ISSN: 2348 - 2117

International Journal of Engineering Technology and Computer Research (IJETCR)
Available Online at www.ijetcr.org
Volume 3; Issue 2; Page No. 157-159

Review: Flexible System Call (Flex SC) Scheduling with Exception-Less System Calls

Jyotshana Upadhyay, Ayushi Agrawal

Jayoti Vidyapeeth Women's University, Jaipur, Rajasthan, India

ABSTRACT

FlexSC implements a "syscall" thread scheduler that is responsible for determining when and on which core system calls will execute. While different operating systems offer a variety of different ser- vices, the basic underlying system call mechanism has been common on all commercial multi processed operating systems for decades. This scheduler is critical to performance, as it influences the locality of user and kernel execution.

Keywords: FlexSC, Operating, System Call, API, MIPS, Kernel

INTRODUCTION

The system call provides an interface to the operating system services. They serve the services provided by the operating system or implemented in the operating system kernel. Application developers often do not have direct access to the system calls, but can access them through an application programming interface (API). The functions that are included in the API invoke the actual system calls. By using the API, certain benefits can be gained:

- **Portability:** as long a system supports an API, any program using that API can compile and run.
- **Ease of Use:** using the API can be significantly easier than using the actual system call.

1. Flexible System Call

User programs invoke system calls by executing the MIPS "syscall" instruction, which generates a hardware trap into the Nachos kernel. The Nachos/MIPS simulator implements traps by invoking the Routine *Raise Exception ()*, passing it a arguments indicating the exact cause of the trap. *Raise Exception*, in turn, calls *Exception Handler* to take care of the specific problem. *Exception Handler* is passed a single argument indicating the precise cause of the trap.

The "syscall" instruction indicates a system call is requested, but doesn't indicate *which* system call to perform. By convention, user programs place the code indicating the particular system call desired in register *r2* before executing the "syscall" instruction. Additional arguments to the system call (when appropriate) can be found in registers *r4-r7*, following the standard C procedure call linkage conventions. Function and system call return values are expected to be in register *r2* on return [1].

While different operating systems offer a variety of different ser- vices, the basic underlying system call mechanism has been common on all commercial multi processed operating systems for decades. System call invocation typically involves writing arguments to appropriate registers and then issuing a special machine instruction that raises asynchronous exception, immediately yielding user-mode execution to a kernel-mode exception handler. Two important properties of the traditional system call design are that: (1) a processor exception is used to communicate with the kernel, and (2) a synchronous execution model is enforced, as the application expects the completion of the system call before resuming user-mode execution. Both of these effects result in performance inefficiencies on modern processors.

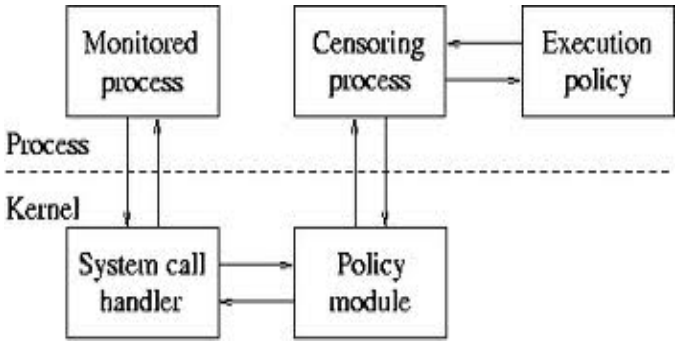

Figure 1: Flexible System Call

2. Synchronous system call

It comes mainly from two sources: one is the overhead in switching between user-mode and kernel-mode and the other is processor architecture (L1 instruction cache and data cache, L2 and L3 cache, TLB, etc.) pollution from the mode switching. They proposed exception-less system calls where a system call was implemented by posting a

Corresponding author: Jyotshana Upadhyay

system call request to memory pages (called syscall pages), without involving exceptions. The execution of system calls is done by a set of kernel only threads. Thus, the invocation of a system call is decoupled from the actual execution.

FlexSC gives two main benefits. The first is, with the decouple between the invocation of a system call and its execution, we can issue as many system calls as possible (by multi-threading: a thread will be blocked once it issues a system call. However, we can switch to other threads and stay in user-mode as long as there are ready threads.). The processor can stay in user-mode or kernel-mode much longer. Flexible system call scheduling and dynamic core specialization are also made possible [2]. The only concern I have with this paper is the benefit from FlexSC is maximized when the program is highly multi-threaded. Then, there are opportunities to batch system calls from multiple threads. For programs with very few threads but invoke a large number of system calls, FlexSC does not seem to help, each thread will be blocked for system calls and we need to switch between user mode and kernel mode frequently.

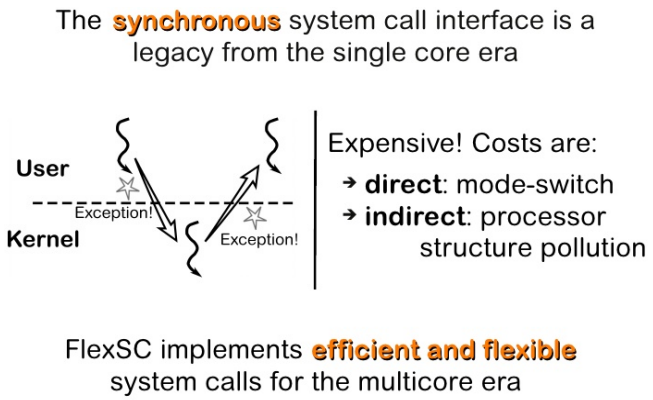

Figure 2: Synchronous system call

3. Exception-Less System Call Interface and Implementation

In this work, we argue for the use of exception-less system calls as a mechanism to improve processor efficiency while multiplexing execution between user and kernel modes in event-driven servers. Exception-less system call is a mechanism for requesting kernel services that does not require the use of synchronous processor exceptions. The key benefit of exception-less system calls is the flexibility in scheduling system call execution, ultimately providing improved locality of execution of both user and kernel code. Exception-less system calls have been shown to improve the performance of highly threaded applications, by using a specialized user-level threading package that transparently converts synchronous system calls into exception-less ones. The goal of this work is to ex-tend the original proposal by enabling the explicit use of Exception less system calls by event-driven applications. In this section, we briefly describe the original exception-less system call implementation (FlexSC) for the benefit of the reader; those readers already familiar with exception-less system calls [3]. For space considerations, this is a simple overview of exception-less system calls; for more information, we refer the reader to the original exception-less system calls proposal the implementation of FlexSC-Threads is compliant with POSIX Threads, and binary compatible with NPTL the default Linux thread library. As a result, Linux multi-threaded programs work with FlexSC-Threads "out of the box" without modification or recompilation. The performance evaluation we present focuses on popular multi-threaded server applications. In particular, we show that our implementation of exception-less system, in conjunction with our specialized threading library, improves performance of Apache by up to 116%, MySQL by up to 40%, and BIND by up to 79% while requiring no modifications to the application.

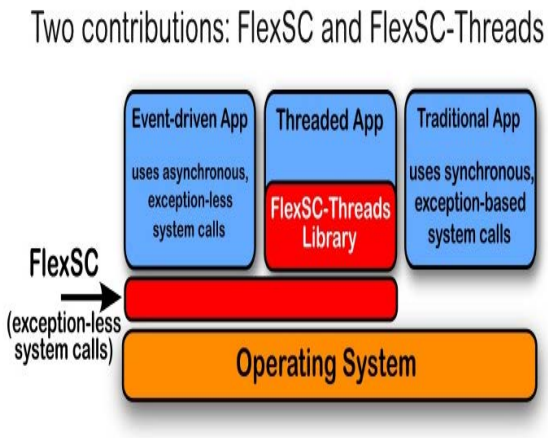

Figure 3: Exception-Less System Call Interface

4. Exception-less system calls

We develop a novel mechanism, called exception-less system call, that allows applications to request operating system services with low overhead and asynchronously schedule operating system work on multiple cores. We quantify the impact of system calls on the performance of system intensive workloads, showing that there are direct and indirect components to the overhead. We propose a new system call mechanism, exception-less system calls, that uses asynchronous communication through the memory hierarchy [4]. An implementation of exception-less system calls, called FlexSC, is described within a commodity monolithic kernel (Linux), demonstrating the applicability of the mechanism to legacy kernel architectures.

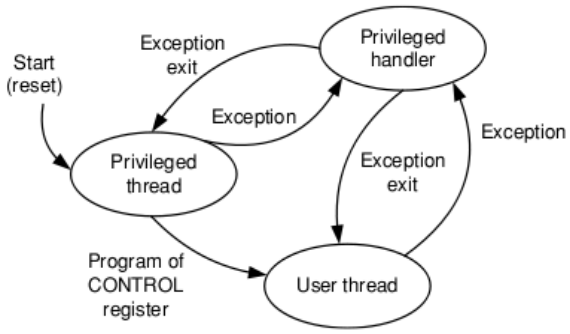

Figure 4: Exception-less system calls with threads

5. Exception-less user-level threading

We develop a new hybrid threading package, FlexSC-Threads, specifically tailored for use with exception-less system calls. The goal of the presented threading package is to translate legacy system calls to exception-less ones transparently to the application.

We experimentally evaluate the performance advantages of exception less execution on popular server applications, showing improved utilization of several processor components. In particular, our system improves performance of Apache by up to 116%, MySQL by up to 40%, and BIND by up to 79% while requiring no modifications to the applications.

6. Exception-less event driven programming

We explore exposing exception-less system calls directly to applications. To this end, we develop a library that supports the construction of event driven applications thataretailoredtorequestoperatingsystemservicesasynchr onously. We show how to port existing event-driven applications to use our new mechanism. Finally, we identify various benefits of exception-less system calls over existing operating system support for event-driven programs [5]. We show how the use of direct use of exception-less system calls can significantly improve the performance of two Internet servers, me cached and nginx. Our experiments demonstrate throughput improvements in me cached of up to 35% and nginx of up to 120%. As anticipated, experimental analysis shows that the performance improvements largely stem from increased efficiency in the use of the underlying processor when pollution is reduced.

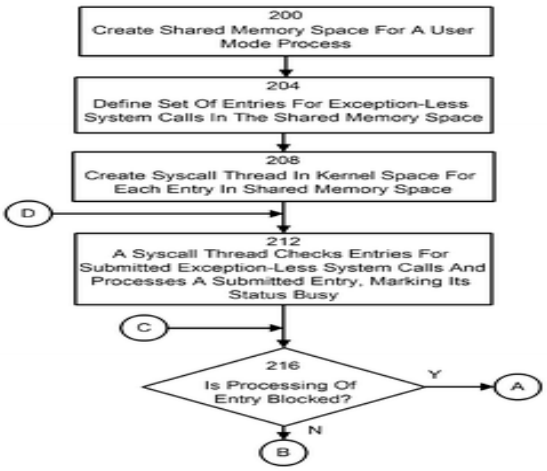

Figure 5: Exception-fewer events driven programming

Conclusion

The system call provides an interface to the operating system services. They serve the services provided by the operating system or implemented in the operating system kernel. exception-less system call, that allows applications to request operating system services with low overhead and asynchronously schedule operating system work on multiple cores.

References

1. BROWN, Z. Asynchronous system calls. In Proceedings of the Ottawa Linux Symposium (OLS) (2007), pp. 81–85.
2. CHAKRABORTY, K., WELLS, P. M., AND SOHI, G. S. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly. In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)(2006), pp. 283–292.
3. ELMELEEGY, K., CHANDA, A., COX, A. L., ANDZWAENEPOEL, W. Lazy asynchronous I/O for event-driven servers. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC) (2004), pp. 21–21.
4. M OGUL, J. C., MUDIGONDA, J., BINKERT, N., RAN-GANATHAN, P.,ANDTALWAR, V. Using asymmetric ingle-ISACMPs to save energy on operating systems .IEEE Micro 28, 3(2008), 26–41.
5. W ENTZLAFF, D., AND AGARWAL, A. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multi-cores. SIGOPS Oper. Syst. Rev. 43, 2 (2009), 76–85.