ECE1759
The V++ Cache Kernel

Summary by: Kevin Harris

1 Introduction

The micro-kernels of today have not been living up to expectations: they have been slower, larger and more unreliable than their creators had in mind. This has been due to the fact that "optimizations" have been made which have compromised the ideal of the micro-kernel by adding code for performance sections of the OS into the kernel. On the other hand, monolithic kernels often have continued to provide the same headaches over and over: the code is too large and unmanageable. The solution to these problems, as posed by David R. Cheriton and his student Kenneth J. Duda is the supervisor mode of the V++ kernel: the V++ Cache Kernel. The Cache Kernel give user level control of system resources through the use of application kernels, like many other micro-kernels. However, unlike most other micro-kernels, much of the implementation of address spaces and threads is also left to the micro-kernel. The Cache Kernel caches system objects in much the same way hardware caches memory. The application kernels are used as the backing store for objects not in the Cache Kernel. This dynamic description of what is "in" the kernel provides a more flexible environment for applications to run in. On the other hand, like many of the operating systems we have seen so far, it leaves much of the complicated "operating systems" coding to application programmers, which may leave the advantages of the Cache Kernel either inaccessible, or unrealized.

In the next section I will briefly discuss the overall design of the kernel. The following section will deal with the objects which are manipulated within the Cache Kernel. Section 4 deals briefly with how exceptions are handled. Section 5 is a quick description of how IPC is achieved in the Cache Kernel. Section 6 addresses the resource management that the Cache Kernel does. Real time issues are discussed in section 7, and finally section 8 concludes the paper.

2 Kernel Design

The Cache Kernel was designed to be a small, fast, micro-kernel implementing only the interface required for caching objects in and out of the kernel , and as few administrative functions as possible. The application kernel is responsible for keeping all state data associated with address spaces and threads within itself. The only state data passed to the kernel upon caching are the performance critical ones. The allocation of physical resources is done in a hierarchical manner, with each application kernel being granted a portion of some resource, and the application kernel administering it (usually dividing it among its children, or applications).

3 What is Cached?

The Cache Kernel supports three basic types of operating system objects, which can be loaded and unloaded from the Cache Kernel: address space objects, thread objects, and kernel objects. Each of these objects contains the encapsulated state of whatever it represents. For example, an address space object will contain a listing of page mappings, or a thread object will contain a reference to its application kernel, as well as its stack, and registers, etc... Objects in the Cache Kernel are known as active objects. In the case of threads it means that they are ready to be executed. It is the application kernels that are responsible for load-

ing and unloading objects into and out of the Cache Kernel. When objects are written into the Cache Kernel, others may be written to backing store (the application kernel) in order to make room for the new object. Through this mechanism, the Cache Kernel avoids the hard ceiling on kernel objects (threads, address spaces) which some other operating systems have.

3.1 Threads

Thread objects that are available for execution are cached in the cache kernel. A strict priority scheduler is used, so scheduling within the kernel is relatively simple. Each application kernel may have several threads cached in the kernel at once (and therefore several ready for execution). When a thread is loaded into the Cache Kernel, its address space must already be loaded, and must be specified. Upon being loaded into the kernel, the load call returns an identifier for the thread to the application kernel, so that it may unload, query, or modify the thread, later. These identifiers are not constant over many loads into and unloads out of the kernel, they may differ every time.

The thread object contains only a little bit of the thread state (the registers and kernel stack for exceptions, and the threads priority), most of the state is held in the application kernel (such as open file table). This simplification within the kernel allows for context switching among threads to be done quickly but may incur a performance penalty when accessing data stored outside the Cache Kernel in the application kernel.

3.2 Address Spaces

Like threads, when address spaces are loaded into the Cache Kernel, the call returns and identifier to the application kernel, for reference later. An address space is basically a list of virtual to physical page mappings, which are loaded in on demand. When mappings are loaded, other mappings may need to be unloaded from the cache. If required, the application kernel chooses the victim mapping and unloads it. In this manner, the application kernel has complete control over paging strategies and memory management. When address spaces are written to backing store, all threads associated with them are automatically written to backing store as well, freeing up many identifiers of threads which would not have been active.

3.3 Kernel Objects

Kernel objects in the Cache Kernel represent the application kernels. They control the loading and unloading of the address spaces and threads to run within that application kernel, as well all exception handling and traps for their threads. All accounting within the Cache Kernel is done on a per application kernel level, so the kernel objects include much of the accounting data (such as which and how many physical pages it has been allocated, the maximum priority for its threads, and cpu usage maximums. Unloading a kernel object from the Cache Kernel is quite an expensive endeavour, because of the hierarchical nature of the design, all address spaces, and thus threads, owned by that kernel must also be unloaded. The authors envision this switching to be needed very rarely, and therefore the huge cost associated with it is inconsequential.

3.4 Locked Objects

It may be desired that some objects are never removed from the Cache Kernel for either the purposes of performance, or simply correctness. These objects may be locked into the Cache Kernel. Each application kernel has a maximum number of objects which it may lock into the kernel, these are usually exception handlers such as the $PageFaultHandler$. If the $PageFaultHandler$ were not locked into the kernel, and it were written to backing store, how would its page be retrieved?

4 Exceptions

Exceptions are handled on a per application kernel basis, each employing its own set of exception handlers, locked into the Cache Kernel. In the example of the *PageFaultHandler* used in the paper, a user-level thread faults on an access to memory. The hardware then traps to the kernel, which forwards the exception to the *PageFaultHandler* for the corresponding application kernel (as found in the thread object). The application kernel, reads the page in, and modifies the address space associated with the thread by adding the appropriate page mapping, and returns to the Cache Kernel, which reloads the thread context, and returns control to the thread.

5 IPC

The Cache Kernel implements interprocess communication through memory based messaging. The kernel provides a means for two processes to share physical memeory, and a means for delivering address valued signals to all threads which share this memory. This can be implemented in software, but is handled in hardware on the ParaDiGM hardware the Cache Kernel runs on. This removes any unneccessary copying that is often found in operating system message passing routines, removing the Cache Kernel from responsibility for the performance-critical data transfer.

6 Resource

Resources are shared between all application kernels, and are distributed to each thread and address space by the owning application kernel. The application kernels are responsible for allocating the resources fairly, and keeping track of how each resource has been allocated. To prevent denial of service attacks, a system application kernel (the System Resource Manager, or SRM) collects usage statistics and periodically recalculates application kernel attributes (like maximum priority, maximum number of physical pages).

6.1 CPU Scheduling

Scheduling within the Cache Kernel is done simply on a strict priority basis. All threads of the highest priority in the kernel share the CPU(s) through a time slicing mechanism. That is to say that any thread may not execute until all threads of a higher priority are blocked, or unloaded from the kernel. The priorities for threads are set by each individual application kernel when the thread is loaded into the kernel (or through a modify interface). The maximum priority which an application kernel can assign to one of its threads is set withing the kernel object by the SRM. The cost of execution is weighted: the cost to the application kernel for threads running with high priority is higher per unit time than that of a thread running at a lower priority. The SRM collects data concearning how much execution cost each application kernel was responsible for, and periodically changes their maximum priorities to compensate.

7 Real Time

The authors advertise that the Cache Kernel could implement a real time operating system in conjunction with other compute based applications. While they don't have any experimental data to show that it is feasible, they argue that the scheduling algorithm clearly allows real time activities. In order for a real time system to co-exist with compute based applications, the real time system would need a way of effective monopolizing the CPU when neccessary. This can be done by using the facility for setting the maximum thread priority for an application kernel. The real time system could be set higher than the maximum priority of any other application kernel. This way, when the real time application kernel loads a thread into the Cache Kernel, it would have the

highest priority, and would therefore run until it blocked, or was unloaded. Also, the allowable CPU usage for this real time application kernel could be set high enough to offset the premium for running at high priority. Depending on how critical the real time system was, maximum CPU usage could be set as high as 100%, implying that application kernel would never be penalized for using too much of the processor.

8 Conclusion

The V++ Cache Kernel can be considered a micro-kernel with twist. It is a compromise between the Exo-kernel which exports all events and attempts to do nothing but export the hardware, and most micro-kernels which implement message passing with all kernel functionality being placed in user-level servers. The Cache Kernel is making the *kernel* even smaller by leaving more of the operating system to be implemented by the application programmers (handling exceptions, memory management). Along with this smaller size, comes less complexity and better manageability.

Another benefit of the Cache Kernel lies in the greater control of resource management for applications than is offered by common monolithic kernels. Policy decisions are left upto the application kernels to implement through the loading and unloading interface offered by the Cache Kernel. Along with this comes forwarding of exceptions again, allowing greater application control.

Unfortunately, the Cache Kernel also has its drawbacks. The system is completely customizable in that the user can implement almost any policy they wish, but extensibility is lacking. Because much of the operating system is left to the application programmers to code, it may get done poorly.

The performance advantages are not very clear. The only results stated were for basic operations, it is not clear how often these basic operations will be done in a loaded version of the kernel. When compared to older monolithic kernels, the authors claim the performance it *comparable*. When the Cache Kernel becomes a full operating system, the performance is bound to degrade, leaving the results no longer *comparable*.

9 References

D.R. Cheriton, K.J. Duda. A Caching Model of Operating System Functionality. *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November, 1994

D.R. Cheriton, H. Goosen, and P.Boyle. ParaDiGM: A highly scalable shared-memory multi-computer architecture. *IEEE Computer*, 24(2), February 1991.

K. Harty and D.R. Cheriton. Application-controlled physical memory using external page cache management. In *ASPLOS*, pages 187-197. ACM, October 1992.

M.J. Zelesko and D.R. Cheriton. Specializing Object Oriented RPC for Functionality and Performance. From Proceedings of the 1996 International Conference on Distributed Computing Systems. 1996

D.R. Cheriton. The V Distributed System. *Comm.ACM*, 31(3):314-333, March 1988.