2018 International Conference on Smart Grid and Electrical Automation

Analysis of Linux kernel's real-time performance

ZhangYanyan[1], Ran Xiangjin[2,3,*]

1. College of Engineering, Jilin Business and Technology College, Changchun, China;
2. College of Earth Sciences, Jilin University, Changchun, China;
3.College of Applied Technology, Jilin University, Changchun, China
zhangyanyanyy@163.com, ranxiangjin@qq.com

Abstract—**The Linux OS is used in various devices, but not in real-time fields mostly. The real-time performance of Linux kernel is analyzed between the 2.6 and 2.4 kernel, using the LmBench tool. A test method is introduced to compare the real-time performance, based on two preemption strategies. It is concluded that the real-time performance of the 2.6 kernel has been greatly improved, which can basically meet the requirements of real-time systems.**

Keywords- Linux; Preemption; Kernel; Real-time performance; LmBench

I. Introduction

With the development of embedded system, the embedded operating system becomes more and more important. In many fields, Linux has demonstrated the superiority characteristic, but since the release of the 2.2 kernel, the preemption has become a hot topic. In order to meet the real-time requirements of the real-time system, many scholars have strongly urged the standard Linux kernel to support the preemptive scheduling. The real-time performance of the standard Linux system is low, and it cannot meet the requirements of real-time until 2.6 version. Therefore, there are many embedded Linux producers using different methods to improve the real-time performance of standard Linux system [1]. MontaVista and TimeSys provide preemptive kernels [2]; REDSonic uses preemption point; LynuxWorks and Red Hat use RTLinux; Lineo uses RTAI to achieve the purpose of real-time[3,4]. On December 17, 2003, the 2.6 Linux kernel was released, and preemptive kernel was supported in this version. Since then, the real-time performance of standard Linux had been greatly improved.

II. Preemption strategy for real time system

In real-time systems, preemption is defined when a process whose priority is higher than the current running process forked in the system, the system interrupts the current running process immediately, and runs the higher priority process[5,6]. This makes priority of the running process the highest one in the system at any time. Although the Linux kernel is non-preemptive before the 2.6 version, caused by the scheduling policy of Linux, the main reason is the time of scheduling. When a high priority user process arrives, if the low priority user process executing in user space, then the high priority process can immediately deprive the execution of the low priority process; but if the low priority user process is executed in the system space, while the process of scheduling action is not allowed to seize the place, so the high priority user process cannot be executed immediately. In order to change this situation, the code which allows preempts added to the 2.6 Linux kernel, so that other processes can be scheduled when the process is executed in the kernel state.

III. The analysis ON THE IMPLEMENT OF preemptive kernel of 2.6 version Linux

The preemptive kernel allows the user process to be preempted during the execution of a system call, which enables the new high priority process to be awakened. This is not occurred safely in any position of the kernel, such as in the critical area, it is not allowed to preempt. A critical area is a sequence of instructions that cannot be executed by more than one process at the same time. In the 2.6 version of the Linux kernel, these parts use spin lock to protect, and the function of the spin lock is enhanced to prevent preemption. Depending on the method, preemption can only occur in the other parts of the unused spin lock. When a higher priority process is awakened, the system call dispatcher can seize the low priority process executed at the moment, as long as the system call code is not under the spin lock protection(spin lock means non preemptive).

Linux kernel can be preempted at any point except for some situations as follows:

1. The kernel is processing the interrupt. In the Linux kernel, the process cannot preempt the execution of interrupt.
2. Write/read lock, spin lock and so on are held by the code of kernel, which protected by these locks. The lock in the kernel is made to ensure the correctness of interruption in concurrent processes running on different CPU execution time of SMP (Symmetrical Multiprocessor) systems. Holding the lock, the kernel should not be preempted, otherwise the CPU will not be able to get the lock.
3. The kernel is executing the scheduler.
4. The kernel is operating the private data structures of every CPU. In SMP, these data structures are implicitly protected, not by spinlocks. If it is allowed to preempt, a rescheduling process may be scheduled to other CPUs, then the variable PCB will be in error status.

In order to ensure that the Linux kernel will not be in the above cases, the kernel uses a variable preempt_count, called kernel preemptive lock. This variable is set in the PCB structure of the process. When the kernel enters into the above conditions, the variable preempt_count increases 1, indicating the kernel does not allow preemption. When the kernel exits from the above several states, the variable count preempt_count decreases 1, and meanwhile executes the preemptive judging and scheduling.

There are two parts to achieve the preemptive kernel. The first is to make the CPU preemptive when returned from the system space(unless the interrupt is in the scheduling forbidden area). The second is to set the scheduling forbidden area in the kernel.

3.1 Scheduling process when returned from system space

When CPU returns from the interrupt that occurs in the system space, in order to be able to carry out the process of preemptive scheduling, two conditional compilations of fragments are added to the returning code from interruption handler, just as follows:

```
/ *fragment 1*/
...... / * save CPU state and register values.*/
#ifdef CONFIG_PREEMPT
    get_thread_info  r8
    ldr    r9, [r8, #TI_PREEMPT]  @ Get the count value
    add    r7, r9, #1      @Increase the value of the counter
    str    r7, [r8, #TI_PREEMPT]
#endif
```

The constant TI_PREEMPT is the displacement of preempt_count in the process control block, which is equivalent to the C statement "current->preempt_count++;"

```
/ *fragment 2*/
#ifdef CONFIG_PREEMPT
    ldr       r0, [r8, #TI_FLAGS]    @Get the flag value
    tst       r0, #_TIF_NEED_RESCHED
    blne      svc_preempt
```

This fragment is the key point, equivalent to:

```
if (need_resched ()) svc_preempt ();
```

This determines whether it can be rescheduled or not. The value of preempt_count in the process control block of the current process determines whether the current process is allowed to run. It can be imagined that when the CPU runs in the user space, and returned to the user space while completed from the system call, the value of preempt_count of the current process must be 0. Therefore, the kernel can use preempt_count value as a means to prohibit / allow the scheduling.

The increase and decrease of the value of preempt_count are achieved by a series of macro operations, defined in the preempt.h header file. If it is not allowed to execute preemptive scheduling, the system executes the code preempt_disable () to forbidden performing preemptive scheduling.

```
#define preempt_disable()       \
do {                    \
    inc_preempt_count();     \
    barrier();              \
} while(0)
```

To prohibit the deprivation, the preempt_count value needs to be increased. This operation of increment is completed by the macro codes preempt_count (). The macro code is very simple; the operation is only to make the preempt_count value increase 1. Executing preempt_disable(), the program enters a restricted area deprivation scheduling. At this point, the system will not allow deprived scheduling.

Correspondingly, preempt_enable () is the end of forbidden area, which will enable preemptive scheduling, if there is a higher priority process ready, the code will have the opportunity to get running. Code as follows:

```
#define preempt_enable()       \
do {                    \
    preempt_enable_no_resched();  \
    preempt_check_resched();     \
}while(0)
```

This operation is more complex, which contains two operations. The first one is to decrease the preempt_count value, completed by the macro operation preempt_enable_no_resched(). Look at the implementation of the macro code:

```
#define preempt_enable_no_resched()   \
do {                        \
    dec_preempt_count();           \
    barrier();                  \
}while(0)
```

The name of this macro operation means “allow stripping, but not scheduling”, because scheduling is the next step. This step is to decrease the preempt_count value, so that the scheduling can be deprived of. However, it is a forbidden area of deprivable scheduling, not of interrupting. The schedule is not occurred for that preempt_count is not 0 when the interrupt happened. But now it can be scheduled when quoting preempt_check_resched().

```
#define preempt_check_resched()   \
do {                      \
    if(unlikely(test_thread_flag(TIF_NEED_RESCHE
D)))  \
    preempt_schedule ();   \
}while(0)
```

At this time, if the conditions are enough, that is, the high priority process or interrupt is ready, it can be scheduled, via calling preempt_schedule () function.

3.2 Set scheduling forbidden area in the kernel

In order to set scheduling forbidden area in the kernel, the SMP structure of the 2.6 version Linux kernel is used. In the system with SMP structure, in order to realize multiple access to shared resources, the concept of "lock" is introduced to guarantee the mutual exclusion among multiple CPU processes. The spin_lock () function is used to compete the SMP access forbidden area and lock, and the spin_unlock() function is used to exit the area and unlock. The SMP structure can be used to implement the operation of setting the scheduler forbidden area in the kernel, thereby reducing the workload. The following codes describe the implementation of spin_lock () and spin_unlock () macros.

```
#define spin_lock (lock)              \
do {                        \
    preempt_disable();            \
    if (unlikely (!_raw_spin_trylock (lock))) \
            __preempt_spin_lock (lock);      \
    }while(0)
```

Here the parameter lock is the data structure which is used to lock. As is mentioned in Section 2.1, preempt_disable() is used to increase the preempt_count value of the current process, which allows the critical resource to be protected, and the scheduling is prohibited.

And for spin_unlock (), its code is as follows:

```
#define spin_unlock (lock) \
do {  \
    _raw_spin_unlock (lock);     \
    preempt_enable ();          \
}while (0)
```

As is described in Section 3.1, preempt_enable() is used to decrease the preempt_count value of current process, relieving the protection of critical resources for scheduling.

The spin_lock() and spin_unlock() codes are added to the kernel, which is divided into preemptive scheduling areas and non-preemptive scheduling areas, thereby the response time of tasks is reduced, and the real-time performance of the system is enhanced.

IV. COMPARISON AND ANALYSIS OF LINUX KERNEL'S REAL TIME PERFORMANCE

It is proved that the standard Linux kernel is enhanced in the real-time performance by testing interrupt response time of the 2.4 and 2.6 versions of the kernel.

In order to test the real-time performance of the kernel, the Lmbench benchmark program is used to test the context switch delay of the system. Lmbench is a software package designed by Mcvoy Larry to evaluate the operating system, which provides a series of small programs to test the real-time performance of the system [7].

Lmbench was compiled into the Linux 2.4 and 2.6 versions of the kernel separately. The two kernels will be tested in the following platform respectively.

1. The platform is X86.
2. The CPU is Pentium4, clocked at 2.8GHz
3. The memory is 2G

The context switching delay of 2.4 version of the kernel is tested first. Every command was tested five times and five sets of data were got, then the average value was calculated. The specific test commands and results are shown in Figure 1.

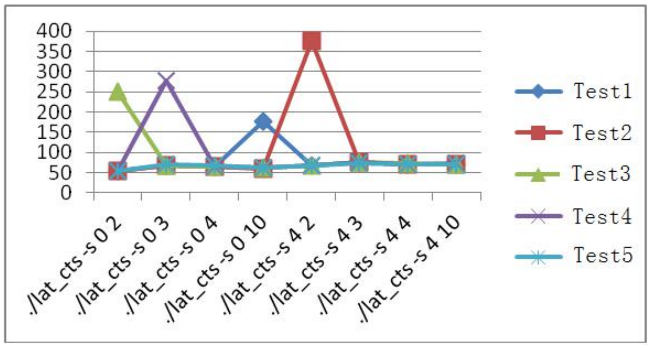

Figure 1.context switch delay of 2.4 version Linux kernel (unit: μs)

It can be seen that most context switching delays are below 80 μs in the 2.4 version of the Linux kernel, but they are not stable, and the max value is up to 375.84 μs(Fig. 1). Context switch delay is an important factor to measure real-time performance of a real-time system, and such a high latency cannot be endured for real time tasks.

The 2.6 version of the Linux kernel was tested on the same hardware platforms and under the same test commands. The specific test commands and results are shown in Figure 2.

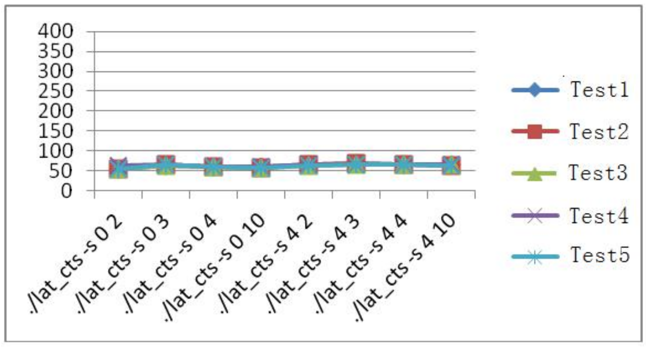

Figure 2. Context switch delay of 2.6 version Linux kernel (unit: μs)

Fig. 2 shows that context switching time is between 50 and 70 μs in the 2.6 version of the Linux kernel, and it is relatively stable. This proves that the addition of preemptive kernel of the 2.6 version of the standard Linux kernel has been able to meet the requirements of real-time systems.

V. CONCLUSION

In the 2.6 version of the Linux kernel, the preemptive kernel is supported. This improves the real-time performance of Linux kernel greatly, which allows the Linux to meet the basic requirements of real-time systems. So the standard Linux system can be modified simply and applied to the higher real time situations.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the funding of this study by the following projects: "Research of Network Intrusion Detection Based on Deep Learning" from the Scientific Research Planning Project of the Education Department of Jilin Province (JJKH20180518KJ); Social Science Fund Project of Jilin province "Research on the cultivation of innovative and entrepreneurial college students based on big data platform"(2016B192); Talent Development Project of Jilin province (20091328); Venture Fund Project for Personnel Studying Abroad of Jilin province (2010273); "The Research on the Mechanism of Application Technology Intellect Cultivation in Diversification Mode of Running School"; "The Teaching Research Project of Higher Education in Jilin province"; "Practice and Research on Reform System of Practical Teaching in Transformation and Development of Local Undergraduate Colleges and Universities".

REFERENCES

[1] Gang W J, Chang G G, Feng X L, et al. Research and design of hard real-time Linux kernel[J]. XI Tong Gong Cheng Yu Dian Zi Ji

Shu/systems Engineering & Electronics, 2006, 28(12):1932-1934, 1947.

[2] Bing Y U, Zhong-Wen L I. Analysis of Linux Real-time Mechanism[J]. Computer Technology & Development, 2007,17(9):41-44,47.

[3] Zhao X, Xia J B. Research and Improvement of Real-time in Linux-system Based on RTAI[J]. Computer Engineering, 2010, 36(14):288-290.

[4] Xin L, Wen J L, Tao H F. Research and Implementation of hard real-time performance of Embedded Linux based on RTAI[J]. 2006,22(11-2):46-48,294.

[5] Liu S, Wang L F, Jiang Z J. Resconstruct Real-time of Linux Based on Multicore Computer[J]. Microelectronics & Computer, 2013, 30(8):120-123.

[6] Shan B, Ru-Zhi X U, Zhu H. Study on methods to improve real-time performances of embedded Linux[J]. Journal of North China Electric Power University, 2006,33(4):65-68.

[7] Jiang L I, Dai S H. Real-time performance test and analysis of Linux[J]. Computer Applications, 2005,25(7):1679-1681.