The Case for Extensible Operating Systems

The Case for Extensible Operating Systems

Margo Seltzer, Christopher Small, and Keith Smith
Harvard University

Abstract

Many of the performance improvements cited in recent operating systems research describe specific enhancements to normal operating system functionality that improve performance in a set of designated test cases. Global changes of this sort can improve performance for one application, at the cost of decreasing performance for others. We argue that this flurry of global kernel tweaking is an indication that our current operating system model is inappropriate. Existing interfaces do not provide the flexibility to tune the kernel on a per-application basis, to suit the variety of applications that we now see.

We have failed in the past to be omniscient about future operating system requirements; there is no reason to believe that we will fare any better designing a new, fixed kernel interface today. Instead, the only general-purpose solution is to build an operating system interface that is easily extended. We present a kernel framework designed to support the application-specific customization that is beginning to dominate the operating system literature. We show how this model enables easy implementation of many of the earlier research results. We then analyze two specific kernel policies: page read-ahead and locking-granting. We show that application-control over read-ahead policy produces performance improvements of up to 16%. We then show how application-control over the lock-granting policy can choose between fairness and response time. Reader priority algorithms produce lower read response time at the cost of writer starvation. FIFO algorithms avoid the starvation problem, but increase read response time.

1 Introduction

A number of recent operating systems research efforts include the design, implementation, and analysis of kernel modifications that improve performance in either general-purpose or application-specific cases. We find specific examples in buffer management, processor scheduling, interprocess communication and networking. In fact, there are few areas of operating systems where we have not seen the need for application-specific modifications.

For example, most operating systems typically implement a file caching mechanism, using least-recently-used (LRU) replacement. The database community has repeatedly demonstrated that other algorithms, such as DBMIN [10], domain-separated pool management [24], Group LRU [22], and HOTSET [25] provide better performance for database applications. Since most systems do not provide applications an interface through which they can alter the page replacement policy, database management systems (DBMS) typically implement their own shared-memory buffer manager in user-space. These user-level buffer managers compete, rather than cooperate, with the operating system for resources. The micro-kernel architecture [2] does not address this particular issue, although it does allow application-specified pagers. Unfortunately, the pager interface does not provide applications the ability to specify page replacement, it allows applications to specify only the backing store for memory pages. A new interface proposal provides this additional capability [19], but the solution is much larger than the problem. It is unclear that every application that cares about page replacement also cares about backing store. The granularity of service that can be specified in current operating systems is too large. Applications may need to replace just one routine in a pager, not the entire pager. Recently, it was shown that application-specific page replacement policies improve the performance of an assortment of applications ranging from conventional linkers to cache simulators to various database applications (e.g. Postgres, cscope, glimpse) [8][9].

A second approach to fixing the paging problem is to redesign the virtual memory system to give applications control over their own memory management [5] [12]. This approach is most useful when user programs use VM page protection to implement logical, application permissions. Appel surveys different uses of VM (e.g. garbage collection, persistent stores, concurrent checkpointing) and proposes new interfaces that would allow

applications to use the system to satisfy these needs more efficiently [5]. Once again, the solution is larger than the problem; in order to provide the required functionality efficiently, entirely new interfaces are required.

The problem of inflexibility is not limited to memory management. Simple schedulers, designed for a time-sharing environment, are being called upon to service real-time, multi-threaded, and distributed applications. The proposed solutions for providing a wider range of scheduling algorithms includes static selection of scheduling algorithms [1], redesigning and relinking the operating system kernel, providing user-level and/or kernel threads [3], and adding complex new primitives [4]. The functionality required is merely the ability for applications to specify their own scheduling disciplines.

Distributed computing paces ever-increasing demands on network technology. Providing high-performance distributed computing requires yet another path through the operating system kernel [11] or kernel implants [17][31]. The high-speed path needed by an input/output stream is not a special case; it is merely an instance of needing to direct the kernel's routing of data. Providing applications the ability to route data efficiently should be the norm, not the exception. The packet filter solution is a solid approach to providing applications the ability to customize their kernel services. However, the customization that is provided for packet filtering is needed in all aspects of the operating system, not just the networking subsystem.

The fundamental problem that is being solved repeatedly is that the operating system interface is not flexible enough to provide applications the performance or functionality they need. In each case described above, the suggested techniques address a set of specific symptoms, but do not solve the underlying problem. The reflective computing community terms these incompatibilities mapping dilemmas [30]. They claim that mapping dilemmas occur when crucial implementation strategies are hidden behind an abstract interface. The reflective community proposes a solution based upon a formal model of base- and meta-protocols. The base protocol is the conventional interface while the meta-protocol is the interface for changing underlying policy decisions [15]. *This* is the correct model for designing operating systems interfaces. As long as policy details are hidden behind high-level interfaces, we will be forced to develop special-purpose implementations There does not exist a single policy that will work for all applications. Instead of searching for the elusive "best" policy, we must concentrate on the development of a framework in which it is possible for each application to use the best policy or algorithm for the task at hand, and leave the arbitration among competing policies to the operating system kernel.

The rest of this paper is organized as follows. Section 2 presents a framework for implementing operating system extensibility. Section 3 discusses several recent research efforts and demonstrates how they map into the framework presented in Section 2. Section 4 discusses the framework in which we evaluate the cost of kernel extensibility and Section 5 presents our experimental results. Section 6 discusses how our framework relates to other ongoing projects in extensible operating systems.

2 A Framework for Extensibility

The key issue in designing a truly extensible kernel is selecting an architecture that supports fine-grain extensibility. Our goal is to allow replacement of the smallest modules that implement policy decisions.

We decompose an operating system kernel into two types of modules: *arbitrators* and *mappers*. Arbitrators manage shared resources and are further decomposed into *allocators* and *schedulers*, responsible for preemptible and non-preemptible resources respectively. Mappers implement mappings between resources; the virtual memory system provides mappings from virtual addresses to physical addresses; name spaces implement mappings between logical names and other logical names or physical resources. Each class of objects requires two sets of interfaces, the user-interface and the tuning-interface. The user-interface is the conventional set of functions typically associated with the object while the tuning-interface is the interface that permits small, incremental changes. These are analogous to the base- and meta-protocols of the reflective community. The user-interface is unimportant for the purposes of this discussion, so the remainder of this section discusses the tuning-interface.

The tuning-interface consists of the set of functions that implement policy decisions for a module. Table 1 shows the tuning-interface for arbitrators. Small incremental changes are implemented by replacing a few of the policies that implement an object; new types of objects are created by implementing a new set of policies. We call this

Interface	Description
granularity	Selects the granularity of allocation or scheduling. This is the blocksize for a disk allocator or a time quantum in a CPU scheduler.
preallocate	Implements projected retrieval policies. This is the read-ahead policy for a file system or a reservation policy for a scheduler.
deallocate	Implements deallocation or completion policies. This is the eviction policy for a buffer manager or preempt strategy in a process scheduler.
priority	Specifies an ordering or value of entities being allocated or scheduled. This is the scheduling priority for processes or an eviction order in a buffer cache.
synchronize	Used with arbitrators that implement caches to synchronize cached values with the values in the object backing the cache. This is the write back policy for a buffer cache.
relative_allocate	Provides allocation for an entity when there are logical relationships to be maintained. This is the block allocation for an existing file in a file system or the processor scheduling algorithm for a multiprocessor scheduler.
new_allocate	Provides allocation of an entity when there is no logical relationship to maintain. This is the block allocation for a new directory in the FFS or the scheduling algorithm for a new, independent process.

Table 1. Arbitrator Tuning-Interface Definitions. Every policy decision made by an arbitration object is exported via a tuning-interface call. Minor modifications to an object are implemented by replacing one or more of the policy decisions. Creation of a new object is implemented by defining a new set of interface functions.

---

the *fine-grain* model of extensibility in that it permits small changes to be made more easily than in other extensible systems.

This fine-grain approach can be contrasted with other approaches to extensibility. The Mach external pager interface provides the ability to replace an entire pager. However, this is too coarse a level of replacement if the application needs to make only minor adjustment of the default kernel policy. For example, although the external pager interface allows applications to specify the backing store for regions of memory, it does not allow those applications to control the replacement policies for those regions. In the fine-grain model, the replacement policy can be modified by specifying an application-specific **`deallocate`** function. In the external pager model, McNamee and Armstrong had to substantially modify the existing external pager interface [19]. Instead of dictating which page to evict, the new interface allows the kernel to ask the external pager to evict a page of its choice. The kernel retains control over which memory object will lose a page frame, but the pager controls which page will be evicted. Unfortunately, this interface requires the implementation of a new external pager for each different policy; the granularity is still too coarse.

The virtual file system architecture of most Unix[1] file systems more closely approximates the fine-grain model. In most Unix systems, the file system abstraction is implemented by the virtual file system (VFS) and virtual node (VNODE) layers [26]. In order to create a new file system with slightly different semantics or behavior than an existing one, the implementor creates a new file system type and associates with it a set of procedures that

1. Unix is a trademark of X/Open.

Buffer Pool Operations
open_buffer_pool(parameters);
close_buffer_pool(bufpool);
sync_buffer_pool(bufpool);
Buffer Operations
get_buffer (bufpool, unique_id);
put_buffer (bufpool, unique_id, buffer, flags);
Policy Operations
enqueue_buffer(bufpool, unique_id, buffer, queue, flags);
buffer = evict_buffer(bufpool, queue)
writeback(bufpool, frequency, queue);

Table 2. Extended interface for a kernel buffer manager. The bufpool parameter identifies the specific pool to which a resource belongs, the unique_id uniquely identifies the resource being buffered, and buffer is a pointer to the buffer itself. The flags parameter is used to indicate buffer status (e.g. dirty, normal_request, prefetch_request) and the queue parameter indicates the appropriate links to traverse in order to enqueue/dequeue a new buffer. The frequency parameter indicates how often the writeback routine should be called to evaluate the buffer queue and dispatch write requests. By controlling the queues on which buffers are placed, the kernel can arbitrate between conflicting policies. The first two classes of calls are user-interfaces while the last set are the tuning-interfaces. The **`enqueue_buffer`**, **`evict_buffer`**, and **`writeback`** routines correspond to an arbitrator's **`priority`**, **`deallocate`**, and **`synchronize`** policies.

---

implements each function in the VFS and VNODE interfaces. Naturally, if the new file system is only slightly different than the old file system, a large number of these functions are identical to those of the original file system, and the code can be reused. However, once the new file system is installed (either by linking into the kernel or by dynamically loading a kernel extension), it becomes available to all running applications; it is not tuned to any specific application. Furthermore, minor modifications to the file system, such as changing the on-disk representation of a directory or inode require rewriting the name translation code. In this case, the framework is correct, but the replaceable component is still too large.

In the fine-grain model, the VFS and VNODE interfaces are more detailed. The file system is actually a combination of six entities: three mappers an allocator and two schedulers (the allocator and schedulers are more detailed invocations of arbitrators). The mappers map logical, user-visible names to internal file identifiers (FIDs), FIDs and file offsets to logical disk addresses, and logical disk addresses to physical disk addresses. The allocator manages disk space. The block allocation policy of a file system such as the Berkeley Fast File System is implemented by specifying the **`new_allocate`** and **`relative_allocate`** functions. The two schedulers are used to arbitrate access to the disk and to file locks. Each of the arbitrators (the disk allocator, the disk scheduler and the lock scheduler) support all the policy decision interfaces described in Table 1.

3 Recent Research Results Map into the Fine Grain Model

Using the framework described above, we can map several recent research results into small incremental changes to default kernel policy. Table 2 shows a canonical buffer management interface. In addition to the per-buffer and per-buffer-pool interfaces typically associated with a buffer pool, we include the relevant policy interfaces that provide applications control over their own buffer management. The buffer manager is a specific case of an arbitrator.

Per-application page replacement is achieved by overriding the default **`enqueue_buffer`** and **`evict_buffer`** routines (the implementations of the **`priority`** and **`synchronize`** policy modules). The **`enqueue_buffer`** routine allows the application to keep its buffer pool sorted in the desired order (e.g. LRU) and

the **`evict_buffer`** routine selects the page to evict. Prefetch or read-ahead is supported by overriding the default **`preallocate`** function.

Assume that the default kernel policy places all buffers in the pool in MRU order and evicts the LRU page. Cao describes six access patterns that require specific replacement policies for optimal performance and suggests two new interfaces, set_priority and set_policy, to implement resource-specific page replacement [9]. These two routines map cleanly into the **`enqueue_buffer`** and **`evict_buffer`** routines described in Table 2. By implementing these routines as instances of generic policy modules (**`priority`** and **`synchronize`**), we create a more flexible solution, capable of implementing any replacement policy rather than merely providing a selection of existing policies.

The Better Update policy proposed by Mogul [21] suggests alternative policies for writing dirty buffers to disk. The periodic update policy (PU) is implemented by specifying a writeback routine with a fixed frequency and writing all dirty buffers in the writeback routine. The approximate interval update policy (AIPU) is only slightly more complicated. The frequency parameter still indicates how often the writeback routine is scheduled. However, since the AIPU policy only writes blocks that have been dirty for a threshold length of time, it must either rely on time stamps provided by the kernel or maintain state describing what buffers were dirty during the past N invocations of the writeback routine. In either case, implementing any of the update algorithms requires replacing the generic **`synchronize`** module.

Another area where we want to introduce small, incremental changes is in scheduling. Many applications are made up of groups of processes working in concert (e.g. client-server systems, synchronized multimedia applications, or multi-threaded systems using the scheduler activations [4] virtual multiprocessor model). The scheduling decisions made by the kernel in these cases should be on a per-group basis, not a per-process basis. In our fine-grained extensibility framework, these models are all provided by a two-level group scheduler. The kernel multiplexes between groups of processes according to time quanta allocations. When a process blocks in the kernel before the expiration of its quantum, the kernel calls the process-specific scheduling module that determines which process should be scheduled next. The selected process does not lose its place on the normal scheduling queue, but is allocated the rest of the quantum. In this manner, if one of the processes of a group blocks in the kernel, the group as a whole can continue to make forward progress.

In the case of a client-server application, a client blocked in the kernel waiting for the results of a synchronous server request passes its time allocation onto the server, decreasing the length of time the client must wait for results. A multi-process multimedia application passes the baton to whichever component needs the time the most.

In the case of scheduler activations, if one of the virtual multiprocessors blocks in the kernel, it means that the thread running on that virtual processor is blocked. Another one of the virtual processors can be scheduled, and requested to run a different thread, so that some thread of the application carries on.

4 Implementing Fine-Grained Operating System Extensibility

We believe that operating systems must be designed from the ground up to provide fine-grain extensibility. Before embarking upon a full kernel implementation project, we wanted to quantify the cost of providing this flexibility. We also wanted to demonstrate other areas, less often investigated in operating systems research, in which small, incremental changes to kernel algorithms could elicit non-trivial performance and/or functional improvements.

We hypothesized that we could construct a simple extensible framework in an existing kernel and use it to measure both the impact of the extensibility as well as the benefit. We selected BSDI BSD/OS 2.0, a 4.4BSD-Lite derivative, as our base system as its architecture is well-understood and well-documented [16], the source code is easily available, and the hardware on which it runs is inexpensive. We extended the kernel interface to allow applications to set policies on a fine grained (per resource) basis. We chose two subsystems for analysis: the file system read-ahead code (our idea being that most people would assume there was little that could be done to improve read-ahead), and the POSIX.1 locking subsystem.

4.1 Interface

While a truly extensible system would allow dynamic downloading of policies at runtime (e.g. spindles [7] or grafts [28]), we wanted to analyze the impact of application-controlled resource management in the context of an existing system. In our implementation, policy modules are statically linked in to the BSD/OS 2.0 kernel at build time. We extended the system to support two new system calls, **`setpolicy`** and **`getpolicy`**. The **`setpolicy`** call takes a resource identifier, a policy type and a policy identifier and two integer parameters, for the use by the specific policy code.[2] The **`getpolicy`** call takes a resource identifier and a policy type and returns the current policy. In all cases, if no policy is explicitly set, the default policy is used.

Inside the kernel, a policy is specified by three functions: **`install`**, **`invoke`**, and **`remove`** (see Table 3). The **`install`** function of a policy is called when that policy is set on a resource. An **`install`** function is responsible for whatever setup is necessary (e.g. allocating storage). The **`remove`** function is called when the policy is removed (because the resource is deallocated, the owning process terminates, or a different policy is installed). The **`remove`** function is responsible for deallocating any storage allocated by the **`install`** function.

Name	Description
`install`	Initialization code, executed once at setpolicy time.
`invoke`	Called when the designated policy is needed.
`remove`	Cleanup code, executed once when policy is overridden or removed.

Table 3. Policy Function Interface. Entrypoints exported by each policy implementation.

4.2 Overhead

Given our prototype implementation, there are two components to the cost of user-specified policy. There is the cost of specifying the policy and the overhead of selecting the correct policy at runtime.

We measured this overhead on a Intel 80486 DX2-66. In our calculation of overhead, we pessimistically assumed that applications specify a large number of non-default policies (100). (The **`setpolicy`** implementation uses a simple linear scan to traverse the list to determine if the policy specification overrides an existing one.) The **`setpolicy`** call takes on average, 23 microseconds (with a standard deviation under 5% on 30 runs of 100,000 calls each). In comparison, a call to **`getpid`** takes 9 microseconds on the same system. The difference is due to the time to traverse the list of 100 policies.

Policies are not carried forward across **`fork`** or **`exec`**. After a **`fork`** or **`exec`**, a process' policies are set to their default values.

The policies are stored in an unordered list chained off the proc structure. At each point in the kernel where a process might have specified a policy, the kernel consults the policy list for the current process. If a policy is found, it is invoked, otherwise the default policy code is invoked.

This is most obviously a proof-of-concept system and is not the implementation of choice for a real system. First, in a system designed around process-specific policies, these policies would have to be maintained in a more efficient fashion (e.g. a system-wide hash table of process-resource tuples). Second, if policies are downloaded at runtime, the cost of policy specification will be substantially greater than the time we measured (1.5 times that of a minimal system call). For example, in the SPIN model, where spindles are written in a type-safe language, the installation time includes copying the code into the kernel and setting up the necessary jump tables. One might also wish to incorporate the compilation time in this calculation. Similarly, in the VINO model, grafts must be

2. The decision to pass two parameters was arbitrary. It is clear that two is not always the correct number of parameters for any policy, but it satisfied the policies we were investigating.

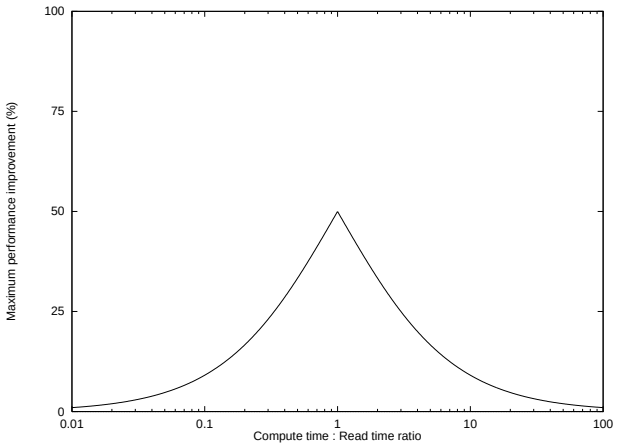

Figure 1. Performance gain from read-ahead. This graph plots the improvement in execution time that can be attained using read-ahead as a function of the ratio between an application's compute time and its I/O time.

precompiled and downloaded, and software fault isolation techniques [30] impose a runtime penalty on top of the installation penalty.

4.3 Read-Ahead

Contemporary file systems use read-ahead to improve read throughput by prefetching data into the file system cache in the hope that it will soon be needed by the user. This simple enhancement optimizes file system performance for the common case of a sequentially read file. There are a variety of situations, however, where current read-ahead policies are inadequate, either because they do not improve performance, or because they degrade it. In many of these cases, the application could provide information about its file access pattern to the kernel, allowing the file system to implement a more intelligent read-ahead policy.

The fraction of a process's total execution time spent blocked on read requests determines the maximum performance gain that can be achieved using file system read-ahead. Ideally read-ahead should produce a perfect overlap of computation and disk reads, eliminating the time a processes might spend blocked during read calls. This ideal is not always attainable, as a process that requires more read time than computation time cannot achieve perfect overlap. Figure 1 shows the maximum performance gain as a function of the ratio between compute time and read time.

Current file systems use a small range of read-ahead policies. The UNIX System V file system [6] and the original implementation of the Berkeley Fast File System [18] issue a read of one additional block per explicitly requested block. Enhanced versions of these file systems [23] [20] cluster logically sequential file blocks on physically contiguous disk blocks, and perform read-ahead by reading full clusters of blocks in each I/O operation. In these systems, there are only two read-ahead policies supported by the file system. Either the file system performs read-ahead, using the one algorithm that it supports, or it does no read-ahead. When the file system detects sequential read requests, it enables read-ahead. The application cannot specify which policy should be used, nor can it provide an alternate policy.

This limited range of read-ahead policies penalizes applications that perform non-sequential I/O. A common example is a database management system using a sorted index to search a database. Since it is unlikely that the records are stored in sorted order, the search issues non-sequential reads from the database file. Existing sequential read-ahead policies are of no use in this scenario, yet the database knows the order in which it will read the records. If the operating system allowed the database to transmit this information to the file system, database records could be pre-fetched using read-ahead. Allowing application control of read-ahead policy also prevents false read-ahead. This degenerate behavior occurs when the file system uses read-ahead to fetch blocks that the application does not read.

Because existing read-ahead policies fail to fully exploit the possible overlap of execution and I/O, applications that perform sequential reads can also benefit from greater control over read-ahead policy. To demonstrate this effect, we implemented application specified read-ahead using the prototype extensible system described in Section 4.1. Applications set read-ahead policy on a per file descriptor basis. The default policy is the read-ahead policy implemented by the base operating system. The BSD/OS implementation of FFS includes the clustering enhancements described by McVoy [20]. The default read-ahead policy exploits this clustering by reading full clusters instead of individual file system blocks. When the file system receives a read request for a block, it determines whether the subsequent block of the file is already in the file system cache. If it is not, the file system issues a request for the entire cluster containing that block. Note that under this policy, if an application reads two blocks of data that span a cluster boundary, the application blocks on the disk, since the second of the two blocks is not yet in the cache.

As an alternative read-ahead policy we allow applications to specify a number of blocks that should be prefetched into the cache with each read request. Like the default read-ahead policy, the new policy only reads full clusters of data. By allowing the application to specify a read-ahead amount, however, an application can guarantee that at least enough data to satisfy the next read call are prefetched into the cache. This mechanism also allows us to turn off read-ahead by specifying a read-ahead size of zero.

To determine the performance gain achievable by customizing file system read-ahead policy, we measured **`gunzip`**, the GNU decompression utility, using a variety of read-ahead policies. Table 4 and Figure 2 show the results. The data show the large effect that read-ahead can have on an application's performance. **`gunzip`** performs better with the default read-ahead policy than it does with no read-ahead (read-ahead size = 0), executing eleven percent faster. The increased overlap of computation and I/O improves processor utilization by fourteen percent. Although this improvement in performance is large, the data also show that there is room for improvement upon the default read-ahead policy. A more aggressive read-ahead policy improves the execution time of **`gunzip`** by an additional six percent, while increasing the CPU utilization to nearly one hundred percent.

The additional performance improvement attained by using a more aggressive read-ahead policy can be explained by considering the read requests issued by **`gunzip`** and the pre-fetching behavior of the default policy. **`Gunzip`** reads its input file 32 kilobytes at a time. This translates to four file system blocks on the test system. Thus, in order to attain the maximum possible benefit from read ahead, the file system must have at least four blocks of pre-fetched data in the cache each time **`gunzip`** issues a read. In the default policy used by BSD/OS, a new cluster is read into the cache only when the final block of the preceding cluster is requested by **`gunzip`**. As a result, any time **`gunzip`** issues a four block read request that spans a cluster boundary, it will block while a new cluster is read from disk. Since the maximum cluster size supported by BSD/OS is eight blocks, at least half of **`gunzip`**'s read requests block when using the default policy.

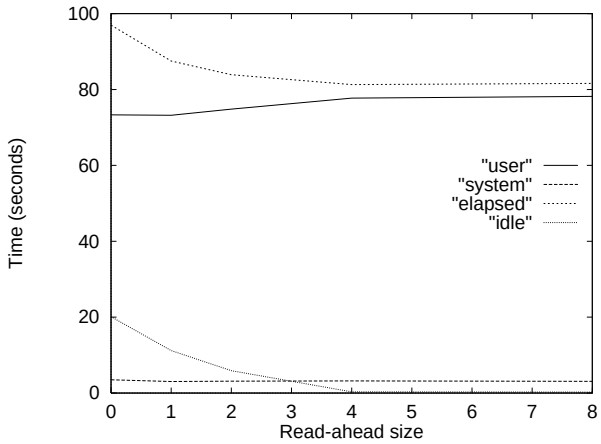

Figure 2. Performance as function of read-ahead size. This graph plots the performance of **`gunzip`** as a function of the file system read-ahead size. Separate lines plot the time (in seconds) spent in user and system modes, the total run time, and the idle time.

Read-Ahead Size	User Time (seconds)	System Time (seconds)	Elapsed Time (seconds	CPU Utilization (%)
0	73.312 (1.126)	3.488 (0.155)	96.960 (1.093)	79.250 (0.267)
1	73.200 (0.844)	3.033 (0.245)	87.496 (0.815)	87.200 (0.122)
2	74.825 (0.705)	3.137 (0.200)	83.892 (0.651)	92.975 (0.046)
4	77.725 (0.555)	3.188 (0.242)	81.289 (0.601)	99.600 (0.000)
8	78.175 (0.792)	3.087 (0.217)	81.618 (0.780)	99.688 (0.035)
Default Policy	74.712 (1.289)	3.188 (0.173)	86.555 (1.374)	90.050 (0.177)

Table 4. Performance impact of read-ahead policy on gunzip. The data in this table were collected running **`gunzip`** on a 21 megabyte compressed file using **`gzip -4`.** The uncompressed size of the input file was 86 megabytes. The left-most column displays the read-ahead size used. The data columns show the amount of time spent executing in user and system mode, the total execution time, and the CPU utilization. For comparison, the performance of the default read-ahead policy implemented by BSD/OS is also provided. All tests were performed eight times on an Intel 80486 DX2-66 system with eight megabytes of RAM, an Adaptec 1542 SCSI controller, and a 422 megabyte HP C2229 disk. The tests were performed with the system running in single user mode. The decompressed output was redirected to /dev/null. Standard deviations are shown in parentheses.

Because **`gunzip`** knows how much data it will request in each read call, it can accurately tell the file system how many blocks to prefetch. In Figure 2 we see that the performance of **`gunzip`** steadily improves as the read-ahead size is increased to four blocks. At that point, **`gunzip`** has reached its maximum performance and additional increases in the read-ahead size do not offer further performance improvements.

When no read-ahead is performed, **`gunzip`** utilizes approximately 80% of the CPU. Most of the remaining 20% of **`gunzip`**'s execution time is spent waiting for disk reads to complete. Because no read-ahead is performed, all of the reads are completely synchronous. Since a perfect read-ahead policy completely overlaps computation with these reads, the maximum performance improvement we can expect from read-ahead is 20%. The data in Table 4 indicate that performance actually improves by 16.2%. The performance improvement is less than we predicted because the amount of time **`gunzip`** spends in user mode increases as the read-ahead size is increased. This is caused by contention between the CPU and the disk controller for the memory bus. The disk controller in the test system uses DMA to transfer data to and from main memory. As the overlap between computation and I/O increases, the contention for the memory bus also increases, leading to a larger average memory access time for **`gunzip`**. Our test platform uses an older bus architecture (ISA). More advanced buses offer wider and faster DMA transfers, which should reduce contention for the memory bus and offer increased performance gains for larger read-ahead sizes.

Read-ahead is a seemingly minor feature of a file system implementation, yet the simple read-ahead policies implemented by current file systems do not provide optimal performance for all applications. A global change in read-ahead policy might improve the performance of some applications, but there can be no single policy that is optimal for all applications. Sequential and non-sequential I/O require different read-ahead strategies. An aggressive read-ahead policy may be well suited to some applications, but may perform poorly with slower applications where the prefetched data is evicted from the cache before the application can use it. In a system that supports fine-grain extensibility, different read-ahead policies can be used by different applications, and a single application can specify the read-ahead policies used for each of its files.

4.4 Locking

The POSIX.1 locking system provides a system call interface to locking bytes or byte-ranges in a file. Although many systems have provided this locking interface for a number of years, applications that require complex lock semantics rarely use the interface, implementing their own lock manager instead [13] [27]. A recent study examined the imple-

mentations found on several widely-used systems and discovered that most implementations use simple and often inefficient data structures (e.g. lists), some implementations make it impossible to provide strict two-phase locking semantics, and in general, the locking services provided by the operating system are unsuitable for database applications [29]. Two implementation details that are particularly egregious for database systems are the semantics of upgrading a read lock to a write lock and the management of lock queues. The former is implemented correctly on our test platform; we focused on offering an alternative to the latter.

BSD/OS implements a lock scheduling algorithm that approximates FIFO/Reader's Priority (FRP). Lock requests are maintained in two sets, one for readers and the other for writers. When a read lock becomes available (e.g. when a write lock is released), all waiting readers are allowed to acquire a read lock on the region, providing reader's priority. When a write lock becomes available (e.g. when the last read lock is released) all writers are scheduled at the same priority level. Due to the interaction between the implementations of the locking subsystem release algorithm and the process scheduling queue, the process that has been waiting the longest for the write lock is given priority. This provides FIFO ordering among writers. There is one exception to this algorithm: when a reader upgrades to a write lock it does not release its read lock, but instead holds it and waits on the write lock wait queue. When the last non-upgrading reader releases its lock, the writers are scheduled. Because the upgrading process has not released its read lock, the other writers can not proceed (because the conflict still exists), and the upgrading process obtains the write lock.

Because the default algorithm gives priority to readers, it is possible for a writer to starve: as long as read requests continue to arrive, and are allowed to move ahead of the queued write request, the writer cannot proceed. In addition, because locks are obtained on arbitrary ranges of bytes, not fixed-size regions, it is possible for any lock request to starve. If a read request is made for the range 0...1, and byte 0 is locked for writing, the read request blocks. A second writer can then obtain a lock on byte 1, jumping past the enqueued request. The first lock may then be released, and acquired by a different writer, blocking the reader. As long as a writer holds a lock on one byte or the other, the read lock will never be granted.

An alternative to the FRP scheme is one in which lock requests are processed in strict FIFO order. An incoming read lock request is never allowed to proceed before a write lock request that arrived earlier. Although this forces reader processes to wait when they could proceed, it avoids starvation.

In existing implementations, it is not possible to provide applications different semantics depending on their needs. Using our extensibility infrastructure we constructed an alternative lock queueing policy (FIFO), where requests are granted in strict FIFO order, to contrast with the default (FRP). We measured the reader and writer throughput of each policy, and the percentage of lock requests that block, under a varying ratio of reader and writer processes. (Note that we do not claim that this is a "better" policy, only that it provides fairness. The default policy would be preferred in cases with low contention, or where readers are judged to have higher priority than writers.)

Intuitively, one expects that both the read throughput and the total throughput will decrease under FIFO, but that write throughput will increase. To test this hypothesis, we simulated a high-contention, I/O bound system. We built a simple test program that requests a read or write lock on byte 0 of a file, performs I/O, computes, and then releases the lock. The test program accepts the tuning parameters described in Table 5. It determines whether to request a read lock or a write lock based on the value of the write percentage parameter, w. It then requests the lock, waiting if the lock is not available. When the lock is obtained the program reads a randomly chosen 8KB record from a 32MB file, computes for 15ms, and releases the lock.

Option	Default	Description
i	1000	number of iterations
p	FRP	lock protocol (FRP or FIFO)

Table 5. Lock Test Parameters.

Option	Default	Description
r	8192	record size
w	50	percent of requests that should be write locks

Table 5. Lock Test Parameters.

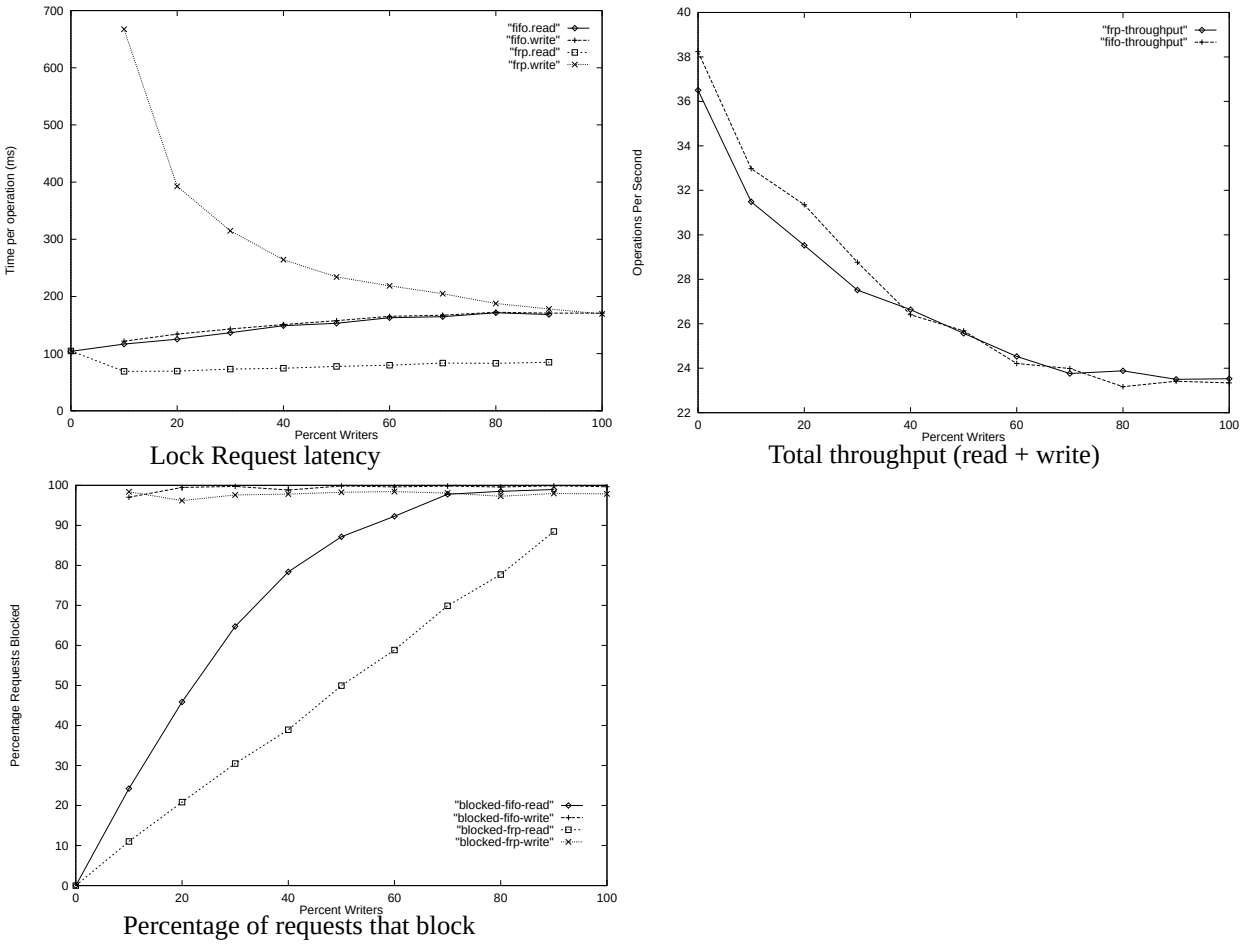

Figure 3. Comparison of lock granting policy. At low write/read ratios, workers are starved, although read latencies are lower. FIFO avoids starvation at the cost of higher read latency.

We first ran the test program with 100% read locks (no lock contention) and varied the number of concurrent invocations of the program until the test system saturated. We used this level of multiprogramming (4) for the tests.

In the case where there is a small number of writers, the wait for a write lock can be quite long. When the operation mix is 95% readers, under the FRP policy a read operation takes 105 ms to complete, while a write operation requires 667 ms. The write operation time decreases quickly as the percentage of write operations increases. Under the FIFO policy, the read operation time tracks the write operation time; readers and writers make roughly equivalent progress under all loads.

We also found that under FRP the amount of time spent obtaining a read lock initially decreased as writers were added. This is because of the time spent waiting for write locks; the total system load decreases because writers are blocked, and readers can complete more quickly.

The total throughput of the two policies is equivalent across the tested workloads. This is not surprising; both read and write operations take the same amount of time (once the lock is acquired), so some number of processes are always making progress.

Although FIFO met the goals of this test, it is not universally preferable. Applications for which reader throughput is more important than writer throughput would choose FRP over FIFO. In some cases, FIFO is the right choice; in others, FRP is better. Offering only one or the other is an unnecessary compromise. Other policies might be preferable at times: for example, a modified FRP might start blocked writers out with priority lower than newly-arrived readers, but guarantee service within a fixed interval. Our proposal is that each application be allowed to specify which policy works best for it.

5 Other Approaches to Extensible Operating Systems

Several current research projects are exploring the domain of extensible operating systems. The particular implementation strategy for achieving extensibility is not the paramount issue, the key issue; is in determining where operating systems make policy decisions and exposing those interfaces, so that operating system functionality may be modified as well as extended. The SPIN system [7] is one example of an extensible kernel. The spindle construct is designed to allow downloading of user-level server code into the kernel for improved performance, but also provides a framework in which kernel behavior may be modified as well. The Aegis system [14] provides another framework in which to support fine-grain extensibility. The low level kernel (the Exokernel) provides only the abstractions of the hardware. Remaining kernel functionality is implemented in user-level libraries. Since most kernel decisions are implemented in these user-level libraries, it seems feasible for applications to select those policies best suited to their needs. The VINO system [28] permits applications to alter normal kernel behavior by supporting application-directed policy using software fault isolation. If the policy interface is sufficiently fine-grained, then that model also adequately supports useful operating system extensibility.

6 Conclusion

We have presented a framework in which operating system functionality can be modified or extended in small, incremental ways. This framework provides a context in which to describe may of the recent research results in operating systems. We have demonstrated that the flexibility required to support fine-grain extensibility is small and that even the simplest mechanisms can be effectively extended by permitting applications to tweak kernel behavior. If operating systems are to continue evolving to meet the needs of tomorrow's applications, this incremental tuning is essential.

7 Bibliography

[1] AT&T, "System V Interface Definition, Third Edition," Volumes 1–3, 1989.

[2] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., Young, M., "Mach: a New Kernel Foundation for UNIX Development," *Proceedings of the 1986 Summer Usenix Conference*, 93–112.

[3] Alfieri, R., "An Efficient Kernel-Based Implementation of Kernel Threads," *Proceedings of the 1994 Summer Usenix Technical Conference*, Boston MA, June 1994, 58–72.

[4] Anderson, T., Bershad, B., Lazowska, E., Levy, H., "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, Monterey CA, October 1991, 95–109.

[5] Appel, A., Li, K., "Virtual Memory Primitives for User Programs," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, 96–107.

[6] Bach, Maurice J., *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[7] Bershad, D., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S., Gun Sirer, E., "SPIN-An Extensible Microkernel for Application-specific Operating System Services," Dept. of Computer Science and Engineering, University of Washington, Seattle, Technical Report 94-03-03.

[8] Cao, P., Felten, E., Li, K., "Application-controlled File Caching Policies," *Proceedings of the 1994 Summer Usenix Technical Conference*, Boston, MA, June 1994, 171–182.

[9] Cao, P., Felten, E., Li, K., "Implementation and Performance of Application-Controlled File Caching," *Proceedings of the First Usenix Symposium on Operating System Design and Implementation*, Monterey, CA, November 1994, 165–177.

[10] Chou, H., DeWitt, D., "An Evaluation of Buffer Management Strategies for Relational Database Systems," Proceedings of the 11th International Conference on Very Large Databases, 1985.

[11] Fall, K., Pasquale, J., "Exploiting In-Kernel Data Paths to improve I/O Throughput and CPU Availability," *Proceedings of the 1993 Winter Usenix Technical Conferenc*e, San Diego CA, January 1993, 327–334.

[12] Harty, K., Cheriton, D., "Application-Controlled Physical Memory using External Page-Cache Management," *Proceedings of the Fifth International Conference on Architectural Support of Programming Languages and Operating Systems*, Boston, MA, October 1992, 187–192.

[13] IBM, *HACMP/6000 Programming Locking Applications* (SC23-2704), IBM Customer Publications Support, 1994.

[14] Kaashoek, F., "The Aegis System," to appear in *Proceedings of the 1995 conference on Hot Operating Systems*, Orcas Island, WA, May 1995.

[15] Kiczales, G., Lamping, J., Maeda, C., Keppel, D., McNamee, D., "The Need for Customizable Operating Systems, *Proceedings of the Fourth Workshop on Workstation Operating Systems*, Napa CA, August 1993.

[16] Leffler, S., McKusick, M., Karels, M., Quarterman, J., *The Design and Implementation of the 4.3 BSD Unix Operating System*, Addison-Wesley, 1989.

[17] McCanne, S., Jacobson, V., "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *Proceedings of the 1993 Winter Usenix Technical Conference*, San Diego CA, January 1993, 259–269.

[18] McKusick, M., Joy, W., Leffler, S., and Fabry, R., "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, Vol 2., No. 3, August, 1984, 181–197.

[19] McNamee, D., Armstrong, K., "Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies," *Proceedings of the Usenix Mach Symposium*, 1990, 17– 29.

[20] McVoy, L., Kleiman, S., "Extent-like Performance from a UNIX File System," *Proceedings of the 1991 Winter Usenix Technical Conference,* Dallas, TX, January, 1991, 33–45.

[21] Mogul, J., "A Better Update Policy," *Proceedings of the 1994 Summer Usenix Technical Conference*, Boston, MA, June 1994, 99–112.

[22] Nyberg, Chris, *Disk Scheduling and Cache Replacement for a Database Machine*, Master's thesis, University of California, Berkeley, July 1984.

[23] Peacock, J. Kent, "The Counterpoint Fast File System," *Proceedings of the 1988 Winter Usenix Technical Conference,* Dallas, TX, February, 1988, 243–249.

[24] Reiter, Allen "A Study of Buffer Management Policies For Data Management Systems." Tech Summary 1619, Math Research Center, University of Wisconsin, Madison, March, 1976.

[25] Sacco, Giovanni Maria and Mario Schkolnick, “A Mechanism for Managing the Buffer Pool In A Relational Database System Using the Hot Set Model, *Proceedings of the 8th Conference on Very Large DataBases*, September, 1982.

[26] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B., “Design and Implementation of the Sun Network Filesystem,” *Proceedings of the Summer 1985 USENIX Technical Conference*, Portland, OR, June 1985, 119–130.

[27] Seltzer, M., Olson, M., “LIBTP: Portable, Modular Transactions for UNIX,” *Proceedings of the 1992 Winter Usenix Conference*, San Francisco CA, January 1992, 9–25.

[28] Seltzer, M., Endo, Y., Small, C., Smith, K., An Introduction to the VINO Architecture, in “VINO: The 1994 Fall Harvest,” Harvard University Technical Report TR-34-94.

[29] Skarra, A., “Using OS Locking Services to Implement a DBMS: An Experience Report,” *Proceedings of the Usenix Summer 1994 Technical Conference*, Boston, MA, June 1994, 73–86.

[30] Wahbe, R., Lucco, S., Anderson, T., Graham, S., “Efficient Software-Based Fault Isolation,” *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Asheville NC, December 1992, 203–216.

[31] Yuhara, M., Bershad., B., Maeda, C., Moss, E., “Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages,” *Proceedings of the 1994 Winter Usenix Technical Conference*, San Francisco CA, January 1994, 153–163.