

2006-01-09

Licentiate presentation

Thank you for the introduction, Lars. I will present my licentiate thesis which is called "Performance and Implementation Complexity in Multiprocessor Operating System Kernels".

2006-01-09

Licentiate presentation

Introduction

This work has been performed as a subproject in the Blekinge Engineering Software Qualities, or the BESQ project. It has been funded by the KK foundation together with Ericsson in Älvsjö, Stockholm. My advisers are professor Lars Lundberg and Dr. Håkan Grahn. The thesis has been organized as a collection of papers, as opposed to a monograph-style thesis.

2006-01-09

Licentiate presentation

Background

Processor Trends

Multiprocessor computers are becoming more and more common. Until recently, multiprocessors generally had two or more physical processor chips. Now, however, Intel, IBM, and Sun among other companies in the processor industry are starting to produce single-chip multiprocessors. This trend has two main tracks: chip multiprocessing and simultaneous multithreading. Chip multiprocessing basically packs two or more processors on one physical processor die. Simultaneous multithreading instead partitions the processor resources and allows multiple threads to share these resources. The images show the IBM Cell and the Sun Niagara processors. The IBM cell processor at the same time incorporates simultaneous multithreading and what can be described as a programmable vector unit. The Sun Niagara chip combines chip multiprocessing with simultaneous multithreading and is capable of executing 32 threads in parallel. In the future it will therefore be increasingly important with software support for multiprocessors.

2006-01-09

Licentiate presentation

Background

Operating System Development

Workplace OS

I would also like to present a real-life story. IBM Workplace OS was a grand plan from IBM to replace the OS/400, AIX, OS/2 and DOS/Windows operating systems with one single system. This system would run on ARM, PowerPC and X86 on everything from small PDAs to large servers, even running several operating systems at once. Still, Workplace OS is almost unheard of today, and why is that? The reason is that IBM was never able to turn Workplace OS into a functioning product. Only an OS/2 personality for PowerPC which had disappointing performance and a custom UNIX personality was released. So after five years and large amounts of money spent, Workplace OS was finally canceled and this story illustrates that operating system development can be very difficult and risky.

2006-01-09

Licentiate presentation

Background

Operating System Development

Operating System Background

For those that need it, I'll next provide an operating system crash course. This figure will be coming back in various forms later during the talk. The bottom box shows the computer hardware which has one or more processors, network interfaces, harddisks, memory and so on. The next layer above contains the operating system kernel. The operating system kernel is a trusted entity that controls the hardware and provides services to programs. In a computer with several processors, it is very important that data structure in the kernel are protected against concurrent modification by multiple processors. Programs execute on top of the kernel. Each program is represented by a process which holds the resources of the program, for example open files. Each process has one or more threads, and these execute the program code. Processes are usually also tied to address spaces, which provide virtual views of the computer's physical memory. Each process therefore gets the impression of having private access to the memory, and also cannot tamper with other processes' memory.

2006-01-09

Licentiate presentation

Background

Research Questions

This thesis has been organized around four research questions, one primary and three secondary research questions. The primary research question is:

How can we find a good tradeoff between performance and development effort when porting a uniprocessor operating system kernel to a multiprocessor and how can we evaluate it in an efficient way?

2006-01-09

- Licentiate presentation
 - Background
 - Research Questions
 - Research Questions, II

Research Questions, II

- Research Question 1 (RQ1)
What is the cost and performance benefit of performing a traditional symmetric lock-based multiprocessor port of an operating system kernel?
- Research Question 2 (RQ2)
What are the lower limits of development effort when performing a multiprocessor port of an operating system kernel? Can multiprocessor support be added to the operating system without modifying the original kernel?
- Research Question 3 (RQ3)
How is it possible to lower the perturbation caused by program instrumentation?

The first secondary research question examines traditional multiprocessor ports: *What is the cost and performance benefit of performing a traditional symmetric lock-based multiprocessor port of an operating system kernel?*

The next secondary research question then examines alternative porting approaches: *What are the lower limits of development effort when performing a multiprocessor port of an operating system kernel? Can multiprocessor support be added to the operating system without modifying the original kernel?*

And finally the third secondary research question is related to tool support for the development of complex software systems such as operating system kernels: *How is it possible to lower the perturbation caused by program instrumentation?*

2006-01-09

- Licentiate presentation
 - Background
 - Research Questions
 - Outline of the papers

Outline of the papers

- Paper I: Scalability vs. Development Effort for Multiprocessor Operating System Kernels. Submitted to the Journal of Systems and Software.
- Paper II: The Design and Implementation of Multiprocessor Support for an Industrial Operating System Kernel. Submitted to the Journal of Systems Architecture.
- Paper III: The Application Kernel Approach - A Novel Approach for Adding SMP Support to Embedded Operating Systems. Submitted to Software Practice and Experience.
- Paper IV: Automatic Low Overhead Program Instrumentation with an API Framework. Presented at the Workshop on Operating System Support, Computer and Computer Architecture, 2005.

The papers in this thesis are the following. Paper I is a survey of the development effort and performance for multiprocessor operating systems and also contains a case study of the multiprocessor evolution of Linux. Paper II presents a port of a uniprocessor operating system to a multiprocessor, and is an extended version of a paper published at the Embedded and Real-Time Computing Systems and Applications conference in August this year. The third paper describes an alternative multiprocessor porting approach and is an extended version of a paper published at the International Parallel and Distributed Processing Symposium in 2004. The fourth paper, finally, describes a framework for program instrumentation and was published at a workshop in February this year.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper I – Multiprocessor OS Survey
 - Overview of Paper I

Overview of Paper I

- Paper I presents a survey of multiprocessor operating system implementations.
- The paper also contains a case study of the multiprocessor evolution of the Linux kernel.
- Contributions
- Comparison of development effort and performance for the techniques.
- Case study of Linux multiprocessor implementation.

Paper I is a survey of multiprocessor operating system implementations, focusing on multiprocessor ports but the paper also presents a case study of the evolution of multiprocessor support for the Linux kernel.

There are three contributions of this paper. First, the paper categorizes existing operating systems into seven implementation techniques. Second, a comparison of development effort and performance for the various techniques is provided, and finally, the Linux study provides insight into the evolution of multiprocessor support for an operating system kernel.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper I – Multiprocessor OS Survey
 - Categorization of Multiprocessor OSes

Categorization of Multiprocessor OSes

Totally 22 operating systems were categorized and the results are shown here. The figures illustrates the organization of the different systems and can be found in the thesis on pages 23–27. 11 of the studied systems use lock-based approaches where locks protect the kernel from concurrent access from more than one processor. Two older systems are completely lock-free, that is implemented using lock-free algorithms, and three systems use asymmetric approaches where processors are responsible for different tasks. One system is based on an ABI-compatible reimplementation focusing on performance for large multiprocessors. A more and more popular approach is virtualization. With virtualization, a layer called the Hypervisor is placed between the hardware and the operating system kernel. Virtualization allows for running many operating systems on shared hardware, and thereby potentially organizing a large multiprocessor as a cluster. The implementation effort and the performance gained varies a lot between the different operating system organizations, which I will show next.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper I – Multiprocessor OS Survey
 - Design Space

Design Space

Note that this figure is an approximation based on source code size, implementation issues and cited experiences. However, we can see that giant locking, that is having a single lock for the entire kernel, together with some asymmetric systems probably provide the lowest implementation effort at the cost of fairly bad scalability. Relaxing the locking scheme provides better performance but also progressively requires more implementation effort. Lock-free systems require a combination of hardware support and complex algorithms, and pure lock-free systems are therefore hard to get scaling on general-purpose hardware. However, lock-free techniques have started to appear even in lock-based systems as a complement to the locks for certain operations. Virtualized systems generally requires low effort since the original kernel can be kept mostly unchanged and can still get good scalability for suitable workloads.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper I – Multiprocessor OS Survey
 - Linux Case Study

Linux Case Study

- The Linux multiprocessor implementation has gone through a transition from giant locking to fine-grained locking.
- 2.0: giant locking
- 2.2: coarse-grained locking
- 2.4: fine-grained locking
- 2.6: the grand locking (initial)
- Performance and implementation was compared between the different versions.
- We measured performance with the postmark benchmark.

I will next talk about the Linux case study. Linux has evolved from a giant locking implementation in version 2.0 through a more coarse-grained implementation in 2.2 to a fine-grained implementation in 2.6. In the Linux case study, we looked at performance, locking structure and implementation size of versions from 2.0 through 2.6. We examine common parts throughout the versions, focusing on the core kernel, the Intel x86 implementation and the ext2 filesystem. The performance was evaluated with the postmark benchmark, which is a benchmark that tries to mimic the behavior of an Internet mail server, which stresses the handling of small files. This benchmark spends much time in-kernel, and therefore requires a good multiprocessor implementation to achieve scalability. I will show the results of the performance benchmark next.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper I – Multiprocessor OS Survey
 - Linux Case Study, Results

Linux Case Study, Results

- The uniprocessor performance for uniprocessor greatly since 2.0
- 2.6 has the best uniprocessor performance
- 2.0 and 2.2 does not scale at all for this benchmark
- 2.6 has very good scalability

This graph shows the relative performance compared to the 2.0 kernel on a uniprocessor. As you can see, there is a large scalability improvement from the 2.0 to the 2.6 versions. While 2.0 and 2.2 doesn't scale at all, 2.6 has very good scalability all the way to 8 processors. Another thing you can note is that the uniprocessor performance is much better with newer kernels.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper I – Multiprocessor OS Survey
 - Linux Case Study, Results II

Linux Case Study, Results II

Kernel	File	locking	locking	locking	locking	locking	locking
2.0.0	100	100	100	0	0	0	0
2.2.0	100	100	100	0	0	0	0
2.4.0	100	100	100	0	0	0	0
2.6.0	100	100	100	0	0	0	0

- The number of lock operations in the kernel reflects the implementation strategy
- 2.0 has only one lock, taken in very few places
- Newer kernels relaxed the giant lock

It is clear that the performance of the Linux kernel has greatly improved, but the implementation has also become significantly larger. The latest 2.0 kernel consisted of less than one million lines of code, while the 2.6 kernel has more than six million lines of code. This table shows the number of lock operations of different kinds in the IA-32, ext2, kernel and memory management parts of the Linux kernel. In the same way, the code size is the number of lines without comments in these parts. We can first see that the giant lock in 2.0 affects very few places in the kernel. In 2.2, the giant locking was relaxed and the places the lock needed to be taken increased greatly. In 2.4 and 2.6 the giant lock has increasingly been replaced by more fine-grained locks, which explains the decrease in the use of the giant lock. At the same time, 2.6 like 2.2 also adds new locking primitives for more efficient handling of some cases.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper II – Traditional MP Port
 - Overview of Paper II

Overview of Paper II

- Paper II presents a traditional multiprocessor operating system port
- The port was performed as an industrial operating system kernel
- We used a traditional lock-based approach for protection of the operating system kernel
- Identifications from this paper
- Technical solutions to common problems in multiprocessor operating systems
- Insights of implementation experience from this port

In this paper, a traditional multiprocessor operating system port using a giant lock is presented. The contributions from this paper is first and foremost that it illustrates technical solutions to common problems in multiprocessor ports but also that it discusses experiences gained during the port.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper II – Traditional MP Port
 - The Industrial Operating System Kernel

The Industrial Operating System Kernel

- Distributed, multi-processor cluster system
- Distributed DBM-qualified database
- Standalone user on telecom systems

This figure shows the general structure of the industrial operating system kernel, which I will call the in-house kernel. The operating system is a cluster system mainly used for telecom applications, and it provides good performance via a in-RAM database and fault tolerance through data replication. There are two kernels used in the cluster, Linux and the in-house kernel. Linux is mainly used as a gateway to the outside world while the in-house kernel generally is used in the processing nodes.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper II – Traditional MP Port
 - The Industrial Operating System Kernel, II

The Industrial Operating System Kernel, II

- The operating system is based around three basic entities
 - Containers
 - Address spaces
 - Processes
- The programming model relies on fast processes
 - Minimal shared state handling, small memory requirements
 - Minimal on disk footprint in memory
 - No paging to disk

Contrary to most other kernels, the in-house kernel separates the concepts of address spaces and processes. The address spaces are instead represented by containers, each of which can hold one or more processes, although there normally is a one-to-one mapping between processes and containers. The programming model encourages short-lived processes since you usually start a new process to handle incoming work. The in-house kernel has therefore optimized the process setup to provide fast processes turnaround. For example, the application code always resides in main memory so there will be no page faults on application code and processes also start with a minimal amount of memory allocated. I will next describe the multiprocessor implementation.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper II – Traditional MP Port
 - Overview of the Multiprocessor Port

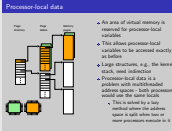
Overview of the Multiprocessor Port

- A giant locking scheme was used for the port
 - Single lock acquired for entire kernel
 - Shared in Linux 2.0
- Hardware interrupt handler (interrupt) is related to one processor
- Portions of the code, e.g., sharing global support and processor being needed to be modified for support multiprocessor
- The reason for this simple approach was to simplify the first part and provide a ground to improve the port later

The implementation of multiprocessor support for the in-house kernel uses a giant locking scheme which is similar to what early versions of the Linux kernel used. This means that the kernel is protected by a single lock which is taken on entering the kernel and released when exiting the kernel again. Another simplification is that hardware interrupts are routed to one processor, with the exception of timer interrupts which are processor-local to activate the scheduler for each processor. We also had to make a number of modifications of the original kernel, for example to add startup code for the processors in the system. The whole reason of using such a simple approach was to get a first version of the multiprocessor kernel running and improving on it later.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper II – Traditional MP Port
 - Processor-local data




Some kernel data, like the currently running thread, need to be processor-local. We have use a virtual memory approach to handle processor-local data which reserves a 4KB region of virtual memory for processor-locals. The advantage with this is that by placing data in this region, the data can still be accessed exactly as before. Since large structures such as the kernel stack does not fit in the 4KB area, a level of indirection has been added for these so that they are accessed through a processor-local pointer to the data, but this still affects very few places in the kernel.

The virtual memory approach to processor-local variables present a special problem for multithreaded address spaces since these would use the same processor-local variables. This has been solved by a lazy method where the address space is copied for each new processor that enters the address space. This is still a fairly uncommon situation since most processes are single-threaded in the in-house kernel.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper II – Traditional MP Port
 - Development experiences

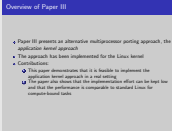


I will now discuss some of the experiences gained from porting the in-house kernel to a multiprocessor. The operating system code is quite large, around 2.5 million lines of code, although the core parts of the kernel that were relevant for the multiprocessor port were smaller, around 160,000 lines of code. The modifications ended up fairly small, with only a few thousand lines of code added and modified, but the implementation still took around two years for me working part-time on the project. The long duration of the project has several reasons, first of all that the system is specialized and quite complex to setup and configure. Another reason is that the work was done off-site, which complicated the interaction with the core developers.

Also, the performance achieved was worse than expected. We saw only 20% improvement for a two-processor machine over the uniprocessor performance in a fairly basic benchmark. So these experiences suggests that there is a need for alternative implementation approaches.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper III – Application Kernel
 - Overview of Paper III

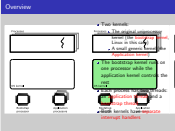


Paper III presents the application kernel approach, which is an asymmetric alternative to the more traditional multiprocessor porting methods. In an earlier publication, I described a prototype implementation of the application kernel approach for a small toy kernel, but the paper included in the thesis describes a more complete implementation for the Linux kernel.

There are two contributions from this paper. First, it shows that it is feasible to implement the application kernel approach in a real setting and second, it also shows that multiprocessor support can be added to a kernel while still keeping the implementation effort low.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper III – Application Kernel
 - Overview



I will next give an overview which shows how the application kernel works. There are two kernels active in this approach. First the original kernel, Linux in this case, that runs on the first processor in the system and will be called the bootstrap kernel from now on.

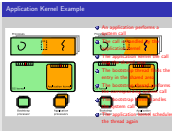
Second, there is a small generic kernel which runs on the other processors in the system and is called the application kernel. The application kernel is inserted as a driver into the bootstrap kernel.

Each process has two threads, one application thread which runs the actual application code and one bootstrap thread which handles communication with the bootstrap kernel, for example through system calls or page faults, and these two threads communicate through a shared area in the processes memory.

Both kernels have separate interrupt handlers, which means that a system call or page fault by the application thread will always end up on the application kernel and vice versa for the bootstrap thread.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper III – Application Kernel
 - Application Kernel Example

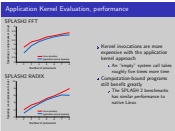


I will now give an example on how the application kernel approach works. In the figure, the active entity is shown with blue. If we assume that the application thread is running some program, it will sooner or later perform a system call. Because of the processor-local interrupt handlers, the call will end up in the application kernel, which enters information about the call into the shared memory area.

At some point in the future, the bootstrap thread will wake up and finds a message in the shared area, and will thereafter perform the corresponding system call. The bootstrap kernel then handles the system call as usually, and when it is finished, the bootstrap thread will thereafter notify the application kernel that it is done and the application kernel can schedule the thread again.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper III – Application Kernel
 - Application Kernel Evaluation, performance



The application kernel approach has similarities with a few other systems. For example, master-slave systems also assign one processor to handle kernel-interaction, but differs from the application kernel approach in that the implementation is added to the original kernel whereas the application kernel is a separate entity.

Just like for master-slave systems, there is an additional cost associated with forwarding kernel operations to another processor. We measured the time an empty system call, and that takes around five times longer under the application kernel approach compared to native Linux. You should note that this varies depending on the type of system call as well as the system load.

Computationally intensive programs can still benefit greatly with the application kernel approach. The graphs here show the speedup from two benchmarks in the SPLASH 2 benchmark suite, and as you can see, the application kernel scalability is similar to that of native Linux. For the RADIX benchmark, we can see that the bootstrap kernel becomes saturated after seven processors.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper IV – Program Instrumentation
 - Overview of Paper IV

Overview of Paper IV

- This paper presents the LOPI framework for binary application instrumentation
- Instrumentation can be used for performance measurement, security, debugging, code coverage analysis etc.
- LOPI tries to minimize the execution time and perturbation of instrumented code
- Main contribution
- Showing optimized implementations for common instrumentation cases

Paper IV presents the LOPI framework for binary instrumentation of applications. Paper IV focuses on support tools for the development of large software systems, so it takes a different aspect than the other papers in the thesis.

Instrumentation means adding probes to a program to study its execution, and is a very useful approach when for example measuring the performance of a program. It is important that instrumentation adds as little perturbation, that is affects, the program behavior as possible, for example to avoid making the instrumented program extremely slow. So LOPI therefore tries to minimize the perturbation caused by instrumentation framework, and the main contribution in this paper is to illustrate the implementation of a number of optimizations for common instrumentation cases.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper IV – Program Instrumentation
 - The LOPI approach

The LOPI approach

- LOPI is a package which provides a low-level interface to instrumenting applications
- The package is general and the user provides the type of instrumentation performed
- The package currently provides optimized instrumentation of function entry and exit

This figure shows the process of instrumenting a program with LOPI. LOPI provides a low-level interface which works on binary object files between the compiling and linking stages of the build process, so LOPI does not require source code of the program to be instrumented. The user of the framework implements the actual instrumentation code which is then called by the framework, and currently the framework provides optimized instrumentation of function entry and exit.

I will next explain some of the optimizations LOPI implements.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper IV – Program Instrumentation
 - LOPI Optimizations

LOPI Optimizations

- Instead of generating new stubs for each instrumentation point, LOPI reuses stubs whenever possible
- LOPI uses one or several stubs for the common case of stack setup
- LOPI performs the instrumentation of function return during runtime by overwriting the return address
- The return stubs are ordered in a stack to encourage reuse

When performing the function instrumentation, LOPI overwrites the function prologue shown in the figure with a call to an instrumentation stub. The purpose of the stub is to call the user-specified instrumentation and then re-execute the overwritten instructions. If there are many instrumented functions, these stubs will often be identical and LOPI then reuses an existing stub. Also, there is a special stub for the most common case of stack setup, which in our set of benchmarks was used in around 80% of the functions.

The function return instrumentation is done during runtime by overwriting the return address, and the return stubs are ordered as a stack to encourage memory reuse.

2006-01-09

- Licentiate presentation
 - Papers
 - Paper IV – Program Instrumentation
 - Evaluation

Evaluation

- We measured LOPI on the SPEC benchmark suite
- Our results show less instructions and fewer cache misses than Dyninst in the case of slightly new branch prediction system

We evaluated LOPI against the state-of-the-art Dyninst instrumentation library. The graphs here show a trace of the quake benchmark from the SPEC benchmark suite which compares the instrumented program with a non-instrumented one, and you can find more graphs and a complete description in the paper. It can be seen that LOPI on the left executes fewer additional instruction cycles than Dyninst.

2006-01-09

- Licentiate presentation
 - Conclusions and future work
 - Conclusions

Conclusions

- PCI overhead between performance and perturbation level
- Future work: how to provide better support for instrumentation
- PCI overhead (SP levels)
- Performance measurement can be improved
- PCI overhead (return address)
- PCI overhead (return address)
- PCI overhead (return address)
- PCI overhead (return address)
- PCI overhead (return address)
- PCI overhead (return address)

The conclusions are organized after the research questions in the thesis. When answering the primary research question about performance and development effort, I will reuse the figure from Paper I. So from the thesis and this presentation it should be clear that there exists a large design space of implementation possibilities which offer different tradeoffs.

For the first secondary research question, we saw in Paper I and Paper II that although traditional methods can be beneficial, they also often require a significant implementation effort. We further saw that the application kernel approach in the third paper seems to be a feasible approach to use when focusing on low effort. Finally, Paper IV demonstrates a number of low-level optimizations which can be used to reduce the perturbation caused by program instrumentation.

2006-01-09

- Licentiate presentation
 - Conclusions and future work
 - Future Work

Future Work

- The application kernel
- Optimization of the application kernel approach
- The industrial operating system
- Reduction of the perturbation
- Support tools
- Virtualized operating system based debugging

The first area of future work is to look into a generalization of the application kernel approach. The vision is to be able to create drop-in multiprocessor support for operating systems with only minimal wrappers for the specific OS. Another possibility is to optimize the application kernel implementation, for example through dynamically moving system-bound applications to the original uniprocessor kernel.

The industrial operating system port could also be improved by relaxing the locking scheme, and also through looking at alternative porting methods such as virtualization techniques.

Another important area for support tools is in kernel debugging, where it would be interesting to look into for example simulator-assisted debugging which could allow debugging backwards in time.

The area we are currently looking into is virtualization techniques, in particular using the Xen Hypervisor and running the in-house kernel on top of it.