On Interrupt Scheduling based on Process Priority for Predictable Real-Time Behavior

Minsub Lee, Juyoung Lee, Andrii Shyshkalov, Jaevaek Seo, Intaek Hong, Insik Shin
Dept. of Computer Science
KAIST, South Korea
insik.shin@cs.kaist.ac.kr

Abstract

Traditionally, kernel services are of a higher priority than user processes. The kernel can preempt the currently executed process in order to perform interrupt handling for the behalf of another process, even though the latter process is of a lower priority than the former. This can be viewed as priority inversion. We propose a new interrupt handling approach that couples interrupt scheduling with the priority of a process associated with the interrupt to handle. We present techniques to derive exact process priorities in handling interrupts for incoming network packets. The proposed approach has been implemented in Linux 2.6, and experiment results show that it reduces interference of lower priority processes to higher-priority process through interrupt handling.

1 Introduction

As the number of hardware devices grows higher, general purpose operating systems are more often used for real-time applications. Such operating systems were not originally designed to satisfy real-time application requirements. Therefore, a number of studies have been conducted to add predictable and efficient task management to commodity operating systems [8],[3].

One of the main design goals of commodity operating systems is the system's responsiveness. Particularly, it implies that interrupt handling must be processed by an operating system immediately. However, interrupts processing, which can take some time, may not be useful for the currently executed task. Additionally interrupt processing time is often deducted from interrupted thread. Because interrupted thread does not necessarily get affected by interrupt processing, this can dramatically distort the real-time performance of the operating system. Therefore, interrupts handling is an important issue to address in order to provide predictability on the execution of real-time tasks.

In Linux, interrupts are processed by kernel threads, which have higher priority than any user thread. From thread scheduler point of view, scheduling is done according to threads' priorities. However, because interrupts are processed to serve different threads with different priorities, the priority inversion may occur. As an example, consider a running user process with priority 17, a sleeping one with priority 21 and an incoming network packet for the second process. This packet will be processed immediately by kernel process, which in turn will delay a thread of higher priority. The more incoming packets, the less predictable is execution of the first thread.

In this paper we addressed the priority inversion problem, which is caused by interrupt processing of network stack. We have designed a technique to process the network interface card interrupts in order of priorities of the threads that require interrupts processing. We have also implemented our technique for Linux kernel version 2.6 and conducted performance evaluations. Our results show that our interrupt handling approach is suitable to real-time environment.

The rest of paper is organized as follows. Section 2 summarizes related work. Section 3 gives an overview of the interrupt handling in Linux. Our approach is described in section 4 and quantitative performance evaluation of our implementation is provided in section 5. Finally, section 6 contains conclusions.

2 Related Work

Several other research projects have investigated interrupt processing distortions on real-time performance of Linux systems. An improved accounting of consumed CPU time during interrupts has been proposed in [3] and [8]. A probabilistic approach [8] has been developed to determine possibly affected processes during top half execution and schedule bottom half with regard to priorities. In this paper however, we describe an approach to find exact processes, and hence priorities, for each interruption caused by incoming network packet. Therefore, our approach can make priority based interrupt scheduling decisions more accurately.

Some researchers focused on other systems than Linux to investigate on similar issues. Scheduling of interrupts

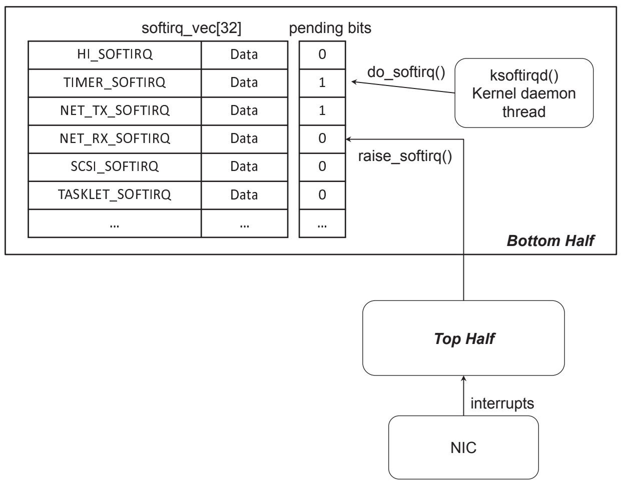

Figure 1. Top and bottom halves of interrupt handling in Linux

and predictable interrupt management has been developed for complex systems [5]. Improved performance of network stack in UNIX operating systems has been proposed by scheduling incoming network traffic with priorities [2]. While these studies focused on providing fairness and increased throughput under high load, our technique focuses on real-time behavior.

Many protocols have been introduced to address the priority inversion problem when tasks are accessing critical sections in a mutually exclusive manner. Such protocols include the Priority Inheritance Protocol (PIP) [7], the Priority Ceiling Protocol (PCP) [6], and Stack Resource Policy (SRP) [1]. While these protocols concern the priority inversion problem within the context of process scheduling, our work concerns the problem taking process scheduling and interrupt handling into consideration together.

3 Interrupt Handling

Interrupts can be caused by hardware as well as by software. In Linux, interrupt handling is done by the kernel, which is invoked every time an interrupt occurs. Interrupts can occur at any time during execution, their number is difficult to predict.

To achieve better performance and responsiveness, interrupt handling in Linux kernel 2.6 is divided into two phases. The first one, called *top half*, starts when an interrupt signal invokes the interrupt service routine (ISR). Then, ISR disables interrupts of the same type and calls the corresponding interrupt handler. Because interrupt handlers execute asynchronously, the processing at this phase should be as quick as possible.

For instance, upon receiving incoming packets off the network, network interface cards (NIC) issue interrupts immediately to alert kernel of their availability. Then, the ISR quickly responds to the interrupts by executing the network card's registered handlers. Most of all, they copy the new packets into the main memory. For remaining processing, network card's registered handler must raise software interrupt (`softirq`) [4], which means setting pending bit to 1 in a softirq vector array (`softirq_vec`) (Figure 1).

All other interrupt processing is deferred to later, so-called the *bottom half* phase. Bottom half usually requires longer processing time than top half. Under heavy load such as high network traffic, the frequency of interrupts is high, and bottom half processing can consume much of the CPU time.

In Linux, bottom half phase is executed periodically by a set of per-processor kernel threads (`ksoftirqd`), which have priority of 15. These kernel threads scan softirq vector array for set pending bits and execute corresponding handlers for further interrupt processing. Additionally, interrupts are processed regardless of the priorities of the processes, which interrupts serve.

Consider a currently running user process with the highest priority, which does not have any network communication. Any incoming network network packet will interrupt the current thread at least once during top half phase processing. Later, since the kernel thread has even higher priority, the bottom half processing may interrupt current thread for even longer time than top half, if the packet is large. Therefore, since the packet is processed for another process with lower priority than current, such scenario is a priority inversion one.

Since the top half cannot be delayed, in this paper we are particularly interested in determining when is the proper time for executing the bottom half phase and which interrupts to process first.

4 Our Approach

This section describes the design and implementation of our new interrupt handling approach in Linux. A key idea of our approach is that some incoming packets are associated with its intended receiver processes (and their priorities) during interrupt handling. Those packets are given process priorities in the top half, and the bottom half is equipped with priority-based scheduling capable of delaying interrupt processing even further, in order to avoid the priority inversion problem.

UDP packets carry process-related information, which is a port number. In the top half, an UDP packet is fetched from the NIC's buffer to the main memory. We then extract a port number out of the packet. This can be easily done (with a single memory lookup), as the port number is stored in a fixed location in the packet. In order to assign priorities to packets, it needs to figure out which sockets are

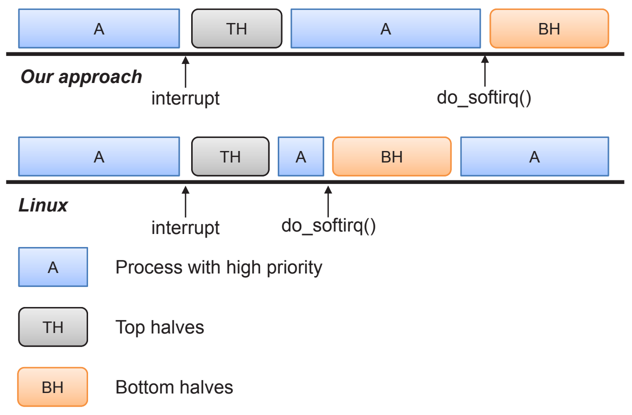

Figure 2. Linux vs. our approach

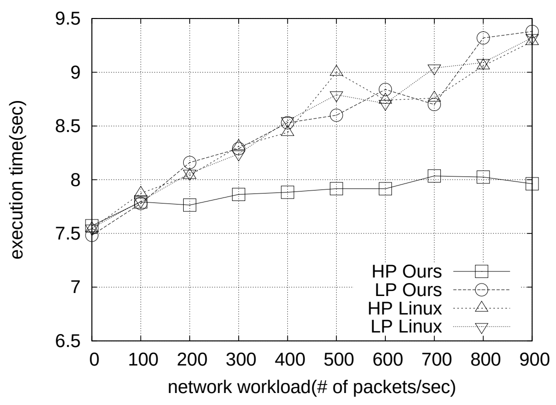

Figure 3. Comparison of execution times

coupled with which processes, which is costly. Hence, we maintain a port number indexed process priority table, and it helps to achieve a faster conversion of a port number to a priority. Each entry of the table is created when a process binds a socket and becomes invalid when the process closes the socket. Once a port number is available, we can simply consult this table to map the port number to the priority of a corresponding process. Then, ISR places the packet into the softirq vector array (`softirq_vec`) according to its corresponding priority. Packets are stored in the queue in a decreasing order of priorities. The further processing of a packet is then deferred to the bottom half.

In the bottom half, a kernel thread (`ksoftirqd`) periodically checks out the softirq vector array for set pending bits. When the pending bit for network packet reception is set, our modified `ksoftirqd` goes through the packet data queue of `softirq_vec` to handle packets one by one, as long as their corresponding priorities are no lower than the priority of the currently executed process. Note that the packet data queue is sorted according to packet's corresponding priorities. When our modified `ksoftirqd` meets a packet with a priority lower than that of the currently executed process, it stops taking care of incoming packets.

Figure 2 shows our new interrupt service approach, in comparison to the original Linux. By using our interrupt service routine, we can reduce interference of interrupt handling to processes by as much as $T_b * N_l - T_d * N_h$, where T_b is the bottom half executing time, T_d is the extra time cost in top half for early demux, and N_h and N_l are the number of packets for higher and lower priority processes, respectively. According to its design, top half extra work is much smaller than that of bottom half, so interference to processes can decrease when N_l is smaller then N_h. It means our interrupt service model can provide less interference to higher priority process.

5 Experimental Evaluation

This section presents experimental results in order to show that our approach is suitable in the real-time environment.

5.1 Experiment Setting

To implement our approach, we make minimum modifications to the Linux kernel and network device driver. We patched Linux kernel 2.6.23. All experiments are performed on the QEMU emulator with a 2.8GHz x86 single core processor and virtual NIC, which is interconnected with a host machine. The host machine has 2.8GHz AMD Phenom CPU, 4GB main memory.

There are two processes executing concurrently. They are a UDP server and a dummy job process. The UDP server handles burst of the packets, and the dummy job process executes Algorithm 1. We set the priority of the UDP server to 20, which follows default Linux settings. We measure execution time of the dummy job processes with priority 17(LP) and priority 21(HP), which are higher and lower than UDP server, respectively. We use a UDP packet as a network workload. The UDP packets are sent from a host machine through the virtual network device. We perform experiments in the original Linux and in our patched Linux, respectively.

Algorithm 1 Dummy Job Algorithm

```
start_time ← current_clock();
i ← ∞;
while(i--) { i←(i+i)/2+(i-i)/2; }
end_time ← current_clock();
```

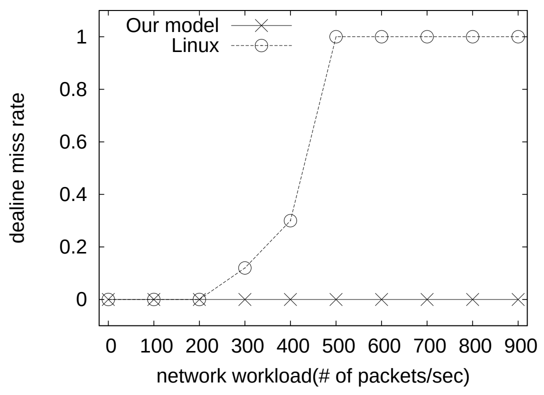

Figure 4. Comparison of deadline miss ratios

5.2 Results

First, we compare execution times of processes with different priority under different network workload. Figure 3 shows measured execution times with the increasing size of burst of packets. In the original Linux, the higher priority process (HP) and the lower priority process (LP) show similar behavior. The reason why two processes show similar behavior is that the network interrupt handling routine preempts both processes without regard to their priorities. Under our interrupt handling approach, the higher priority dummy job process (HP) shows stable execution times. This means that HP is less affected from the interrupts which are associated with the lower priority process (LP). In the top half, the early demux procedure assigns the priority of a receiver process to a bottom half interrupt handling routine. By using this information, the bottom half scheduler is able to "delay" interrupt handling if it is intended for the process with a lower priority than the current process. Such delaying bottom half interrupt handling can reduce the number of times that HP should yield to the interrupt handler. The execution time gap between under original Linux and under our patched Linux process represents how much of bottom halves has been delayed.

In our interrupt service approach, the execution time of HP is only increased about 6% when the client sends 900 packets per second. While the higher priority process shows stable behavior over different network workload, the lower priority process shows similar behavior to other processes under the original Linux setting.

Second, we compare the deadline miss ratio of processes of higher priorities than the UDP server in both Linux and our patched Linux. To measure the deadline miss ratio, we execute the dummy job periodically every 8 seconds and its deadline is 8 seconds. Figure 4 shows the result of this experiment. Under our interrupt service approach, it misses no deadlines. This shows that our interrupt service model is suitable to real-time environment, as it can reduce the interference from lower priority processes through interrupt handling.

6 Conclusion

This paper presents the design and implementation of a new Linux interrupt handling approach for incoming packets. It couples packets with the priorities of their receiver processes, and their interrupt handling is performed according to priorities. This approach is able to prevent the priority inversion problem, in particular, between the currently executed process and the receiver process of a packet under interrupt handling.

We demonstrate the effectiveness of this approach by implementing it over Linux. Experiments show that it effectively provides the predictable execution of processes of higher priorities. In this paper, only UDP packets are covered. Our future work includes accommodating more sophisticated protocols, such as TCP. While TCP employs flow control, delaying interrupt handling can defer TCP acknowledge and may put the TCP communication unstable. We plan to incorporate TCP packets addressing such concerns.

Acknowledgement

We thank anonymous reviewers for their constructive comments. This research was supported in part by IT R&D program of MKE/KEIT of Korea [2009-KI002090, Development of Technology Base for Trustworthy Computing], and KAIST ICC, KIDCS, KMCC, OLEV, and URP grants.

References

[1] T. P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, March 1991.
[2] P. Druschel and G. Banga. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation*, 1996.
[3] K. J. Jung, S. G. Jung, and C. Park. Stabilizing execution time of user processes by bottom half scheduling in linux. In *Proc. of Euromicro Conference on Real-Time Systems*, 2004.
[4] R. Love. *Linux Kernel Development*. Novell Press, 2005.
[5] G. Parmer and R. West. Predictable interrupt management and scheduling in the composite component-based system. In *RTSS*, 2008.
[6] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS*, 1988.
[7] L. Sha, J. P. Lehoczky, and R. Rajkumar. Task scheduling in distributed real-time systems. In *Proceedings of the International Conference on Industrial Electronics, Control, and Instrumentation*.
[8] Y. Zhang and R. West. Process-aware interrupt scheduling and accountability. In *RTSS*, 2006.