DEPARTMENT OF INFORMATION TECHNOLOGY

SELF STUDY REPORT

ON

BARRELFISH: OPERATING SYSTEM

PRESENTED BY–

Rishabh Dev

2K14/IT/056

CERTIFICATE

I (Rishabh Dev) hereby solemnly affirm that the project report entitled "**BARRELFISH OPERATING SYSTEM"** is being submitted by me in partial fulfilment of the requirements for the award of the degree of *Bachelor of Technology* in Computer Engineering, to the *Delhi Technological University*, is a record of bona fide work carried out by me under the guidance of Ms. Geetanjali Bhola. The work reported in this report in full or in part has not been submitted to any University or Institute for the award of any degree or diploma.

Place: DTU, Bawana Road, Delhi-110042

Date: 2 March 2016

ACKNOWLEDGEMENT

I would like to express my greatest gratitude to the people who have helped and supported me throughout my project.

I am grateful to my guide, Ms. Geetanjali Bhola for her continuous support for the project, from initial advice to contacts in the early stage of conceptual inception and through ongoing advice and encouragement to this day.

I wish to thank them for her undivided support and interest, which inspired and encouraged me to go the right way.

At last, I want to thank my friends who appreciated me for my work and motivated me, and finally to God who made all the things possible.

Rishabh Dev

2K14/IT/56

CONTENTS

INTRODUCTION: BARRELFISH

Computer hardware is changing and diversifying faster than system software. A diverse mix of cores, caches, interconnect links, IO devices and accelerators, combined with increasing core counts, leads to substantial scalability and correctness challenges for Operating system designers. Such hardware, while in some regards similar to earlier parallel systems, is new in the general-purpose computing domain. We increasingly find multicore systems in a variety of environments ranging from personal computing platforms to data centers, with workloads that are less predictable, and often more OS-intensive, than traditional high-performance computing applications. It is no longer acceptable (or useful) to tune a general purpose

OS design for a particular hardware model: the deployed hardware varies wildly, and optimizations become obsolete after a few years when new hardware arrives. Moreover, these optimizations involve tradeoffs specific to hardware parameters such as the cache hierarchy, the memory consistency model, and relative costs of local and remote cache access, and so are not portable between different hardware types. Often, they are not even applicable to future generations of the same architecture. Typically, because of these difficulties, a scalability problem must affect a substantial group of users before it will receive developer attention.

We attribute these engineering difficulties to the basic structure of a shared-memory kernel with data structures protected by locks, and in this paper we argue for rethinking the structure of the OS as a distributed system of functional units communicating via explicit messages. We identify three design principles:

(1)make all inter-core communication explicit,

(2) make Operating system structure hardware-neutral,

(3) view state as replicated instead of shared.

The model we develop, called a multikernel is not only a better match to the underlying hardware (which is networked, heterogeneous, and dynamic), but allows us to apply insights from distributed systems to the problems of scale, adaptivity , and diversity in operating systems for future hardware. Even on present systems with efficient cache-coherent shared memory, building an OS using message-based rather than shared data communication offers tangible benefits: instead of sequentially manipulating shared data structures, which is limited by the latency of remote data access, the ability to pipeline and batch messages encoding remote operations allows a single core to achieve greater throughput and reduces interconnect utilization. Furthermore, the concept naturally accommodates heterogeneous hardware.

The contributions of this work are as follows:

• We introduce the multikernel model and the design principles of explicit communication, hardware-neutral structure, and state replication.

• We present a multikernel, Barrelfish, which explores the implications

of applying the model to a concrete OS implementation.

• We show through measurement that Barrelfish satisfies our

goals of scalability and adaptability to hardware characteristics,

while providing competitive performance on contemporary

hardware.

Main goals of an Operating System -

1. Convenience:-

The primary goals of operating system is to make computer system easier for user i.e. Operating system makes interaction b/w user and hardware.

2. Efficiency:-

The secondary goal of Operating system is to allocate the system resources to various application program as efficient as possible.

3.Portable:-

Make application software portable and versatile.

4.Security:-

Provide isolation, security and protection among user programs.

5.Reliability:-

Improve overall system reliability.

WHY DO WE NEED BARRELFISH:

- Cores are increasingly diverse

Concerns with power consumption and heat management have limited the ability of chip manufacturers to continue to provide more processing power via faster clock speeds. Consequently, to deliver the ever-increasing performance to which we've all grown accustomed, processor makers have turned to developing chips with multiple cores to gain efficiency.

The problem is that current operating systems were not designed to support computers with large numbers of processing cores. Efforts are afoot to make existing operating systems work well on existing hardware, but such attempts are incremental at best, and within five or 10 years, we'll need a new paradigm. Computer hardware is changing and diversifying faster than system software. A diverse mix of cores, caches, interconnect links, IO devices and accelerators, combined with increasing core counts, leads to substantial scalability and correctness challenges for OS designers.

The developed model, called a multikernel , is not only a better match to the underlying hardware but allows us to apply insights from distributed systems to the problems of scale, adaptivity, and diversity in operating systems for future hardware.

Even on present systems with efficient cache-coherent shared memory, building an OS using message-based rather than shared-data communication offers tangible benefits: instead of sequentially manipulating shared data structures, which is limited by the latency of remote data access, the ability to pipeline and batch

messages encoding remote operations allows a single core to achieve greater throughput and reduces interconnect utilization.

- Messages cost less than shared memory

Of late, shared-memory systems have been the best fit for PC hardware in terms of both performance and good software engineering, but this trend is reversing. This shows scalability issues for cache-coherent shared memory on even a small number of cores. Although current Operating systems have point-solutions for this problem, which work on specific platforms or software systems, we believe the inherent lack of scalability of the shared memory model, combined with the rate of innovation we see in hardware, will create increasingly intractable software engineering problems for OS kernels.

- Messages are getting easier

the convenience of shared data is somewhat superficial. There are correctness and performance pitfalls when using shared data structures, and in scalable shared-memory programs (particularly high-performance scientific computing applications), expert developers are very careful about details such as lock granularity and how fields are laid out within structures. By fine-tuning code at a low level, one can minimize the cache lines needed to hold the shared data and reduce contention for cache line ownership. This reduces interconnect bandwidth and the number of processor stall incurred when cache contents are stale.

OS ARCHITECTURE

In this section we present our OS architecture for heterogeneous multicore machines, which we call the multikernel model. In a nutshell, we structure the OS as a distributed system of cores that communicate using messages and share no memory. The multikernel model is guided by three design principles:

1. Make all inter-core communication explicit.
2. Make OS structure hardware-neutral.
3. View state as replicated instead of shared.

These principles allow the OS to benefit from the distributed systems approach to gain improved performance, natural support for hardware heterogeneity, greater modularity, and the ability to reuse algorithms developed for distributed systems.

1. Make inter-core communication explicit

Within a multikernel OS, all inter-core communication is performed using explicit messages. A corollary is that no memory is shared between the code running on each core, except for that used for messaging channels. As we have seen, using messages to access or update state rapidly becomes more efficient than shared memory access as the number of cache-lines involved increases. We can expect such effects to become more pronounced in the future.

This approach also enables the OS to provide isolation and resource management on heterogeneous cores, or to schedule jobs

effectively on arbitrary inter-core topologies by placing tasks with reference to communication patterns and network effects. Furthermore, the message abstraction is a basic requirement for spanning cores which are not cache-coherent, or do not even share memory. Finally, a system based on explicit communication is amenable to human or automated analysis. The structure of a message passing system is naturally modular, because components communicate only through well-defined interfaces. Consequently it can be evolved and refined more easily.

2. Make OS structure hardware-neutral

A multikernel separates the OS structure as much as possible from the hardware. This means that there are just two aspects of the OS as a whole that are targeted at specific machine architectures – the messaging transport mechanisms, and the interface to hardware (CPUs and devices). This has several important potential benefits. Firstly, adapting the OS to run on hardware with new performance characteristics will not require extensive, cross-cutting changes to the code base (as was the case with recent scalability enhancements to Linux and Windows). This will become increasingly important as deployed systems become more diverse.

A final advantage is to enable late binding of both the protocol implementation and message transport. For example, different transports may be used to cores on IO links, or the implementation may be fitted to the observed workload by adjusting queue lengths

or polling frequency.

3. View state as replicated

Operating systems maintain state, some of which, such as the Windows dispatcher database or Linux scheduler queues, must be accessible on multiple processors. Traditionally that state exists as shared data structures protected by locks, however, in a multikernel, explicit communication between cores that share no memory leads naturally to a model of global OS state replicated across cores. Finally, a potentially important optimization of the multikernel model (which we do not pursue in this paper) is to privately share a replica of system state between a group of closely-coupled cores or hardware threads, protected by a shared-memory synchronization technique like spinlocks. In this way we can introduce (limited) sharing behind the interface as an optimization of replication.

IMPLEMENTATION

While Barrelfish is a point in the multikernel design space, it is not the only way to build a multikernel. In this section we describe our implementation, and note which choices in the design are derived from the model and which are motivated for other reasons, such as local performance, ease of engineering, policy freedom, etc. – we have liberally borrowed ideas from many other operating systems.

1. System structure

The multikernel model calls for multiple independent OS instances communicating via explicit messages. In Barrelfish, we factor the OS instance on each core into a privileged-mode CPU driver and a distinguished user-mode monitor process. CPU drivers are purely local to a core, and all inter-core coordination is performed by monitors. The distributed system of monitors and their associated CPU drivers encapsulate the functionality found in a typical monolithic microkernel: scheduling, communication, and low-level resource allocation.

2. CPU drivers

The CPU driver enforces protection, performs authorization, time slices processes, and mediates access to the core and its associated hardware (MMU, APIC, etc.). Since it shares no state with other cores, the CPU driver can be completely event-driven, single threaded, and non-pre-emptible. It serially processes events in the form of traps from user processes or interrupts from devices or other cores. This means in turn that it is easier to write and debug

than a conventional kernel, and is small enabling its text and data to be located in core-local memory.

3. Monitors

Monitors collectively coordinate system-wide state, and encapsulate much of the mechanism and policy that would be found in the kernel of a traditional OS. The monitors are single-core, user-space processes and therefore schedulable. Hence they are well suited to the split-phase, message-oriented inter-core communication of the multikernel model, in particular handling queues of messages, and long-running remote operations.

On each core, replicated data structures, such as memory allocation tables and address space mappings, are kept globally consistent by means of an agreement protocol run by the monitors. Application requests that access global state are handled by the monitors, which mediate access to remote copies of state.

4. Memory management

Although a multikernel OS is itself distributed, it must consistently manage a set of global resources, such as physical memory. In particular, because user-level applications and system services may make use of shared memory across multiple cores, and because OS code and data is itself stored in the same memory, the allocation of physical memory within the machine must be consistent – for example, the system must ensure that one user process can never

acquire a virtual mapping to a region of memory used to store a hardware page table or other OS object.

However, one benefit has been uniformity: most operations requiring global coordination in Barrelfish can be cast as instances of capability copying or retyping, allowing the use of generic consistency mechanisms in the monitors. These operations are not specific to capabilities, and we would have to support them with any other accounting scheme.

Page mapping and remapping is an operation which requires global coordination – if an address space mapping is removed or its rights are reduced, it is important that no stale information remains in a core's TLB before any action occurs that requires the operation to have completed. This is implemented by a one-phase commit operation between all the monitors.

A more complex problem is capability retyping, of which revocation is a special case. This corresponds to changing the usage of an area of memory and requires global coordination, since retyping the same capability in different ways (e.g. a mappable frame and a page table) on different cores leads to an inconsistent system. All cores must agree on a single ordering of the operations to preserve safety, and in this case, the monitors initiate a two-phase commit protocol to ensure that all changes to memory usage are consistently ordered across the processors.

CONCLUDING REMARKS

It would be wrong to draw any quantitative conclusions from our large-scale benchmarks; the systems involved are very different. An enormous investment has been made in optimizing Linux and Windows for current hardware, and conversely our system is inevitably more lightweight (it is new, and less complete). Instead, they should be read as indication that Barrelfish performs reasonably on contemporary hardware.

We make stronger claims for the micro benchmarks. Barrelfish can scale well with core count for these operations, and can easily adapt to use more efficient communication patterns (for example, tailoring multicast to the cache architecture and hardware topology). Finally we can also demonstrate the benefits of pipelining and batching of request messages without requiring changes to the OS code performing the operations.

Since the Barrelfish user environment includes standard C and math libraries, virtual memory management, and subsets of the POSIX threads and file IO APIs, porting applications is mostly straightforward. In the course of this evaluation we ported a web server, network stack, and various drivers, applications and

libraries to Barrelfish, which gives us confidence that our OS design offers a feasible alternative to existing monolithic systems.

Nevertheless, bringing up a new OS from scratch is a substantial undertaking, and limits the extent to which we can fully evaluate the multikernel architecture. In particular, this evaluation does not address complex application workloads, or higher-level operating system services such as a storage system. Moreover, we have not evaluated the system's scalability beyond currently-available commodity hardware, or its ability to integrate heterogeneous cores.

Computer hardware is changing faster than system software, and in particular operating systems. Current OS structure is tuned for a coherent shared memory with a limited number of homogeneous processors, and is poorly suited to efficiently manage the diversity and scale of future hardware architectures.

Since multicore machines increasingly resemble complex networked systems, we have proposed the multikernel architecture as a way forward. We view the OS as, first and foremost, a distributed system which may be amenable to local optimizations, rather than centralized system which must somehow be scaled to the network like environment of a modern or future machine. By basing the OS design on replicated data, message-based communication between cores, and split-phase operations, we can apply a wealth of experience and knowledge from distributed systems and networking to

the challenges posed by hardware trends.

Barrelfish, an initial, relatively unoptimized implementation of the multikernel, already demonstrates many of the benefits, while delivering performance on today's hardware competitive with existing, mature, monolithic kernels.

References

- http://www.barrelfish.org/
- Avi Silberschatz,Peter Baer Galvin and Greg Gagne, Operating System Concepts Eight Edition
- P. E. McKenney and J. Walpole. Introducing technology into the Linux kernel: A case study. Operating Systems Review, July 2008
- http://www.barrelfish.org/bffaq.html
- B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximising locality and concurrency in a shared memory multiprocessor operating system. In Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation.
- S. Borkar. Thousand core chips: a technology perspective. In Proceedings of the 44th Annual Design Automation Conference.
- https://en.wikipedia.org/wiki/Operating_system