www.ijecs.in
International Journal Of Engineering And Computer Science ISSN: 2319-7242
Volume 5 Issue 1 January 2016, Page No. 15454-15457

An Implementation Of IPC Using Direct Thread Switching

Dongha Shin[1], Sunghoon Son[2], Eunyoung Kim[3]
[1]Department of Computer Science, Sangmyung University,20, Hongjimun 2-gil, Jongno-gu, Seoul 03016, Korea
dshin@smu.ac.kr
[3]Department of Computer Science, Sangmyung University,20, Hongjimun 2-gil, Jongno-gu, Seoul 03016, Korea
shson@smu.ac.kr
[2]Department of Computer Science, Sangmyung University,20, Hongjimun 2-gil, Jongno-gu, Seoul 03016, Korea
201211186@sangmyung.kr

Abstract: Microkernel approach restructures the existing operating system by removing all nonessential components from the kernel and implementing them as user-level programs. The main function of the microkernel is to provide a communication facility, such as inter-process communication (IPC), between the client program and the various services that are also running in user space. Since IPC occurs very often in microkernel, the performance of IPC highly affects the overall performance of the system. The performance of IPC decreases tremendously when there are many threads in ready state because many threads need to execute before IPC receiver thread begins. One of solutions to this problem is direct thread switching, which schedules the receiver thread immediately after the sender thread send a data. In this paper, we implemented synchronous IPC for microkernel and adopted direct thread switching to improve IPC performance. Carrying out extensive performance measurement studies, we showed that direct thread switching enormously improves the performance of IPC.

Keywords: IPC, microkernel, direct thread switching. ARM architecture

1. Introduction

Microkernel operating system usually implements basic mechanisms in the kernel and other policies as service programs. Since service programs have its own address space, they communicate each other via inter-process communication (IPC). IPC is a mechanism that transfers a data between source thread and destination thread. Since IPC is performed very frequently in microkernel, IPC performance is very important [1]. However when there are many threads of ready state, IPC performance is decreased because many threads need to execute before IPC destination thread execute. Many studies have been proposed to improve the IPC performance and direct thread switching is known to be effective in solving this problem. Direct thread switching can improve IPC performance by scheduling destination thread immediately after source thread sends a data to destination thread [2][4].

In this paper, we implemented synchronous IPC for microkernel and used direct thread switching to improve IPC performance. By the results of IPC performance measurement of uC/OS-II, FreeRTOS and the kernel implemented in this paper, we confirmed that IPC performance improves immensely when using direct thread switching. Also we analyzed the IPC performance differences when using direct thread switching.

In section 2, the concepts of IPC and direct thread switching are introduced. In section 3, the implementation of IPC is presented. We describe the results of performance evaluation of IPC. Finally, section 5 concludes the paper.

2. Preliminaries

In this section, we explain basic concept of synchronous IPC and direct thread switching. We also explain how to use ARM performance monitor [5] which is necessary to measure IPC execution cycle.

2.1 Synchronous IPC

For synchronous (rendezvous-style) IPC, a sender thread must wait until message delivery is completed. Once a receiver thread receives IPC message, the state of the sender thread becomes ready [3]. Synchronous IPC is a prerequisite for a number of IPC performance enhancement techniques such as temporary mapping, lazy scheduling, and direct thread switching [4]. Various L4 microkernels also support synchronous IPC as basic communication mechanism. Recent versions of L4 also provide asynchronous notification as well to provide a rich programming environment.

2.2 Direct thread switching

Generally when the current thread is ceased to run, the kernel calls thread scheduler to select the next thread to run and switches to the thread. During an IPC call, the scheduler is invoked to select the next thread to run among threads in the ready queue. If there are too many threads in the ready queue before the receiver thread is selected, the message delivery is delayed. One of the solutions to this problem is direct thread switching [2][4]. If a thread gets blocked during an IPC call, the kernel switches to a readily-identifiable runnable thread, which then executes on the original thread's time slice, usually ignoring priorities. Since there is no intervention of other threads between sender thread and receiver thread, direct thread switching improves IPC performance immensely.

However, the fact that direct thread switching ignores thread's priority sometimes causes some problems. Modern L4 versions, concerned about correct real-time behavior, retain direct thread switch where it conforms to priorities, and else invoke the thread scheduler [4].

2.3 ARM performance monitor

ARMv7 provides system performance monitor functionality using Coprocessor15 (cp15) [5]. Performance monitor consists of a cycle counter and at most 31 event counter registers. Of these 32 registers, we use PMCR, PMCNTENSET, PMOVSR, PMCCNTR, PMUSERENR, and PMIMTENCLR. In order to measure processor execution cycles for a given time interval, we first read the PMCCNTR register right before the time interval and also read the register immediately after the interval. Then, the difference of two PMCCNTR values is the processor execution cycles of the interval.

3. Implementation

In this section, we explain various fields of TCB related to IPC implementation. We also explain IPC send and receive system calls and the implementation of direct thread switching.

3.1 Thread control block (TCB)

TCB has many IPC-related fields such as tid, state, message, and rbtid/sbtid/rrwtid.

3.1.1 tid

tid field contains thread's id. It is used to identify sender thread and receiver thread during an IPC call.

3.1.2 state

The field contains the thread's state. It may have one of the values, READY, RUNNING, or WAITING.

3.1.3 message

This field stores message during an IPC call. The message is stored at message field of receiver thread's TCB.

3.1.4 rbtid, sbtid, and rwtid

There are many fields which contain the copies of tid. rbtid stores the thread id of sender thread while receiver thread is blocked during IPC receive call. sbtid stores the thread id of receiver thread while sender thread is blocked during IPC send call. rwtid is used to store the thread id of sender thread while receiver thread is blocked during IPC send call to other thread.

3.2 System calls

In this study, we implement IPC send and receive system calls.

3.2.1 send system call

The current thread uses send system call to send a message to another thread. If there exists a receiver thread that is waiting for the message, the kernel moves the receiver thread into ready queue and delivers the message to the receiver thread. If there exists no such thread, the system call requests message send and makes the current thread become blocked.

3.2.2 receive system call

The current thread uses receive system call in order to receive a message from other thread. If there already

is a sender thread, the kernel reads the message at the sender's TCB and makes the sender thread ready. If there exists no such thread, the system call requests message reception and makes the current thread blocked.

3.3 Direct thread switching

In this study, direct thread switching is implemented in receive system call. While the original receive system call returns immediately after inserting the receiver thread into ready queue, the receiver thread executes immediately after it is inserted into ready queue. The receiver thread executes during the remaining time slice of the current thread, while the current thread is inserted into ready queue.

4. Measurement and Analysis

In this section, we measure the performance of direct thread switching. For comparison, we present the IPC performance of other embedded operating system such as uC/OS-II and FreeRTOS.

4.1 IPC performance

In order to understand the effect of direct thread switching, we measures the IPC performance of four different systems: uC/OS-II[6][7], FreeRTOS [8][9], in-house microkernel (ARM-Kernel) and in-house microkernel with direct thread switch (ARM-Kernel-DTS). All the measurements were conducted on Cortex-A8 based BeagleBone board. Performance measurement program utilized ARM performance monitor registers [5] to count CPU execution cycle during an IPC call.

Table 1: IPC performance in each system

Thread types	uC/OS-II	FreeRTOS	ARM-kernel	ARM-kernel-DTS
Busy wait	-	260×10^6	297×10^6	1392
Busy wait & Sleep	92×10^6	116×10^6	120×10^6	1015
Sleep	650	2500	333	299

in cycles

Table 1 shows the IPC performance in each system. The measurement is done between the point before the system call IPC send in a sender thread and the point after the system call IPC receive in a receiver thread. The lower value means better performance. Each row of Table 1 represents the type of other threads running together with a sender and a receiver thread. The thread type 'Busy wait' means all other threads are executing a busy wait loop, the thread type 'Sleep' means all other threads are executing a sleep loop and the thread type 'Busy wait & Sleep' means other threads are executing either a busy wait loop or a sleep loop.

As shown in Table 1, the thread type 'Busy wait' has the biggest clock cycle in each system, since this case makes most other threads ready in the ready queue and the receiver thread need to wait long after the sender thread execute the system call IPC send. The thread type 'Sleep' has the smallest clock cycle in each system, since this case makes a few other threads ready in the ready queue and the receiver thread wait short in the ready queue after the sender thread execute the system call IPC send. We can see that ARM-kernel-DTS that uses IPC with direct thread switching shows best performance in all thread types.

4.2 Analysis on the effects of direct thread switching

Table 2 compares the IPC performance of ARM-kernel-DTS with ARM-kernel in order to verify the effect of direct thread switching.

Table 2: Analysis on the effects of direct thread switching

No. of threads	ARM-kernel	ARM-kernel-DTS
0	456	378
30	170×10^6	843
60	340×10^6	1025
90	510×10^6	1104
120	680×10^6	1146
150	805×10^6	1128
180	1020×10^6	1153
200	1133×10^6	1186

in cycles

In Table 2, Number of threads means the number of threads other than sender thread or receiver thread. As shown by Table 2, when direct thread switching is not enabled, the IPC performance degrades as the number of threads increase. This is because the receiver thread begins to run only after other threads in the ready queue are scheduled first. However, using direct thread switching, the number of threads cannot affect the IPC performance. On the other

hand, by direct thread switching, the length of ready queue does not affect the execution of receiver thread since the receiver thread execute as soon as it enters the ready queue.

5. Conclusions

In this paper, we developed synchronous IPC for in-house microkernel. We also implements direct switching mechanism for IPC performance enhancement. Experimental study reveals that direct thread switching improves the performance of IPC immensely. Also we analyze the IPC performance differences using direct thread switching.

Acknowledgements

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. B0101-15-0644). This paper is a revised version of a conference paper (in Korean) presented in the Proceedings of Fall Conference of Institute of Embedded Engineering of Korea, Jeju, South Korea, in November 14, 2015.

References

[1] Jochen Liedtke, Toward Real Microkernels, Communications of the ACM 39(9), 1997.
[2] Jochen Liedtke, Improving IPC by kernel design, In Proceedings of the 14th ACM Symposium on Operating System Principles, 1993.
[3] Trent Jaeger, Jonathon E. Tidswell and Alain Gefflaut, Synchronous IPC over Transparent Monitors, Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System, 2000.
[4] Kevin Elphinstone and Gernot Heiser, From L3 to seL4: What Have We Learnt in 20 Years of L4 Microkernels?, In Proceedings of the 24th ACM Symposium on Operating Systems Principles, 2013.
[5] ARM Limited, ARM Architecture Reference Manual, ARM DDI 0406C.c, Chapter C12 The Performance Monitors Extension, 2014.
[6] J. Labrosse, MicroC/OS-II: The Real-Time Kernel, 2nd Ed., CMP Books, 2002.
[7] MicroC/OS-II, http://www.micrium.com, 2015.
[8] Real Time Engineers Ltd., FreeRTOS Reference Manual – API Functions and Configuration Options, 2014.
[9] FreeRTOS, http://www.freertos.org, 2015.

Author Profile

Dongha Shin received the B.S. degree in Computer Engineering from Kyungpook National University in 1980, the M.S. degree in Computer Engineering from Seoul National University in 1982 and the Ph.D. degree in Computer Science from University of South Carolina in 1994. During 1982-1996, he stayed in ETRI as a technical staff to study expert systems, word processing systems, file systems and language processing systems. During 1997-current, he is a professor of Computer Science Department in Sangmyung University.

Sunghoon Son received the B.S., M.S. and Ph.D. degree in Computer Science from Seoul National University in 1981, 1983 and 1999 respectively. During 1999-2004, he stayed in ETRI as a technical staff to study operating systems, embedded systems and multimedia systems. During 2004-current, he is an associate professor of Computer Science Department in Sangmyung University.

Eunyoung Kim is currently an undergraduate student in Computer Science from Sangmyung University. Her research interest includes real-time kernels and embedded systems.