L4-LINUX BASED SYSTEM AS A PLATFORM FOR EPICS IOC-CORE

J. Odagiri, N. Yamamoto and T. Katoh, KEK, Oho 1-1, Tsukuba, JAPAN

Abstract

The EPICS Input/Output Controller (IOC) core-program, iocCore [1], is now portable to multi-platforms. The Linux operating system, among them, seems to be a promising candidate for a platform to run iocCore, considering the recent high appreciation in desktop and server use as well as control fields.

The Linux kernel, however, is not suitable for time-critical applications, since it responds to external events with unpredictable latency. We summarize three known causes of the latency, and then discuss some of the different solutions and how they affect the functionality of iocCore.

As a possible alternative, we propose an approach that dispatches user-level processes by a real-time kernel aiming at a consistency of availability with predictable responsiveness.

1 INTRODUCTION

In EPICS 3.14, the latest version, iocCore can run on Linux with OSI (Operating System Interface) libraries supplied as a part of the distribution [1]. The OSI libraries interface iocCore and POSIX threads (POSIX 1003.1c) with a real-time extension (POSIX 1003.1b).

However, the real-time extension can support only so-called “soft” real-time applications, where relatively rare misses to the deadline can be acceptable. It applies even if the POSIX threads are implemented on top of kernel-level threads because the sources of the unpredictability are inherent in the Linux kernel, itself.

In section 2, the causes of unpredictable latency in the Linux kernel are summarized. Section 3 describes the advantages and disadvantages of some of the real-time extensions of Linux in view of the availability to iocCore. In section 4, we propose a solution based on L4-Linux, a port of the Linux kernel onto a real-time kernel named L4 [2]. Conclusions are given in Section 5.

2 SOURCES OF UNPREDICTABLE LATENCY

There are three sources of unpredictable latency in the Linux kernel with widely different impacts: non-preemptive kernel, interrupt disabling and address space switching.

2.1 Non-preemptive Kernel

A process under Linux runs in kernel-space while it executes a system call. The term “non-preemptive kernel” implies that the Linux kernel does not switch the execution from a process in kernel-space until it invokes the scheduler in its own context. Real-time kernels, however, must switch the execution upon return from an interrupt handler if the interrupt gets a higher priority process ready to run. The Linux kernel does not allow preemption in the kernel, since it cannot protect the consistency of kernel data if multiple processes concurrently run in the kernel. A higher priority process that gets ready to run has to wait until the process currently running in the kernel exits from the kernel. While typical latency due to non-preemptiveness is known to be several tens of milliseconds, it can reach 100 milliseconds or more in the worst case [3, 4].

2.2 Interrupt Disabling

Mutual exclusion to keep the consistency of kernel data is required between a process in the kernel and an interrupt handler, and between interrupt handlers as well. Since interrupt handlers can not sleep, the only way to achieve it is to disable interrupts while the process or an interrupt handler executes the critical section. Even though the typical execution path of such critical sections is very short compared to the whole execution path of a system call, it can reach, in time, up to several hundreds of microseconds [5]. It can even be much longer in special cases [4].

2.3 Address Space Switching

Linux gives each process an independent address space, in more concrete terms, a set of mapping tables to translate a virtual address into a physical address. Some of the entries of the tables are cashed in a Translation Look aside Buffer (TLB) of the Memory Management Unit (MMU). Upon process switching, all of the cashed entries need to be flushed to invalidate the old mapping. The process newly switched in starts without having the cashed entries, and then cases TLB misses as it evolves. This brings in unpredictable latency of tens of microseconds.

3 POSSIBLE SOLUTIONS

3.1 Real-time Tasks in Linux Kernel Space

Some approaches, such as RTLinux [5] and RTAI [6], introduce a hardware abstraction layer, a small kernel, under the Linux kernel. The real-time kernel, as well as its tasks, resides in Linux kernel space, and it takes complete precedence on interrupt management and CPU scheduling over the Linux kernel. This approach is free from all of the three causes of unpredictable latency. First, the real-time tasks have essentially nothing to do with the non-preemptive-ness of the Linux kernel because they do not call for Linux services. Second, the Linux kernel is not allowed to disable interrupts in this system. Third, the real-time tasks run in the Linux kernel space, which has been mapped into address spaces of every Linux process. No matter which process an external event interrupts, the real-time tasks are ready to handle it without switching any address spaces.

In this approach, however, all iocCore programs, including the run-time database, need to be moved into the Linux kernel in order to use real-time tasks for the threads of iocCore. It seems not only unrealistic, but also unfavorable, since debugging an application easily crashes the Linux kernel. On the other hand, if iocCore resides in user-space, it must run under the standard Linux kernel.

3.2 User processes Under Improved Linux

If the causes of latency, at least the dominant latency that stems from the non-preemptive-ness, are removed from the Linux kernel, user processes can be put to use for real-time applications. This approach is the most convenient way to run iocCore, since the OSI libraries based on the POSIX threads can benefit from the improvement just by replacing the Linux kernel, as long as the POSIX threads use kernel level threads. Possible evolution of the Linux kernel is clearly isolated from that of iocCore at the POSIX interface.

The most promising way to achieve this seems to be to make the Linux kernel preemptive by converting SMP spin-locks to a measure for mutual exclusion between multiple processes on a single processor. This approach includes the changes to have the kernel honor specified priorities and to switch the process on a return from interrupt if it is possible and required. This improves the latency down to, *typically*, around one millisecond [4].

As some argue, however, the predictability of the latency still remains essentially at a soft real-time level. It is true that the requirement to the spin-lock is essentially the same regardless of whether it is for between processes on different CPUs, or for between processes on a single processor. However, it does not necessarily mean that the usage of spin-locks for SMP in the existing Linux kernel is optimized enough to ensure real-time responsiveness when it is converted. The average performance of SMP stays even if a few parts of the kernel code allow a process to run too long with holding a spin-lock. It, however, directly affects the longest latency if the spin-lock is used to make the Linux kernel preemptive. Checking up on the whole kernel codes to fix such problematic parts should take much effort, since the Linux kernel is too huge and complex to be fully analyzed. Furthermore, the check must be iterated every time any part of the Linux kernel is modified, or a new driver module is added to the kernel.

3.3 User processes under real-time kernel

The two approaches discussed in the previous subsections aim at either high predictability or high availability. To be available, the threads of iocCore must run in user-space. To be predictable, the Linux kernel must be excluded from the execution path of dispatching the threads. If a system allows threads in user-space to be dispatched by a real-time kernel, consistency of availability and predictability is possible.

This approach should be free from any unpredictable latency due to the non-preemptiveness of the Linux kernel. It must also be able to avoid the disabling of interrupts by the Linux kernel. It accepts, though, the unpredictability due to address space switching for the cost to use user-space. The threads must be able to issue real-time kernel system calls, as well as Linux system calls, when predictable responsiveness is not necessary. At present, candidates for this approach are very few. One is LXRT, an option of RTAI [6]. Another is L4-Linux, as described in the next section.

4 IOC-CORE ON L4-LINUX

4.1 What is L4-Linux?

L4-Linux was developed at Dresden Institute of Technology in corporation with IBM Watson Research Center [2]. It is a port of Linux kernel as a server task on top of a real-time micro kernel, named L4, or its successor, named Fiasco [7], as illustrated in Fig. 1.

L4 is a preemptive micro kernel, which provides its tasks with only three primitives, threads, address spaces, and Inter Process Communication (IPC). The entity of the Linux server, as well as its “processes”, is an L4 task. The Linux “processes” call the Linux server for a service through an IPC call of L4. Page faults caused by a process are also transformed into IPC with the Linux server, and then handled in the same way as standard Linux systems. Every Linux “process”, being a L4 task, can also issue L4 system calls.

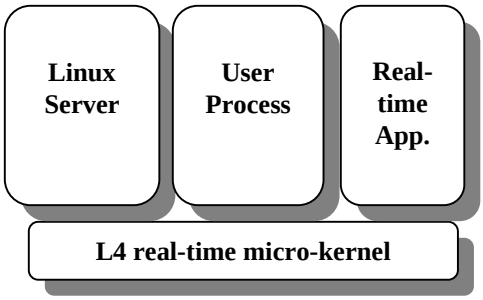

Figure 1: Architecture of L4-Linux

4.2 L4-Linux as a real-time platform

L4-Linux was designed in order to run a real-time application and non-real-time Linux applications on a single computer. Linux processes, hence, in themselves are not for real-time applications. However, the basic architecture of L4-Linux allows a process to be turned into a real-time process, basically, if it is given a higher priority than the Linux server. The process can preempt execution from the Linux server because both the "process" and the Linux server are just a task under L4's scheduling. The "process" can keep running, as long as it does not issue any Linux system calls, until it suspends itself by calling an L4 system call, or being preempted by another real-time process with higher priority.

A problem, unfortunately, was found in relation to virtual memory management. In L4-Linux, there are two different sets of page tables for the virtual memory space of a process. One is in the Linux server and the other is in the L4 kernel. The former is the one that the Linux server manipulates as does the standard Linux kernel. The latter is the real page tables that the MMU refers to. The two sets of page tables are forced to be equal upon page faults. This doubly layered memory management has two negative impacts on "real-time processes". First, a newly created thread through a Linux clone system call shares the page tables with its creator, not in the L4 kernel, but in the Linux server. The cloned thread gets its own page tables in the L4 kernel. This results in the need for address space switching upon "thread switching". The other one, which is more serious, is a coherency problem between the two sets of page tables. For example, a process can cause page faults even after a mlockall system call has been issued to make itself memory-resident, because mlockall returns with only the page tables in the Linux server updated, but leaving the real ones unchanged. Page faults to update the real page tables decimate the assumption that real-time processes run without depending on the Linux server.

With our patches to work around the coherency problem, interrupt response was measured with heavy disk I/O activity as a background. Including 3 times of "thread switching" into the real-time thread in the worst case, the worst latency was measured to be roughly 700 microseconds in 10^5 times of trials on a Celeron 300 MHz CPU [8]. It also includes the latency due to disabling of interrupt by the Linux server. L4-Linux can be configured so as not to allow the Linux server to disable interrupts. If this option is enabled and the number of context switching is reduced, the latency may be reduced down to around 100 microseconds.

4.3 iocCore on L4-Linux

In order to confirm the ability of the system to run iocCore, OSI-libraries for L4-Linux were implemented, though this is not complete. Each of their functions relies only on L4 system calls to preserve real-time responsiveness, as far as it is supposed to be called constantly from the iocCore threads. Some other functions that create threads or semaphores issue Linux system calls for resource allocation at a cost of losing the responsiveness temporarily, assuming that they are called only in the initialization sequences of iocCore. Channel Access- related threads, however, rely on Linux for TCP/IP socket services, and they work in much the same way as they do on the standard Linux kernel. Other than that, iocCore threads can run with lower predictable latency under L4 kernel control.

5 CONCLUSIONS

Though real-time tasks in the Linux kernel space can ensure the lowest and most predictable latency, running iocCore in the Linux kernel is an unfavorable solution. On the contrary, an improved Linux kernel gives the highest availability to iocCore, but its responsiveness essentially remains at a soft real-time level. If a platform allows threads in user-space to be dispatched by a real-time kernel, it can provide iocCore with not only an available basis, but also predictable responsiveness. L4-Linux is one of the possible candidates for this kind of approach.

REFERENCES

[1] http://www.aps.anl.gov/epics/
[2] http://os.inf.tu-dresden.de/L4/LinuxOnL4/
[3] http://www-online.kek.jp/~nakayosi/
[4] http://www.mvista.com/realtime/
[5] http://www.rtlinux.org/
[6] http://www.aero.polimi.it/projects/rtai/
[7] http://os.inf.tu-dresden.de/fiasco/
[8] J. Odagiri et.al., "Porting EPICS to L4-Linux Based System, " PAC 2001, Chicago, USA, Jun. 2001