real-time process, it is associated with an integer which would be used as the minor number of the rt_FIFO character device to the Linux process.

To see from the Linux side, a character device called “rtf” is registered by the function register_chrdev(RTF_MAJOR,”rtf”,&rtf_fops) within rtf_init(). Usually all rtf devices have a major number of 63. Then the specific rt_fifos are created by “mknod” command as /dev/rtf0 /dev/rtf1 with minor number of 0,1 respectively. The structure rtf_fops includes all file operations such as open, seek, read, write and etc.[11] For example, if an application wants to use /dev/rtf1, the real-time process will create a rt_FIFO with minor number of 1,and the Linux process could open the device file of /dev/rtf1, and then the real communication starts.

Although both are copy to and remove from a buffer, the send and receive operations to the rt_FIFO from the kernel space are different from those from the user space. Memory operation with the kernel space is done through kernel function memcpy(), while memory operation between the kernel and user space is done through memcpy_fromfs() and memcpy_tofs() functions.

7. RTLinux Kernel Minimization and Optimization

Based on the design architecture of RTLinux, if we want to use it in a specific application in some fields, the regular Linux is still a need. However, the big size of a regular Linux kernel presents a big problem for RTLinux to be used in some low-memory embedded systems such as RAM-disk based systems. Therefore, minimization of Linux kernel is significant.

The following ways could help us build a miniRTLinux kernel of size 1.44MB: 1) Reduce the system size by exploiting redundancy, deletion and compression; 2) Reduce run time sizes by shrinking executables, compressing output, and using multi-call binaries;

3) Pick the minimum list of libraries for the specific usage. The detailed explanation could be found in [12].

After kernel minimization, given a specific field application, there is still one measure can be employed to further improve the performance of an embedded system based on the kernel optimization. Basic idea is to tune the system setting to an optimized point so that more run-time memory space is left and higher execution speed could be achieved by ways such as allowing the system to drop log/data files to a central server system and then remove them from the kernel until needed again; reducing the amount of resource allocation (e.g. setting the maximum number of IDE device to 2 instead of 8, setting console number to be 4, and etc.); picking optimized values for free pages, page-cache and etc.; configuring reduced size for file systems(e.g. a directory depth of 1024 or maximum filename length of 1800 characters are rarely needed and thus could be reduced.). On-demand loading is also a good candidate since there are very few resources really needed on-site. Much can be moved to off-site as long as there are mechanisms available to hook them up on demand via scripts, cron and etc.[12]

8. Conclusions

In this paper, general discussions about real-time system requirements, unsuitability of Linux for real-time application, and real-time system design mechanisms are presented in the beginning. More details are then given to all the key points about how a RTLinux kernel is designed, minimized and optimized. Although some of the statements are extracted from the design documentation from original authors based on personal understanding and reorganizations, the contribution to the Linux public domain of this paper is that it gives a detailed explanation about the three skeleton kernel programs rtl_sched.c, rt_time.c and rt_fifo.c in a specific embedded real-time Linux: uCLinux. More detailed comments were added to the original source code.

References

[1] Borko Furht, Dan Grostick, et al. *Real-time UNIX systems: design and application guide*. Kluwer Academic Publishers Group, Norwell, MA, USA, 1991.
[2] Victor Yodaiken and Michael Barabanov *RTLinux Version TWO* Design documentation about RTLinux in FSMLabs 1997. http://www.fsmlabs.com
[3] QNX company website. http://www.qnx.com
[4] Wind-River Company website. http://www.windriver.com.
[5] SGI Company Website: http://www.sgi.com/software.
[6] Michael Barabanov *A Linux-based Real-Time Operating System*, MS thesis, June,1997. Http://www.fsmlabs.com/developers/white_papers.
[7] Lileo Company website. http://www.uclinux.org.
[8] uCsimm Project website: http://www.ucsimm.com.
[9] M.Beck,and etc, *Linux Kernel Internals*, 2nd edition, Addison-Wesley ,1999
[10] Binoy Ravindran, *Project 1 Specification*, ece4984,spring, 2000 at Virginia Tech
[11] Alessandro Rubini and etc *Linux Device Drivers* 2nd edition, O'Reilly 2001
[12] Nicholas Mc Guire *MiniRTL Hard Real-Time Linux for Embedded Systems*, 1998 Http://www.hofr.at