The Sprite Network Operating System

John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch
University of California at Berkeley

Sprite is an experimental network operating system under development at the University of California at Berkeley. It is part of a larger research project called SPUR, whose goal is the design and construction of a high-performance multiprocessor workstation with special hardware support for Lisp applications.[1] One of Sprite's primary goals is to support applications running on SPUR workstations, but we hope that the system will also work well for a variety of high-performance engineering workstations. Currently, Sprite is being used on Sun-2 and Sun-3 workstations.

Driving forces. The motivation for building a new operating system came from three general trends in computer technology: networks, large memories, and multiprocessors.

In an increasing number of research and engineering organizations, computing now occurs on personal workstations connected by local-area networks. Larger, time-shared machines are used only for those applications that cannot achieve acceptable performance on workstations. Unfortunately, workstation environments tend to suffer from poor performance and difficulties of sharing and administration, due to the distributed nature of the systems. In Sprite, we hope to hide the distribution as much as possible, while providing the sharing and communication of time-shared machines.

Sprite implements a set of kernel calls that provide sharing, flexibility, and high performance to networked workstations.

The second technology trend driving the Sprite design is the availability of ever-larger physical memories. Today's engineering workstations typically contain four to 32 megabytes of physical memory, and we expect memories of 100 to 500 megabytes to be commonplace within a few years. We believe that such large memories will change the traditional balance between computation and input/output by permitting all of a user's commonly accessed files to reside in main memory. The "RAMdisks" available on many commercial personal computers have already shown this capability on a small scale. One of our goals for Sprite is to manage physical memory in a way that maximizes the potential for file caching.

The third driving force behind Sprite is the imminent arrival of multiprocessor workstations. Workstations with more than one processor are currently under development in several research organizations (UCB's SPUR, Digital Equipment Corporation's Firefly, and Xerox's Dragon are a few prominent examples), and we expect multiprocessor workstations to be available from several major manufacturers within a few years. We hope that Sprite will facilitate the development of multiprocessor applications, and that the operating system itself will be able to take advantage of multiple processors in providing system services.

0018-9162/88/0200-0023$01.00 © 1988 IEEE

Design goals. Our overall goal for Sprite is to provide simple, efficient mechanisms that capitalize on the three technology trends affecting the system's design. In areas where technology factors did not suggest special techniques, we modeled the system as closely as possible after Berkeley Unix.

The technology trends had only a minor impact on the facilities Sprite provides to application programs. For the most part, Sprite's kernel calls are similar to those provided by the 4.3 BSD version of the Unix operating system. However, we added three facilities to encourage resource sharing: a transparent network file system, a simple mechanism for sharing writable memory between processes on a single workstation, and a mechanism for migrating processes between workstations to take advantage of idle machines.

Although the technology trends did not have a large effect on Sprite's kernel interface, they did suggest dramatic changes in the kernel implementation, relative to Unix. This is not surprising, since networks, large memories, and multiprocessors were not important issues in the early 1970s when the Unix kernel was designed. We developed the Sprite kernel from scratch, rather than modifying an existing Unix kernel. Some interesting features of the kernel implementation are

- The kernel contains a remote procedure call (RPC) facility that allows each workstation's kernel to invoke operations on other workstations. The RPC mechanism is used extensively in Sprite to implement other features, such as the network file system and process migration.
- Although the Sprite file system is implemented as a collection of *domains* on different server machines, it appears to users as a single hierarchy shared by all workstations. Sprite uses a simple mechanism called *prefix tables* to manage the name space; these dynamic structures facilitate system administration and reconfiguration.
- To achieve high performance in the file system, and also to capitalize on large physical memories, Sprite caches file data on both server and client machines. A simple cache consistency mechanism guarantees that applications running on different workstations always use the most up-to-date versions of files, in exactly the same fashion as if the applications were executing on a single machine.
- The virtual memory system uses ordinary files for backing storage; this simplifies implementation, facilitates process migration, and may even improve performance relative to schemes based on a special-purpose swap area. Sprite retains the code segments for programs in main memory, even after the programs are complete, to allow quick start-up when programs are reused. Finally, the virtual memory system negotiates with the file system over physical memory use, permitting the file cache to be as large as possible without degrading virtual memory performance.
- Sprite guarantees that processes behave the same whether migrated or not. This is achieved by designating a *home* machine for each process and forwarding location-dependent kernel calls to the process' home machine.

Application interface

Sprite's application interface contains little that is new. Kernel calls are very similar to those provided by the Berkeley versions of Unix. Indeed, we have ported many traditional Unix applications to Sprite with relatively little effort.

Three unusual aspects of the application interface can be summed up in one word: sharing. First, the Sprite file system allows sharing of all disk storage and I/O devices in the network by all processes, so they need not worry about machine boundaries. Second, the virtual memory mechanism allows sharing of physical memory between processes on the same workstation, so they can extract the highest possible performance from multiprocessors. Third, Sprite implements process migration, which allows job offloading to idle workstations and, thereby, sharing of processing power.

File system. Almost all modern network file systems, including Sprite's, have the same ultimate goal: *network transparency*. Network transparency means that users should be able to manipulate files in the same ways they did under time-sharing on a single machine; the distributed nature of the file system and the techniques used to access remote files should be invisible to users under normal conditions. MIT's Locus system was one of the first to make transparency an explicit goal[2]; other file systems with varying degrees of transparency are Carnegie Mellon's Andrew[3] and Sun's NFS.[4]

Most network file systems fail to meet the transparency goal in one or more ways. The earliest systems (and even some later systems, such as 4.2 BSD) allowed remote file access only with a few special programs (for example, rcp in 4.2 BSD); most application programs could only access files stored on local disks. Second-generation systems, such as Apollo's Aegis,[5] allow any application to access files on any machine in the network, but special names must be used for remote files (for example, "file" for a local file, but "[ivy]file" for a file stored on the Ivy server). Third-generation network file systems, such as Locus, Andrew, NFS, and Sprite, provide *name transparency*—that is, file location is not indicated directly by name, and groups of files can be moved from one machine to another without changing their names.

Most third-generation systems still have some nontransparent aspects. For example, in Andrew and NFS only a portion of the file system hierarchy is shared; each machine must also have a private partition that is accessible only to that machine. In addition, Andrew and NFS do not permit applications running on one machine to access I/O devices on other machines. Locus appears to be alone among current systems in providing complete file transparency.

Sprite, like Locus, provides complete transparency, so applications running on different workstations see the same behavior they would see if all the applications were executing on a single time-shared machine. A single file hierarchy is uniformly accessible to all workstations. Although it is possible to determine where a file is stored, that information is not needed in normal operation. There are no special programs for operating on remote files, as opposed to local ones, and no operations that can be used only on local files. Sprite also provides transparent access to remote I/O devices. Like Unix, Sprite represents devices as special files; unlike most versions of Unix, Sprite allows any process to access any device, regardless of device location.

Shared address spaces. The early versions of Unix did not permit memory sharing between user processes, except for read-only code. Each process had private data and stack segments, as shown in Figure 1. Since then, extensions to allow read-write memory sharing have been implemented or proposed for several versions of Unix, including System V, SunOS, Berkeley Unix, and Mach.

There are two reasons for providing shared memory. First, using a collection of processes in a shared address space is the most natural way to program many applications. It is particularly convenient when an application consists of mostly independent subactivities (for example, one process to respond to keystrokes and another to respond to packets arriving over a network); the shared address space allows them to cooperate to achieve a common goal (for example, managing a collection of windows on a screen). The second motivation for shared memory is the advent of multiprocessors. Decomposing an application into pieces that can be executed concurrently requires rapid communication between pieces. The faster the communication, the greater the degree of concurrency that can be achieved. Shared memory provides the fastest possible communication, hence the greatest opportunity for concurrent execution.

Sprite provides a particularly simple form of memory sharing; when a process invokes the Proc_Fork kernel call to create a new process, it may request that the new process share the parent's data segment (see Figure 2). The stack segment is still private to each process; it contains procedure invocation records and private process data. For simplicity, Sprite's mechanism provides all-or-nothing sharing; a process cannot share part of its data segment with one process and part of it with another.

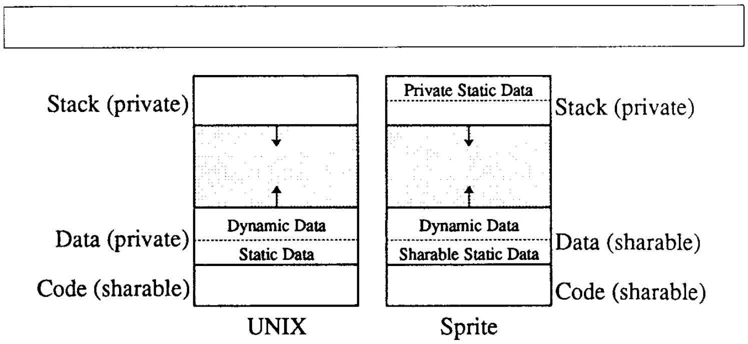


Figure 1. The organization of virtual memory as seen by user processes in traditional Unix (left) and Sprite (right). In both systems there are three distinct segments. The lower portion of the data segment contains static data known at compile time, and the upper portion expands to accommodate dynamically allocated data. In Unix, processes may share code, but not data or stack. In Sprite, the data segment may be shared between processes, including both statically allocated and dynamic data. Private static data may be stored at the top of the stack segment.

We expect multiprocess applications to synchronize using hardware mutual-exclusion instructions (for example, test-and-set) directly on shared memory. In most cases it will not be necessary to invoke the kernel, so synchronization can be accomplished in just a few instructions. The kernel participates only when it is necessary to delay process execution (for example, to wait for a lock to be released). For these situations, Sprite provides kernel calls that put a process to sleep and wake it up later. This permits efficient implementation of synchronization primitives.

Process migration. In an environment that has a workstation for each person, many machines will be idle at any given time. To allow users to harness this idle computing power, Sprite provides a new kernel call, Proc_Migrate, that will move a process or group of processes to an idle

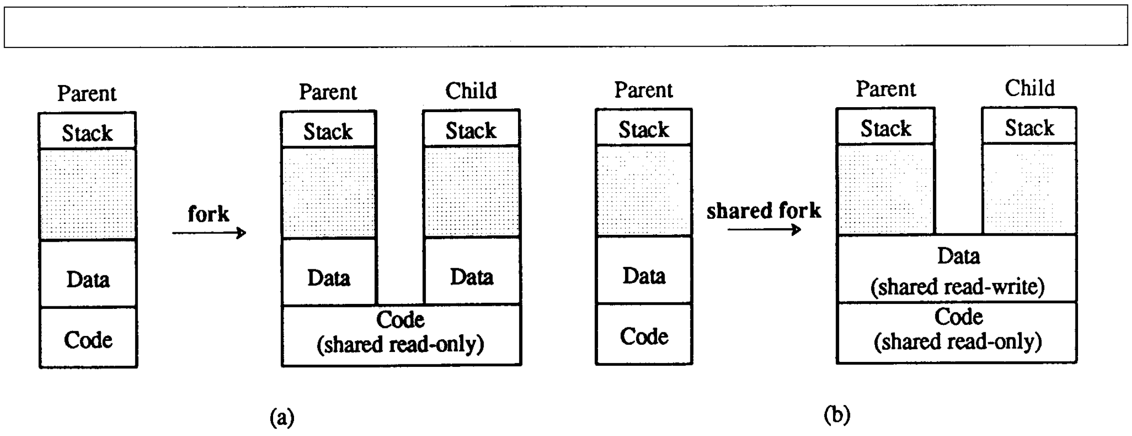


Figure 2. The Unix fork operation (a) creates a new process that shares code with its parent while using private copies of the data and stack segments. Sprite provides both the traditional fork and a shared fork (b) in which the child shares its parent's data as well as code.

machine. Processes sharing a heap segment must migrate together. Sprite keeps track of which machines are idle and selects one as the target for the migration. The fact that a process has migrated is transparent both to the migrated process and to the user, as described below. The only noticeable difference after migration will be a reduction in the home machine's load.

Initially, we expect migration to be used in two ways. First, shell commands for manual migration will allow users to migrate processes in much the same way the Unix shell allows users to place processes in the background. Second, a new version of the Unix Make utility, called Pmake, recompiles programs when their source files change. Make invokes recompilations sequentially, but Pmake is organized to invoke multiple recompilations concurrently, using process migration to offload the compilations to idle machines. We hope to see more and more automatic uses of migration, like Pmake, in the future.

The idea of moving work to idle machines is not a new one. Unfortunately, the most widely available facilities (for example, the rsh command of 4.2 BSD Unix and the rex facility of Sun's Unix) provide only *remote invocation*, which is the ability to initiate new processes on other machines, but not the ability to move processes once they have started execution. Process migration, which allows processes to be moved at any time, has been implemented in several systems (for example, Locus,[2] V,[6] and Accent[7]) but is not widely available. For Sprite, we decided to implement process migration. We think the additional flexibility migration provides is particularly important in a workstation environment. For example, if remote invocation is used to offload work onto an idle machine and then the machine's user returns, either the foreign processes have to be killed or the machine's user receives a degraded response until the foreign processes are complete. In Sprite, the foreign processes can be migrated away.

One of the most important attributes of Sprite's migration mechanism is its transparency, both to the process and to the user. When migrated, a process will produce exactly the same results as if it were not migrated; Sprite preserves the environment of the process as it migrates, including files, working directory, device access, environment variables, and anything else that could affect process execution. In addition, a migrated process appears still to be running on the user's home machine; it will appear in listings of processes on that machine and can be stopped, killed, or debugged just like the user's other processes. In contrast, rsh does not preserve the working directory and other aspects of the environment, and neither rsh nor rex allows a remotely executing process to be examined or manipulated in the same fashion as local processes. Other implementations of process migration tend not to provide complete transparency to users, although they do provide complete transparency to the migrated processes. (How Sprite achieves execution transparency is described in a later section.)

Basic kernel structure

Application programs invoke kernel functions via a collection of kernel calls. Sprite's basic flow of control in a kernel call is similar to that in Unix: user processes execute "trap" instructions to switch to the supervisor state, and the kernel executes as a privileged extension of the user process, using a small per-process kernel stack for procedure invocation within the kernel.

Two features of Sprite's basic kernel structure support multiprocessor and network operation. First, a multithreaded synchronization structure allows the Sprite kernel to run efficiently on multiprocessors. Second, a remote procedure call facility allows kernels to invoke operations remotely over the network.

Multithreading. Many operating system kernels, including Unix, are *single-threaded*, which means that a single lock is acquired when a process calls the kernel and released when the process puts itself to sleep or returns to user state. In these systems, processes are never preempted while executing kernel code, except by interrupt routines. The single-threaded approach simplifies kernel implementation by eliminating many potential synchronization problems between processes. Unfortunately, it does not adapt well to a multiprocessor environment. With more than a few processors, contention for the single kernel lock will limit system performance.

In contrast, the Sprite kernel is *multithreaded*, which means that several processes may execute in the kernel at the same time. The kernel is organized in a monitor-like style with many small locks, instead of a single overall lock, protecting individual modules or data structures. Many processes may execute in the kernel simultaneously as long as they do not attempt to access the same monitored code or data. The multithreaded approach allows Sprite to run more efficiently on multiprocessors, but the multiplicity of locks makes the kernel more complex and slightly less efficient since many locks may have to be acquired and released over the lifetime of each kernel call.

Remote procedure calls. In designing Sprite for a network of workstations, one of our most important goals was to provide a simple, efficient way for the kernels of different workstations to invoke each others' services. The mechanism we chose is a kernel-to-kernel RPC facility similar to the one described by Birrell and Nelson.[8] We chose RPC rather than a message style because RPC provides a simple programming model (remote operations appear just like local procedure calls) and because the RPC approach is particularly efficient for request-response transactions, which we expected to be the most common form of interaction between kernels.

The RPC implementation consists of *stubs* and *RPC transport*, as shown in Figure 3. Together they hide the fact that the calling procedure and the called procedure are on different machines. Each remote call has two stubs, one on the client workstation and one on the server. The client stub copies its arguments into a request message and returns values from a result message, so the calling procedure is not aware of the underlying message communication. The server stub passes arguments from the incoming message to the desired procedure and packages results from the procedure, so the called procedure is not aware that its real caller is on a different machine. Birrell and Nelson modified their compiler to generate the stubs automatically from a specification of procedure interfaces. To avoid changing our C compiler, we hand-generated the stubs for the 40 or so remote operations used in the Sprite kernel. Although this was workable, it would have been more convenient if an automated stub-generator had been available.

The second part of the RPC implementation is RPC transport. It delivers messages across the network and assigns incoming requests to kernel processes that execute the server stubs and called proce-

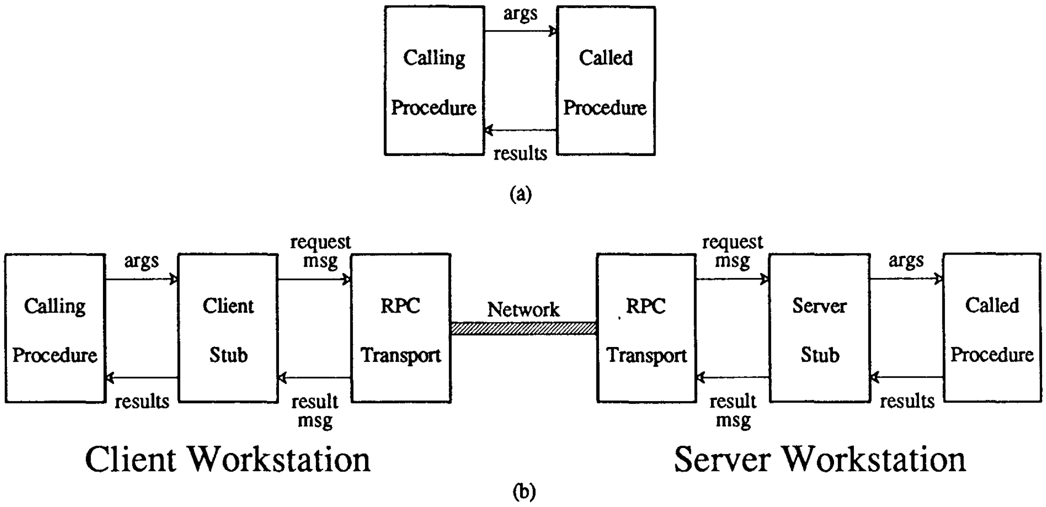


Figure 3. Sprite's remote procedure call mechanism makes it appear as if a remote procedure can be invoked directly (a). The actual situation (b) is that stub procedures copy procedure arguments and results into and out of messages, and a transport mechanism delivers the messages reliably and assigns server processes to requests.

dures. The goal of RPC transport is to provide the most efficient possible communication between the stubs while ensuring that messages are delivered reliably. Sprite's RPC transport uses two techniques to gain efficiency: implicit acknowledgments and fragmentation.

Since network transmission is not perfectly reliable, each request and response message must be acknowledged; if no acknowledgment is received within a reasonable time, the sender retransmits. To reduce the overhead associated with processing acknowledgment packets, Sprite uses the scheme described by Birrell and Nelson, where each request or response message serves as an implicit acknowledgment for the previous response or request message from that client, respectively. In the common case of short, closely spaced operations, only two packets are transmitted for each remote call: one for the request and one for the response.

The simplest way to implement RPC is to limit the total size of the arguments or results for any given RPC so that each request and response message can fit into a single network packet. Unfortunately, the maximum allowable size for a network packet is relatively small (about 1500 bytes for Ethernet), so this approach would result in high overhead for bulk transfers. The delays associated with sending a request, dispatching to a server process, and returning a response would be incurred for each 1500 bytes. Since remote file access is one of RPC's most common uses, we were unwilling to accept this performance limitation.

Sprite's RPC mechanism differs from the Birrell-Nelson scheme in that it uses fragmentation to ship large blocks of data (up to 16 kilobytes) in a single remote operation. If a request or reply message is too long to fit in a single packet, RPC transport breaks the message into multiple packets (fragments), which it transmits in order without waiting for acknowledgment. The receiving RPC transport reassembles the fragments into a single large message. A single acknowledgment for all the fragments uses the implicit acknowledgment scheme described above. When packets are lost in transmission, the acknowledgment indicates which fragments have been received so that only lost fragments are retransmitted.

Sprite kernels trust each other, and we assume that the network wire is physically secure (all workstations on the network must run the Sprite kernel or some other trustworthy software). Thus, the RPC mechanism does not use encryption, nor do the kernels validate RPC operations except to prevent user errors and detect system bugs. The RPC mechanism is used only by the kernels and is not directly visible to user applications.

Figure 4 shows the measured performance of the Sprite RPC mechanism. Figure 4a shows that the minimum round-trip time for the simplest possible RPC is about 2.8 milliseconds between Sun-3/75 workstations, with an additional 1.2 milliseconds for each kilobyte of data. Figure 4b shows that throughputs greater than 700 kilobytes per second (nearly 60 percent of the total Ethernet bandwidth of 10 megabits per second) can be achieved between two workstations if each RPC transfers a large amount of data. Without fragmentation (at most 1500 bytes transmitted per RPC) the throughput is reduced by more than a factor of two. The measurements in Figure 4 are for operations between kernels. User-visible performance is slightly worse; for example, a user process can achieve a throughput of only 475 kilobytes per second when it reads a file

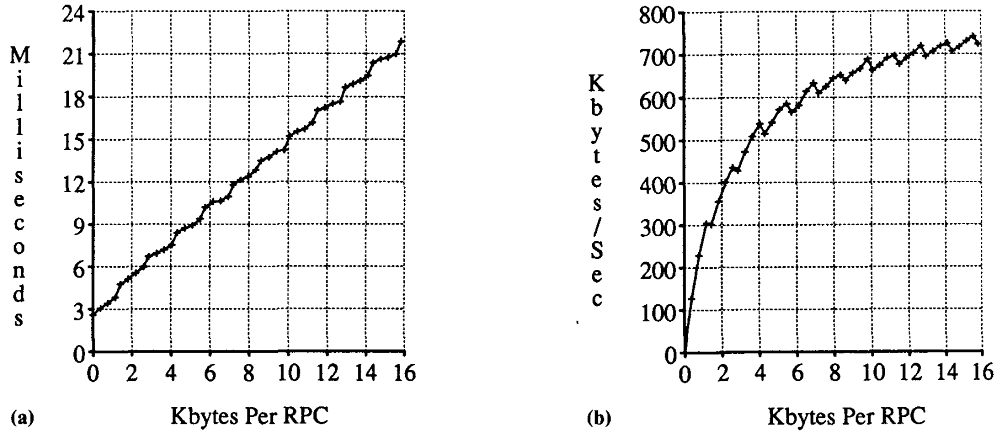


Figure 4. Measured performance of Sprite's remote procedure call mechanism between Sun-3/75 workstations. The test consisted of one kernel invoking a remote procedure in another kernel, passing to it the contents of a variable-size array as an argument. The called procedure returned immediately. Part (a) shows the round-trip time for an individual RPC as a function of the amount of data passed to the remote procedure; (b) shows the throughput when repeated RPCs are made. Larger transfers, which use fragments on 1500-byte boundaries, are most efficient. The jumps in the curves occur at the points where additional packets become necessary.

that is cached in the main memory of a remote server and the kernel makes four-kilobyte RPC requests.

Managing the file name space—prefix tables

In designing the Sprite file system for a network environment, we were particularly concerned about two implementation issues: how to manage the file name space in a way that simplifies system administration, and how to manage the file data in a way that provides high performance. Furthermore, we felt that it was important to provide easy administration and high performance without compromising users' ability to share files.

To users, the Sprite file system is a single hierarchy, just as in time-shared Unix. To system administrators, the file system is a collection of *domains*, which are similar to file systems in Unix. Each domain contains a tree-structured portion of the overall hierarchy. The domains are joined into a single hierarchy by overlaying the leaves of some domains with the roots of other domains as illustrated in Figure 5. (In Unix terms, the subdomains are *mounted* on their parents; the leaves where mounting occurs, such as "/a" in Figure 5, are called *mount points*.) As the operating system traverses the components of a file name during name lookup, it must move automatically from domain to domain to keep the domain boundaries from being visible to users.

The interesting naming issues are how to keep track of the domain structure and how to handle file names that cross domain boundaries. These issues are particularly interesting in a network environment where the domains may be stored on different servers and where the server configuration may change frequently. Unix and most of its derivatives (such as NFS) use static *mount tables* to keep track of domains; the mount tables are established by reading a local configuration file at boot-time. This makes it difficult for the systems to respond to configuration changes. In our NFS clusters, for example, any change to the domain structure typically requires each user to modify the configuration file on their workstation and reboot. Even in small clusters we have found that such changes occur distressingly often.

In Sprite, we use a more dynamic approach to managing the domain structure, which we call *prefix tables*. Each client machine's kernel maintains a private prefix table. Each entry in a prefix table corresponds to a domain; it gives the full name of the top-level directory in the domain (that is, the common prefix shared by the names of all files in the domain), the name of the server on which that domain is stored, and an additional token to pass to the server to identify the domain (see Table 1). Prefix tables are not normally visible to user processes.

Locating a file. In Sprite, as in Unix, application programs refer to files by giving either an *absolute path name* for the file (one starting at the file system root, such as "/d/k/p/r" in Figure 5) or a *relative path name*, which is interpreted as starting at a previously specified working directory (if the working directory is "/d/k/p" in Figure 5, then the relative name "r" refers to the same file as "/d/k/p/r"). To look up an absolute path name, a client kernel matches the name against all entries in its prefix table and chooses the entry with the longest matching prefix. In the example of Figure 5, the file name "/d/k/p/r" will match three entries in the table, of which the entry for server Z has the longest prefix. The client strips the prefix from the file name and uses the RPC facility to send the remainder of the name ("p/r") to the server, along with the token from the prefix table entry (5). The server uses the token to locate the root directory of the domain, looks up the remainder of the file name, and replies with a token identifying the file. The client can then issue read, write, and close requests by making RPCs to the server with the file's token.

Sprite handles working directories by opening the working directory and storing its token and server address as part of the process' state. When a file name is specified relative to the working directory, the client kernel uses the token and server address corresponding to the working directory rather than those from a prefix table entry. Thus, absolute and relative path name lookups appear identical to the server.

There are several cases where the initial server that receives a file name cannot completely process the name. These correspond to situations where the file's name crosses a domain boundary. For example, ".." components in a name (which refer to the parent directory) could cause it to ascend back up the hierarchy and out the top of the domain; or the name could refer to a symbolic link containing an absolute file name for a different domain; or a relative path name could start at the current working directory and descend into a new domain. In each of these cases, the initial server processes as many components of the file name as it can, then returns a new name to the client instead of a file token. The client takes the new name, processes it with its prefix table, and sends it to a new server. This process repeats until the name is completely resolved (see Welch and Ousterhout[9] for details).

The prefix approach bypasses the root domain (and its server) when looking up absolute names of files in nonroot domains. Since a large fraction of all name lookups involves absolute path names, we expect this approach to reduce the load on the root server and increase the scalability of the system relative to schemes that require root server participation for every absolute path name. It may also let the system provide limited service even when the root server is down.

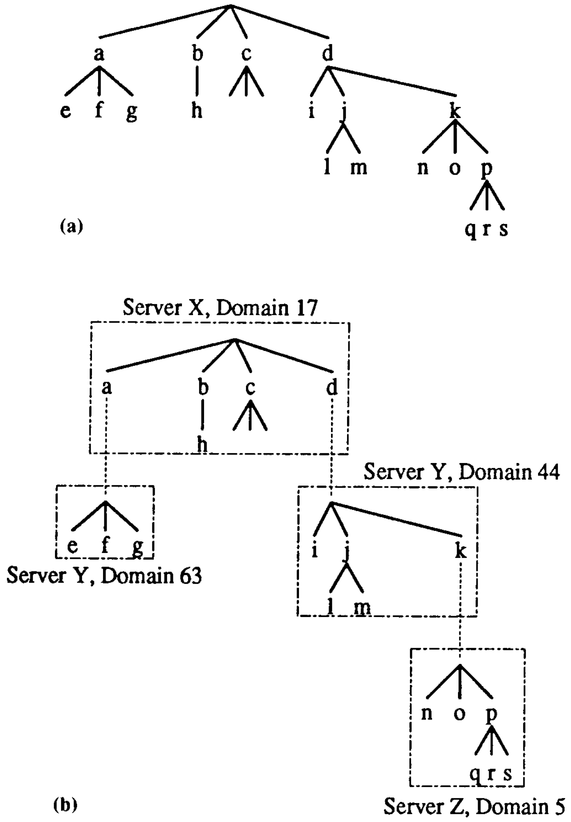


Figure 5. Although the Sprite file system behaves as if it were a single hierarchy (a), it is actually divided up into domains (b). Each domain may be stored on a different server.

Managing prefix tables. One of the greatest advantages of prefix tables is that they are created dynamically and updated automatically when the system configuration changes. To add a new entry to its prefix table, a client broadcasts a prefix name to all servers. The server storing the domain replies with its address and the token corresponding to the domain. The client uses this information to create a new prefix table entry. Initially, each client starts out with an empty prefix table and broadcasts to find the entry for "/." As it uses more files, it gradually adds entries to its prefix table.

How does a client know when to add a new prefix to its table? The file at the mount point for each domain is a special link, called a *remote link*, which identifies the file as the mount point for a new domain. For example, in Figure 5 the file "/d/k" in server Y's domain is a remote

Table 1. A prefix table corresponding to the domain structure of Figure 5.*

Prefix	Server	Token
/	X	17
/a/	Y	63
/d/	Y	44
/d/k/	Z	5

*Prefix tables are loaded dynamically, so they need not hold complete file information at any given time.

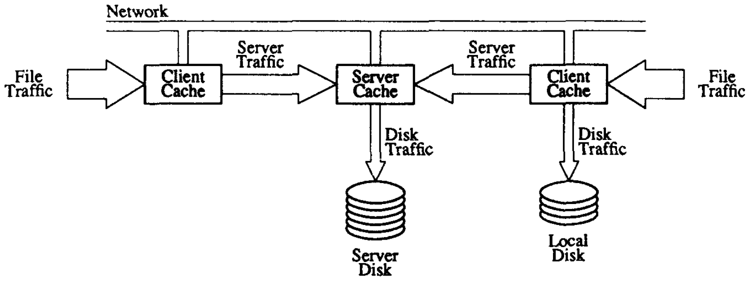


Figure 6. Caches in the Sprite file system. When a process makes a file access, it is presented first to the cache of the process' workstation (file traffic). If not satisfied there, the request is passed either to a local disk, if the file is stored locally (disk traffic), or to the server where the file is stored (server traffic). Servers also maintain caches to reduce their disk traffic.

link. A remote link is similar to a symbolic link in that it stores a file name; for remote links, this is the prefix name (that is, the file's absolute name). Whenever a remote link is encountered in file name lookup, the server returns to the client the prefix name and the remainder of the name being looked up. The client uses the broadcast protocol to make a new prefix table entry and then reprocesses the remainder of the name. Remote links do not store any network address information; they simply indicate the presence of a domain. This feature permits the system to adapt quickly to changes in configuration.

Prefix table entries are treated as hints and are adjusted automatically as the system configuration changes. When a client sends an open request to a server, it is possible for the request to fail with a timeout (if the server has crashed) or a rejection (if the server no longer stores the domain). In either case, the client invalidates the prefix table entry for the domain and rebroadcasts. If the domain has moved, the new server will respond to the rebroadcast, and the client will establish a new prefix table entry and retry the open. In this case, the configuration change will be invisible to user processes. If the server has crashed, then the broadcast will timeout; each additional open will also broadcast and timeout. During the time the server is down, user processes will receive errors analogous to disk-off-line errors in time-shared Unix. Eventually, the domain will become available again, and the next open will reestablish the prefix table entry. Adding a new domain to the file system requires only adding a remote link at the mount point for the domain and arranging for the server to respond to requests.

Managing file data—client and server caches

The Sprite file system is implemented using large caches of recently used file blocks stored in the main memories of both clients and servers. The caches provide two benefits that are especially important when most of the workstations are diskless. First, the caches improve file system performance by eliminating disk accesses and network transactions. Second, they reduce the loading on the network and the servers, which increases the scalability of the system. Sprite's caches use a consistency protocol that allows applications on different workstations to share files just as if they were running on a single time-sharing system.

Basic cache design. Each client and server workstation maintains a large cache of recently accessed file blocks, as shown in Figure 6. The caches are organized on a block basis, rather than a whole-file basis as in the Andrew file system,[3] and are stored in main memory rather than on a local disk. Blocks are currently four kilobytes. Each block in the cache is identified by a token for a file and a block location within the file. When the Fs_Read kernel call is invoked to read a block of a file, the kernel first checks its cache and returns the information from the cache if it is present. If the block is not in the cache, the kernel reads it from disk (if the file is on a local disk) or requests it from a server; in either case, the block is added to the cache, replacing the least-recently used block. If the block is requested from a server, the server checks its own cache before issuing a disk I/O and adds the block to its cache if the block was not already there.

Sprite uses a *delayed-write* approach to handle file writes. When an application issues an Fs_Write kernel call, the kernel simply writes the block into its cache and returns to the application. The block is not written through to the disk or server until it is ejected from the cache or 30 seconds have elapsed since the block was last modified. This policy is similar to the one used in time-shared Unix. It means some recent work may be lost in a system crash, but it provides much higher performance to applications than a policy based on write-through, since the application can continue without waiting for information to be flushed to disk. For applications with special reliability requirements, Sprite provides a kernel call to flush one or more blocks of a file to disk.

Cache consistency. When clients cache files, a consistency problem arises: What happens if one client modifies a file that is cached by other clients? Can subsequent references to the file by the other clients return "stale" data? Most network file systems, such as Sun's NFS, provide only limited guarantees about consistency. In NFS, for example, other clients with the file open may see stale data until they close the file and reopen it. Sprite guarantees consistency; each Fs_Read kernel call always returns the most up-to-date data for a file, regardless of how the file is being used around the network. This means that application programs running on different workstations under Sprite behave as if they were all running on a single, time-shared Unix system.

To simplify the implementation of cache consistency, we considered two separate cases. The first case is *sequential write-sharing*, where a file is modified by one workstation, read later by another workstation, but never open on both workstations at the same time. We expect this to be the most common form of write-sharing. The second case is *concurrent write-sharing*, where one workstation modifies a file while it is open on another workstation. Our solution to this situation

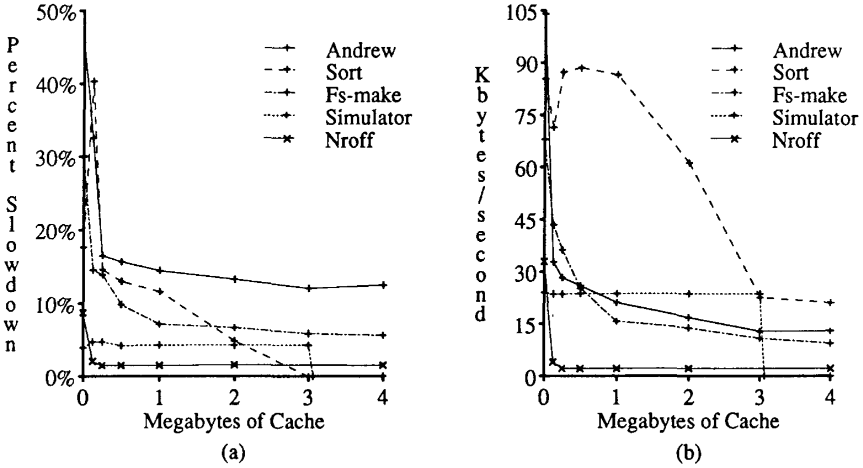


Figure 7. Client degradation and network traffic as a function of maximum client cache size for diskless Sun-3/75s with client caches using an unloaded Sun-3/180 file server. For each point the cache size was allowed to vary up to the given maximum. Part (a) plots degradation, which is the additional time required by a diskless workstation to complete the benchmark, relative to the time to complete the benchmark with a local disk and four-megabyte cache; (b) plots network traffic, including bytes transmitted in packet headers and control packets as well as file data.

is more expensive, but we do not expect it to occur very often.

Sprite uses version numbers to handle sequential write-sharing. When a client opens a file, the server returns the file's current version number, which the client compares to the version number associated with its cached blocks for the file. If they are different, the file must have been modified recently on some other workstation. In this case, the client discards all cached blocks for the file and reloads its cache from the server when the blocks are needed. Because of Sprite's delayed-write policy, the server does not always have current file data (the last writer need not have flushed dirty blocks back to the server when it closed the file). Servers handle this situation by keeping track of the last writer for each file; when a client other than the last writer opens the file, the server forces the last writer to write all its dirty blocks back to the server's cache. This guarantees that the server has up-to-date file information whenever a client needs it.

For concurrent write-sharing, where the file is open on two or more workstations and at least one of them is writing the file, Sprite disables client caching for that file. When the server receives an open request that will cause concurrent write-sharing, it flushes dirty blocks back from the current writer (if any) and notifies all clients having the file open that they should not cache the file anymore. Cache disabling is done on a file-by-file basis, and only when concurrent write-sharing occurs. A file may be cached simultaneously by several active readers.

There are two potential disadvantages to Sprite's cache consistency mechanism. First, it results in substantially slower file access when caching has been disabled. Fortunately, measurements and simulations in Nelson et al.[10] and Ousterhout et al.[11] show that files tend to be open for only short periods and are rarely write-shared, so cache disabling seldom occurs. Second, the Sprite approach depends on the fact that the server is notified whenever a file is opened or closed. This prohibits performance optimizations (such as name caching) in which clients open files without contacting the files' servers. Our benchmark results in Nelson et al.[10] suggest that such optimizations would provide little performance improvement.

It is important to distinguish between consistency and correct synchronization. Sprite's mechanism provides consistency; each read will return the most up-to-date data. However, the cache consistency mechanism will not guarantee that applications perform their reads and writes in a sensible order. For this to occur, applications must synchronize their actions on the file using the Fs_Lock system call or other available communication mechanisms. The cache consistency provided by Sprite simply eliminates the network issues and reduces the problem to that of timesharing systems.

File system performance. To measure the benefits of caching, we ran a series of file-intensive benchmark programs on Sun-3/75 workstations. A single Sun-3/180 file server was used for all client I/O and paging traffic. Because the benchmarks do not involve file sharing, they do not measure the overhead associated with cache consistency. (For descriptions of the benchmarks and additional performance measurements, see Nelson et al.[10])

Figure 7 shows that diskless workstations with caches of a few megabytes can

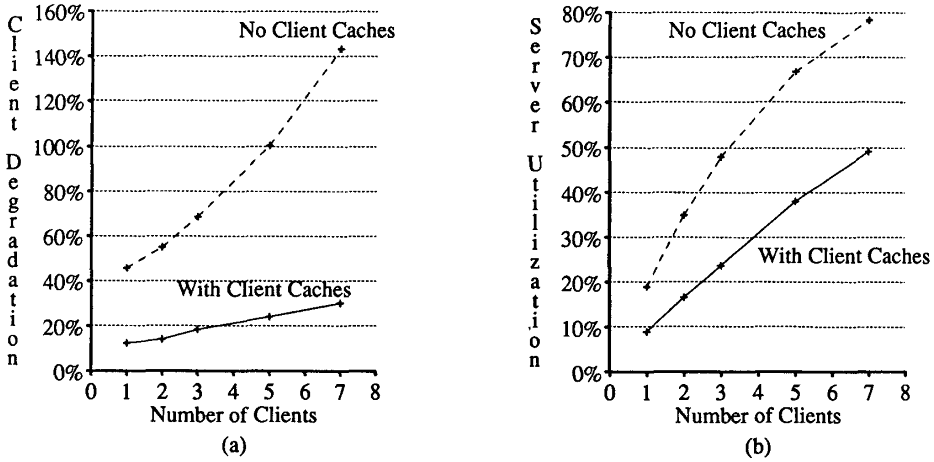


Figure 8. Effects of server contention when multiple diskless clients ran the most intensive benchmark (Andrew) simultaneously on different files using Sun-3/75 workstations. Andrew, written by M. Satyanarayanan,[3] is a composite benchmark that includes directory searches, file copying, version checking, and compilation. Part (a) shows the additional time required by each diskless client to complete the benchmark, relative to a single client running with local disk and cache; (b) shows server CPU use. When client caches were enabled, they were allowed to grow to four megabytes.

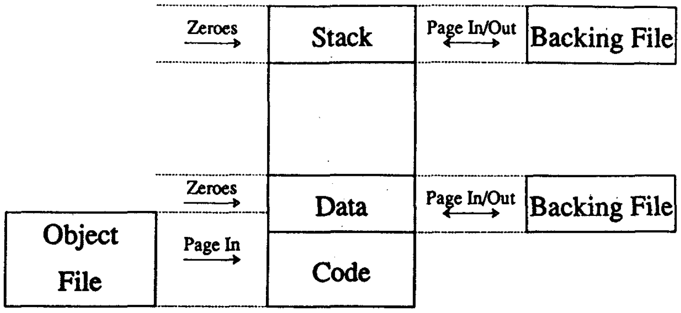


Figure 9. Sprite's paging structure. The code is paged in on-demand from the process' object file; since the code is read-only, it need not be written to backing storage and can be reloaded from the object file when needed. An ordinary file is used to back each data and stack segment. Initialized portions of the data segment are read in from the object file on first reference, then written to the backing file during page replacement and reused from there. For the stack segment and the uninitialized portions of the data segment, pages are filled with zeros on first reference, then paged to and from the backing files.

achieve performance within one to 12 percent of workstations with local disks, whereas diskless workstations without caches typically run 10 to 40 percent slower than workstations with disks. It also shows that client caching reduces network traffic by a factor of four or more. Without client caching, we believe that Ethernet's 10-megabit-per-second bandwidth will be a major bottleneck for next-generation workstations with five to 10 million instructions per second of processing power (for example, SPUR or the Sun-4 family). Even with client caching, faster networks will be needed to support the next generation of workstations after that.

Figure 8 shows that client caching reduces the server load by about a factor of two and suggests that a single server could support 10 or more active clients without excessive performance degradation. Normal users are rarely as active as the benchmark in Figure 8; Howard et al.[3] and Nelson et al.[10] estimate that one instance of the benchmark presents a load equivalent to at least five average users. This suggests that a Sun-3/180 Sprite file server can support at least 50 user workstations.

In comparisons with Sun's NFS, Sprite completed the Andrew benchmark 30 percent faster and generated only about one-fourth the server load. Since our NFS servers can support 10 to 20 clients, the NFS comparison supports our estimate of at least 50 clients per Sprite file server. (See Nelson et al.[10] for more information on the NFS comparison.)

Virtual memory

Sprite's virtual memory implementation is traditional in many respects. For example, it uses a "clock" algorithm variation for its page replacement mechanism and uses a straightforward extension of the time-shared Unix mechanism to provide shared read-write data segments. These and other aspects of the virtual memory system are described in detail by Nelson.[12]

This section focuses on three aspects of the virtual memory implementation where we intentionally deviated from Unix to better use networks and large physical memories. First, Sprite uses ordinary files for backing storage to simplify process migration, to share backing storage between workstations, and to capitalize on server caches. In addition, Sprite provides "sticky segments" and a dynamic tradeoff of physical memory between the virtual memory system and the file cache; these mechanisms were implemented to make the best possible use of physical memory as a cache for programs and files.

Backing storage. Backing storage is the portion of disk used to hold pages that have been swapped out of physical memory. Most versions of Unix use a special disk partition for backing storage and manage that partition with special algorithms. In networked Unix systems, each machine has its own private disk partition for backing storage. In contrast, Sprite uses ordinary files, stored in the network file system, for backing storage. A separate backing file is used for each data and stack segment, as illustrated in Figure 9. Each workstation is assigned a separate directory in which to create backing files for its processes.

There are several advantages to paging from files. First, it simplifies the implementation of virtual memory by reusing the existing file mechanisms. Second, it provides flexibility not present when each machine uses a private partition for backing storage. Many workstations may store their backing files in the same file system domain; this uses disk space more efficiently than schemes based on statically allocated private partitions. The network file system also simplifies backing file allocation on local disks or remote servers and simplifies process migration by making all backing files accessible to all workstations.

Backing files also have interesting performance consequences. In Sprite, remote backing files are cached in the main memories of servers, just like all other files. Our initial measurements show that a client can read random pages from a file in the server's cache faster than from a local disk, which means that a server with a large cache may provide better paging performance than a local disk. We think that CPU and network speeds are likely to increase at a much faster rate than disk speeds over the next few years, which will make remote paging to and from a server's cache even more attractive in the future.

Sticky segments. When a program starts execution, the pages in its code and data segments are loaded on-demand from the program's object file when page faults occur. To reduce this cost for frequently invoked programs, Sprite keeps a program's code pages in memory even after the program exits. The pages remain in memory until they are replaced using the normal clock mechanism. We call this mechanism *sticky segments*. If the same object file is reinvoked, then the new process can be started more quickly by reusing the sticky segment. If the object file is modified between executions, then the sticky segment will be discarded on the next execution. Data and stack segments are modified during execution, so they cannot be retained after the process completes.

Double caching. Double caching (caching the same file block in two different memory locations) is a potential issue because the virtual memory system is a user of the file system. A naive implementation might cause pages being read from backing files to end up in both the file cache and the virtual memory page pool; pages being eliminated from the virtual memory page pool might simply get moved to the file cache, where they would have to age again before being sent to the server. To avoid these inefficiencies, the virtual memory system bypasses the local file cache when reading and writing backing files. A similar problem occurs when demand-loading code from its executable file. In this case, the pages may already be in the file cache (for example, because the program was just recompiled). If so, the page is copied to the virtual memory page pool and the block in the file cache is given an infinite age so that it will be replaced before anything else in memory. The sticky segment mechanism will cache the page in the virtual memory system, so it is not necessary to keep it in the file cache as well. For the portions of object files corresponding to data pages, Sprite permits double caching to provide faster program start-up (the dirty data pages are discarded on program exit, but clean ones can be quickly reloaded from the file cache).

Although the virtual memory system bypasses its local file cache when reading and writing backing files, the backing files will be cached on servers. This makes servers' memories into an extended main memory for their clients. Servers do not cache backing files for their own processes, since this would constitute double caching; they only cache backing files for their clients.

Virtual memory-file system negotiation. The virtual memory system and file system have conflicting needs for physical memory. File system performance is best when the file cache is as large as possible, while virtual memory performance will be best when the file cache is as small as possible so that most of the physical memory may be used for virtual memory. To get the best overall performance, Sprite allows the file cache on each workstation to grow and shrink in response to changing demands on the machine's virtual memory and file system. This is accomplished by having the two modules negotiate over physical memory usage. The result is that small I/O-intensive programs, like compilers, may use almost all of the memory for a file cache, while large CPU-bound programs may use almost all of the memory for their virtual address spaces.

The file system and the virtual memory system manage separate pools of physical memory pages. Each module keeps an approximate time-of-last-access for each page (using different techniques in each module). Whenever either module needs additional memory (because of a page fault or a miss in the file cache), it compares the age of its oldest page with the age of the oldest page from the other module, replacing whichever is older. This allows memory to flow back and forth between the virtual memory page pool and the file cache, depending on the needs of the current applications.

We also considered more centralized approaches to trading off physical memory between the virtual memory page pool and the file cache. One possibility would be to access all information through the virtual memory system. To access a file, it would first be mapped into a process' virtual address space and then read or written just like virtual memory, as in Apollo's Aegis system[5] or Mach.[13] This approach

Table 2. The time required to migrate a process on Sun-3/75 workstations.*

Action	Cost or speed
Migrate smallest possible process	190 msec
Flush dirty pages	585 Kbytes/sec
Demand-load pages	545 Kbytes/sec
Transfer info for open files	14 msec/file
Flush file cache	585 Kbytes/sec

*The total time depends on how many dirty pages the process has (these must be flushed to the server during migration), how large its address space is (pages must be loaded on-demand on the process' new host), how many open files it has, and how many dirty blocks for those files are cached locally (they must be flushed). "Smallest possible process" refers to a process with no open files and one page each of code, data, and stack.

Table 3. Costs and benefits of process migration, measured by running several compilations concurrently.*

Program	Execution time		Improvement
	Local	Migrated	
One compilation	15.5 sec	15.9 sec	−3%
Two compilations	30 sec	17 sec	43%
Three compilations	45 sec	18 sec	60%
Four compilations	60 sec	20 sec	67%

*In the "local" column, all the compilations were run concurrently on a single machine. In the "migrated" column, one compilation was run locally and each of the others was migrated to a different workstation (except for the "one compilation" row, where the single compilation was migrated).

would eliminate the file cache entirely; the standard page replacement mechanisms would automatically balance physical memory use between file and program information.

We rejected the mapped-file approach for several reasons, the most important one being that it would have forced us to use a more complicated cache consistency scheme. Since a mapped-file approach requires a file's pages to be cached in a workstation's memory before they can be accessed, we would not have been able to implement cache consistency by refusing to cache shared files. A second reason for rejecting the mapped-file approach is that we wished to retain the Unix notion that I/O devices and files are accessed in exactly the same fashion; a mapped-file framework, with the assumed ability to access bytes in random order, does not seem natural for device I/O, which is most often sequential.

Process migration

Sprite's implementation of process migration differs from other implementations, such as those in the V System,[6] Accent,[7] or Locus,[2] in two major ways. The first difference is the way in which a process' virtual memory is transferred between machines, and the second difference is the way migration is made transparent to the migrated process.

The simplest approach to process migration is

- "freeze" the process (prevent it from executing any more);
- transfer its state to the new machine, including registers and execution state, virtual memory, and file access;
- "unfreeze" the process on its new machine so that it can continue executing.

The virtual memory transfer is the dominant cost in migration, so various techniques have been applied to reduce it. For example, V uses precopying, where the process continues executing while its memory is transferred. The process is then frozen, and any pages that have been modified are recopied. Accent uses a "lazy" approach in which the virtual memory image is left on the old machine and transferred to the new machine one page at a time when page faults occur. Locus checks for a read-only code segment and reopens it on the new machine, rather than copying it from the old machine; this allows the process to share a preexisting copy of the code on the new machine, if there is one.

In Sprite, backing files simplify the transfer of the virtual memory image. The old machine simply pages out the process' dirty pages and transfers information about the backing files to the target machine. If the code segment already exists on the new machine, the migrating process shares it, as in Locus. Pages get reloaded in the process' new machine on demand, using the standard virtual memory mechanisms. Thus, the process need only be frozen long enough to write out its dirty pages. The Sprite approach requires processes to be frozen longer than with either V or Accent, but it requires less data copying than V and does not require page fault servicing by the old machine after unfreezing on the new machine.

The second, and more important, issue in process migration is achieving transparent remote execution. A migrated process must produce the same results it would produce if it were not migrated, and special coding must not be required for a process to be migratable. For message-based systems like V and Accent, transparency is achieved by redirecting the process' message traffic to its new home. Since processes communicate with the rest of the world only by sending and receiving messages, this is sufficient to guarantee transparency. In contrast, Sprite processes communicate with the rest of the world by invoking kernel calls. Kernel calls are normally executed on the invoking machine (unless they make RPCs to other kernels), and some kernel calls will produce different results on different machines. For example, Sprite kernels maintain shared environment variables; Proc_GetEnviron may return different results on different machines.

Sprite achieves transparency in a fashion similar to Locus by assigning each process a *home node*. A process' home node is the machine on which the process was created, unless the process was created by a migrated process; in this case, the process' home node is the same as the home node of its parent. Whenever a process invokes a kernel call whose results are machine-dependent, the kernel call is forwarded to the process' home node (using the RPC mechanism) and executed there. This guarantees that the process produces the same results as if it were executing at home. To the outside world, the process still appears to be executing at home. Its process identifier does not change; it will appear in a process listing on the home node; and it can be debugged and terminated in the same way as other processes on the home node.

For each kernel call, we thus had two choices: either transfer all the state associated with the call at migration time so that the call can be executed remotely, or forward home all invocations of the call made by migrated processes. For calls that are invoked frequently, such as all the file system calls, we chose the first course (this was particularly simple for files, since the cache consistency mechanism already takes care of moving the file's data between caches). For infrequently invoked calls, or those whose state is difficult or impossible to transfer (for example, calls that deal with the home node's process table), we chose the forwarding approach.

Table 2 gives some preliminary measurements of process migration costs. If a process is migrated when it starts execution (before it has generated many dirty pages), the migration requires only a few hundred milliseconds on Sun-3/75 workstations. We expect this to be the most common scenario. The other major use of migration will be to evict migrated processes from a workstation whose user has just returned. In this case, the major factor will be the number of dirty pages. Even in the worst case (all memory dirty), all processes can be evicted from an eight-megabyte workstation in about 15 to 20 seconds. Table 3 shows that remote execution costs are acceptable (less than five percent penalty over executing at home for a compilation benchmark) and that migration may allow much more rapid completion of a collection of jobs. (See Douglis and Ousterhout[14] for more information on process migration in Sprite.)

As of this writing, all features discussed are operational—except for the code to choose a target for process migration and to evict migrated processes when a workstation's user returns, which is currently under development. In addition, Sprite supports the Internet protocol family (IP/TCP) for communication with other systems, and Sun NFS protocol support is planned. The Sprite kernel contains approximately 100,000 lines of code, about half of which are comments. All but a few hundred lines of code are in C; the remainder are written in assembler. Sprite currently runs on Sun-2 and Sun-3 workstations. Recently, we began using it for all of our everyday computing, including maintaining Sprite. We plan to port Sprite to the SPUR multiprocessor as prototypes become available later in 1988. We hope that Sprite will be portable enough to run on a variety of workstation platforms, and that it will be attractive enough for people outside the Sprite group to want to use it for their everyday computing.

In conclusion, we hope that Sprite will provide three overall features: sharing, flexibility, and performance. Users want sharing so that they can work cooperatively and use hardware resources fully. Sprite provides sharing at several levels: tightly coupled processes on the same workstation may share memory; processes everywhere may share files; and users may share processing power using the process migration mechanism. System administrators want flexibility so that the system can evolve gracefully. Sprite provides flexibility in the form of prefix tables, which allow user-transparent reconfiguration of the file system, and in the form of backing files, which allow workstations to share backing storage. Finally, everyone wants performance. Sprite provides high performance by using a special-purpose RPC protocol for communication between kernels and by using physical memory as a flexible cache for both programs and files. □

Acknowledgments

Adam de Boor implemented the Pmake program and has assisted in many other areas of Sprite development, including porting the X11 window system. David Anderson, Jim Larus, Rita Ousterhout, Gerald Popek, Carlo Séquin, and the *Computer* referees provided helpful comments on early drafts of this article.

The work described here was supported, in part, under Defense Advanced Research Projects Agency Contract N00039-85-R-0269 and, in part, under National Science Foundation Grant ECS-8351961.

References

1. M. Hill et al., "Design Decisions in SPUR," *Computer*, Nov. 1986, pp. 8-22.
2. G. Popek and B. Walker, eds., *The Locus Distributed System Architecture*, MIT Press, Cambridge, Mass., 1985.
3. J. Howard et al., "Scale and Performance in a Distributed File System," *ACM Trans. Computer Syst.*, Feb. 1988.
4. R. Sandberg et al., "Design and Implementation of the Sun Network Filesystem," *Proc. Usenix 1985 Summer Conf.*, June 1985, pp. 119-130.
5. P. Leach et al., "The Architecture of an Integrated Local Network," *IEEE Trans. Selected Areas in Comm.*, Nov. 1983, pp. 842-857.
6. M. Theimer, K. Lantz, and D. Cheriton, "Preemptable Remote Execution Facilities for the V-System," *Proc. 10th Symp. Operating Syst. Principles*, Dec. 1985, pp. 2-12.
7. E. Zayas, "Attacking the Process Migration Bottleneck," *Proc. 11th Symp. Operating Syst. Principles*, Nov. 1987, pp. 13-24.
8. A. Birrell and B. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Syst.*, Feb. 1986, pp. 39-59.

9. B. Welch and J. Ousterhout, "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System," *Proc. Sixth Conf. Distributed Computing Syst.*, May 1986, pp. 184-189.

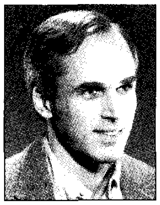
10. M. Nelson, B. Welch, and J. Ousterhout, "Caching in the Sprite Network File System," *ACM Trans. Computer Syst.*, Feb. 1988.

11. J. Ousterhout et al., "A Trace-Driven Analysis of the Unix 4.2 BSD File System," *Proc. 10th Symp. Operating Syst. Principles*, Dec. 1985, pp. 15-24.

12. M. Nelson, "Virtual Memory for the Sprite Operating System," Tech. Report UCB/CSD 86/301, June 1986.

13. M. Accetta et al., "Mach: A New Kernel Foundation for Unix Development," *Proc. Summer Usenix*, July 1986, pp. 93-112.

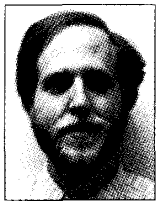
14. F. Douglis and J. Ousterhout, "Process Migration in the Sprite Operating System," *Seventh Int'l Conf. Distributed Computing Syst.*, Sept. 1987, pp. 18-25.



John K. Ousterhout is an associate professor in the Department of Electrical Engineering and Computer Sciences at the University of California at Berkeley. His interests include operating systems, distributed systems, user interfaces, and computer-aided design. He and his students have developed several widely used programs for computer-aided design, including Magic, Caesar, and Crystal. Ousterhout is now leading the development of Sprite, a network operating system for high-performance workstations.

Ousterhout is a recipient of the ACM Grace Murray Hopper Award, the National Science Foundation Presidential Young Investigator Award, the IEEE Browder J. Thompson Award, and the UCB Distinguished Teaching Award.

He received a BS degree in physics from Yale University in 1975 and a PhD degree in computer science from Carnegie Mellon University in 1980.



Andrew R. Cherenson is currently employed at Silicon Graphics, Mountain View, California. His research interests include operating systems and distributed systems. He was a systems programmer at the Department of Chemistry, Harvard College, and at the Research Institute of Scripps Clinic, La Jolla, Calif.

He recently received an MS degree from the University of California at Berkeley and received an AB degree in biochemical sciences from Harvard College in 1981. He is a member of IEEE and ACM.

Frederick Douglis is currently a PhD candidate in the Department of Electrical Engineering and Computer Science, University of California at Berkeley. His research interests include process migration and archival storage.

He received an MS degree in computer science from the University of California in 1987 and a BS degree in computer science from Yale University in 1984. Douglis is a member of IEEE and ACM.

Michael N. Nelson is a PhD candidate at the University of California at Berkeley. His research interests include operating systems and distributed systems.

He received the MS and BA degrees in computer science from the University of California in 1986 and 1983, respectively.

Brent B. Welch is a PhD candidate at the University of California at Berkeley. His research interests include network operating systems and distributed systems in general.

He received an MS degree in computer science from the University of California in 1986 and a BS degree in aerospace engineering from the University of Colorado, Boulder, in 1983.

Questions regarding this article may be addressed to Ousterhout at the Computer Science Division, Dept. of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.