First Remarks on a Market Based Approach to Resource Scheduling for the GNU Hurd Multiserver Operating System*

Neal H. Walfield
neal@gnu.org
http://www.gnu.org/software/hurd

Libre Software Meeting, Dijon, France
July 6, 2005

Abstract

Through the use of a multiserver capability-based architecture, the GNU Hurd has attempted to increase security and flexibility relative to traditional Unix-like operating systems. This shift away from a monolithic design requires a reevaluation of conventional operating system praxis to determine its degree of continued applicability. Resource scheduling appears particularly defunct in this regard: to make smarter scheduling decisions, monolithic systems cross component boundaries to gain insight into application behavior. This introspection is incompatible with a multiserver architecture and its elimination, as observed in Mach, the current microkernel used by the GNU Hurd, results in noticeable performance degradation. To this end, I propose that rather than have the operating system provide virtualized resources, i.e. schedule the contents of resources on behalf of applications, it offer near raw access to the principals which they must multiplex as required thereby relieving e.g. the memory manager of paging decisions. The resource managers must still partition the physical resources among the competing principals. For this, I suggest a market based solution in which principals have a periodically renewed credit allowance and lease the required resources. This approach also suits adaptive and soft real-time applications.

Keywords: *Operating Systems, Security, Extensibility, Resource Management, Resource Allocation, Resource Scheduling, Imprecise Calculations, Market*

1 Introduction

The Hurd aims to provide a secure and flexible operating system foundation without losing the ability to support legacy applications[Bus94]. To achieve this, the Hurd employs a capability based multiserver architecture. Unlike a monolithic design, services traditionally provided by the kernel such as file systems and protocol drivers are broken out of the nucleus and placed in user-space servers to which clients issue remote procedure calls (RPCs). The messages carry capabilities each of which conveys the authority to invoke a set of methods on an object. On top of this framework, the Hurd provides a tightly integrated POSIX personality such that programs can transparently take advantage of many Hurd features or do so with minimal modification.

This paradigm shift away from a monolithic design requires reflection on the strategies conventionally employed in operating systems. In particular, resource management in the form of resource accouting and resource scheduling.

In monolithic systems, processes either directly use resources or ask the kernel for some services. In this framework, resource accounting is relatively straightforward as applications either act on their own or the kernel acts on its behalf. In a multiserver system, tasks often interact: clients contact servers which perform operations for them. From the kernel's perspective and often the operating system personality's, the servers are untrusted and cannot be relied upon to correctly report the amount of resources used by the client.

Monolithic kernels also take advantage of the centralized model to peer into various subsystems so as to gain insight into how applications use resources thus

allowing them to better predict processes' requirements and usage. Since resource use is divorced from resource management, e.g. file systems are simply user processes completely separate from the CPU, memory and I/O schedulers, the kernel simply cannot access this type of information.

As seen on GNU Mach, performance is extremely poor and anecdotal evidence suggests that this is part of the reason. Assuming good resource scheduling requires application usage patterns, we need to reunite resource usage and resource scheduling. That is, we either reintegrated components back into the kernel where scheduling occurs or we move resource scheduling to the tasks where the resources are actually used.

The approach that I propose in this paper is the latter: the operating system provides near physical resources to applications which must do any required virtualization. Hence, we strip the illusion that each task has its own virtual machine with infinite resources. This has a double advantage: as applications are in the best position to predict how they will use resources in the future (e.g. with respect to page eviction), they are now in a position to schedule them; and because applications know exactly how much of each sparse resource is available, they can better adapt. A similar approach has been successfully employed in V++[HC92], Exokernel[EGK95] and Nemesis[Han99].

This does not completely free the resource managers from having to schedule the resources they manage: they must still distribute the resources among competing principals. To this end, I propose a market based solution to resource allocation in which principals are given periodic allowances with which they can reserve resources. A market based solution is desirable as: there is a huge volume of literature interested in market dynamics, i.e. the field of economics; and moreover, markets have the desirable property of solving sparse resource allocation problems with little more information than supply and demand[Cle96].

2 Mach's Multiserver Framework

Operating systems often treat physical memory, i.e. core, as a cache of backing store. Protocol drivers translate parts of backing store to expose structured data sources. A file system, for instance, exposes files and may reside on one or more hard disks or be accessed over a network. Operating systems can use virtual memory to provide a task with a consistent view of a portion of backing store independent of whether it is actually cached in core. To support this mode of operation, the operating system provides a mechanism to logically bind parts of virtual memory address spaces to segments of backing store. On Unix-like operating systems, the kernel provides the `mmap` system call. When a client accesses a part of its virtual memory address space which is bound to backing store but not in core, the hardware suspends the thread and notifies the kernel so that it can bring the data into core, install the mapping and resume the thread.

Which parts of backing store are in core at any time are usually left to the discretion of the operating system. Typically, backing store is only brought into core once it has been referenced, i.e. not necessarily at bind time. When the available physical memory is busy caching other portions of backing store (that is to say, there is memory pressure), the operating system must choose memory to return to backing store, i.e. evict, so as to make memory available. Operating systems often employ a form of the least recently used (LRU) policy combined with some predictive algorithms. This is done transparently to the user applications[1]: other than the wall clock difference, they observe nothing.

The ability to provide integrated interpretations of backing store affects flexibility. On monolithic operating systems this is often done by patching the kernel either at compile time or at runtime through the loading of modules. The problem with this approach is that the code executes in a privileged domain and thus poses a security risk: an error in the code can potentially cause the whole system to crash; and malicious code can create havoc. The number of interpreters actually loaded on a secure system is thus relatively small and restricted to those which the administrator trusts. Moreover, using the interpreters, e.g. mounting a file system, is often itself a privileged operation. An important part of the multiserver approach is to provide a mechanism to allow untrusted applications to service virtual memory faults. Mach does this through its so-called external memory management API.

Briefly, on Mach, tasks can create memory objects, internally associate them with backing store and provide a capability to clients. Clients then map the memory objects into their address space using the `vm_map`

[1] Privileged applications are often allowed to wire memory. POSIX also provides the `madvise` function which permits applications to provide hints about backing store use to the operating system, however, there is no requirement that the operating system consider them.

system call. When the client faults, Mach finds the memory object backing the faulting address and queues a page-in request on the memory object. When the server dequeues the message, it fetches the data from backing store and returns it to Mach which then installs the mapping and resumes the faulting client[YTR+87].

This model, like the Unix model, provides applications the illusion of running alone on a machine with infinite resources: all mappings seem to be in core; they never have to yield the CPU; and performing I/O is without effort. This model is useful when resources are plentiful. As resources become scarce, they must be scheduled. Monolithic systems violate component boundaries to gain better insight into how resources are used.

There are two deficiencies with this model: when reading from or writing to backing store, servers allocate resources including memory, CPU and I/O bandwidth on behalf of their clients inhibiting resource accounting; and Mach provides tasks no mechanism to influence the page replacement policy.

2.1 Resource Accounting

That servers not part of the TCB must allocate resources on behalf of their clients suggests that resource accounting is incomplete. If so, this is a serious design flaw: systems which execute potentially malicious code must have mechanisms in place to constrain allocation so as to mitigate attempts at denial of service attacks.

The first problem is that Mach has no way to identify the actual resource principals: resource principals often cross protection domains. The clearest example is perhaps that a single user runs multiple processes: the processes run at the user's behest, thus all resources they allocate are allocated on the user's behalf and the user should pay for them. The obvious solution of designating users as the resource principals is problematic: users are a Hurd concept and completely foreign to Mach. Using the protection domain to enforce user resource limits is also suboptimal as then each task must be limited to the total of the real principal's resources divided by the maximum number of tasks the principal may create. This approach results in resource underutilization.

The problem is a bit subtler than suggested above: when a task issues an RPC it can be considered to run in the context of the server. By way of analogy, the server is a virtual machine and the RPC interface its set of instructions which authorized clients can issue. This is a variation of the confused deputy problem[Har88]: the server uses its own authority to allocate resources which it does not directly manage (i.e. unlike the file system to which it mediates access) for clients.

If servers are to allocate resources on clients' behalves, for instance, when a client reads from a file, the server must traverse the meta-data and read the data from backing store into memory, the server allocates memory, CPU time and I/O bandwidth, then a mechanism needs to be provided to charge the principal. Within this model, resource containers[BDM99] appear a tenable solution[2]. There also remains the the complication that within the Mach framework, clients do not knowingly trigger faults and thus cannot pass the required resources at that time. Instead, they must pass the resources at map time or a facility must be introduced to allow a client to add resources to a container.

2.2 Resource Scheduling

Resource scheduling, with the notable exception of hard real time systems to which the Hurd does not aspire, is primarily a question of performance. Of course, a system where it takes so long to perform an operation that the result is no longer useful once it has been obtained does not help either even if it is the most secure or correct system. The performance of the Hurd running on GNU Mach is, however, problematic. Currently, applications executing on the Hurd run approximately an order of magnitude slower than on a modern GNU/Linux distribution.

Attributing this solely to the page eviction scheme is unfair: there are a number of engineering problems with Mach. First, the last real work on GNU Mach was done in 1994 while it was still being maintained by the University of Utah. GNU Mach is at best optimized for machines with tens of megabytes of ram and processors running at approximately 100 MHz. Moreover, many of the algorithms don't scale well, in particular the use of a single handed clock to approximate LRU. There are also some areas where GNU Mach was never completed. In the page out path, for instance, GNU Mach sends a single page to a server at a time even if it could bundle several together.

More difficult to solve are design problems: monolithic systems are able to see that a user is, for instance,

[2]There are a number of trust issues here. Applying game theory, we can adopt that in the first instance we are willing to trust a server with small amounts of resources until it proves itself untrustworthy at which point we no longer make use of its services.

reading a file. Mach only sees raw I/O or, at best, access of a memory object. The Linux memory manager makes up-calls to the various components to shrink their caches when memory becomes sparse. It also detects access like streaming I/O which require knowledge of the file system[vR02]. Because these components exists in separate servers in a multiserver system, Mach is unable to do this.

This problem has been considered by [MA90, LCC]. Both propose solutions where the server backing a memory object may optionally provide a paging policy. This offers a potential performance improvement. More important, however, is providing this mechanism to applications who are actually using the memory.

3 Self-Managed Tasks

Under Mach's extensibility framework, because tasks are not aware of page faults, they cannot directly request the data and thus pay for it on their own account. A better interface would have faults forwarded to tasks where they can then determine what memory to evict if required and acquire the necessary resources to bring the data into core. This requires changing the way in which memory is bound to the virtual memory address space: it must now be the responsibility of the tasks to manage this translation. Microkernels also make bad resource scheduling decisions because they do not know what applications are doing: they do not have the insight that their monolithic counterparts have. L4 provides better primitives for this mode of operation[Gro04].

Having tasks manage their own resources is not a new idea: V++[HC92], Exokernel[EGK95] and Nemesis[Han99] all shift the burden from the nucleus to the operating system personality or processes.

Denning argues that we cannot trust application input with respect to resource scheduling because resources are multiplexed on their behalf [Den68]. Applications are not, however, giving advice: they are, in the first instance, managing their own resources.

Programs which are explicitly programmed to take advantage of these mechanisms will perform best. It is not always desirable or feasible to change programs at the source level. In this case, a library can be used to mediate the resource scheduling. The default library can provide an approximation of LRU for memory scheduling and highest priority first round robin scheduling. Because it is linked directly to the applications, it can intercept various standard library calls such as `malloc`, `mmap` and `madvise`. For applications willing to make simple modification, hooks can be provided, for instance, to drain memory caches. Finally, the behavior of the library can be influence by environment variables which can be set either by the application distributor, e.g. Debian, or the user.

There are several classes of applications which, if they could manage their own resource scheduling, would perform far better: database systems, scientific applications, garbage collectors and multimedia applications. Also, soft real-time applications can scale better knowing the resources available to them rather than trying to guess.

Applications and servers often cache results. In the simple case where the data is unmodified, it can be dropped and reread from backing store later. Slightly more complicated is the case where the data is the result of a render or other calculation. For instance, a document viewer may keep rendered pages in memory; and a file system may cache the `d_entry` structures. As long as there is sufficient memory, this is not a problem. When there is memory pressure, in both of these cases, it could be less expensive to simply drop the data and recreate it later, if required, then to consume I/O bandwidth sending the data to swap. This type of decision can only be made by the applications themselves.

The reason that monolithic kernels cross component boundaries is to gain insight into how applications will use a resource. Of course, they still only have limited knowledge as they can only detect known access patterns. The entity with the best knowledge of how an application will use a resource is, of course, the application itself. It may not have complete knowledge of its exact access patterns and resource requirements, however, Stonebraker notes that, for instance, INGRES generally knows at the beginning of its examination of a block which block it will access next, however, because it is not necessarily the one next in the file, the operating system will have no way to have known[Sto81]. Stonebraker calls for an interface to inject application specific knowledge into the resource scheduler so that database managers no longer have to implement their own virtual memory system. Other classes of applications can profit as well: scientific applications can often break calculations down fitting them to the amount of memory available[CE97], garbage collectors often do not follow the LRU acceess pattern [HFB04] and multimedia applictions, for instance, can decrease quality thereby reducing the number of dropped frames [SM98, DG01].

4 Resource Distribution

If tasks manage their own resources, this raises a new problem: physical resources need to be divided between the competing principals. One approach is to use a market.

This technique has already been explored by V++[HC92]. One policy that it chose was to separate resources. It is our intention to unify the resources under a single currency as it has been observed that resources can often be traded one for the other. For instance, when bandwidth is plentiful, it doesn't make sense to compress data. As bandwidth contention increases, if CPU is available, compression can be used to send more data per unit time. The incentive is that the principal will perform better: is pays less overall for the same service hence it is in its own benefit to be adaptive[Wel96].

We don't want to use a protection domain as the resource principal[BDM99]. Although this is sometimes the case, a process such as a web server might manage a number of network connections and thus need to allocate resources among them. This is difficult particularly when it comes to resources allocated indirectly, for instance, network bandwidth. Alternatively, and which is particularly true in a multiserver operating system, a single task can span several protection domains: a file server may allocate memory on behalf of a client. The client should be changed for the memory.

Denning states that memory management and CPU scheduling are tightly coupled: to make progress, a process needs its working set in memory; having memory alone is insufficient as the purpose of obtaining the memory was to continue execution[Den68].

Resources can often be traded one for another. Stratford suggests that a carefully chosen video format would allow an application to adapt either to disc bandwidth, CPU and memory when decoding the video[SM98]. Domjan uses the example that applications can either render or transcode depending on the availability of resources and can do so fairly quickly[DG01].

To support these modes of operation, principals will be given a periodic allowance proportionate to their static priority in the system. It is important that they are not able to accumulate wealth. This provides an incentive to applications to make the best of the available resources and avoids priority inversion. The latter problem arises when a low priority principal waits to accumulate sufficient wealth such that is able to acquire most of the resources of the system thereby resulting in a denial of service attack. This is often the case in multiuser systems where many users log in occasionally. To prevent this, credits which are not used decay. Thus minimal savings are possible. This also has the desirable property that when principals starts they have a slight, brief inflated priority and thus able to bootstrap relatively quickly. Many programs use a bit of resources when starting or when responding to user input and then spend the rest of their time idle.

We design the market such that resources are distributed fairly based on externally assigned priorities (based, for example, on how much a customer has paid or the class of tasks being performed). As is often the case in real markets, agents' demand vary based on external conditions. To maximize resource utilization, the market should distribute the available resources among the active agents based on their relative priorities. However, long term resource commitments must cost more as at any time agents become active and the system must dynamically adjust. A data backup program, for instance, may be given a small allowance. While the system is under pressure, it may only be able to acquire a few kilobytes per second of bandwidth. However, when the system goes idle, it should be able to utilize all of the available bandwidth.

A second important criteria is that hording wealth must not be possible. A low priority agent may remain idle while high priority agents run continuously. A malicious agent could wait until there is significant demand and then strain the economy by purchasing a significant amount of resources thereby denying a high priority agent its due and resulting in a denial of service. By penalizing agents for not spending their allowance, the market need only consider the wealth of active agents when determining price. As agents become active, the system can gracefully adapt: when there is significant resource pressure, there are many competing agents and a single agent cannot significantly change the overall wealth; when there is no resource pressure, it doesn't matter.

Reservation vectors contain a series of possible allocation schemes with relative utilities. The scheduler will grant the reservation with the highest utility which the principal can afford. Reservations include a mandatory part which the principal requires and zero or more optional parts each of which would increase service[HFL96].

Since multiple tasks can use the same principle identifier to make reservations, we need to provide a

medium to determine what utility to grant.

5 Conclusion

Simply importing resource scheduling strategies from monolithic systems into multiserver systems has been shown to not work. The major problem in this case is that monolithic kernels can use component introspection. Rather than attempt to introduce mechanisms to make this available to the microkernel, I have suggested that processes schedule their own resources. This is desirable as the processes themselves know best which resources they need when and can thus better schedule and adapt than any external entity.

With multiple entities competing for the available sprarse resources, the operating system must still provide a distribution mechanism. To this, I have proposed a market based approach. The reason that we have the applications themselves manage their resources is because Mach did not have enough information to manage them intelligently. In the proposed approach, the operating system has even less knowledge. Markets require very little information to work well. In our case, they know the relative priorities of the principals as well as the supply and demand. Moreover, there is a large body of literature which covers how markets distribute resources fairly. There has been some attempt at applying these in the domain of computer science, however, other than the work on V++, I am not aware of much work done at the operating system level.

To date, I have, minus the frame reclamation code, nearly completed an initial implementation of the physical memory manager. The server proper consists of approximately 5000 lines of heavily comment C code. Most of the complexity is in dealing with super pages. In particular, deallocating a part of a shared super page greatly complicates reference counting. A simple virtual memory management library is also present but does not yet do any paging. Currently, I am working on the I/O interfaces and an initial file system. When that code is completely, we should have a sufficient framework that I can fully invest myself in implementing and analyzing the resource scheduling mechanisms and policies outlined in this paper.

6 Acknowledgements

I would particularly like to thank Marcus Brinkmann for many fruitful discussions and his work on the other interesting part of the port of the Hurd to L4—the capability system. Isabel Hünig asked me to explain most every detail and helped clarify my thought process. Thanks to Espen Skoglund, I visited the Universität Karlsruhe during the summer of 2002 and started the port of the Hurd to L4. He, as well as Kevin Elphinstone and Volkmar Uhlig, provided useful input and guidance at this stage.

References

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[Bus94] Michael Bushnell. Towards a new strategy of OS design. *GNU's Bulletin*, 1(16), January 1994.

[CE97] Michael Cox and David Ellsworth. Application-controlled demand paging for out-of-visualization. Technical Report NAS-97-010, NASA Ames Research Center, July 1997.

[Cle96] Scott H. Clearwater. Why market-based control? In Scott H. Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific Publishing, 1996.

[Den68] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[DG01] Hans Domjan and Thomas R. Gross. Managing resource reservations and admission control for adaptive applications. In *30th International Conference on Parallel Processing*, September 2001.

[EGK95] Dawson R. Engler, Sandeep K. Gupta, and M. Frans Kaashoek. AVM: Application-level virtual memory. In *Fifth Workshop on*

Hot Topics in Operating Systems (HotOS-V), pages 72–77, May 1995.

[Gro04] System Architecture Group. L4 experimental kernel reference manual version x.2 revision 5. Technical report, Dept. of Computer Science Universität Karlsruhe, June 2004.

[Han99] Steven M. Hand. Self-paging in the nemesis operating system. In *Third Symposium on Operating Systems Design and Implementation*, pages 73–86, 1999.

[Har88] Norm Hardy. The confused deputy (or why capabilities might have been invented). Technical report, Key Logic, 1988.

[HC92] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. October 1992.

[HFB04] M. Hertz, Y. Feng, and E. Berger. Page-level cooperative garbage collection, 2004.

[HFL96] David Hull, Wu-chun Feng, and Jane W.S. Liu. Operating system support for imprecise calculations. In *AAAI Fall Symposium on Flexible Computation*, November 1996.

[LCC] Paul C. H. Lee, Meng Chang Chen, and Ruei-Chuan Chang. In-kernel policy interpretation for application specific memory caching management.

[MA90] Dylan McNamee and Katherine Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. In *First USENIX Mach Workshop*, October 1990.

[SM98] Neil Stratford and Richard Mortier. An economic approach to adaptive resource management. November 1998.

[Sto81] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[vR02] Rik van Riel. Page replacement in the Linux 2.4 memory management. 2002.

[Wel96] Michael P. Wellman. Market-oriented programming: Some early lessons. In Scott Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific, River Edge, New Jersey, 1996.

[YTR+87] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *11th Operating Systems Principles*, pages 63–76, November 1987.