Infokernel: An Evolutionary Approach To Operating System Design

Andrea Arpaci-Dusseau
Remzi Arpaci-Dusseau

Technical Report #1465

January 2003

Infokernel: An evolutionary approach to operating system design

Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau
Department of Computer Sciences
University of Wisconsin–Madison

Abstract

Although information about the internals of the operating system is useful to applications, such information is rarely exposed. In this paper, we argue that all information about the internal state, algorithms, *and* costs *of the OS should be exported. We term an OS that exports this information an* infokernel. *We show that access to information is useful so that applications can both* adapt *to and* control *the behavior of the OS. Given that it is relatively easy to modify existing systems to expose information, we believe this is a practical way to obtain many of the benefits of more radical extensible systems.*

1 Introduction

"As a general rule, the most successful man in life is the man who has the best information." *Benjamin Disraeli (1804-1881)*

It is well-known that access to accurate, timely information is the key to making intelligent decisions. Sophisticated applications and middleware components that want to tune themselves to the underlying OS for the best performance require information about the current state of the OS, the algorithms the OS employs, and the cost of various operations. Despite this value, operating systems have traditionally hidden much of their information, adhering to the principles of encapsulation and modularity; after all, even Lampson encourages developers to "keep secrets of the implementation" [11].

We argue that the functionality of the entire system can be significantly expanded by having the OS expose more information; specifically, applications are able to *adapt* their behavior to the OS as well as *control* the OS in new ways. We name an OS that has been modified to expose all possible internal information an *infokernel*. With an infokernel, functionality that previously required changes to the OS can now often be placed outside of the OS. This functionality can be located either directly in an application or in a library acting on behalf of applications.

The infokernel approach is in direct contrast to other research in extensible systems that has advocated a complete restructuring of the OS [3, 7, 9, 16]. Although these systems will allow more flexibility than an infokernel, this extra functionality comes at a high cost: discarding the hundreds of millions of dollars and thousands of developer years spent on commodity operating systems [22]. We believe that within modern OS research, an evolutionary approach is essential. Introducing a radically new operating system that will be accepted by users and ported to the wide range of hardware systems is not a realistic goal. Therefore, we advocate a design in which any existing OS can be transformed into an infokernel.

In this paper, we begin by addressing the primary issues, advantages, and challenges to building an infokernel. We then discuss two case studies where applications can control an OS that has been converted to an infokernel; specifically, applications can control the layout of their files on disk and can modify the page replacement algorithm. Finally, we compare infokernel to three related philosophies: exokernel [7, 9], open implementation [10], and gray-box systems [2].

2 Infokernel Issues

"It is a very sad thing that nowadays there is so little useless information." *Oscar Wilde (1854 - 1900)*

In this section, we discuss four questions related to exposing information from an operating system. What types of information exist in the OS? Why is this information useful to applications? What are the goals of an infokernel? How can an infokernel be implemented?

2.1 What information exists in the OS?

From the perspective of an application, the operating system has a number of pieces of useful information. We believe that this information can be grouped into three categories: internal state, algorithms, and costs. Within each category, the information may be either *static* or *dynamic*. We define each of these as follows.

Internal state: any variables that are internal to a traditional OS. Static variables are configurable, but rarely change (*e.g.*, only when new hardware is installed, the machine is rebooted, or when an administrator interacts with the system); examples include the length of a time slice for a given priority, the interval at which dirty file blocks are flushed to disk, or the maximum number of file descriptors. Dynamic variables can change at any time; examples include the current priority of each runnable process, the value of each page reference bit, and the number of messages waiting to be sent over the network.

Algorithm: the code that is executed by the OS. The algorithms of the OS will be of interest at different levels of detail to different applications. For example, two different applications may be interested in the allocation policy of the file system; however, one application may simply want to know whether or not extent-based allocation is performed, whereas the other application must know precisely how each block of a new file will be allocated on disk. One can view the algorithm statically or dynamically as well; the static description is the general behavior of the code, whereas the dynamic description captures the instructions or events that are currently executing.

Cost: the overhead (*e.g.*, time, space, or power) of performing a given operation within the OS at a given time. A static cost is relatively constant within the given system (*e.g.*, what is the cost of switching between threads versus processes?), whereas a dynamic cost is circumstantial, depending upon the current internal state of the OS (*e.g.*, what is the cost of reading a particular byte of a file, given the state of the buffer cache and the position of the disk head?).

2.2 Why is OS information useful?

There are two primary reasons why applications find information about the OS useful: applications can *adapt* their behavior to that of the OS and applications can *control* the behavior of the OS. We discuss these two uses in more detail.

2.2.1 Adaptation

To improve their performance, sophisticated applications can adapt their behavior to each category of information provided by the OS. The basic idea is that with more information, an application knows which operations are the most efficient to perform at any given time. Depending upon the type of information used, the *granularity* of adaptation can differ significantly. At the coarsest level of adaptation, a different version of the application can simply be instantiated on different systems; this usually occurs when reacting to static information. At the finest level of adaptation, a different decision is constantly made; this is usually a reaction to dynamic information.

To illustrate how applications adapt to OS information, we consider examples ranging from coarse to fine adaptation. First, a greedy, CPU-bound process that knows the scheduler gives priority to interactive processes (*i.e.*, static algorithmic knowledge), can periodically print a character to boost its priority. At a slightly finer level of adaptation, a memory-intensive application that knows the amount of physical memory currently available (*i.e.*, dynamic state), can process its data in multiple passes [23], appropriately limiting its working set to avoid thrashing. Alternatively, a process that knows the amount of time remaining in its time slice (*i.e.*, dynamic state), may decide not to acquire a contentious lock if it expects to be preempted before finishing its critical section. Finally, a web server can improve its average response time by reordering how it handles its requests; specifically, it can service those requests which are expected to complete the fastest (*i.e.*, dynamic cost) [6].

2.2.2 Control

More surprisingly, applications can also use information to *control* the future state of the OS or to change the policies seen by the end user. The basic idea is that, given knowledge of how the OS behaves, the application can probe the OS, or change its normal inputs, so that the OS reacts in a certain, controlled manner.

We illustrate by using examples of where applications loosely control OS prefetching and the TCP congestion control. First, consider an application that knows the file system performs prefetching after observing a sequential access pattern; if the application knows that it will not access these blocks, then it can squelch the prefetching by issuing an intervening read to a random block. Second, with knowledge of the congestion control algorithm and access to internal state such as the window size and observed round-trip time, an application may implement a less aggressive sending algorithm, such as TCP Nice [20]; if the application calculates that its desired window size is smaller than that set by the default algorithm, it can reduce the amount of data it sends by a corresponding amount.

2.2.3 Discussion

In many cases, there is an equivalence across the three categories of information; that is, internal state, algorithmic information, or cost may each be sufficient on their own. Consider our previous example of a web server that improves its average response time by servicing first those requests that are expected to complete the fastest [6]. Although the most direct approach is to use cost information from the OS, the server can approximate this itself from either internal state or algorithmic knowledge. For example, if the OS exposes the contents of the buffer cache [19], the location of the disk head, and the location of each file on disk (*i.e.*, dynamic internal state), then the web server can compute the relative costs for each page on its own (or, pragmatically, at least order the requests based upon hitting in the buffer cache). Alternatively, and more radically, if the OS exposes its algorithms, the web cache can simulate the OS policy given the observed stream of past requests to infer the current state of the OS [4]. Although the overhead, accuracy, and complexity within the application will be different, the end result may be roughly the same.

2.3 What are the goals of an infokernel?

To allow adaptation and control in a pragmatic way, an infokernel should have the following properties.

Export all information. Because one cannot know *a priori* what information is useful to applications, all information from the OS is exported. The only information that is concealed is for security or privacy (*e.g.*, the data read and written by applications).

Incur low burden on developer. It is easy to modify an existing OS to expose information; if this task is onerous, then few infokernels will be created, particularly for pieces of information whose benefits are not yet demonstrated.

Allow partial information. If the onus of exporting all information is too high, or if a developer (or more likely, a company) feels that trade secrets will be compromised, then the OS can expose a subset of information.

Be flexible. New formats of information are expressible, since one cannot predict all types of information that will exist in future systems.

Be portable. Applications can be ported easily across platforms, even to systems that export different types of information.

Incur low overhead. The overhead of updating and accessing information is low.

2.4 How can one implement an infokernel?

We briefly discuss two open questions for building an infokernel. First, how can one provide an interface that meets the conflicting requirements of flexibility, portability, and efficiency? Second, how can the information be expressed?

Some of the infokernel goals contradict one another; in general, exposing more information breaks modularity, making applications more difficult to port across infokernels. Consider an application that wants to know the next page to be evicted by the OS: if the application is developed on an infokernel that uses Clock replacement, the application examines the clock hand position and the reference bits; if this application is run on an infokernel with pure LRU replacement, the application must instead examine the position of the page in the LRU list. This may require a significant change to the application.

Thus, a two-tiered approach is needed for the infokernel interface. At the first level, the OS exports information in the form that is most convenient (*e.g.*, marking pages read-only and mapping them into the address space of each process or by leveraging existing interfaces such as `/proc`). This information is platform specific. At the second level, an *infolibrary* provides a portability layer; this layer converts the OS-specific formats into a more standard interface. Applications are free to skip the infolibrary to access system-specific information.

A further challenge is to determine appropriate representations for the three categories of information. Representing internal state is relatively straight-forward: variables are a natural match. Functionality allowing applications to block until a variable contains a particular value may be useful as well [9]. Representing algorithms and costs is not as simple. Many representations for algorithms at various levels of detail are possible, ranging from exporting the source-level or assembly *code*, to leveraging a *specification language* [8, 15], to simply exporting a well-known *name* that captures the spirit of the algorithm (*e.g.*, LRU, FIFO, LFU, MRU). Finally, exposing costs has the additional challenges that this information is not already directly available in the OS and it depends upon the hardware. Given that applications often use cost to choose between operations, an infokernel may be able to export only relative, and not absolute, costs.

3 Examples

"Everybody gets so much information all day long that they lose their common sense."
Gertrude Stein (1874 - 1946)

In the previous section, we gave a number of short examples where an infokernel would be useful to applications. In this section, we briefly elaborate on two additional examples that have been implemented in previous systems to demonstrate the benefits of extensibility. We focus on control, rather than adaptation, since we believe the ability of an infokernel to supply control is more challenging.

3.1 File Layout

Although I/O-intensive applications, such as database systems and web servers, benefit from controlling the layout of their data on disk [18], traditional file systems do not provide this control. Thus, controlling file layout has been used to demonstrate the power of extensible systems [9]. We believe that this functionality can also be built on an infokernel that uses an FFS-like file system; we refer to this application-level service as PLACE [13]. For simplicity in our discussion, we consider a version of the service that allows files to be placed within a particular *cylinder group* (abbreviated with simply "group") on the disk.

PLACE needs algorithmic and dynamic state information from the infokernel: the file allocation algorithm used by the OS, the group in which each file or directory is allocated, and the current "fullness" of each group. We assume that the algorithmic information is at a sufficient level of detail to reveal the following properties of FFS-like allocation [12]: a new directory is placed in the "least full" group, files are placed in the same group as their parent directory, and data blocks are usually placed in the same group as their corresponding inode.

PLACE initializes itself by creating directories such that at least one directory, D_i exists in each group, i. To ensure a directory is allocated in the target group, PLACE fills the non-target groups with dummy data such that the target directory is the least "full" and the FFS allocation algorithm will choose it for the next directory. When an application wishes to create a file in a particular group, it contacts PLACE. To allocate a file named /a/b/c in group i, PLACE creates the file under directory D_i, and thus, with its algorithmic knowledge, knows that the file will be allocated in the appropriate group. Through the infokernel interface, PLACE can verify that the inode and all the data blocks of the file are in the desired group and take corrective action if needed. PLACE then renames the file to the user-specified name.

3.2 Page Replacement

It is well known that different applications benefit from different page replacement algorithms [5], and modifying the replacement policy of the OS has been used to demonstrate the flexibility of extensible systems [16]. This functionality can also be approximated in an infokernel environment; we call this service REPLACE.

To support REPLACE, an infokernel must export the page replacement algorithm and enough dynamic state such that REPLACE can determine the next victim page. As a simple example of how page replacement can be controlled with an infokernel, consider the case where the page P will be the next page evicted from the OS cache and an LRU-replacement policy is being used. If the application knows that it will access P soon in the future, it can probe P to adjust its position in the LRU list and ensure that it stays in the cache. More generally, one replacement policy can be converted to another by probing pages at some frequency and recency before they are evicted. Our prototype of REPLACE, implemented for the buffer cache of NetBSD 1.5, is able to convert an LRU replacement policy into MRU, LFU, LRU-K [14], and EELRU [17]; converting to LRU-K, we have measured I/O-based applications with improvements up to 33%.

4 Related Philosophies

"The idea is to try to give all the information to help others to judge the value of your contribution; not just the information that leads to judgment in one particular direction or another."
Richard Feynman (1918-1988).

One could view an infokernel as a very limited subset of an exokernel [7], which has the goal of separating protection from management. When developing Aegis, the authors found three important principles: expose allocation, expose names, and expose revocation. Exposing allocation and revocation provide control to applications, whereas exposing names (*e.g.*, page numbers and bookkeeping data structures, such as freelists and cached TLB entries) provides information. This point is made more explicitly in the second exokernel paper [9], in which "expose information" is added as a fourth principle. In contrast, an infokernel exposes information directly, but not control.

There are two additional differences between the exokernel and infokernel philosophies. First, an infokernel is designed as an evolution of an existing operating system, as opposed to a completely new exokernel. We believe that an infokernel is more feasible to implement and deploy, while providing many of the same benefits. Second, an infokernel strives to expose all types of information (*i.e.*, internal state, algorithms, and costs), whereas an exokernel focuses on internal state; however, given that the goal of an exokernel is to remove all resource management, one could reasonably argue that an exokernel does not have interesting algorithms to expose.

The philosophy of the Open Implementation project [10] is similar to ours, although they do not specifically target operating systems. The goal of an open implementation is to tailor the behavior of a module, while still hiding "unnecessary" details of its implementation; in other words, understanding how a module is implemented helps clients use it more appropriately. The authors propose several ways for changing the interface between clients and modules; most of the approaches have the clients specify their anticipated usage or requirements of the module, and thus do not involve exposing information from the implementation. However, in one of their suggestions, clients choose a particular implementation from an available list (*e.g.*, BTree, LinkedList, or HashTable). This approach is directly related to our proposal of exposing the algorithms employed by the OS.

Finally, there is a relationship between infokernel and the authors' own work on gray-box systems [2]. The philosophy of gray-box systems also acknowledges that information in the OS is useful to applications; however, a gray-box system takes the more extreme position that the OS cannot be modified and thus applications must either assume or infer all information. The limitation of the gray-box approach is that its assumptions may be incorrect and its inferences may impose significant overhead.

However, an infokernel is not a panacea; in some cases, the OS may not be able to obtain and export all information. For example, the OS may interact with information that is directly controlled by hardware or another software component (*e.g.*, the OS is unlikely to know the exact location of the disk head, although it can infer this information by combining knowledge of the last disk request serviced with rotational speeds [21]); thus, unless every component in the system exposes all of its internal information, gray-box techniques will still be needed. Furthermore, in networked and distributed settings, it may not be possible to explicitly expose information. For example, to implement gang-scheduling across autonomous systems, the local OS must know which processes are scheduled elsewhere; however, by the time this information has been communicated, it may have changed. Thus, inferencing is the only suitable option [1].

Finally, gray-box techniques are likely to be useful when implementing the portability library for a new infokernel. A major challenge of the infolibrary is to handle cases where the infokernel does not export all information defined by the standard interface. Gray-box techniques may be able to infer this information, rather than have the infolibrary report that the information is not available. In summary, we believe that infokernel and gray-box techniques are complementary.

5 Conclusions

"Errors using inadequate data are much less than those using no data at all." *Charles Babbage (1791-1871)*

Through the ages, many people more eloquent than we have extolled the virtues and the vices of information. In this paper, we have joined this dialog by arguing that operating systems should expose all available information. We believe that such an OS, or infokernel, is pragmatic because it is evolutionary; this approach will gain some of the benefits achieved by more radical extensible systems, while still retaining the large code base of modern, commodity operating systems.

References

[1] A. C. Arpaci-Dusseau. Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. *ACM Transactions on Computer Systems (TOCS)*, 19(3):283–331, August 2001.

[2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *The 18th Symposium on Operating Systems Principles (SOSP)*, October 2001.

[3] B. N. Bershad, S. Savage, E. G. S. Przemyslaw Pardyak, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[4] N. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Management. In *USENIX Annual Technical Conference*, June 2002.

[5] P. Cao, E. W. Felten, and K. Li. Application-Controlled File Caching Policies. In *Proceedings of the 1994 USENIX Summer Conference*, January 1994.

[6] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection Scheduling in Web Servers. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[7] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[8] J. V. Guttag, J. J. Horning, S. Garland, K. Jones, A. Modet, , and J. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[9] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Malo, France, October 1997.

[10] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. C. Murphy. Open implementation design guidelines. In *International Conference on Software Engineering*, pages 481–490, 1997.

[11] B. W. Lampson. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 33–48, Bretton Woods, NH, December 1983. ACM.

[12] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[13] J. Nugent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A Gray-Box Approach To Controlling Data Layout: Techniques and Implementation. Technical Report Computer Sciences 1456, University of Wisconsin, Madison, 2002. Submitted for Publication.

[14] E. J. O'Neil, P. E. O'Neil, and G. W. ikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 297–306, 1993.

[15] S. Sankar and R. Hayes. ADL: An Interface Language for Specifying and Testing Software. In *Proceedings of the Workshop on Interface Definition Languages*, January 1994.

[16] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extens ions. In *OSDI II*, 1996.

[17] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, Atlanta, GA, May 1999.

[18] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[19] R. Van Meter and M. Gao. Latency Management in Storage Systems. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, October 2000.

[20] A. Venkataramani, R. Kokku, and M. Dahlin. Tcp-nice: A mechanism for background transfers. In *Operating Systems Design and Implementation*, December 2002.

[21] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading capacity for performance in a disk array. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation*, pages 243–258, San Diego, 2000. USENIX Association.

[22] G. P. Zachary. *Show-Stopper!: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft.* Free Press, October 1994.

[23] H. Zeller and J. Gray. An Adaptive Hash Join Algorithm for Multiuser Envronment s. In *The 16th International Conference on Very Large Data Bases (VLDB)*, pages 186–197, 1990.