Experiences from Implementing Multiprocessor Support for an Industrial Operating System Kernel

Simon Kågström, Håkan Grahn, and Lars Lundberg

Department of Systems and Software Engineering, School of Engineering
Blekinge Institute of Technology, Ronneby, Sweden
{ska, hgr, llu}@bth.se

Abstract

The ongoing transition from uniprocessor to multiprocessor computers requires operating system support. There is a large body of specialized operating systems which require porting in order to work on multiprocessors. In this paper we describe the design and implementation of a multiprocessor port of a cluster operating system kernel. The multiprocessor support is implemented with a giant locking scheme, which allowed us to get an initial working version with only minor changes to the original code. We also discuss performance and implementation experiences.

1. Introduction

A current trend in the computer industry is the transition from uniprocessors to various kinds of multiprocessors. Apart from SMPs, many manufacturers are now presenting chip multiprocessors or simultaneous multithreaded CPUs [5] which allow more efficient use of chip area. This trend requires support from operating systems.

We are working on a project together with a producer of large industrial systems in providing multiprocessor support for an operating system kernel. The operating system is proprietary fault-tolerant industrial operating system primarily used in telecommunications with (soft) real-time response time which runs on clusters of uniprocessor Intel IA-32 computers. The system can use either the Linux kernel or an in-house kernel, which performs better than Linux. In this paper, we describe the design and implementation of the initial multiprocessor support for the in-house kernel. More technical details about the implementation can be found in [1]. Some structure names and terms were changed to keep the anonymity of our industrial partner.

The rest of the paper is structured as follows. Section 2.1 describes the operating system. Section 3 discusses the design of the multiprocessor support and Section 4 describes a performance evaluation. We thereafter discuss some experiences we made in Section 5 and conclude in Section 6.

2. The Operating System

2.1. General OS structure

Figure 1 shows the architecture of the operating system. The system exports a C++ or Java API to application programmers for the clusterware. The clusterware runs on top of one of the two kernels and provides access to a replicated RAM-resident database, cluster management, and a CORBA object broker. The cluster consists of processing nodes and gateway nodes. Processing nodes handle the workload and usually run the in-house kernel, while gateway nodes run Linux and act as front-ends to the cluster. The gateway nodes also provide logging support for the cluster nodes and do regular database backups to disk.

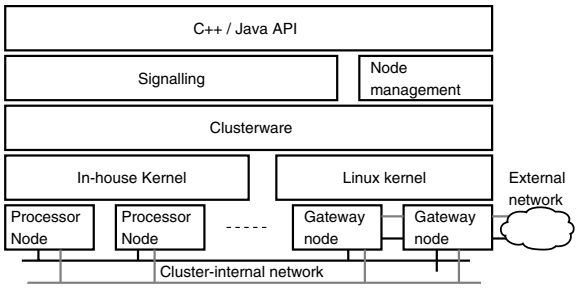

Figure 1. The operating system architecture.

The operating system employs an asynchronous event-driven programming model, with short-lived processes activated by incoming workload. The system supports two types of processes, *static* and *dynamic*. Static processes are restarted on failure and can either be unique (one for the entire cluster) or replicated (one per node in the cluster, for lower communication costs). Dynamic processes are created when referenced by another process and usually run short jobs, e.g., checking and updating an entry in the database. Dynamic processes are often tied to database objects on the local node for fast access to the database.

Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)
1533-2306/05 $20.00 © 2005 **IEEE**

2.2. The Process and Memory Model

The in-house kernel base user programs on three basic entities: *threads*, *processes*, and *containers*. There is kernel-level support for threading, and threads define the basic unit of execution. The in-house kernel separates the concepts of process and protection domains. Processes are resource holders, while containers define the protection domain (an address space). To support the asynchronous programming model, the kernel supplies very fast creation and termination of processes. There are several mechanisms behind the fast process handling. Each code package (object code) is located at a unique virtual address range in the address space and all code packages also reside permanently in memory. This allows fast setup of new containers since no new memory mappings are needed for object code and also means that there will never be any page faults on application code. Further, the in-house kernel does not use disk swapping, which simplifies page fault handling and reduces page fault latency.

The paging system uses a two-level paging structure on the IA-32 where 4KB pages are mapped into 4MB *page tables* which in turn are mapped into the complete 4GB address space in a *page directory*. A global page directory containing application code, kernel code, and kernel data is created during kernel startup, and this page directory can thereafter serve as basis for subsequent page directories since containers share most of the address space. Containers start with only two memory pages allocated, one containing the page table and the other the first 4KB of the process stack. Because of this, the container can use the global page directory, only replacing the entry mapping the process stack, global variables, and part of the heap.

3. Design of the Multiprocessor Support

For the first multiprocessor implementation, we employ a simple locking scheme where the entire kernel is protected by a single, "giant" lock (see Chapter 10 in [4]). The giant lock is acquired when the kernel is entered and released again on kernel exit. The advantage of the giant locking mechanism is small changes to the code since most of the uniprocessor semantics of the kernel can be kept. For the initial version, we deemed this important for correctness reasons and to get a working version early. However, the giant lock has performance shortcomings especially for kernel-bound workloads.

3.1. CPU-local Data

Some structures in the kernel need to be accessed privately by each CPU. For example, the currently running thread and the kernel stack must be local to each CPU. To avoid having to convert every private structure into an array, we adopted an approach where each CPU always runs in a private address space, accessing CPU-local data in private memory. To achieve this, we reserve a 4KB virtual address range for CPU-local data and map this page to different physical pages for each CPU. We modified structure declarations to place the data in a special ELF-section, which then is mapped and page-aligned by the boot loader.

The CPU-local page approach presents a few problems, however. First, some CPU-local structures are too large to fit in one page of memory. Second, handling of multithreaded processes must be modified with our approach as described in the next section. The kernel stack, which is 128KB per CPU, is one structure which is too large to store in the CPU-local page. The address of the kernel stack is only needed at a few places, however, so we added a level of indirection to set the stack pointer register through a CPU-local pointer to the kernel stack top.

3.2. Multithreaded Processes

The CPU-local page presents a problem for multithreaded containers. Using a single address space for all CPUs on a multiprocessor would cause the CPU-local virtual page to map to the same physical page for all CPUs, i.e., the CPU-local variables would be the same for all CPUs. To solve this problem, a multithreaded container needs a separate page directory for every CPU executing threads in the container. Since multithreaded containers are fairly rare, we chose a lazy method for handling multithreaded containers. Our method allows singlethreaded containers to run with the same memory requirements as before, while multithreaded containers require one extra memory page per CPU which executes in the container. The method requires only minimal scheduler changes.

Figure 2 shows the handling of multithreaded containers on multiprocessors in the in-house kernel. The figure shows the container memory data structure, which (as before) has a container page directory pointer and an initial page directory entry, but is extended with an array of per-CPU page directory pointers. When the process starts up it will have only one thread as in Figure 2a. The singlethreaded process starts without a private page directory and instead uses the global page directory. The global page directory is modified with a page table for the process stack, global variables and part of the heap. This state will be kept as long as the process is singlethreaded and uses moderate amounts of heap or stack space.

When the process becomes multithreaded the first time, as shown in Figure 2b, a new *container page directory* is allocated and copied from the global page directory, with the current CPU set as owner. Apart from setting the owner, this step works exactly as in the uniprocessor version and

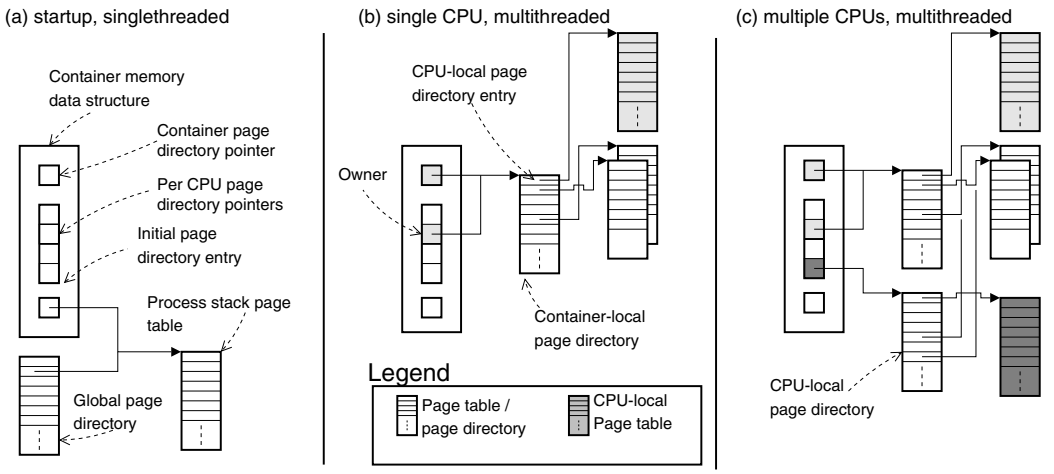

Figure 2. Handling of container address spaces in the in-house kernel for multiprocessor computers

since thread stacks reside outside the 4MB process stack area, multithreaded processes will soon need a private address space even on uniprocessor hardware. As long as only one CPU executes in the process, only one page directory will be used. However, when another CPU schedules a thread in the process, a single page directory is no longer safe. The container page directory must be copied to a new CPU-local page directory, requiring one page directory per CPU active in the container. This is shown in Figure 2c.

One complication with this scheme is page fault handling. If two or more CPUs run in a container, a page fault will be generated for the CPU-local page directory. We therefore modified the page fault handler to always update the container page directory beside the CPU-local page directory. However, there can still be inconsistencies if the fault is caused by the owner of the container page directory. A later access on the same page from another CPU will then cause a spurious page fault. We handle this situation lazily in the page fault handler by checking if the page was already mapped in the container page directory, in which case we just copy the entry to the faulting page directory. This situation is fairly uncommon since it only affects page directory faults, i.e., unmapped 4MB areas.

4. Evaluation

We have performed an initial evaluation of our multiprocessor implementation where we evaluate contention on our locking scheme as well as the performance of the multiprocessor port. We ran all performance measurements on a two-way 300MHz Pentium II SMP. For the performance evaluation, we constructed a benchmark application with two processes executing a loop. The loop performs system calls at configurable intervals so that we can vary the proportion of user to kernel execution. Apart from the benchmark processes, around 100 system threads handling database replication, logging etc., were also running in the system. We measure the time from the start of the benchmark application until it finishes.

Consistent speedups on the multiprocessor is seen only when our benchmark application executes almost completely in user-mode, so the presented results refer to the case when the benchmark processes run only in user-mode. Executing the benchmark with the multiprocessor kernel on a uniprocessor gives a modest slowdown of around 2%, which suggests that our implementation can be used even on uniprocessor hardware. Running the benchmark on the multiprocessor gives a 20% speedup over the uniprocessor kernel, which was less than we expected. Since the two benchmark processes run completely in user-mode and does not interact with each other, we expected a speedup close to 2.0.

Table 1. Kernel/user time for UP and SMP.

	User-mode	Kernel	Spinning
UP	64%	36%	< 0.1%
SMP	55%-59%	20%-22%	20-23%

Table 1 shows the lock contention during the benchmark run, both for the uniprocessor (when acquiring the lock always succeeds directly) and for the multiprocessor. From the table, we can see that the multiprocessor spends 20%-23% of the time spinning for the giant lock. Since the in-kernel time is serialized by the giant lock, the theoretically maximum speedup we can achieve on a dual processor system is $\frac{36+64}{36+\frac{64}{2}} \approx 1.47$ according to Amdahl's law. There are several reasons why the speedup is only 1.2 for our benchmark. First, the in-kernel time is high due to running sys-

tem processes. Second, some heavily accessed shared kernel structures, e.g., the ready queue, cause cache lines to move between processors, and third, as a consequence of the high in-kernel time, time is wasted spinning on the lock.

5. Implementation Experiences

The implementation of multiprocessor support for the in-house kernel was more time consuming then we had expected. The project has been ongoing part-time for two years, during which a single developer has performed the multiprocessor implementation whereas we initially expected a first version within approximately six months. The reasons for the delay are manifold.

First, the development of a multiprocessor kernel is generally harder then a uniprocessor kernel because of inherent mutual exclusion issues. We also performed most of the implementation off-site, which made it harder to get assistance from the core developers. A highly specialized and complex build and setup process led us to spend significant amount of time on configuration issues and build problems. Further, the code base consists of over 2.5 million lines of code, of which around 160,000 were relevant for our purposes. The complexity and volume of the code meant that we had to spend a lot of time to understand the code.

We did consider a number of other approaches apart from the giant lock for the initial implementation. Master-slave systems (refer to Chapter 9 in [4]) also allow only one processor in the kernel at a time. In master-slave systems, one "master" processor is dedicated to handling kernel operations while the other processors ("slaves") only run user-level applications. Similarly, the application kernel approach [2] executes all kernel operations on one processor, but allows the uniprocessor kernel to be kept as-is while multiprocessor support is added as a loadable kernel module. Neither of these approaches provide any additional performance benefit over giant locking, and incrementally improving the giant locking with finer-grained strategies is easier.

In the end, around 1% of the relevant code base was modified. We wrote around 2,300 lines of code in new files and modified 1,600 existing lines for the implementation. The new code implement processor startup and support for the locking scheme whereas the modified lines implement CPU-local data, acquiring and releasing the giant lock, etc.

Future work related to the multiprocessor port will center around the following. Since the speedup with the giant lock is low, we will investigate a more fine-grained locking scheme, starting with coarse subsystem locks. First, we are planning to use a separate lock for low-level interrupt handling to get lower interrupt latency. Another area of possible improvements is the scheduler were we will investigate dividing the common ready queue into one queue per processor as e.g., recent versions of Linux do [3]. Finally, we would like to further explore CPU-affinity optimizations for short-lived processes. Currently, processes will not be moved from the processor they are started on, but for short-lived processes it could also be beneficial to restart them on the last processor they executed on.

6. Conclusions

In this paper, we have described the design and implementation of a multiprocessor port of a cluster operating system kernel. Since our focus was to get an initial version with low engineering effort, we chose a simple "giant" locking scheme where a single lock protects the entire kernel. Our model where CPU-local variables are placed in a virtual address range mapped privately on different CPUs limited the source code changes and we also showed how this method can be applied to multithreaded processes with a very small additional memory penalty. Our experience illustrates that refactoring of a large and complex uniprocessor kernel for multiprocessor operation is a substantial undertaking, but also that it is possible to implement multiprocessor support without intrusive changes to the original kernel, changing around 1% of the core parts of the kernel.

Acknowledgments

The authors would like to thank the anonymous reviewers and the PAARTS-group for their valuable comments. This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (`http://www.bth.se/besq`).

References

[1] S. Kågström, H. Grahn, and L. Lundberg. The Design and Implementation of Multiprocessor Support for an Industrial Operating System Kernel. Technical Report 2005:3, ISSN: 1103-1581, Blekinge Institute of Technology.

[2] S. Kågström, L. Lundberg, and H. Grahn. A novel method for adding multiprocessor support to a large and complex uniprocessor kernel. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, USA, April 2004.

[3] R. Love. *Linux Kernel Development*. Sams, Indianapolis, Indiana, 1st edition, 2003.

[4] C. Schimmel. *UNIX Systems for Modern Architectures*. Addison-Wesley, Boston, first edition, 1994.

[5] L. Spracklen and S. G. Abraham. Chip multithreading: Opportunities and challenges. In *Proceedings of the 11th International Conference on High-Performance Computer Architecture (HPCA-11)*, pages 248–252, San Francisco, CA, USA, February 2005.