Kernel data – First class citizens of the system

Dhammika Elkaduwe, Philip Derrin and Kevin Elphinstone

National ICT Australia* and University of New South Wales
Sydney, Australia
firstname.lastname@nicta.com.au

Abstract

Kernel memory is a resource that must be managed carefully in order to ensure the efficiency and availability of the system. The use of an inappropriate policy would lead to suboptimal performance and even make the system susceptible to denial of service attacks.

In this paper, we argue that user-level managers, with their domain specific knowledge, can better manage the kernel memory consumption of their clients than a static in-kernel policy; and we present the kernel memory management scheme of *seL4*, where kernel memory is represented as named, first class objects which are created and managed by user-level managers according to a suitable policy. The scheme is flexible enough to express a wide range of policies, and allows multiple policies to coexist.

1 Introduction

Traditionally, operating systems perform two roles. They provide users with higher-level abstractions to operate upon, and they are also responsible for the arbitration of resources between competing users or applications. The abstract resources provided by the operating system require underlying hardware resources to implement the abstractions. For example, the state and metadata associated with a thread of execution is stored in a thread control block (TCB), which requires physical memory for storage. This applies similarly to other abstractions provided by the kernel, such as address spaces requiring page tables, IPC ports requiring port control blocks, capabilities requiring capability storage nodes. The resource ultimately consumed by the kernel to implement its abstractions is physical memory (we ignore devices for now), which is a limited resource. Any resource which is limited requires careful management, and kernel data is no exception.

Without careful physical memory management in the kernel, the system as a whole is susceptible to a range of problems, including denial of service attacks, poor predictability, underutilisation and general inefficiency, and covert communication channels.

Denial of service (DoS) attacks are characterised by an attacker preventing authorised users from obtaining services. The common method of launching a resource based DoS attack is to overload the system [LIJ97] with a burst of operations that consumes resources. To guard against such attacks the system must be capable of:

1. accounting resource use by identifiable entities in the system,
2. using the accounting information to detect attacks, and
3. recovering from any such attack.

Accounting for resource consumption is a non-trivial task, especially when resources are shared between entities. For example, consider a shared address space or a shared server — the kernel is ignorant of the user-level configuration and cannot account for these resources in a sensible manner. Banga et al. [BDM99] claim that accounting resources towards a process obstructs proper resource accounting in large-scale server systems. The *Scout* system [SP99], which targets multimedia, accounts resources to a *path* — a stream of data flowing through several subsystems. These exemplify the need for flexibility in resource accounting — users must be capable of selecting the resource principal and account resources towards it.

On the other hand, detecting a resource based DoS attack is hard. It is difficult to distinguish between a legitimate high load and an attack. User-level managers, with their domain specific knowledge, are able to detect such an attempt with greater precision than the kernel.

Having detected an attack, the next problem is recovery: resources must be reclaimed, either from the attacking clients or from elsewhere, to maintain system functionality. Once again, allowing a user-level manager to decide which resources to reclaim has obvious advantages — removing resources from users without knowing their importance to the system might have significant impact on performance.

Some systems require more elaborate policies rather than a simple limit on the kernel memory consumption. A real-time system, for instance, requires predictable

*National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

behaviour. However, if the kernel data of one application can be evicted from the cache by another, the predictability is lost. This warrants a cache colouring scheme over kernel data. Users should be able to define where to place the kernel objects in memory such that one application will not affect the temporal behaviour of another — a temporal partitioning policy for kernel data. Advantages of temporal partitioning in the context of real-time systems can be found elsewhere [BA03]. A default kernel policy will not be expressive enough to capture such needs — it must be sufficiently general to serve a wide range of applications, and will always provide less optimal performance than a finely tuned policy. Applications are often ill-served by the default policy [AL91, Sto81], and can benefit significantly by managing their own resources [EGK95, Han99]. With their domain specific knowledge, application level resource managers can much better manage the resource usage of their clients than a general default policy.

2 Related Work

Several operating systems manage their kernel data carefully. *Eros* [SSF99] and the *Cache kernel* [CD94] both view kernel physical memory as a cache of the kernel data and as such can evict cache entries when cache capacity is exceeded. While these systems are capable of guarding against DoS attacks, they do not guarantee predictability. Applications need to compete for space in the kernel cache, which is also a limited resource. A malicious task can flood the cache and thus degrade the performance of other tasks.

Scout [SP99] accounts resources towards a special abstraction called *path*, and limits the resource usage. Such limits would result in either underutilisation of system resources or overcommitment. The *K42* kernel [IBM02] segregates kernel memory into pinned memory and paged memory. Kernel pinned memory contains all code and data required to do paging I/O, while the rest of the kernel data can be paged out to backing stores. These systems carefully manage kernel resources in such a way as to fulfil a single overall system policy. A single policy however, is designed with a particular focus and therefore will *hurt* the performance of applications that do not fall within the original focus.

The *SPIN* system [BSP+95] allows uploading of code at run time to the kernel, thus changing the system policy. However, the policy is still global, which may not suit all the applications on the system.

Haeberlen and Elphinstone [HE03] implemented a scheme of paging kernel data from user space. When the kernel runs out of memory for a thread, it will be reflected to the corresponding *kpager*. The kpager can then map any page it possesses to the kernel, and later preempt the mapping. However, the kpagers can not control the type of data that will be placed in the mapped page, leading to performance problems if a kpager inadvertently reclaims a page containing essential data.

This paper presents the *seL4* (Secure Embedded L4) kernel-data management model. It is a microkernel which addresses the issue of kernel data management by exposing *all* dynamically allocated kernel data as first class objects. The kernel itself does not create these objects; instead it provides an interface though which sufficiently authorised user applications can create, manage and destroy them. Additionally, it provides mechanisms for delegating the authority over an object in a controlled manner. The resulting system allows user-level managers to define a suitable policy over the kernel data managed by them, and allows coexistence of different policies.

3 seL4 Overview

seL4 is a microkernel, and provides the same three basic abstractions as L4 [Lie95]: threads, address spaces and interprocess communication. These abstractions are provided via named, first-class kernel objects. Each kernel object implements a particular abstraction — a thread, for instance, is implemented by a *TCB* object. Each of these objects supports one or more operations, which authorised users may perform. Users obtain kernel services by invoking these operations on the kernel objects.

Authority over objects are conferred via capabilities [DVH66]. In brief, a capability can be viewed as an object reference coupled with a set of permissions over that object. The object reference identifies a kernel object, while the permissions dictate the kind of operations the user can perform with it. Possession of a capability with appropriate permissions is a necessary and sufficient condition for invoking operations on the corresponding kernel object.

If capabilities are to be used to confer authority, they must be tamper-proof. In seL4, they are stored inside kernel objects called *CNodes* — arrays of capabilities, which may be inspected and modified only via invocation of the CNode object itself — and therefore are guarded against user tampering. Capabilities are also immutable; while user-level programs may specify some of a capability's properties at the time it is created, those properties may only be changed by removing the capability and replacing it with another.

Each thread in the system is associated with a *capability space*, or *CSpace*. The CSpace is represented by a guarded page table [Lie94], or GPT. It is essentially a hierarchical collection of CNode objects, which contain capabilities to kernel objects (including other CNodes). The CSpace defines a local name space for the thread. It maps each valid *capability index* to an object reference and a set of permissions. User space threads invoke kernel objects by performing system calls specifying the corresponding capability index in their local name space.

3.1 seL4 Kernel Objects

Presently, the seL4 API defines six types of kernel object, associated with the abstractions it provides.

TCB objects implement threads, which are seL4's basic unit of execution.

Endpoint objects implement inter-process communication *(IPC)*. Users send and receive messages by invoking capabilities to these objects. Like L4 IPC, this operation is synchronous; each thread invoking an endpoint is suspended until a partner is available.

Asynchronous Endpoint objects are used to implement asynchronous IPC. Rather than containing queues of waiting threads, they contain a buffer which is used to store the content of a message after a sender has resumed execution.

CNode objects are arrays of 2^n (where $n > 0$) capabilities. They provide the abstraction of the capability address space, CSpace. Each CSpace is constructed from a tree of CNodes; invoking a CNode allows a user-level server to manipulate a region of an address space mapped by the tree of which that CNode is the root.

User Data objects provide storage to back virtual memory pages accessible to the user. Load and store instructions can be considered invocations of these objects. Their size may be any power of two which is at least as large as the smallest possible virtual memory mapping on the host architecture.

Untyped Memory objects provide the storage from which all other kernel objects may be allocated. Like user data objects, their size may vary, though it is limited to powers of two. They can be invoked to return capabilities to new kernel objects — including smaller untyped memory objects.

4 User Level Object Management

With the exception of a small amount of statically allocated physical memory for the kernel's stack and code, all the kernel memory is managed by user-level managers. Most importantly, dynamic allocation of memory for kernel objects is not done by the kernel. Instead, the kernel provides a secure interface, through which users can manage these objects — they are created, maintained and destroyed by user level managers. For example, a thread is created by the user-level managers by creating a TCB object, an address space by creating a CNode object, and so on. Any object creation operation must specify the area of memory that will be used to store the object; this is done by providing a capability to an Untyped Memory object.

Once an object is created, the user-level manager that created it can delegate all or part of the authority it possesses over the object to one or more of its clients. It does this by granting each client a capability to the kernel object, thereby allowing the client to obtain kernel services by invoking the object.

This model allows user-level managers to account and enforce suitable policies over the kernel memory used by their clients. The explicit creation of kernel objects allows more elaborate policies than simply limiting the kernel memory consumption of each client.

5 Future Work

We have identified a number of areas of the seL4 API which need further investigation to validate our approach.

Virtual memory implementations available on modern hardware sometimes dictate the page table format. For example, the ARM and x86 platforms feature two-level page tables. For such architectures, we believe that it will be necessary to define an interface between the hardware page table and the capability space. This may take the form of a new platform-specific kernel object type, which allows a user-level task to build page tables from its set of available capabilities.

The mapping mechanism for tracking capability derivations introduces a significant space overhead to CNodes. Furthermore, the kernel needs to traverse this structure and invalidate capabilities that point to objects within a given region of memory before creating new ones in it that region; this may, depending on the mapping structure, incur a significant time overhead. We are investigating methods of reducing these costs.

Minimising crosstalk between applications is an important issue in real-time systems. Cache pollution — where one application causes the cached working set of another to be evicted from the cache — is one place where applications can experience such unwanted interaction. We envisage reducing this by implementing a *page colouring* scheme over kernel objects.

6 Conclusions

The seL4 kernel API is free of kernel resource management policy — it does not require the kernel to make any decisions about how, where or when to allocate kernel memory. Instead, it provides a secure interface for creating, managing and destroying kernel objects from the user space. Thus, it allows user-level resource managers to account and manage the resource usage of their clients.

We believe that our mechanism is powerful and flexible enough to express a wide range of resource policies concurrently on isolated subsystems.

References

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *4th ASPLOS*, pages 96–107, 1991.

[BA03] Michael D. Bennett and Neil C. Audsley. Partitioning support for the L4 microkernel. Technical Report YCS-2003-366, Dept. of Computer Science, University of York, 2003.

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *3rd OSDI*, pages 45–58, New Orleans, LA, USA, Feb 1999. USENIX.

[BSP+95] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *15th SOSP*, pages 267–284, Copper Mountain, CO, USA, Dec 1995.

[CD94] David R. Cheriton and K. Duda. A caching model of operating system functionality. In *1st OSDI*, pages 14–17, Monterey, CA, USA, Nov 1994.

[DVH66] J.B. Dennis and E.C. Van Horn. Programming semantics for multiprogrammed computers. *CACM*, 9:143–55, 1966.

[EGK95] Dawson R. Engler, Sandeep K. Gupta, and M. Frans Kaashoek. AVM: Application-level virtual memory. In *5th HotOS*, pages 72–77, May 1995.

[Han99] Steven M. Hand. Self-paging in the Nemesis operating system. In *3rd OSDI*, pages 73–86, New Orleans, LA, USA, Feb 1999. USENIX.

[HE03] Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *8th Asia-Pacific Comp. Syst. Arch. Conf*, volume 2823 of *LNCS*, Aizu-Wakamatsu City, Japan, Sep 2003. Springer Verlag.

[IBM02] IBM K42 Team. *Utilizing Linux Kernel Components in K42*, Aug 2002. Available from http://www.research.ibm.com/K42/.

[Lie94] Jochen Liedtke. Page table structures for fine-grain virtual memory. *IEEE Technical Committee on Computer Architecture Newsletter*, 1994.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *15th SOSP*, pages 237–250, Copper Mountain, CO, USA, Dec 1995.

[LIJ97] Jochen Liedtke, Nayeem Islam, and Trent Jaeger. Preventing denial-of-service attacks on a μ-kernel for WebOSes. In *6th HotOS*, pages 73–79, Cape Cod, MA, USA, May 1997. IEEE.

[SP99] Oliver Spatscheck and Larry L. Petersen. Defending against denial of service attacks in scout. In *3rd OSDI*, New Orleans, Louisiana, Feb 1999.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *17th SOSP*, pages 170–185, Charleston, SC, USA, Dec 1999.

[Sto81] Michael Stonebraker. Operating system support for database management. *CACM*, 24:412–418, 1981.