OVERCOMING UNIX KERNEL DEFICIENCIES IN A PORTABLE, DISTRIBUTED STORAGE SYSTEM

UCRL--102665

DE90 007256

Mark Gary

Lawrence Livermore National Laboratory
Livermore, California

ABSTRACT

The LINCS Storage System at Lawrence Livermore National Laboratory was designed to provide an efficient, portable, distributed file and directory system capable of running on a variety of hardware platforms, consistent with the IEEE Mass Storage System Reference Model. Our intent was to meet these requirements with a storage system running atop standard, unmodified versions of the Unix operating system. Most of the system components run as ordinary user processes. However, for those components that were implemented in the kernel to improve performance, Unix presented a number of hurdles. These included the lack of a lightweight tasking facility in the kernel; process-blocked I/O; inefficient data transfer; and the lack of optimized drivers for storage devices. How we overcame these difficulties is the subject of this paper. Ideally, future evolution of Unix by vendors will provide the missing facilities; until then, however, data centers adopting Unix operating systems for large-scale distributed computing will have to provide similar solutions.

INTRODUCTION

With the advent of the IEEE Mass Storage System Reference Model,[1] storage systems and system components are being created that are capable of running on a variety of architectural platforms. One such system is the LINCS Storage System at Lawrence Livermore National Laboratory (LLNL).[2,3]

The LINCS Storage System was designed to provide an efficient, portable, distributed file and directory system capable of running on a variety of hardware platforms. It consists of a set of cooperating, distributed, multitasking servers (disk bitfile server, tape bitfile server, name server, etc.).[2] These servers communicate with each other and application clients over separate control and data associations (bidirectional communications links) using a common communication library (see Figure 1).[4] By allowing servers and clients to reside on different machines, this approach lets network designers make the best use of their machine resources.

From the standpoint of ease of development and portability among different machine architectures, we decided that Unix would be the best operating-system base. In addition to running on Unix, however, our system was required to efficiently manage a very large number of large files in a distributed environment with heterogeneous machines and operating systems. Thus the LINCS Storage System was designed with these general requirements in mind without the constraint of being tied to a particular operating system or architecture. The result is a fast, scalable file and directory system capable of running alongside native file systems.

Our intent was to meet the design requirements with a storage system running atop standard, unmodified versions of the Unix operating system. In developing the design, however, we found that the Unix system-call interface and kernel lack many of the facilities that are critical to the performance the LINCS system.

Other sites have observed some of the same problems in designing Unix-based storage systems, but their solutions do not necessarily meet our needs. In common with NASA-Ames, for example, we encountered a number of limitations in the Unix file system itself: the amount of time spent during crash recovery checking possibly millions of files; file size and directory structure organizational limitations imposed by the file system; a myriad of security deficiencies;[5] the lack of a large-scale, rapid-access archive capability; the absence of an automatic

---

file migration system. While NASA-Ames addressed these problems by modifying the native file system,[6] the LINCS bitfile servers manage a set of devices not known to the Unix file system.

This paper describes how LLNL system designers compensated for six Unix kernel deficiencies with respect to the requirements of the LINCS Storage System:

- Lack of a lightweight tasking facility in the kernel
- Process-blocked I/O
- Data copies between user and system space
- Lack of drivers for leading-edge devices
- Driver and kernel code not written for storage-system demands
- Lack of kernel facilities needed for day-to-day operation

The LINCS Storage System currently runs on Amdahl's UTS operating system (based on AT&T System V), although portions were developed under Berkeley 4.3 Unix and some servers have been ported to ULTRIX† and SunOS† systems. Each server is a separate Unix process that is multitasked using a lightweight tasking library.[7] The tasking library is a coroutine-based stack manager and scheduler which provides reentrant code and simplified memory sharing without the overhead of heavyweight process forking and context switching. It handles the actual scheduling of tasks within a process, while the Unix process scheduler handles the scheduling of the process itself.

It is important to note that, while we ran into the deficiencies outlined below when implementing the LINCS Storage System on UTS, they are not unique to that system. These deficiencies are common to most Unix systems.

UNIX KERNEL DEFICIENCIES AND SOLUTIONS

Lack of a Lightweight Tasking Facility in the Kernel

Most commercially available Unix kernels do not contain a lightweight tasking facility. We realized, as did the developers of the Mach[8] and Tunis[9] operating systems, that such a facility was needed to enable writers of drivers and kernel-resident servers to handle multiple concurrent operations as many threads of execution rather than as one queue-driven control thread. Such explicit concurrency simplifies writing and maintaining drivers, and extends much more easily to kernel implementations that run on multiprocessors, which we expect to be common in the near future.

We provided lightweight kernel tasking by implementing our tasking library in the kernel. As mentioned above, our tasking library is a stack manager and scheduler. It allocates space within which it manages task stacks. It uses Unix `setjmp()` and `longjmp()` calls to transition between tasks (see Figure 2). Besides providing tasking primitives, this library also provides primitives for timing, memory allocation, and synchronization.

In a standard Unix implementation, when an event such as the completion of an I/O occurs, Unix "Vs"[10] a semaphore associated with the I/O to return control to the kernel stack waiting for the operation. In our environment, when a "V" occurs for an event to be handled by the tasking library, the event is recognized and control is transferred from the active kernel stack to the appropriate task stack managed by the tasking library. When the task completes, the tasking library transitions back to the active kernel stack. `setjmp()` and `longjmp()` are used to save the current set of registers in a jump buffer and to resume operation with a previously saved register set, allowing simple transitions between stacks.

Process-Blocked I/O

When a Unix process issues an I/O operation, the entire process blocks (suspends) until that I/O is complete. This is not acceptable for multitasked server processes, as an I/O request from

any task within a server causes all of that server's tasks to block even though they may be working on widely differing operations for different clients.

Various asynchronous I/O facilities exist in different Unix implementations, including ULTRIX "nbuf" system calls (multiple-buffered I/O operations). Efficiently using these methods requires the `select()` system call. Some Unix systems do not support `select()` at all; other systems, including UNICOS, have other, nonstandard, asynchronous I/O facilities; those that do support `select()` have problems. For example, the number of I/O descriptors upon which `select()` can operate may be inadequate for large, active storage systems, and not all systems support `select()` for I/O to all devices. Although we are currently pursuing implementations based on the `select()` call, we have developed an alternative solution which also provides more efficient data transfer.

To avoid blocking an entire server process with one task's I/O, the tasking library queues a process's I/O requests until all of its tasks are either waiting for I/O or are quiescent. All of the process's I/O requests are then submitted using a pseudo-device-driver call, which only then blocks the process. The process is unblocked and allowed to run as soon as any one of the requested I/Os is complete. Therefore, one task's I/O does not interfere with the operation of other tasks within the same process.

To implement this solution, we designed servers called bitfile movers.[1,11] Movers are defined in the IEEE Mass Storage Systems Reference Model as modules that transfer data between two channels. The LINCS movers were designed to transfer data between a client and a device. Two movers are implemented in the kernel; one accesses magnetic disks, and the other accesses magnetic tape. Other movers, with lower performance requirements, are implemented in user space; one such mover maintains tape header information on disk.

LINCS movers are multitasked in both the kernel-space and the user-space implementations. Movers receive control requests from LINCS servers in the form of record structures. These structures contain information about requests (volume number, data location, length) and the outcome of these I/O requests (success, failure) when returned to the server. A server's request structures are queued and submitted to a mover using a LINCS pseudo-device-driver call.

Kernel-space movers translate I/O requests into direct driver calls (interfacing at the base of the block I/O layer), while user-space movers translate them into Unix file-system calls (read, write). A mover sends or receives data to or from an address provided in the request structure. Mover tasks exist for each request of a bitfile server task. In this way, once one I/O request completes, the server process can be unblocked while a second I/O is pending, its state saved in a separate kernel task stack awaiting completion.

Using a portable tasking interface and a common communications interface allows a mover to be ported between kernel space and user space with only minor modifications to the device-driver–operating-system interface. It is important to note that a kernel-space mover eliminates the problem of process-blocked I/O, while a user-space mover, without `select()`, does not. Because they use standard Unix read/write calls, user-space movers block when a task does file I/O. They do have the advantage of allowing the storage system to be written portably. This allows the system to run on machines without kernel modifications if process-blocked I/O is not a handicap.

Data Copies Between User and System Space

User-space bitfile servers lose efficiency if data is repeatedly copied between user and system space. For example, a bitfile server might read data from a device into user space and then send it to the requesting client. Using standard Unix block I/O, this involves a copy of the data from kernel space into the bitfile server's user space (`copyout()`), a copy of the data from user to system space (`copyin()`) and another `copyout()` to get the data to the client (see Figure 3). This is true even if the client is on the same machine. These copies between user and system space are expensive. Ideally, to achieve the

highest possible performance, the data should flow from the disk device directly onto the network.

The LINCS architecture eliminates extra data copies by separating control and data associations. A mover obtains the source or destination address for a request from the bitfile server and sends data directly to, or receives data directly from, the client with whom the data is associated. Therefore data flows directly from movers to clients without passing through the bitfile server.

This technique is particularly useful when the client is on a different physical machine on the network. In this case, the data can flow directly from disk (or any medium) to the remote client without the usual `copyout()`, `copyin()` steps required to get it on the network (see Figure 4). Likewise, in the case where the client is another kernel mover (i.e., file migration between disk mover and tape mover) the data flow is kept entirely within the kernel with no extra data copies (see Figure 5).

Lack of Drivers for Leading-Edge Devices

Large-scale storage systems have traditionally been developed on mainframes because these processors typically supported leading-edge storage and network devices. Leading-edge devices are defined here to be high-speed, large-capacity devices and very fast networks. Unix has only recently started making inroads into the mainframe and supercomputer markets from its historical position in the minicomputer and workstation arenas. Because of this, Unix systems do not yet support, or are slow in supporting, the leading-edge devices required by large storage systems.

Even though we implemented our system on a mainframe-based Unix product, we had to write drivers for IBM 3480 tapes, for the the Storage Technology 4400 Automated Cartridge System (ACS), and for the NSC HYPERchannel. We are currently migrating to standard vendor drivers for these devices as they become available.

Driver and Kernel Code Not Written for Storage System Demands

Existing Unix storage and network drivers, as well as other kernel routines, are not always written for very large volumes of data throughput. In some cases, too few resources (e.g., buffers, data structures) are dedicated to operations. In other cases, drivers (e.g., disk drivers) are written to write and retrieve only a few consecutive blocks of data. Kernel resources such as callout structures or open-file data structures can easily be exhausted by large data transfers, or when hundreds of jobs are outstanding because of inoperative network nodes.

It is not always possible to add more resources by simply changing a compile-time or run-time parameter. Further, increasing the size of a kernel resource often has wide-ranging effects throughout the kernel. This is particularly true of structures that are created for each process, such as the process structure itself and the user structure, since increases here can dramatically increase the size of a kernel. Time spent traversing larger list structures can degrade performance, and more memory for kernel structures increases paging for user processes.

We solved these problems by rewriting or modifying the appropriate kernel code. Determining which code segments require enhancement and which kernel resources might be strained by a storage system is not always simple. It was not until heavy load testing on LLNL's identical development storage processor that some of these problems became obvious.

We improved data throughput from drivers by providing larger or variable-size data transfers. Some improvements were achieved through simple code optimization, while other improvements required code modification, mainly network drivers, to gracefully handle recovery from errors due to inoperative nodes in our distributed environment. For example, we modified our NSC HYPERchannel driver to convey error information to the data link layer for intelligent probing of inoperative nodes.

As we convert to industry standard protocols (e.g., TCP/IP), we expect to encounter problems with set timeouts. To handle machines with

widely differing throughput abilities in environments in which large blocks of data are moved, it is often necessary to increase data timeouts, make them flexible, or add special data-buffering schemes.

Lack of Kernel Facilities Needed for Day-to-Day Operation

Unix kernels often do not provide the kernel hooks needed for day-to-day storage system operation. This again comes from the fact that Unix is only now beginning to appear in large data center environments. Until such support is supplied, the only option is to add the missing pieces oneself.

Although present in some Unix implementations, one of the most important items missing from our Unix systems was a facility providing atomic redundant writes of critical information. Safely managing tape headers on disk required such a facility. To accomplish this we implemented shadowed atomic updates in the mover which handles tape headers.[3]

Another important facility we added was a circular-buffer trace facility in the kernel. This trace buffer was useful when implementing drivers and other kernel modifications, and it is useful in helping diagnose network problems. We use this facility to supplement the standard system trace routines, logging information from drivers and tracing packets flowing to and from the network.

Other additions and kernel modifications we made included installing code to force automatic reboot upon system crashes, modifying text tables for correct console printing, and adjusting tuning parameters and algorithms to best meet our environment's requirements.

CONCLUSION

The LINCS Storage System is a fast, scalable, distributed storage system implemented on top of the Unix operating system. To satisfy the performance requirements of a large scale storage system a number of kernel modifications and additions were made to overcome a number of Unix kernel deficiencies. These deficiencies included: the lack of a lightweight tasking facility in the kernel, process-blocked I/O, inefficient data copies between user and system space, and the lack of optimized drivers for leading-edge storage devices. It remains our goal, however, to use standard, unmodified Unix operating-system bases for our system. Meeting this goal will require that Unix system developers appreciate the problems we encountered and offer suitable, possibly similar solutions in future Unix releases. In the meantime, we are investigating various asynchronous I/O facilities, including enhanced `select()` calls, for systems upon which kernel modification is not an option.

ACKNOWLEDGEMENT

I would like to thank Samuel Coleman, Richard Watson, Richard Wolski and Gary Shaw for their valuable contributions to this paper. This work was performed by Lawrence Livermore National Laboratory under contract number W-7405-ENG-48 under auspices of the U.S. Department of Energy.

REFERENCES

1. Miller, Stephen W., "A Reference Model for Mass Storage Systems," Advances in Computers, Vol. 27, 1988, pp. 157–209.

2. Hogan, Carole, L. Cassell, J. Foglesong, J. Kordas, M. Nemanic, and G. Richmond, "The Livermore Distributed Storage System: Requirements and Overview," DIGEST OF PAPERS, Tenth IEEE Symposium on Mass Storage, May 7–10, 1990.

3. Foglesong, Joy, G. Richmond, L. Cassell, C. Hogan, J. Kordas, and M. Nemanic, "The Livermore Distributed Storage System: Implementation Experiences," DIGEST OF PAPERS, Tenth IEEE Symposium on Mass Storage, May 7–10, 1990.

4. Fletcher, John, "APST Interfaces in LINCS," Lawrence Livermore National Laboratory (internal publication), July 23, 1985.

5. Hogan, Carole B., "Protection Imperfect: The Security of Some Computing

Environments," Operating Systems Review,Vol. 22, No. 3, July 1988, pp.7–27.

6. Richards, J., T. Kummell, and D. G. Zarlengo, "A UNIX-MVS Based Mass Storage System for Supercomputers," DIGEST OF PAPERS, Ninth IEEE Symposium on Mass Storage, October 31-November 3, 1988, pp 25–28.

7. Fletcher, John G., "SMILE," Lawrence Livermore National Laboratory (internal publication), April 15, 1988.

8. Rashid, Richard F., "Threads Of A New System," Unix Review, August 1986, pp. 37-48.

9. Ewens, P.A., R.C. Holt, M.J. Funkenhauser, D.R. Blythe, "The Tunis Report: Design of a UNIX-Compatible Operating System," University of Toronto Technical Report CSRI-176, January 1986.

10. Dijkstra, E.W., "Cooperating Sequential Processes," Programming Languages, ed. F. Genuys, Academic Press, New York, 1968.

11. Kitts, David, Sam Coleman, and Bruce Griffing, "Bitfile Mover," DIGEST OF PAPERS, Ninth IEEE Symposium on Mass Storage, October 31–November 3, 1988, pp 25–28.

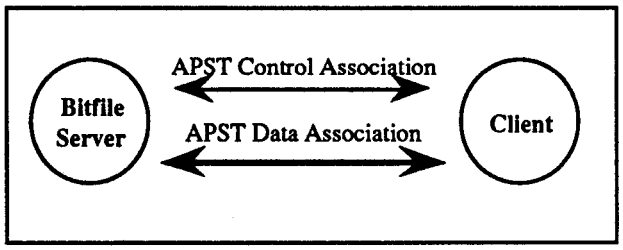

Figure 1. Separation of Control and Data

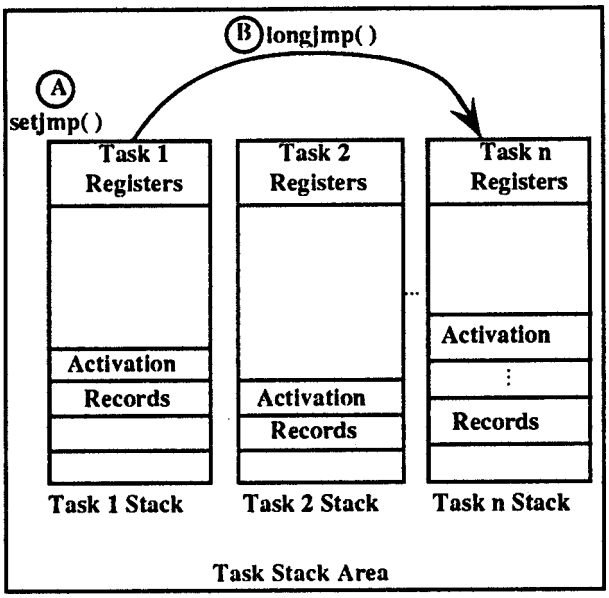

Figure 2. Task Transitions.

(A) Task 1's registers saved using setjmp( ).

(B) Control switched to Task n using longjmp( ) andTask n's saved registers.

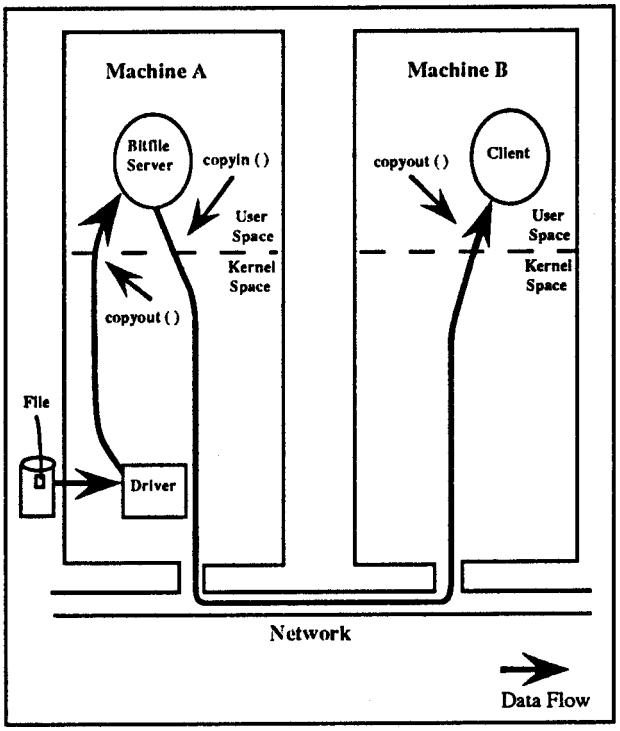

Figure 3. Data Flow Without A Kernel Mover

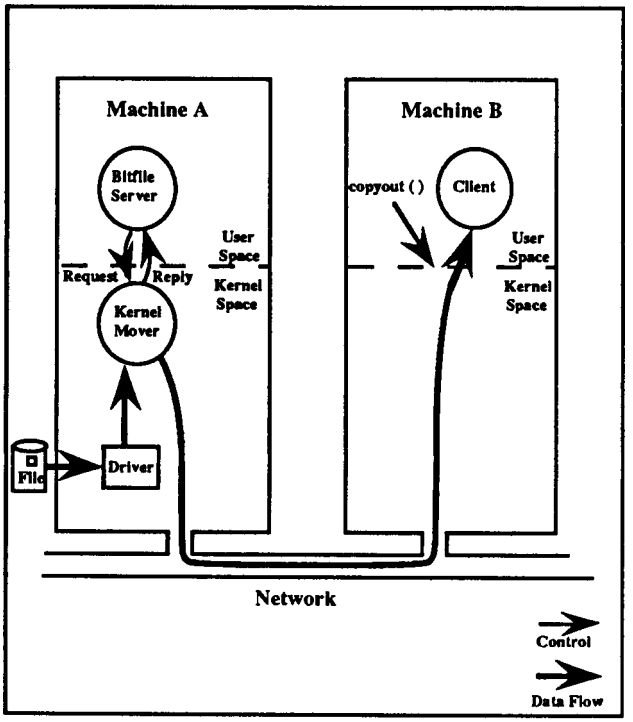

Figure 4. Data Flow With A Kernel Mover

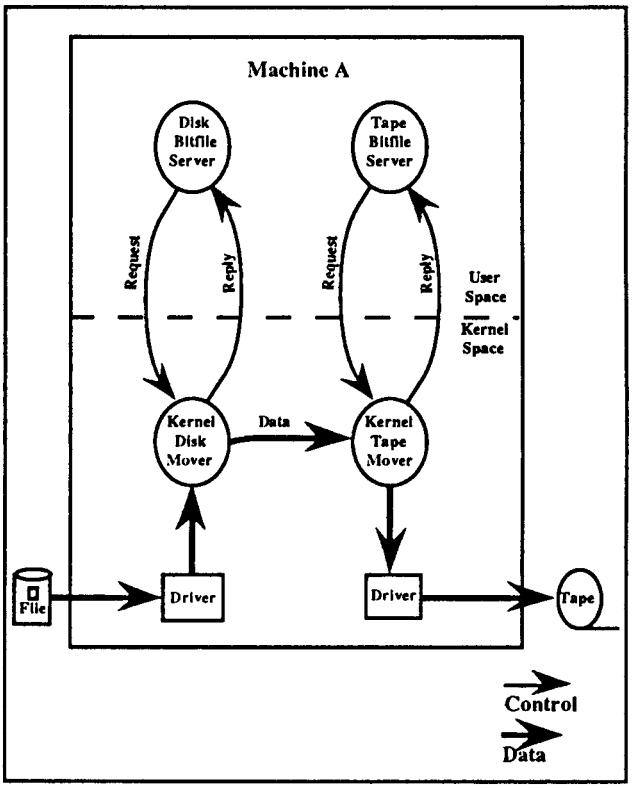

Figure 5. Data Flow, Disk To Tape Migration With Movers