A communication mechanism for resource isolation

Matthieu Lemerre Vincent David
CEA, LIST
LaSTRE, BP 94 91191 Gif sur Yvette
matthieu.lemerre/vincent.david@cea.fr

Guy Vidal-Naquet
SUPELEC
91190 Gif sur Yvette
guy.vidal-naquet@supelec.fr

Abstract

Sharing resources between multiple untrusted clients requires a shared service that provides access to the resources upon client requests. But executing these requests needs other resources, like memory or CPU time, which must be carefully allocated.

In this paper, we investigate a communication mechanism that allows access to shared services without changing existing allocation decisions. This is achieved by systematically using the new *resource lending* principle, that allows a service to use the resources of its clients to perform the request. We present an easily understandable design model for this communication mechanism named the *thread lending model*, that completely avoids any allocation by the service, and demonstrate its implementation in our prototype OS Anaxagoros. We finally investigate the consequences of using this model on the structure and implementation of the shared services.

1. Introduction

Many applications like multimedia systems [1], modern PDAs [2], data centers for avionics, require executing real-time and general purpose tasks on the same computer. This brings the need for a unified platform and a methodology that allows dynamic sharing of high-level resources (like network or hard drive access), while permitting the use of different allocation policies, some of which being deterministic. We say that an allocation policy is *deterministic* when the actual allocation is always the same as the requested allocation. In other words, allocation is predictable. Deterministic allocation is absolutely necessary for running safety-critical hard real-time tasks.

One important factor that breaks allocation determinism is resource consumption when accessing shared services. The service has to spend resources (e.g memory and CPU time) to handle the client requests. If these resources are not correctly managed, this can lead to denial of service: for instance this happens whenever the service allocates resources using a First-Come First-Served (FCFS) policy. Using `malloc` to store data sent by clients in microkernel servers is a good example of this. For most resources, deterministic allocation only requires exact accounting of all resources used.

Deterministic allocation of CPU time is special because it not only requires correct accounting of resources, but also not to change the scheduling decisions. For instance, if the kernel randomly encounters semaphores (sleeplocks) or randomly executes interrupt handlers, CPU time allocation becomes non-deterministic, i.e. the actual schedule becomes very different from the planned schedule.

Because of these problems, most systems aiming at deterministic allocation (e.g. RTOSes) generally do so by sticking to a simple system structure, and simple allocation policies algorithms. Most use static allocation, except for CPU time. CPU time is allocated using fixed-priority scheduling because it allows predictability in CPU allocation even when there are sleeplocks, at the cost of complex analysis. Building a full-featured, efficient system which allows deterministic allocation for arbitrary allocation policies is a challenge.

To allow deterministic allocation for any policy, we propose the *independence of policy from implementation* property. A policy is independent from the OS implementation when it is defined solely by its allocation policy algorithm. We achieve it mainly by systematic use of the *resource lending principle*, which allows transferring the right to use a resource without changing the allocation. This paper presents a communication mechanism that allows implementing this resource lending principle.

The paper is structured as follow: Section 2 presents different existing means to achieve deterministic resource allocation and explains why lending of resources is appropriate to solve this problem. Section 3 presents the design and implementation of the communication mechanism using the thread lending model. Section 4 explains how this impacts the shared services structure, and provides some guidelines for writing services that are accessed using this communication mechanism. Section 5 presents related work. Section 6 gives the state of our prototype and discuss further research directions.

IIES'09 March, 31th 2009, Nuremberg.

2. Deterministic resource allocation in presence of shared services

There are really only three ways to ensure deterministic resource allocation in presence of shared services :

2.1 Defining a new policy for the service's resources

The first possibility is to distribute the service's resources carefully among the different client requests. The service executes the client requests using its own pool resources, which we call a *subpool*. Choosing how to distribute its own resources to serve the concurrent requests is done using what we call a *subpolicy*.

For instance, a network service would reserve some amount of memory for network buffers, and a subpolicy would have to be defined to know how much each client can have. A task creation service could allow the creation of only a static number of tasks. Free address space is another example of subpool, and can be exhausted if e.g. a new memory mapping is created on each client request.

Defining subpolicies is tedious and error prone: using the FCFS policy makes the service vulnerable to denial-of-service attacks. Not using it means reserving some resources, which may remain unused and thus are wasted. In addition, separation of the resource used by a client into many different subpools makes accounting less tractable, and therefore less predictable.

Moreover, subpools are often overprovisioned, which produces internal fragmentation and wastes resources. (Stripes in Figure 1(2.1) represent CPU time lost due to fragmentation caused by using a subpool of CPU time for S, when using static scheduling).

Using subpools for CPU-time means that the server only executes requests when it is scheduled, and the clients send requests asynchronously to the service. The subpolicy determines the order of treatment of the requests (FCFS in Figure 1(2.1)). The communication thus has a high latency. Subpools of CPU time is suitable for network or disk services, which have to define a scheduling policy to access their device. But they are not suitable for access to other services such as memory or screen.

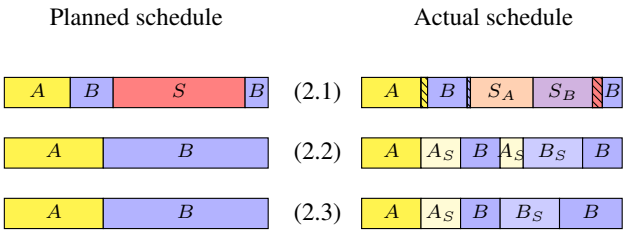


Figure 1. Accounting for CPU time spent in a service, either (2.1) by changing the planned and actual schedules and creating a subpolicy; (2.2) by allowing differences between the planned and actual schedules; or (2.3) by servicing the requests using the planned schedule. S_A is time of S spent processing requests for A, while A_S is time of A spent processing requests in S.

2.2 Making changes in allocation part of the policy

This is the most common choice for CPU time in real-time operating systems. The priority inheritance protocol or priority ceiling protocol are two examples of this: by defining the order in which a lock is taken, they allow scheduling analysis. The scheduling decisions depend not only on the scheduling algorithm, but also on the semaphores potentially encountered. Schedulability analysis becomes difficult, even on simple tasks.

Direct use of semaphores does not allow secure sharing of resources, and requires knowledge of the implementation to perform the analysis. Using synchronous communication primitives [3, 4] with priority inheritance as in [2] solve these problems (Figure 1(2.2)).

There are some drawbacks of this approach: complexity in scheduling analysis (mostly due to the presence of the semaphores), and provisions due to priority inheritance tend to oversize the CPU time needed. Finally, few scheduling algorithms allow precise analysis in the presence of sleeplocks even with priority inheritance, so a system with these services has to use one of these algorithms.

2.3 Using the existing allocations

The last alternative is to reuse the existing allocations: in other words, processing of the requests in the service is done using the resources allocated to the client it serves. This avoids the need to define subpolicies, avoids the subpool problems, and is independent of the allocation policies used. As resources are not divided into subpools and their allocation depends only on the allocation algorithms, the allocation problem for the entire system becomes far more tractable.

To use existing allocations, some kind of transfer of resources (also called resource donation [5, 6]) is necessary, so that the resources allocated to the client can be used by the service. Monolithic system calls is an example of transfer of CPU time: the processing of a `getpid()` UNIX system call is typically done when the caller is scheduled. The thread-tunneling mean of communication [7], also called doors call in Spring [8] or synchronous PCT in exokernels [9], is another example (Figure 1(2.3)).

However, for other resources like memory, transferring the resources involves the allocation policy, which has to change the resources status from "owned by the client" to "owned by the service". This requires management by the allocation policy, and thus some overhead every time resources are transferred.

To solve this problem, a solution is to *lend* resources, i.e. to give the right to use them temporarily, instead of transferring them. The client gives to the service the right to use the resources, but retains ownership: it can revoke this right to use the resources at any time. As ownership is not modified, allocation does not change and the allocation

algorithm does not need to be involved; this makes resource lending efficient.

Examples of lending of memory can bee seen, e.g. on L4 [10, 11] or EROS [12]. But we propose an OS architecture where lending of resources is the rule, not the exception. This poses serious engineering problems which will be examined below.

3. Design considerations

This section presents a communication mechanism based on resource lending.

3.1 Context

Some contextual details are necessary to understand our particular design of the communication system.

Goal. The goal of our prototype OS, Anaxagoros, is to achieve a high degree of security and isolation. Security in resource allocation is an important part, and is achieved by making all services *policy-free*: *they must not define the policy of the resource they provide, nor change the allocation of resources they do not provide*. All policies are defined by user-provided allocation algorithms.

Policy-freedom is a natural goal for operating systems, but most systems focus only on allowing replaceable policies [13], without taking into account resources spent in services to process clients requests. Not changing the allocation is reduced to the weaker goal of accounting for resources spent in the service, which is not sufficient for precise control over CPU time allocation.

Structure. The operating system structure is that of a microkernel, with all services cleanly separated. Separation of mechanism from policy is implemented by putting mechanism and policy of each kind of resource in separate services. The *accessor* service provides the mechanisms for multiplexing the resources among the applications, and the *allocator* implements the allocation policy.

The separation into small services makes it easier to ensure that accessor services do not impose any policy or change existing allocations.

Access control. A mechanism is needed to track the rights that a task has on a resource. These mechanisms can be divided into two main categories [14]: capabilities and ACL.

We chose capabilities over ACL for several reasons:

- Storage for capabilities is naturally associated to clients, rather than resources, which allows proper accounting of the memory used by the client[1]. A list of capabilities is abbreviated *C-list*.
- Capabilities are discretionary, which allows transferring rights to use a resource without notifying the service that protects this resource, and can thus implement low-overhead resource lending.
- Capabilities allow constant-time access control, which is important for real-time systems.
- Capabilities do not need a separate namespace to designate resources, thus economizing an additional namespace mechanism.

[1] However, some services require an object table that allows revoking the capability and store some state associated to a client. Entries in this object table are limited and therefore constitutes a resource that requires careful allocation.

3.2 Service call: the thread lending model

Our service call mechanism is centered around the resource lending mechanism, represented by Figure 2.

Transfer of CPU time. The principle of transfer of CPU time is that *execution of the request in the service happens when the client should have run*. This is achieved using a mechanism similar to the doors in Spring [8]. It allows low-overhead, low-latency client-service communication like L4 synchronous IPC [3, 15], without interfering with the scheduler. In particular, preemption can happen in services at anytime.

The services become by nature multi-threaded, with one thread per simultaneous client connection. All these threads consume memory for their stack, and the number of clients is not *a priori* bounded, so memory for the stack is also lent by the client to avoid introducing a memory subpolicy.

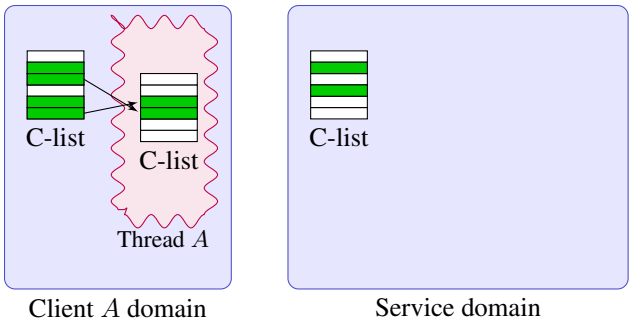
Resource lending with explicit designation. With the exception of CPU time, the resources lent to the service (represented by capabilities or page table entries) are explicitly designated by the client, like in exokernels [9]. This allows management of the resource by the client, and forbids the service to take more resources than the client wants to lend. Libraries, provided with the service, are used to know how much of a resource to lend.

CPU time is not explicitly designated because it is impossible to statically know how much time will be needed for the service to complete the request. Timeouts can be provided to limit this time, but are seldom necessary, as clients usually depend on their service and hence, trust them.

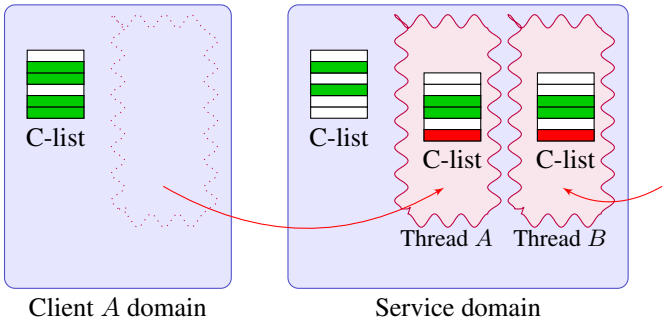
Thread lending model. When the client lends resources to the service, the metadata for access control (i.e. capabilities and page table entries) of the resources lent need to be stored in memory. So as not to consume service memory, memory for this metadata has to be lent as well.

This motivates the "thread lending model": in Anaxagoros, there are two kinds of *principals* (i.e. resource holders), the *domain* and the *thread*. A capability may be invoked if one of the current domain or the current thread possess it.

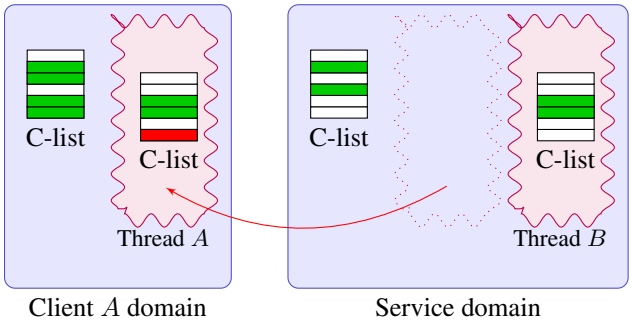
To perform a request to a service (Figure 2), the client first copies the metadata from the domain to the thread. Then the service call "lends" the thread to the service. This gives the service access to the thread metadata, as well as to its CPU time (threads are the units of dispatch of CPU time). Notice the analogy with the implementation of "passive call" in object-oriented languages: here the thread is a stack used



(a) The client copies the capabilities to lend from the C-list in its domain to the one of its thread



(b) The client lends its thread to the service. A return capability is inserted by the kernel in the C-list of the lent thread. When the thread is preempted, execution resumes in the service. Other clients may simultaneously lend their thread to the service.



(c) The service invokes the return capability, and the thread returns in the client domain. The service may eventually copy some capabilities in the thread to give them to the client.

Figure 2. Steps for service call with thread lending

to pass metadata parameters, while the service is the invoked object.

This mechanism allows *transient* lending, i.e. lending only for the duration of the call. The client can allow the service to copy some of the thread metadata into its own domain; this implements non-transient lending. It consumes service memory, though, and so needs a subpolicy: that's why we try to maximize transient lending, which is simpler.

Implementation. In the implementation, threads are composed of two memory pages. One is accessible only by the kernel, and is used to pass metadata parameters (capabilities and memory mappings). One is writable, and is used to pass data parameters and as a stack for the service. It is ensured to be mapped only in the current domain for security reasons. Services are small and a 4kB stack is more than sufficient.

In the thread metadata, there is room for two first-level page table entries. Thread-local mappings is obtained by simply copying these mappings into the first-level page table of the domain whenever a thread enters a new domain.

This model is quite efficient: on a Core 2 Duo processor, we measured 1500 cycles for a calling/returning sequence. This includes switching of address space, changing the IO ports rights, and lending a stack plus 8MB of thread-local mapping.

4. Design principles for shared services

Shared services must be policy-free, i.e. not change existing allocations and separate mechanisms from policy. But one of the main challenges is that they are by nature multi-threaded, and must deal with possible sudden revocation of lent resources. Indeed, the client requests are processed in the service using the resources lent by the requester. At anytime, the access to these resources can be revoked, e.g. because the client is externally killed.

This has three concrete consequences: the service must deal with access to revoked capabilities and memory, and revocation of CPU time can stop the service in the middle of an operation. Dealing with revoked capabilities is easy: the kernel returns an error when invoking such a capability. Handling revoked memory is easy by making the service receiving its page faults. This triggers an exception-like mechanism in the code. Handling stop during an operation is more difficult because data consistency must be ensured; but this problem can be solved using the following principles.

Data exportation. Services are designed to reduce their data to the absolute minimum. In particular, services are stateless, in the sense that they do not associate any data with a client (this would allow DoSes). Data is either global or associated to a ressource.

This is mainly achieved by exporting as much data as possible to the client, for instance by eliminating abstractions in the service (like [9]). The client sends the needed data along with the requests. This reduces the amount of memory consumed by the service, avoids most needs for non-transient lending of memory, allows simpler, shorter operations in the service, minimizes possibilities for covert channels, and alleviates most needs for notification when the client is destroyed.

RISC interface. We use a RISC-like and low-level approach to service interface. Instead of providing complex operations (e.g. `mmap` or `writev`), the service provides simple operations (e.g. like individual page-table modifications, or `write`), and means to perform several operations in batch. There are no abstractions and resources are named physically by the capability, as in exokernels [9].

This significantly simplifies the code of the service, and is more flexible as each application can optimize its use of the service. Libraries provide convenient high-level operations using these interfaces. To retain good performance, multiple operations can be batched into one service call.

Synchronization exportation. This novel design principle says that in case of several threads simultaneously accessing the same resource, *the service should ensure coherency to a state associated to a resource, only if not doing so would deny proper use of other resources*. The basic idea is that if a policy gives access to the same resource to several clients, it is their responsibility to ensure proper synchronization (e.g. mutual exclusion), not that of the service, except when it allows an attack on the service.

For instance, if multiple clients can write to the same buffer in a service, it is of no use to ensure that the write operations appear atomic, because in general the clients have to cooperate to ensure proper order of these writes anyway. However, proper reference counting of the number of times a page is mapped (as in [16]) is important, else an attacker could manage to write to a page table.

This principle has a number of advantages. First, many of these kind of synchronizations are best solved with the intervention of the scheduler (e.g. sleeplocks may put a task to sleep until the operation can be performed), which makes the scheduling policy dependent of the service. Second, the common case of unshared resource does not require any synchronization, and thus is faster.

A similar principle alleviates the need for many TLB flushes: while this does not lead to access to sensitive data (this happens when a previously writable page frame now contains this data), the memory system only performs TLB invalidations upon explicit client request.

Other synchronizations. To cope with unexpected stop in processing, one way is to ensure that all operations should only change a coherent global state into another coherent global state atomically.

Statelessness and synchronization exportation drastically reduce the amount of data that needs to be synchronized. The RISC design makes all operations require only a small incremental change of state. All this greatly simplifies synchronizations, to the extent that doing it with atomic instructions is generally simple.

When this is not the case, we provide a special "rollforward lock". In those, when a thread encounters a lock whose holder has been preempted (or destroyed), this thread execute the critical section instead of the holder, until the lock is released, before pursuing its own execution. This ensures the lock is non-blocking. All operations are done in userspace, which makes this lock efficient (overhead ≈ 40 cycles). The lock is semantically equivalent to masking interrupts in the kernel.

5. Related work

On independence of policy from implementation and lending of resources. Many previous systems tried to separate the policies from the operating system, but needed to make some compromises on this property.

Nemesis [7] reduced the dependence of CPU time scheduling policy from the implementation (called "QoS crosstalk") by minimizing the amount of processing done in shared services. Our approach extends this by also correctly managing CPU time (and other resources) *inside* these shared services.

A weak form of "independence of scheduling from implementation" can be seen as "correct CPU time accounting", which is handled by many systems (e.g. [1, 2, 17]).

Allocation of service memory to serve the client is a common problem. The L4 map operation has been used to lend memory to the kernel [10, 11] or a window system [6]. Memory can also be lent in EROS [12], although the page swapping policy is imposed. Other approaches are to limit kernel memory allocation, e.g. by caching [15, 9] , but this imposes policies.

Synchronous communication. One of the most common alternative to the thread lending model is synchronous communication [3, 4], which also allows low-latency, low-overhead communication. The natural way to use them as a communication mean between a client and a service is as a single-threaded server, in which transient resource lending is not needed: as a service cannot serve more than one client simultaneously, it can use its resources (e.g. stack or address space) without risking exhaustion.

However, synchronous communication is vulnerable to other denial of resource attacks [5, 18], which must also be carefully handled.

The blocking semantics of synchronous IPC make it equivalent to that of using a protected semaphores, and execution determinism is achieved by emulation of a protocol such as priority inheritance protocol. But as for semaphores, this renders analysis more complex.

Virtual-machine monitors. Virtual-machine monitors such as Xen [16] allow replaceable resource allocation policies which are sole responsible of the resource allocation, and are thus policy-free. But this is achieved by using static bounds on the number of virtual machines, and sharing of high-level resources like the network makes QoS crosstalk occur, because asynchronous communication is used (section 2.1).

6. Conclusion

Summary. This paper presented the motivations for communication based on resource lending, that allows the use of simple, deterministic allocation policies. We provided an easy to understand model for communication based on resource lending called the thread lending model. This model can be implemented efficiently, as is demonstrated by the first results of our prototype operating system Anaxagoros.

But it imposes some constraints on the design and implementation of shared services, that can be overcome using a set of design principles given in Section 4.

State of the prototype. Anaxagoros is a prototype implementing the communication based on the thread lending model, and with services following these guidelines. The protection and communication subsystem, the memory subsystem (which is similar to the one in Xen [16]) are written, as well as an incomplete thread and scheduling service which resembles [19], but with exact accounting of CPU time. Userspace services comprehends a VGA service, PCI service and an incomplete network service.

Discussion. Our goal is to obtain a full-featured highly-secure policy-free OS, and to prove that this goal is achievable using the proposed methodology. We are confident about performance: initial microbenchmarks are satisfactory; other systems with similar structure [9, 7] proved to be very efficient; the fine control over allocation policies should allow efficient resource management, in particular for real-time applications. Services are multithreaded and ready for multicore architectures. The overhead due to multiple protection domains can always be eliminated by regrouping them later. The only compromises to QoS crosstalk we made so far were masking interrupts or using the rollforward lock to small sequence of instructions, negligible compared to the crosstalk induced by the hardware.

The next steps are to finish the system and evaluate its performance with macro benchmarks.

Embedded systems. The presented communication mechanism can be adapted for use in embedded systems. When the system does not have MMU, the system structure can still be used as an organisational method for sharing resources among the clients. Many embedded systems now have at least a memory protection unit that provides segment-based protection without virtual memory translation. They would work really well with our sytem: e.g. lending the stack is simply giving to the service the right to use the client's stack segment.

Moreover, many optimisations can be applied when the system is static. For instance, when capabilities are never dynamically copied, they can all be stored contiguously in the kernel.

Hardware-induced crosstalk. Even if we remove all software-induced crosstalk, it still remains the hardware-induced one. This crosstalk is mainly due to the unpredicability of caches, a problem exacerbated on multicore systems. A further research direction would then be to limit this crosstalk using software techniques (like page coloring), and to study other hardware mechanisms for cache partitioning.

References

[1] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves for multimedia operating systems," Tech. Rep. CS-93-157, Carnegie Mellon University, 1993.

[2] U. Steinberg, J. Wolter, and H. Hartig, "Fast component interaction for real-time systems," in *Proceedings of ECRTS'05*, pp. 89–97, July 2005.

[3] J. Liedtke, "Improving IPC by kernel design," in *Proceedings of SOSP'93*, (Asheville, NC), Dec. 1993.

[4] J. S. Shapiro, D. J. Farber, and J. M. Smith, "The measured performance of a fast local IPC," in *Proceedings of IWOOOS '96)*, (Washington, DC, USA), p. 89, IEEE Computer Society, 1996.

[5] J. Liedtke, N. Islam, and T. Jaeger, "Preventing denial-of-service attacks on a microkernel for weboses," in *Proceedings of HotOS-VI*, (Cape Cod, MA), May 5–6 1997.

[6] N. Feske and C. Helmuth, "A Nitpicker's guide to a minimal-complexity secure GUI," in *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, (Washington, DC, USA), pp. 85–94, IEEE Computer Society, 2005.

[7] T. Roscoe, *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge, April 1995.

[8] G. Hamilton and P. Kougiouris, "The Spring nucleus: A microkernel for objects," Tech. Rep. TR-93-14, Sun Microsystems Laboratories, Inc, April 1993.

[9] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole, "Exokernel: an operating system architecture for application-level resource management," in *Proceedings of SOSP '95*, pp. 251–266, ACM Press, 1995.

[10] A. Haeberlen and K. Elphinstone, "User-level management of kernel memory," in *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, (Aizu-Wakamatsu City, Japan), Sept. 24–26 2003.

[11] D. Elkaduwe, P. Derrin, and K. Elphinstone, "Kernel design for isolation and assurance of physical memory," in *1st Workshop on Isolation and Integration in Embedded Systems (IIES'08), Glasgow, UK*, April 2008.

[12] A. Sinha, S. Sarat, and J. S. Shapiro, "Network subsystems reloaded: a high-performance, defensible network subsystem," in *Proceedings of the USENIX Annual Technical Conference 2004*, pp. 19–19, USENIX Association, 2004.

[13] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf, "Policy/mechanism separation in Hydra," in *Proceedings of SOSP '75*, (New York, NY, USA), pp. 132–140, ACM, 1975.

[14] B. Lampson, "Protection," *ACM Operating System Review*, vol. 1, pp. 18–24, January 1971.

[15] J. Shapiro, *EROS: A capability system*. PhD thesis, University of Pennsylvania, 1999.

[16] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of ACM SOSP '03:*, pp. 164–177, ACM Press, 2003.

[17] M. B. Jones, D. Rosu, and M.-C. Rosu, "CPU reservations and time constraints: efficient, predictable scheduling of independent activities," *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 198–211, 1997.

[18] J. S. Shapiro, "Vulnerabilities in synchronous IPC designs," in *IEEE Symposium on Security and Privacy, Oakland, CA*, 2003.

[19] B. Ford and S. Susarla, "CPU Inheritance Scheduling," in *Usenix OSDI'96*, pp. 91–105, 1996.