The File System Belongs in the Kernel

Brent Welch
welch@parc.xerox.com
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

This paper argues that a shared, distributed name space and I/O interface should be implemented inside the operating system kernel. The grounding for the argument is a comparison between the Sprite network operating system and the Mach microkernel. Sprite optimizes the common case of file and device access, both local and remote, by providing a kernel-level implementation. Sprite also allows for user-level extensibility by letting a user-level process implement the naming and I/O interfaces of the file system. Mach, in contrast, provide general interprocess communication and does not define a file system protocol in the kernel.

1 Introduction

This paper argues that the file system is a mature enough abstraction that it should be implemented inside the operating system kernel for optimal performance. Data storage and high-level naming are fundamental features of today's computer systems, along with process and communication abstractions. In the Sprite network operating system, a shared network file system is the basis for a distributed system, and it is implemented inside the operating system kernel. The kernel provides the framework for a transparently distributed hierarchical name space and for an I/O stream abstraction with UNIX semantics. The Sprite file system provides higher performance than NFS and AFS [Nelson88] [Ousterhout90]. Unlike these two systems, the Sprite file system retains the full semantics of the UNIX I/O stream abstraction even in cases of network sharing [Welch90].

In comparison, the central abstractions in Mach are ports, messages, tasks, threads, and memory objects. These are lower-level, more general purpose abstractions than those of a file system. Therefore, many would argue that the operating system should provide only these general-purpose facilities and defer higher-level facilities like the file system to implementation at user-level. However, I argue that our experiences with UNIX have taught us the value of the file system abstractions, so they merit efficient support in a kernel-level implementation. Consider the following quote from Butler Lampson's article *Hints on Computer System Design* [Lampson83].

> "If the interface is used widely enough, the effort put into designing and tuning the implementation can pay off many times over. But do this only for an interface whose importance is already known from existing uses. And be sure that you know how to make it fast."

The file system satisfies both of Lampson's criteria; its importance is well-known and there are caching techniques that make it perform well. The fundamental differences between the VM interface and the file system interface will discussed in more detail below.

Aiming at a high-level interface has two benefits. First, the interface provides more functionality so its clients are simpler. In particular, the Sprite file system abstraction handles network transparency. Second, in focusing on a high-level interface, the designers have more freedom to make different design decisions. Sprite optimizes network access by providing a kernel-to-kernel RPC protocol, and it optimizes remote file access by providing a distributed caching scheme. It optimizes performance in general by putting the implementation inside the kernel. The hierarchical file system name space has names for a variety of object types, not just files.This requires careful design of the file system interfaces, which are described in more detail below. The final touch is to add an upcall facility[Clark85] so that a user-level process can implement the naming and I/O interface for functions we did not wish to put into the kernel. Thus Sprite takes a hybrid approach. It chose a well-known, high-level, widely used interface and supports it in the kernel, yet provides an escape hatch to user-level for more arbitrary functionality.

In contrast, a microkernel requires an external file system implementation in order to be generally useful. The strategy taken by Mach is to layer file access on top of the virtual memory system by using mapped files. The section below describes why this is not necessarily a good idea, especially in a network environment. Also, one of the nice features of the UNIX file system is the hierarchical name space. The only names implemented in the Mach microkernel are ports, which are too low-level to be useful by themselves. Instead, a name service is also required for the microkernel to be useful. In short, you'll need a file system anyway, and layering it on top of a virtual memory-oriented, message-passing kernel is not the best way to go about it. The next few sections elaborate on this point.

2 The File System vs. Virtual Memory

The fundamental difference between the virtual memory interface and the file system interface is that the virtual memory interface is used on every load and store instruction, while the file system interface is invoked at a few well defined moments. As a result, considerable hardware support is provided for the VM system so it performs well. Translations from virtual to physical addresses are cached in fast registers, and CPUs are designed so they can make virtual memory references with little or no penalty. However, this model does not extend well to networks of loosely coupled machines. Work by Kai Li has demonstrated that it is feasible, but his work is applicable to a certain class of applications that share memory at a relatively infrequent rate [Li89].

In contrast, the file system is oriented towards access to slow devices such as disks and terminals. In these cases, it is more natural to use read and write procedures that move data from the slower devices into main memory. Open and close procedures provide convenient points at which the system can set up state about the underlying device in order to optimize later read and write calls. In particular, setup can be performed to optimize network accesses. Sharing resources is also easier via the file system interface because the calls into the file system interface provide well defined points at which to check for consistency among shared copies. In contrast, network shared memory schemes must rely on complex page protection schemes that grant ownership of a page to a particular process.

Consider the hard case for network sharing when processes on different machines are sharing and modifying the same page of data. If the page is shared via the VM interface, the system must arrange page protections so that one process has read-write privileges, while

all other processes have no access rights. The later processes will suffer a page fault and the cost of copying a whole hardware pageframe across the network when they try to access the shared page. Recall that this can occur as often as every load and store instruction. In contrast, the Sprite file system uses a simple trick to handle the hard case. It simply disables the caching for the shared file, forcing the read and write calls to go over the network to the server for the file[Nelson88]. I/O operations are serialized in the main memory file cache of the server. In most cases less data is transferred over the network, too, because only the modified bytes need to be sent to the server in this case, not the whole page. This example illustrates one of the fundamental differences between the file system and VM. A virtual memory page always has to be resident in physical memory in order to access it, while file system data can be streamed into memory from the device, often using a highly tuned hardware channel interface. Granted, the file system interface implies a copy operation from the device into the user's memory, but this occurs when the programmer invokes read, plus it can be further optimized by caching the data in main memory.

Another difference between the two systems is the way they access memory. Accesses to file system data is by and large sequential, while virtual memory accesses are more randomized. Systems that layer the file system on top of mapped files often suffer performance problems because of this. The classic bug is that copying a large file can throw out the working set of the running programs, slowing down all those load and store instructions. In Sprite, the file system and the virtual memory system each maintain their own pool of memory pages. When they run out, they negotiate over who should give up a page. The two systems compare their oldest LRU times, and system with the oldest time gives up a page. This technique is refined by biasing against the file system. The VM system cheats and adds 20 minutes to its time before telling the file system. This means that no VM page that has been accessed within the last 20 minutes will be thrown out in favor of a file system cache page [Nelson88].

3 Micro vs. Monolithic

"So what?", you say, given efficient IPC, you could perform all those tricks in the file system that accompanies the microkernel. However, the microkernel makes a fundamental trade-off between security clearance procedures and performance. Smaller programs are easier to certify as secure, so the goal of the Mach microkernel is to have a small kernel and a set of small system servers. However, by moving system services out of the kernel address space there are inevitable performance consequences.

First, the resources controlled by the file system are ultimately accessed via kernel-resident device drivers. Note that in Sprite, this includes access to the network as well as to disks and other peripheral devices. Therefore, the execution path ultimately has to end up inside the kernel. If the file system is implemented in a separate address space, then file system accesses suffer additional traps into the kernel in order to effect the change of address space. Furthermore, each additional address space that is crossed places more load on the MMU. For example, the Sun MMU is designed to support efficient access to 8 or 16 address spaces, one of which is reserved for the kernel. The DECstation has a 64 entry TLB, but it can get away with this because most of the operating system kernel resides in a memory range that is mapped one-for-one to physical pages, thus bypassing the small TLB. Processor caches have similar problems. Often a change of address space requires a cache flush. In the Suns, the cache is only flushed when one of the hardware contexts is reused, but this

will happen more often when there are more address spaces placed in the critical path of the file system. Finally, each address space crossing adds more instructions to the execution path. There is an inevitable amount of glue code, however small, associated with the boundary. If the message abstraction is added, then there is even more code to pack and unpack things from the messages.

Thread scheduling is another potential source of overhead in the microkernel approach. Not only does the execution path cross address spaces, but a new thread must be found to execute in the other address space. This overhead is not present in a kernel-resident system. The application thread traps into the kernel and continues to execute in the kernel's protected address space. After that, communication between various kernel-provided services is achieved with a simple procedure call.

Recently, Bershad has extended this idea to apply to user-level IPC, eliminating the need to schedule a new thread [Bershad90]. In Bershad's LRPC mechanism an execution stack is mapped into a server process for each client that is bound to it. The client thread traps into the kernel when crossing the protection boundary, but it continues to execute on the shared execution stack after the kernel fixes up the address space. Using this technique, Bershad reports a null round-trip time of 125 microseconds on a 3 MIP C-vax. In contrast, Draves reports the same time for an optimized version of Mach IPC, but on a 12 MIP DECstation [Draves90].

In spite of recent advances in IPC by Bershad, which I applaud, the fact remains that execution paths will be longer in services that are provided outside the operating system kernel. Extra glue code at the interfaces, traps into the kernel, and MMU effects from changing address spaces all contribute to overhead. Given that the file system is as fundamentally important as the VM interface, and the potential problems with implementing one in terms of the other, it seems reasonable to provide an efficient, kernel-level implementation of the file system.

4 Benefits of a Shared File System

A UNIX file system supports two main abstractions, pathnames and I/O streams. These abstractions were derived from earlier work in Multics [Feirtag71]. Experiences with these abstractions have shown that the notions of device-independent naming and I/O are extremely useful, and that the lack of them in a network environment is frustrating. Accordingly, Sprite extends these file system abstractions to a network environment. Additionally, Sprite provides process migration so that cycles can be shared across the network. The combination of a shared file system and process migration makes a network of Sprite workstations into a powerful computing platform.

If the file system is chosen as the basis for the system, a number of simplifications are possible. First, the file system can act as the name space for the system. UNIX, for example, uses special files to represent peripheral devices. Additionally, Sprite uses special files to represent user-level server processes known as *pseudo-devices* [Welch88]. The services implemented as pseudo-devices include a TCP/IP protocol server, terminal emulators, and the X display server. More details about the pseudo-device mechanism will be given below.

Another simplification possible in Sprite is that regular files are used as virtual memory backing store as opposed to having preallocated, dedicated swap space. This is especially convenient in a network of diskless workstations. First, it is not necessary to preallocate

swap space on disk as it is in most UNIX systems. Second, a remote file server can share a swap directory among many clients. This approach is valuable in today's networks of workstations with large memories and applications with large working sets. The Sprite network at Berkeley uses a single, 600 Meg disk for the backing store of over 40 hosts.

The shared file system also simplifies the implementation of process migration. An address space is moved to a new host simply by paging it all out to the shared file system and demand paging it back it at the new site. Similarly, there is no difficulty with data files or device access after a migration because all file system resources are uniformly available on all hosts. I will admit, however, that the algorithm to correctly migrate an open I/O stream while preserving the semantics of shared UNIX I/O streams was tricky to get right.

5 Sprite Features and Development History

Sprite is a 4.3 BSD UNIX compatible operating system with extensions for a distributed file system, process migration, multi-threaded address spaces, and a multi-threaded kernel for use on a multiprocessor. The kernel was coded from scratch in C, from the device drivers and boot code up through the system call layer. The project began with professor John Ousterhout and 4 graduate students: myself, Andrew Cherenson, Fred Douglis, and Mike Nelson. Later students included Mendel Rosenblum, Mary Baker, John Hartmann, Ken Sherriff, and Jim Mott-Smith. Adam de Boor, Robert Bruce, and Mike Kupfer were valuable staff members.

The facilities implemented in the Sprite kernel include:

- A debugnub to support remote kernel debugging.
- Device drivers.
- A kernel-to-kernel RPC network protocol.
- Address spaces with virtual memory.
- Multiple threads of execution per address space.
- A transparently distributed hierarchical name space.
- An transparently distributed I/O interface.
- A local file system.
- Host monitoring and failure recovery.
- Integration of user-level services into the name space and I/O interfaces.
- Process migration.

The features listed above are given in the rough order they were implemented, although there was considerable overlap. Before the "official" start of Sprite, I had modified SunOS 1.1 to use prefix tables to distribute the name space uniformly among workstations [Welch86b], and to use a kernel-to-kernel RPC protocol for network communication [Welch 86a]. Implementation of the true Sprite kernel began in the Summer of 1985 using Sun2 workstations. The debugnub was built early so we could use a symbolic debugger via a remote workstation running UNIX. File service was initially provided by the prototype Sprite file server. This approach let us defer writing the first local Sprite file system, including disk drivers and a file system format, until the Summer of 1986. Work on the distributed caching system followed soon after the native file server was up. In the fall of 1987 the system sources were moved to a native Sprite file server, a Sun3 with 16 Meg of main memory, and all development continued using Sprite itself. Work continued on graceful failure recovery, user-level extensions, and process migration. During the '87-88 academic year I

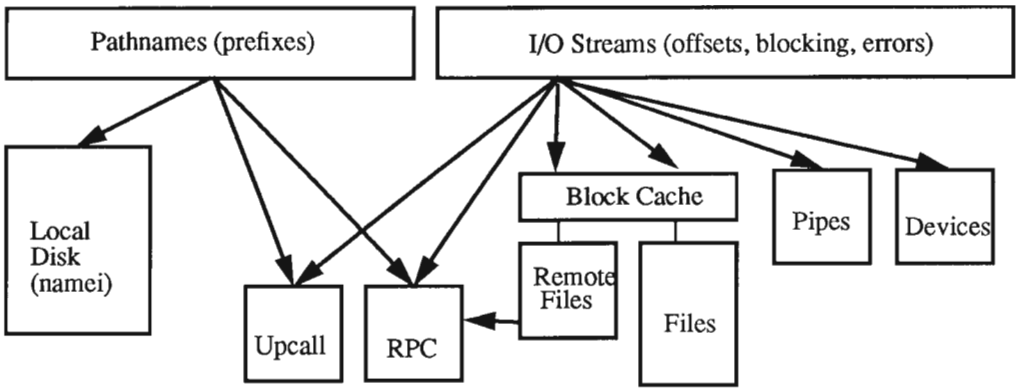

Figure 1. An overview of the Sprite file system architecture. The two primary interfaces involve pathnames and I/O streams. The Upcall module forwards operations to user-level processes. RPC forwards operations across the network to other Sprite kernels.

redesigned and reimplemented the file system architecture to better support the many features that had crept into the system. In 1989, Sprite was ported to the Sun4, DECstation, and SPUR multiprocessor [Hill86]. In the Fall of 1989 the Sprite network expanded to support a number of regular users that included professors and grad students working on other projects. The most recent work on Sprite is the log-structured file system[Rosenblum90][-Rosenblum91] and extensive measurements of the system's day-to-day performance [Baker91]. Today the main Sprite network at Berkeley includes 1 primary file server, a Sun4 with 128 Meg of main memory, 3 auxiliary file servers, and about 40 workstations. The time from initial development to day-to-day usage by non-developers was about 5 years, which matches experiences with other operating system projects [Lauer81].

In contrast, Mach began as a modification of a 4.2 BSD kernel. This gave it a nice head start, and it was quickly used on a number of multiprocessors because it improved on the BSD virtual memory interface that was oriented heavily towards the VAX architecture. In 1986, a USENIX paper hailed Mach as a "new kernel foundation for UNIX development." [Accetta86] However, 5 years later, the microkernel and its accompanying set of server processes is still under development. My conclusion is that it is ultimately cleaner to start from scratch, although there is an initial start-up penalty as basic features are reimplemented.

6 Sprite File System Architecture

The internals of the file system were rewritten in an object-oriented style during the period of about one year (academic year '87-88) to clean up and simplify the interactions among various features. The redesign introduced a generalized *object descriptor*. An object descriptor is a main-memory data structure maintained by the kernel, not a disk-resident representation of an object. A basic object descriptor has a type, uid, server ID (a sprite Host ID), a reference count, and a lock bit. Objects are specified internally by a tuple of <type, serverID, uid>. This base data structure is subclassed* for the implementations of various object types. The object-oriented approach allowed clean separation of different object

*. The various kinds of object descriptors would be the result of subclassing if Sprite were written in C++. However, all the object-oriented features described here were hand crafted in C.

implementations, as well as some sharing. At the same time, the interface between the top-level, generic layers of the file system and the lower-level, object-specific layers was cleaned up based on experiences with the initial design. A diagram of the file system architecture is given in Figure 1.

The pathname interface illustrates the three basic cases handled by the Sprite kernel. The server for a pathname may be the local kernel, in which case the file system implementation is accessed by an ordinary procedure call within the Sprite kernel. The server may be remote, in which case the kernel-to-kernel RPC protocol is used to pass the pathname to the server. Finally, the server may be a user-level process, in which case an upcall mechanism, which is described below, is used to pass the operation up to user level. Thus there are three orthogonal cases that are supported by a Sprite kernel, a local, kernel-resident module, a remote module, and a user-level module.

The importance of this approach is that it extends the general features implemented in upper levels of the kernel to local, remote, and user-provided objects. Notable, high-level features include the name space, error recovery, and blocking I/O. Thus, the focus of the file system architecture has relatively little to do with actual disk management. The focus is on extending the high-level abstractions of pathnames and I/O streams to the network environment.

7 The Sprite Name Space

Sprite uses a prefix table mechanism to implement a uniformly shared, hierarchical name space. Each Sprite kernel keeps a cache of pathname prefixes. The prefixes define the way server domains are coalesced into a single hierarchy. The Sprite naming protocol ensures that servers export their domains consistently so that all hosts, and therefore all processes, see exactly the same name space. In contrast, the V-system and Mach 3.0 use a prefix cache that is maintained on a per-process basis by library routines. While this is advertised as a feature that allows custom name spaces, I believe this is a case where generality is not what you want. Users, administrators, and developers enjoy the simplicity of a single, shared name space. The resulting, fully shared file system supports cross-compilation and easy maintenance for all architectures from any workstation.

The Sprite prefix implementation and the naming protocol are very simple. After matching a pathname against the prefix cache, the remaining pathname is sent to the server identified in the cache. The server traverses its directory structure, expanding symbolic links if necessary, until the lookup terminates or the pathname leaves its domain. A symbolic link to an absolute pathname is one way a pathanme exits, and specifying ".." in the server's root directory is the other way. If the server processes the whole pathname, it performs the requested operation (create, delete, rename, mkdir,...). Otherwise, the server returns the remaining pathname to the client. Relative pathnames bypass the prefix match and are sent to the server of the current working directory.

Mount points are handled by placing a special symbolic link at the mount point. The contents of the symbolic link is the absolute pathname of the mount point, and the link has a different file type than ordinary symbolic links so that the server knows when it hits a mount point. After expanding this link, the server returns the new pathname to the client along with an indication of how much of the pathname is the prefix of the mount point so that the client can add the prefix to its cache. Note that there is nothing in the link but its own name, so some other mechanism is used to locate the server for that name. Currently, Sprite uses

broadcasts to locate the server. After locating the server, the client reiterates the lookup procedure. Bootstrapping is achieved by broadcasting for the server of "/".

There are a number of good properties of the name space, and one limitation. First, clients are simplified. They do not iterate through directories or expand symbolic links, in contrast to the NFS naming protocol. The prefix mechanism completely replaces the UNIX mount mechanism, so servers are no more complex. The most important property is that the name space remains uniform across machines because it is the contents of the symbolic links at the mount points that defines how domains fit together, not a per-host or per-process configuration file. An advantage in common with all prefix caching schemes is that the root server is usually bypassed because clients quickly cache prefixes for the domains they use. Mount points can occur an any directory, so it is possible to nest server domains arbitrarily.

The primary limitation of the Sprite scheme is the use of broadcast to locate servers. This choice was made for simplicity, but it obviously limits the range of the name space. A general solution would be make an upcall in the case that the broadcast fails so that a user-level process can take arbitrary action to locate the server. For example, the Domain name server or some other name service could be used. This solution has the nice property that the kernel implements a lightweight mechanism (broadcast) that works for the common case, but can rely on the escape hatch to user-level in the hard case.

8 Separating Naming and I/O

A mistake that is easy to make when extending a file system to handle more than just files is to blur the distinction between the naming and I/O interfaces. The SunOS implementation, for example, has a *vfs* interface that is primarily concerned with the mount protocol, and a *vnode* interface that includes both naming and I/O operations. These interfaces stemmed from the original UNIX design where the namei procedure was the central core of name lookup. The problem is that namei handles both mount points and directory scanning. When generalizing the implementation to handle remote file systems, namei was retained and the interfaces to mounts and directory scanning were generalized. This erroneously lumps name lookup operations with I/O operations together into the vnode interface. In contrast, the Sprite prefix abstraction replaces the mount abstraction and goes further by hiding the notion of directory scanning altogether.

Consider the open system call that maps from a pathname to an I/O stream. In Sprite, this is broken into two, high-level operations: name_open and io_open. The name_open procedure returns attributes of the named object. The io_open procedure uses these attributes to create an open I/O stream. The name_open and io_open procedures may be implemented by different servers, and this is implemented cleanly by branching through the object-oriented naming and I/O interfaces. Note that a stateless protocol like NFS has no notion of an io_open operation.

The clean separation of naming and I/O allows for a number of optimizations. First, simple objects like devices can rely on a file server to implement the naming interface on their behalf. Special files are used to represent devices and pseudo-devices in the name space. Furthermore, a file server can have special files that represent devices and pseudo-devices on any machine in the network. Contrast this with NFS, which doesn't support remote device access, or even RFS, which only supports accesses to devices on the file server. These systems are limited by the vnode (or equivalent) interface that lumps naming and I/O

together.

Another obvious optimization is that in the case of regular files, name_open sets up enough state to support an I/O stream to the file. This eliminates the need for io_open to contact the file server a second time. This is a subtle, but important optimization for the common case of file access.

Both of the above optimizations are implemented cleanly by introducing a one-procedure interface on the file servers that has different implementations for different *file* types (i.e., for each kind of special file used on the server). After a file server finds a file with a general directory traversal procedure*, its name_open procedure calls through the interface to a type-specific procedure that extracts attributes from the disk-resident file descriptor and takes any special action needed for that type. It is at this point that the cache consistency protocol for regular files is invoked, for example. The attributes are returned to the client for use in calling the io_open procedure.

There are two areas in which Sprite and the Mach microkernel provide similar features, interprocess communication and user-level extensibility of a kernel-level abstraction.

9 Interprocess Communication

The Mach kernel provides a communication mechanism between threads on the same host. Threads inherit or create communication *ports* that are the destinations for messages. A *send right* to a port can be passed in a message so that communication patterns among threads can be built up. Network communication is achieved by using a user-level server, the *netmsgserver*, that maintains a mapping between local and remote ports and forwards messages over the network using a network protocol to a peer netmsgserver. Note that this design means that there are 4 user-level processes involved in a network message exchange: the two processes that wish to exchange the messages and the two netmsgserver processes that forward the message across the network. Extrapolating the microkernel philosophy, the code for the protocol stacks may also be implemented in user level. In the most general design, the function of the netmsgserver might even be separated from the implementation of the network protocol stack. In that case there would be an additional two processes involved in network communication. Another paper in this Symposium describes optimizations to this scheme [Barrera91].

In contrast, Sprite provides two special-purpose communication mechanisms, both of which are basically hidden behind the file system interface. The first is its network RPC protocol that is used solely for communication among Sprite kernels. If a kernel operation needs to be carried out on a remote machine, the kernel-to-kernel RPC protocol is used to invoke it. The other mechanism is an upcall facility that is used in a similar way, except that it forwards the operation up to a user-level process that implements a pseudo-device. The upcall mechanism is somewhat analogous to a Mach kernel using a port to communicate with a user-level external pager process. The equivalent of network RPC is not defined by the Mach kernel. Instead, it is left up to the netmsgserver implementation. Note that the two Sprite mechanisms compose nicely. If a remote, user-level process needs to be invoked, then the network RPC protocol is used first. At the remote host, the operation is converted

*. We are ignoring the effects of the prefix mechanism. This discussion applies to the final server involved in a pathname resolution, the one in whose domain the pathname terminates.

into an upcall by calling through the object-oriented interface

The network RPC protocol is based on the Birrell-Nelson RPC protocol that uses implicit acknowledgments so that ordinarily an RPC requires only two network packets[Birrell84]. Their basic model was extended to optimize bulk data transfer. Large messages are fragmented into multiple packets, and the whole batch is acknowledged by the reply packet (or subsequent request if it was the reply that was fragmented). A custom implementation allows other optimizations. An RPC request or reply is composed of two buffers plus the header. One buffer, the parameter block, is used to marshall small arguments. The other buffer refers to a large, uninterpreted block of data, usually in user-space, that can remain in place until copied onto the network by the network interface. Packet headers and parameter blocks are automatically byte-swapped at a low level, but only if the receiver has a different byte order. Packet headers contain a boot time-stamp so that crashes and reboots can be easily detected. These optimizations tune the RPC protocol for its primary use as the file system's network transport protocol.

While the RPC protocol is designed to optimize network traffic, the upcall mechanism is designed to reduce context switching and data copying. The buffer space for the upcall messages is kept in the server process's address space, not in the kernel. This allows the kernel to copy data directly between address spaces of the client application and the server process. A write on a pseudo-device can be made asynchronous at the server's option. In this case, write messages are allowed to accumulate in the server's buffer until another type of operation occurs, or until the buffer fills up. The server can also use a read buffer to decouple client reads from the generation of the data by the server. In this case, the server adds data to the read buffer as it is generated, and the kernel copies data out of the buffer in response to read operations by other processes. For example, the X server diverts mouse and keyboard input to read buffers associated with different windows, and applications read the data at their leisure. The select call is also optimized by keeping state bits inside the kernel. The kernel can test the state of a pseudo-device without a context switch to the process. The server updates the state bits as part of the upcall protocol, and it can notify the kernel directly when a pseudo-device changes state.

The Sprite RPC protocol is relatively efficient. A null call takes 2.45 msec between Sun3 class hosts, and about 1 msec between Sun4 and DECstation 3100 class hosts. This is about the same time it takes to exchange a byte of data between user-level processes on the same host using UNIX pipes. Using 16 Kbyte block sizes, Sun3 workstations can transfer data at 800 Kbytes/sec on a 10 Mbit/sec ethernet, while Sun4 workstations can achieve 900 Kbytes/sec. Other systems have implemented faster protocols. Amoeba claims the fastest RPC time in with a 1.4 msec null RPC on a Sun3 [Renesse88]. The x-kernel group reports a Sprite RPC implementation that makes a null RPC between Sun3 hosts in 1.73 msec [Hutchinson89]. The improvements by the x-kernel result from careful design of the protocol stacks, while I suspect that much of the performance of the Amoeba system stems from an assembly language implementation. The Sprite upcall mechanism has not been tuned at all, so it is about as expensive as exchanging data with UNIX pipes, or about 1 msec on a DS3100. This is the time to make a null ioctl on a pseudo-device.

10 User-Level Extensibility

The purpose of the Sprite upcall mechanism is to allow user processes to extend the kernel's

pathname and I/O interfaces. A user process can implement any semantics it chooses for a pseudo-device or a pseudo-file-system. Unlike simple message passing, the value of this approach is that general purpose features provided by the kernel, in particular network transparency, are inherited by the user-level server processes. For example, the X server lives under the pathname /hosts/*hostname*/X0. (The /hosts/*hostname* directories are just ordinary directories that are used for the few files needed on a per-host basis.) A process wishing to display a window on a particular host merely needs to open the corresponding pseudo-device. The kernel's RPC protocol is used to forward operations to the particular host. Similarly, NFS access is provided by a user-level pseudo-file-system that maps Sprite file system requests onto the NFS protocol. The server process runs on a single workstation, yet the NFS pseudo-file-system is transparently integrated into the distributed name space using the prefix table mechanism described earlier. Another feature that is inherited is blocking I/O. The server process can respond just like a device driver in order to cause the client process to block. As a result, the select system call can be used to wait on a set of devices and pseudo-devices that are located throughout the network.

Mach is similar in that the interface to a memory object can be exported to user-level [Young90]. In this case a user-level process responds to kernel requests to create and destroy memory objects, and to fetch and store pages. This facility allows a number of interesting applications, including network shared memory, compressed paging, and even remote file systems. In this case, features of the kernel-resident VM implementation are inherited by the external pager. This includes the general notion of memory objects, and more detailed features like copy-on-write.

Overall, the notion of an escape hatch to a user-level implementation is quite useful. A key difference between message passing among user processes and an upcall from the kernel is that with upcalls the kernel performs some processing on behalf of the user-level applications. The alternative in a pure message passing kernel is to put some amount of system software into runtime libraries. The fundamental difference, however, is that it is more difficult to share data structures among libraries, while the kernel has its own address space in which to maintain critical, shared data structures. Recall the differences between the kernel-resident prefix tables in Sprite vs. the per-process prefix tables in V and Mach.

Exporting a kernel interface via upcall is so useful that I have regretted the cases where it is not done. Notably, the cache consistency protocol is not exported via upcall, so caching data from pseudo-devices and pseudo-file-systems is not supported. As a result, the Sprite-to-NFS gateway provides absolutely no caching, and the Andrew benchmark runs twice as slow through the gateway as it does with a native Sprite file server. (In this case, the user-level UDP/IP server is also in the loop.) Also, as mentioned earlier, an upcall would be very useful in the case where the broadcast for a prefix fails. There is no fundamental reason why these features could not be implemented, it is just a small matter of programming.

The main drawback with Sprite's use of upcall is that ioctl is the only way to get at arbitrary functionality in the server process. Ioctl is perfectly general because it takes a command ID, and input buffer, and a reply buffer. While this is obviously clumsy, it has proved sufficient for user-level implementations of sockets and the TCP/IP protocols, terminal emulation, and the X display server. The ioctl model also assumes a request-reply pattern of interaction, while Mach ports can provide more general patterns of communication.

Configuration	Copy	Compile	Total	Penalty
DS3100 Sprite Local	22	98	120	
DS3100 Sprite Diskless	34	93	127	6%
DS3100 Mach 2.5 Local	29	107	136	
DS3100 Mach 2.5 NFS	58	147	205	50%
Sun4 Mach 2.5 Local	37	122	159	
Sun4 Sprite Local	44	128	172	
Sun4 Sprite Diskless	56	128	184	7%
Sun4 SunOS 4.0.3 Local	54	133	187	
Sun4 SunOS 4.0.3 NFS	92	213	305	63%
Sun3 Sprite Local	52	375	427	
Sun3 Sprite Diskless	75	364	439	3%

Table 1. Comparison of local and remote file system performance. The times are in seconds. The last column gives the percent slowdown of the benchmark when using a remote file system under the same OS

11 Sprite Performance.

Performance of Andrew file system benchmark, a benchmark that copies, stats, and compiles a large program, shows how well Sprite performs in the remote case. The numbers given in Table 1 were measured by Ousterhout in a series of measurements of UNIX systems [Ousterhout90]. This is a part of Table 7 from that paper.

The microkernel was unavailable to him at the time Ousterhout made his measurements. While a recent paper has reported that the microkernel with the single-server UNIX emulator can perform about as well as the Mach 2.5 kernel on the Andrew benchmark [Golub90], this makes no statement about the performance in the remote case, nor for performance with a multi-server configuration.

The primary reason for the performance advantage of the Sprite workstations is the differences in the file caching protocol. Sprite uses a delayed write strategy on both diskless clients and servers, while NFS writes data from a client through to the servers disk. The effectiveness of the Sprite caching system is presented in [Nelson88] and [Welch91]. As much as 50% of the data generated by Sprite clients is deleted before being written back to the server. This result is from long term (i.e., months long) measurements of clients that use the standard 30-second delay policy inherited from UNIX.

Recent work by Rosenblum has dramatically improved the performance of Sprite in the local case, as well. The log-structured file system aggressively optimizes writing performance, which is becoming the bottleneck as large main memory caches reduce the percentage of reads that go through to the disk [Rosenblum90][Rosenblum91].

12 The Cost of Complexity

The danger, of course, with providing a fancy distributed file system is in complexity. Consider the following sources of complexity.The cache consistency protocol relies on state maintained by the server, and this state has to be recovered after a server reboots. Users can abort operations with down servers, or they can wait for automatic recovery. During process migration the server's state has to be updated, and the semantics of shared UNIX I/O

streams (e.g., the shared seek offset) have to be maintained so that migration is transparent to the processes involved [Douglis90].Finally, the system supports a variety of "file system" objects, including devices, files, pipes, and user-level server processes.

The main cost of this complexity is in development time. As described earlier, it was about 5 years before Sprite was stable enough to be used by outsiders, although I began using Sprite for all my day-to-day work 2 years before that. Complexity doesn't necessarily imply larger programs. They do get larger as "small" features are added incrementally over time. However, major re-writes often reduce code size and simplify things. The file system benefited considerably from a re-write after initial experiences with process migration, crash recovery, and the upcall mechanism. Another of Butler Lampson's quotes is:

> "Plan to throw one away; you will anyhow." [Lampson83]

13 The Size of the Sprite Kernel

The sizes of the Sprite kernel modules are given in Appendix 1. Overall, the kernel contains about 95,000 lines of code, excluding comments, and it compiles into about 1 Megabyte on a DS3100. The largest modules are the file system (38% of lines of code), process manager (10%), which includes process migration, network and device drivers (12%), virtual memory (%8), and the RPC protocol (4%). The remaining third of the kernel is split among a miscellaneous group of modules that implement signals, synchronization primitives, the scheduler, a timer, a host monitor that triggers recovery, a debug nub to support remote debugging, malloc, free, printf, and support for profiling with the UNIX gprof program.

The file system is broken into a number of modules. The largest, **fs** (8%), contains an emulation library for 4.3 BSD system calls that used to be linked into applications via the C library. The parts of file system that directly manage disks is relatively small. **fsdm** (1%) has generic code to handle file descriptors and superblocks, while **lfs** (6%) and **ofs** (3%) implement particular block layouts. One layer higher, **fscache** (3%) maintains a cache of local and remote file blocks, while **fsconsist** (1%) implements the network consistency protocol. The **fslcl** (2%) module implements a directory hierarchy on a local disk, while **fsprefix** (1%) implements a transparently distributed name space. Local and remote implementations of various file system objects are implemented in **fsio** (3.5%) and **fsrmt** (3.5%), respectively. The **fspdev** (3%) module implements an upcall l facility. The **fsutil** (1%) module maintains the table of object descriptors (similar to vnodes) and contains other supporting routines.

Note that a Sprite kernel leaves out some things that are found in other UNIX kernels, notably protocol stacks and terminal emulation. The only network protocol in the Sprite kernel is the kernel-to-kernel RPC protocol. The TCP/IP protocols are implemented in a user-level process as a pseudo-device. The socket interface is implemented in a library that makes ioctl calls on the TCP/IP pseudo-device to setup and destroy network connections. The kernel-resident terminal driver is very crude, relying on a more sophisticated terminal emulator that runs as part of a window system. Thus Sprite takes a hybrid approach, putting performance critical features into the kernel, yet exporting the file system interface via upcall in order to allow for user-level extensibility.

14 The Size of the Mach Microkernel

Table 3 gives the sizes of the code in the directories that make up a microkernel for the DECstation 3100. For comparison, the compiled size of the emulator library and the single server are also given.The Mach microkernel is about 65% the size of a Sprite kernel in the number of non-comment code lines. This is approximately equal to leaving out the Sprite file system, although other modules in the Sprite kernel depend on the file system. The single process UNIX server is about the same size as the microkernel, although I suspect their may be some dead code in the UNIX server. I do not have access to a multi-server implementation, so I cannot comment on the size of that system. All-in-all, the code size comparison comes out a wash. By the time a file system is added back onto the microkernel, both systems are about the same size.

15 Other Comparisons

Ease of Development - A microkernel is hailed as providing an easier environment in which to develop system services. After all, they are ordinary user processes so they can be debugged in the normal ways. However, Sprite has always had a symbolic debugging facility for its kernel. The debugnub implements enough functionality to support the ptrace interface used by UNIX debuggers. It communicates with the debugger over a serial line, or over the ethernet. It is possible to set breakpoints and even single step the kernel. Debugging a multi-server environment might even be more difficult because system services are distributed into different address spaces. In this case, it isn't as easy to trace the execution of something like exec that involves the process manager, the file system, and the virtual memory system.

Virtual Memory- Mach provides an excellent internal interface to the machine-dependent MMU facilities. One of the most difficult aspects of porting an operating system is dealing with a new MMU. Much of Mach's success is due to the pmap interface and its support for multiprocessors. Sprite, too, has a decent internal interface between the machine-dependent and machine-independent parts of the kernel. It also runs on multiprocessors. However, the Mach VM interface reflects a long history of experience with VM in the Rig and Accent kernels.

Directory	Description	Procedures	Lines	Text	Bytes
boot_ufs	Bootstrap file system	30	3006		
chips	Common device code.	235	6411		
ddb	Kernel debugger	111	2836		
device	Device interface	91	3912		
inline	Compiler support	12	360		
ipc	Messages and ports	224	13097		
kern	Tasks and threads	387	11573		
scsi	Generic SCSI	113	4225		
vm	Virtual memory	137	9209		
pmax	DS3100-specific code	190	6016		
Total	A DS3100 Microkernel	1550	60645	473576	574880
Emulator	The emulator library/process	81	2421	65536	73728
Server	The monolithic UNIX server	1424	63816	503808	615600

Table 3. Sizes for a DECstation 3100 microkernel. The kernel modules are not linked separately, so I have not given compiled sizes for each module. The total size of the emulator and server are given for comparison

Real Time - The key to real time performance is a preemptible kernel, not necessarily the use of a microkernel. Early versions of Sprite had a scheduler that preempted kernel threads, although this feature was eliminated in order to simplify the scheduler. The Sprite kernel is multi-threaded internally, and with some effort the scheduler could revert to a preemptive one. Another important issue for real time is guaranteed I/O bandwidth. Again, a microkernel *per se* makes no guarantees in this regard. I suspect that the next version of SunOS will be readily converted to a real time kernel because it is multi-threaded internally and has a preemptive scheduler[Powell91]

Multiple Personalities - Mach has done a good job of extracting UNIX emulation in to a library and an associated server process. However, Sprite also emulates the BSD and Ultrix system call interfaces with a simple library. The bulk of UNIX compatibility is in the file system, which is already provided. Another approach to emulation is taken by the Chorus microkernel. Different system services are downloaded into the kernel depending on the needs of the applications [Guillemont91]. I prefer this approach to the Mach multi-server approach because I feel that kernel-resident modules will perform better. As for Mac and DOS emulation, I do not see a real need for this given the multi-tasking facilities available with Windows 3.0 and Mac's System 7. Only a small set of programmers will want to program UNIX-style on DOS or a Mac, while commercial developers and users will almost certainly prefer their native DOS and Mac environments.

16 What I Would Do Differently

If I were to write a new operating system today (which I am not), I would like to pick and choose from the excellent work done by various research groups. For example, the Mach pmap interface, the x-kernel protocol stacks, the Sprite distributed file system, and Bershad's LRPC would be an excellent starting point. The other crucial factor is support from the hardware vendors in the form of device drivers and bootstrap code. There are a number of difficulties with this approach, however. A practical matter is that each of these have been developed in their own environment, so integration would be difficult, even if it were possible to get the source code. The other, central problem is that the interfaces to these subsystems are not always well defined. Often the top-level interface is published, but the interfaces relied on by the implementations are not specified.

The next time around I would still not choose message passing. Instead, I would provide two communication mechanisms in the kernel, LRPC for local communication and network RPC for remote communication. I believe that the procedure call model is right abstraction to present to programmers. I would provide, in the kernel, a high-level distributed name space using a prefix table mechanism. I would retain the file system's I/O interface as the default interface, but I would also allow other interfaces to be associated with pathnames. Thus the kernel would still provide the default, high-performance file system. However, user processes could use the hierarchical name space for binding to arbitrary interfaces implemented by other, user-level processes. Finally, I would leverage more on dynamic loading so that only the needed kernel modules need to be loaded on any particular machine, and so that turn-around time in development would improve.

17 Conclusions

First, while it is always true that you can build things with general purpose, low-level facilities, it is also true that choosing a higher-level abstraction gives you more flexibility in the

implementation. A higher-level abstraction also does more for its clients, although it may restrict them in some ways. For Sprite we chose the file system interface as opposed to a message passing interface. As a result, we had the flexibility to implement simple, efficient mechanisms for its transparently distributed name space and its remote file access. The file system also provides an overall structure to the system that provides a benefit in uniformity and simplicity. The danger in choosing a high-level abstraction is that it may not be appropriate for every application. However, in the case of the file system I can safely assert that this abstraction is useful to a wide range of applications.

Second, while it is important for performance to provide kernel-level support, an upcall mechanism that integrates user-level processes into kernel abstractions is a good idea.The kernel can provide the basic framework for the system, such as a high-level name space or a copy-on-write VM facility, yet the user-level processes can extend the system in new ways without modifying the kernel. This is different than pure message passing where the kernel makes no statements about the overall structure of the system. By now, I think we know enough to provide some higher-level functionality in the kernel.

18 Acknowledgments

I would like to that all the helpful feedback from the reviewers, especially Dave Nichols. I would like to thank the program committee and Alan Langerman for giving me the opportunity to present these ideas to this audience.

19 References

Accetta86. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young, Mach: A New Kernel Foundation for UNIX Development. *Proc. of the Summer 1986 USENIX Conference*, July 1986.

Baker91. M. Baker, J. Hartman, M. Kupfer, K. Shirriff and J. Ousterhout, Measurements of a Distributed File System, *(to appear) Proc. of the 13th Symp. on Operating System Prin., Operating Systems Review*, Oct. 1991.

Bershad91. B. N. Bershad, T. E. Anderson, E. D. Lazowska and H. M. Levy, User-Level Interprocess Communication for Shared Memory Multiprocessors, *ACM Transactions on Computer Systems*, 9, 2 (May 1991), 175-198.

Birrell84. A. D. Birrell and B. J. Nelson, Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems* 2, 1 (Feb. 1984), 39-59.

Clark85. D. Clark, The Structuring of Systems Using Upcalls, *Proc. of the 10th Symp. on Operating System Prin., Operating Systems Review* 19, 5 (Dec. 1985), 171-180.

Douglis90. F. Douglis, Transparent Process Migration for Personal Workstations, PhD Thesis, Sep. 1990. University of California, Berkeley.

Draves90. R. Draves, A Revised IPC Interface, *Proc. of the Mach Workshop*, Oct. 1990, 101-121

Feirtag71. R. Feirtag, E. Organick, The Multics Input/Output System, *Proc. of the 3rd SOSP*, 1971, 35-41.

Golub90. D. Golub, R. Dean, Forin and R. Rashid, UNIX as an Application Program, *Proc. of the Summer 1990 USENIX Conference*, June 1990, 87-96.

Guillemont91. M. Guillemont, J. Lipkis, D. Orr and M. Rozier, A Second-Generation Micro-Kernel Based UNIX; Lessons in Performance and Compatibility, *Proc. of the Winter 1991 USENIX Conference*, Jan. 1991, 13-22.

Hill86. M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A.

Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, Design Decisions in SPUR, *IEEE Computer* 19, 11 (November 1986).

Hutchinson89. N. C. Hutchinson, L. L. Peterson, M. B. Abbott and S. O'Malley, RPC in the x-Kernel: Evaluating New Design Techniques, *Proc. 12th Symp. on Operating System Prin., Operating Systems Review* 23, 5 (December 1989), 91-101.

Barrara91. J. S. Barrera. III, A Fast Mach Network IPC Implementation, *(to appear) Proc. of the 2nd Mach Symposium*, Nov. 1991.

Lampson83. B. Lampson, Hints for Computer System Design, *Proc. 9th Symp. on Operating System Prin., Operating Systems Review* 17, 5 (October 1983), 33-48.

Lauer81. H. C. Lauer, Observations on the Development of an Operating System, *Proc. of the 8th Symp. on Operating System Prin., Operating Systems Review* 15, 5 (Dec. 1981), 30-36.

Li89 .K. Li and P. Hudak, Memory Coherence in Shared Virtual Memory Systems, *ACM Transactions on Computer Systems* 7, 4 (Nov. 1989), 321-359.

Nelson88. M. Nelson, B. Welch and J. Ousterhout, Caching in the Sprite Network File System, *ACM Transactions Computer Systems* 6, 1 (Feb. 1988), 134-154.

Ousterhout90. J. Ousterhout, Why Aren't Operating Systems Getting Faster As Fast as Hardware?, *Proc. of the Summer 1990 USENIX Conference.*, June 1990, 247-256.

Powell91. M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein and M. Weeks, SunOS Multi-thread Architecture, *Proc. of the Winter 1991 USENIX Conference*, Jan. 1991, 65-80.

Renesse88. R. Renesse, H. Staveren and A. W. Tanenbaum, Performance of the World's Fastest Distributed Operating System, *Operating Systems Review* 22, 4 (Oct. 1988), 25- 34.

Rosenblum90. M. Rosenblum and J. K. Ousterhout, The LFS Storage Manager, *Proc. of the Summer 1990 USENIX Conference*, June 1990, 247-256.

Rosenblum91. M. Rosenblum and J. K. Ousterhout, The Design and Implementation of a Log-Structured File System, *(to appear) Proc. of the 13th Symp. on Operating System Prin.*, Operating Systems Review, Oct. 1991.

Welch86a. B. B. Welch, The Sprite Remote Procedure Call System, Technical Report UCB/Computer Science Dpt. 86/302, University of California, Berkeley, June 1986.

Welch86b. B. B. Welch and J. K. Ousterhout, Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem, *Proc. of the 6th ICDCS*, May 1986, 184-189.

Welch88. B. B. Welch and J. K. Ousterhout, Pseudo-Devices: User-Level Extensions to the Sprite File System, *Proc. of the 1988 Summer USENIX Conf.*, June 1988, 184-189.

Welch90. B. B. Welch, Naming, State Management, and User-Level Extensions in the Sprite Distributed File System, PhD Thesis, 1990. University of California, Berkeley.

Welch91 .B. B. Welch, Measured Performance of Caching in the Sprite Network File System, *(to appear late '91) Computing Systems.*

Young87. M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. CHew, W. Bolosky, D. Black and R. Baron, The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System, *Proc. 11th Symp. on Operating System Prin., Operating Systems Review* 21, 5 (Nov. 1987), 63-76.

Module	Description	Procedures	Lines	Text	Bytes
dbg.ds3100	Debug Nub	27	1509	13680	24672
dev	Devices	117	4159	83296	127472
dev.ds3100	DS3100 drivers	67	5350		
fs	FS syscalls & BSD compat.	158	8018	76784	97296
fscache	Cache manager	70	2747	24032	27184
fsconsist	Consistency protocol	35	1338	11584	14080
fsdm	Local disk manager	8	889	5904	7344
fsio	Local I/O objects	100	3490	26352	31760
fslcl	Local directories	43	2086	16896	19104
fspdev	Upcall to user-level	68	3155	24784	28176
fsprefix	Distributed naming	29	1200	10704	11824
fsrmt	Remote I/O objects	91	3644	28816	32336
fsutil	Table management	62	1547	13792	16464
lfs	Log Structured FS	132	5394	46096	52064
libc	Printf, etc.	80	5133	39104	57056
libc.ds3100	ditto	1	99		
mach.ds3100	Trap handlers & Ultirx compat.	135	5595	63840	80384
main.ds3100	System start-up	3	294	2480	3648
mem	Malloc/free	17	841	6128	22544
net	Network interface	42	1718	37760	45408
net.ds3100	Network devices	18	875		
ofs	Old disk layout	51	2724	23360	2744
proc	Process manager	225	9747	77776	96000
proc.ds3100	ditto	1	52		
prof	Kernel profiling	8	198	2464	3904
prof.ds3100	mtrace, etc.	7	288		
raid	Disk arrays	116	3230	32 *	128 *
recov	Network recovery	40	1253	10448	12144
rpc	Remote Procedure Call	79	3709	38704	50928
rpc.ds3100	ditto	1	8		
sched	Scheduler	24	814	9440	11888
sig	Signals	31	1189	9200	1097
sync	Monitors, condition vars	42	1493	16152	17348
sys	Syscalls and miscellany	38	1847	9312	17184
timer	Timer and callout	17	463	8560	15168
timer.ds3100	Timer device	11	247		
utils	Hash and Trace	17	710	6656	9200
utils.ds3100	ditto	3	39		
vm	Virtual Memory	198	6565	77328	103200
vm.ds3100	ditto	49	1068		
TOTAL	A DS3100 Sprite kernel	2261	94725	828576	1085632

Appendix 1. Module sizes for the Sprite kernel. The count of lines of code exclude comment blocks ar cpp directives. The Text and Bytes columns give the text size and total size of the compiled modules as reported by the UNIX size command. The compiled sizes are for the DECstation 3100. Machine-depende modules are named with a .ds3100 suffix. They are linked together with corresponding machine-indepe dent modules so separate compiled sizes are not given. Compiled sizes for the Sun3 are roughly 75% th size for the other, RISC-based architectures. The presence of compatibility code in both fs and mach.ds31 is because the compatibility code is being rewritten. There is considerable overlap between the two.

* raid is only compiled for the Sun4 architecture. The compiled sizes are for stubs.