Proceedings of Sixth ACM Symposium on Operating Systems Principles (November 1977) 33-42.

# The DEMOS File System*

Michael L. Powell

Los Alamos Scientific Laboratory
Los Alamos, New Mexico 87545

## ABSTRACT

This paper discusses the design of the file system for DEMOS, an operating system being developed for the CRAY-1 computer at Los Alamos Scientific Laboratory. The goals to be met, in particular the performance and usability considerations, are outlined. A description is given of the user interface and the general structure of the file system and the file system routines. A simple model of program behavior is used to demonstrate the effect of buffering data by the file system routines. A disk space allocation strategy is described which will take advantage of this buffering. The last section outlines how the performance mechanisms are integrated into the file system routines.

## 1. Introduction

File system design is a fairly well explored area of operating system research. Many modern file systems offer device-independent user interfaces, hierarchical file names, flexible file operations and powerful mechanisms for manipulating individual files and groups of files [Ritchie] [Feiertag] [Bobrow]. However when considering high-speed computational machines, such as the Cray Research, Inc. CRAY-1 [CRI], ease of use and flexibility are often sacrificed in the names of efficiency and size of operating system. The systems currently running at the Los Alamos Scientific Laboratory (LASL) are examples of this type of system. None support hierarchical file names and all require significant programmer effort in order to derive the most performance from the machine.

The kinds of work done at LASL on these machines varies widely. There is a heavy time-sharing load consisting of interactive text editing and many small user jobs. However, most of the machine resources are consumed by "number crunching" programs which may run for minutes or hours. Large portions of this workload are being transferred to the CRAY-1. Moreover, because of the unique characteristics of the CRAY-1, new kinds of applications will be developed to run on it. Therefore, the DEMOS file system must support jobs with radically varying I/O requirements.

The classical method for improving I/O performance is buffering. Buffering means that data is transferred from a secondary storage device to areas of main memory not currently being manipulated by the program. This allows computation to be overlapped with I/O activity, and thus reduces the total amount of time for a job to execute. File systems for most large computational computers require the user to take explicit actions with respect to buffering data.

One of the primary mechanisms for achieving performance goals in DEMOS is buffering. Buffering is done on two levels: by the disk controller in its local memory, and by the file system routines in main memory buffers. This buffering is performed without any special actions on the part of the user. The buffering done by the disk controller has a fixed strategy. The buffering by the file system routines is based on the ob-

* Work done under the auspices of the USERDA.

served I/O activity on each file. By adjusting the buffering strategy for each file dynamically, the best use may be made of system resources.

Another important technique used in DEMOS for improving file system performance is the localizing of references to small physical areas of the secondary storage device. The most common device, the moving head disk, can access data at the current arm position much faster than it can access data farther away. Also, data at the current arm position can be accessed more economically if several records can be accessed in one revolution of the disk.

## 2. Design Objectives

Characteristics of the future workload of the CRAY-1 are difficult to predict with any accuracy. Since the hardware and software differ significantly from that currently in use, simple extrapolation may not be appropriate. However, lower bounds on the necessary performance may be estimated from the characteristics of the current workload. Since much of the work which will initially be done on DEMOS has been done on these other systems, their characteristics provide initial design data for the CRAY-1 system.

The distribution of sizes of disk files is useful in choosing an allocation unit for disk space. Figure 1 shows the cumulative distribution of file sizes in bytes [Collins] for the central file system at LASL. The central file system is the main repository for permanent files. The top curve shows the percentage of files versus file size while the bottom curve shows the percentage of total file space versus file size. The top curve indicates that most files are relatively small, almost 50% contain less than 40K bytes. On the other hand, a few large files occupy most of the space in the central file system. Although the CRAY-1 will be able to directly access substantially more disk space than the current machines can, similar behavior is expected for DEMOS files. This would indicate that the allocation unit should be less than 40K bytes. Otherwise, each small file will occupy considerably more disk space than its data requires.

File access characteristics are significant in determining the performance of a file system. The sustainable data bandwidth is one measure of file system performance. At LASL, there are two general kinds of usage for the current machines: interactive time-sharing through LTSS (Livermore Time-Sharing System) and single-user batch through CROS (Chili Ridge Operating System). At night, when there are few interactive users, LTSS effectively runs in a multi-programmed batch mode. In the interactive environment, most jobs have modest I/O requirements. However, because of the large number of jobs in the system, a substantial amount of I/O activity is required for swapping. In the batch environment, individual jobs have much greater I/O requirements, although swapping rates are much lower. Data accumulated at LASL [Keller] show sustained transfer rates of 5-6 megabits/second for each CPU are typical when the interactive system is busy. Although they tend to be lower, batch mode I/O rates frequently are in this same range. Data gathered at Lawrence Livermore Laboratory [Sloane] which runs the same system with a workload similar to LASL shows sustained data rates of 9-10 megabits/second per CPU. The data rate is higher due to the larger number of I/O channels on their machines, but indicates that even higher data rates could be necessary to effectively use the greater processing speed of the CRAY-1.

Gene Amdahl has speculated that the I/O bandwidth requirement is proportional to the instruction execution rate [Amdahl]. The evaluation of the CRAY-1 done at LASL indicates that it operates in the 20-60 million floating point operations per second (MFLOP) range, whereas a CDC 7600 operates in the 2-6 MFLOP range [Keller]. This would mean a factor of 4-10 improvement in I/O bandwidth over observed LTSS performance in order to satisfy Amdahl's criterion.

Analysis of the distribution of the data transfer size is necessary in order to determine whether the required transfer bandwidth can be achieved. If the amount of data transferred by a particular request is small, a large percentage of the time required to complete the request is consumed by locating the appropriate data, rather than transferring it. Data gathered at Lawrence Livermore Laboratory [Sloane] shows that users tend to favor smaller transfer sizes. Except near the standard buffer size, the number of requests of a particular size decreases as the size increases. The system, however, performs few I/O operations except to swap jobs in and out of memory. Its requests are clustered around the standard memory field sizes (see Figure 2). The overall mean transfer size is 150 Kbytes, with the mean user request about half this value and the mean system request about half again as much.

Since the median file size is so much smaller than any of these values, it can be concluded that the larger files are accessed far more frequently than smaller files. This observation has intuitive appeal in that programs and control information are gen-

erally small, whereas input, output and work files are generally large. The latter files are accessed more frequently, although the former may be greater in number.

A summary of the characteristics of the environment is: the file system will have to support a bandwidth of 20-60 megabits/second or higher; there will be a preponderance of small files which will be accessed relatively infrequently; there will be a smaller number of large files which will be heavily accessed; most user requests will be small, although a significant fraction will be large; and virtually all system requests will be to swap programs.

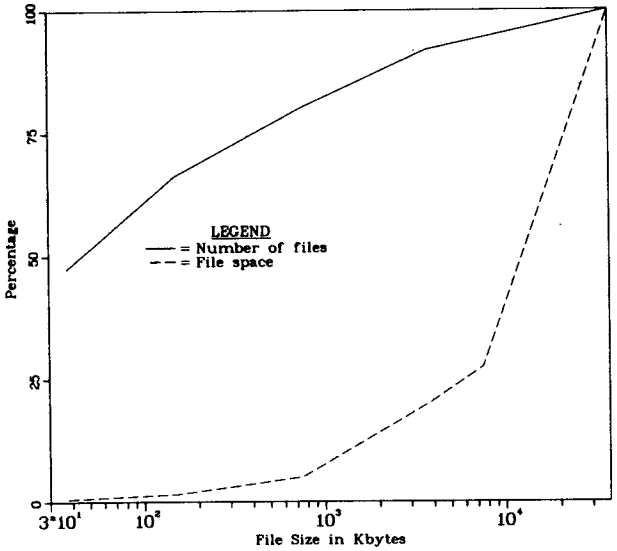


Figure 1. CUMULATIVE FILE SIZE DISTRIBUTION

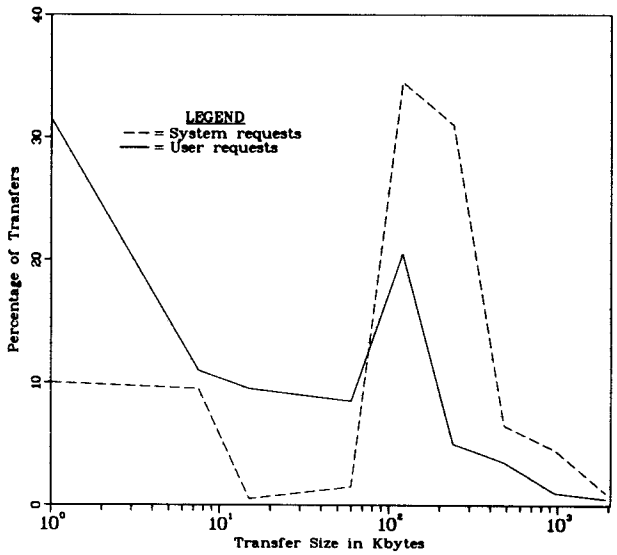


Figure 2. TRANSFER SIZE DISTRIBUTION

## 3. File System Overview

The basic unit of storage in the file system from the user's point of view is an 8-bit byte. A file may be viewed as an ordered sequence of bytes. The bytes of a file may be processed sequentially or randomly.

Files have hierarchical names. The hierarchical names are implemented through special files called directory files. Data stored in a directory file establishes a correspondence between file names and files. The directory files may be visualized as nodes in a tree. Directories point to other files which may be directory files or data files. The root of the directory tree is the system root directory file. At the leaf nodes are file descriptors which contain such information as the file's owner, classification, and size, as well as the location on the disk of the file's data. For each file descriptor being used, there is a unique path through the directory tree from the system root directory to the file descriptor.

Files are initially created with no data in them. As data is written to the file, blocks are allocated to hold the data. A block contains 4096 bytes and occupies a sector on the disk. Programs do not have to deal with blocks, however, and may read or write (sequentially or randomly) any group of contiguous bytes in the file. I/O operations are initiated by the user program through messages to the file system routines. The data is automatically buffered by the system as described below. I/O operations may be done synchronously (that is, the program will not proceed until the operation is complete) or asynchronously (the program may proceed, but must later check the status of the operation).

The DEMOS file system will be used in conjunction with a central mass storage system, the Common File System (CFS). Currently under development, the CFS will eventually serve most of the LASL computers as a permanent and archival store. The CFS has a directory structure identical to the DEMOS file system. When the CFS is operational, such facilities as automatic migration and staging of files will be available.

## 4. File System Routines

The file system routines form a flexible structure for the implementation of I/O operations. Each main routine is implemented as a separate task in the system. Each task manipulates its own subset of control information, and communicates with the other tasks through messages. The user interface is isolated in one task; the buffer management is in another; the device-dependent information is in a third.

### 4.1. The request interpreter

The request interpreter provides the user interface to the file system. It receives messages from the user program and requests buffers from the buffer manager. When the buffer manager gives a buffer to the request interpreter, it is full of data (on a read operation) or ready to be filled (on a write operation). The actual I/O is performed implicitly through the buffer manager.

The request interpreter moves all information to or from the user's memory field. It accepts requests from the user program and provides status information when they are complete.

The request interpreter performs two primary functions: parameter validation and request translation. Parameter validation consists of ensuring that the addresses specified by the user are valid and that the requested operations are permitted on the file. Request translation is interpreting the request's parameters and making the appropriate calls on the buffer manager.

### 4.2. The buffer manager

The buffer manager maintains the pool of buffers for I/O operations. The buffer pool is shared by all open files, and may vary dynamically in size. In the worst case, buffers are allocated on a demand basis and reclaimed on a least-recently-used basis. In this case, file blocks are written to disk as a result of the buffer reclamation algorithm.

In practice, buffers will be allocated according to the strategy routines described below. Buffers will be queued to be written to the disk after being filled by the request interpreter.

Buffers are either associated with a particular file, or available for use. Requests are made for buffers based on a system file index and a block number within the file. If the requested buffer is in memory it is simply returned. If not, an empty buffer is allocated and sent to the disk driver to be filled.

### 4.3. The disk driver

The disk driver performs all of the actual disk positioning and data transfer operations related to a file request. In the disk driver is the device dependent information for the file system. The disk driver task manages I/O to all disk devices. The disk driver performs three major functions: device mapping, disk queueing, and interrupt processing.

The disk driver receives control as a result of one of two possible circumstances. Either the buffer manager has sent more buffers to be filled or emptied, or an interrupt has occurred indicating that an I/O operation has completed. In the first case, the requests are processed and placed in a queue. In the second case, any additional operations are started, and, if appropriate, filled or emptied buffers are returned to the buffer manager.

The device mapping function of the disk driver is primarily a translation from a system file index and a block number within a file to a channel and unit number, and a physical disk address. In the case of an output operation, if there is no block assigned to a block number (in other words, a new block is being written), a call is made on the allocation routine to allocate one.

Requests are placed in queues so that the interrupt routine can quickly select the next operation to be performed without having to search for one. Requests are queued by cylinder in shortest seek time first (SSTF) order. Within a cylinder, requests are queued by sector number and are issued in shortest latency time first (SLTF) order.

The interrupt routine has very little to do. It gets control whenever a sector has been transferred and after each positioning function completes. If the previous opera-

tion finished successfully, then it selects the next operation in the queue and initiates it. If the previous operation caused an error, no new operation is begun and control is given to an error processing routine. If the operation which completed was a transfer operation, then the filled or emptied buffer is sent back to the buffer manager.

## 5. Performance Mechanisms

There are two basic notions in improving I/O performance. The first is that I/O operations ought to proceed in parallel with computation in order to minimize the amount of time the program must wait for its data to be moved in or out of memory. The second is that the length of time an I/O operation takes should be reduced as much as possible. The first concept is the primary motivation for buffering; the second is the motivation for optimizing disk scheduling and for localizing references to small physical areas of a disk.

Suppose a simple model is chosen for I/O requests. In this model it is assumed that a single program running on the CPU is requesting consecutive sectors on the disk. The program requests a sector, performs some computation, and then requests the next sector. This model will be used to investigate the effects of buffering on I/O performace and to determine the feasibility of using system buffering to meet the performance goals.

The next two sections deal with buffering. The first discusses buffering in the disk controller; the second discusses buffering by the file system routines in main memory. The third section describes the disk allocation mechanism which helps achieve the bandwidth goal.

### 5.1. Disk controller buffering

The CRAY-1 disk controller contains enough memory to hold two 4096-byte disk sectors. On an input operation, data is first read from the disk into one of the buffers. This transfer takes place at disk speed (35 megabits/second). When the buffer is full, the data is transferred to main memory at channel speed (232 megabits/second). While this sector is being sent to the CPU, the next sector is being read into the other controller buffer. If a read operation is issued for the next sector before the sector following it passes the read/write heads, that sector may be moved into main memory with no delay. The controller will continue in this manner, reading one sector beyond the sector requested by CPU. Figure 3 illustrates the operation of the controller.

Using the model described above, suppose that the compute time per sector is a constant, C, and that C < S, where S is the time for one sector to pass under the read/write heads. In this case, only one memory buffer is required, since it is always available in time for the next sector to be read into it.

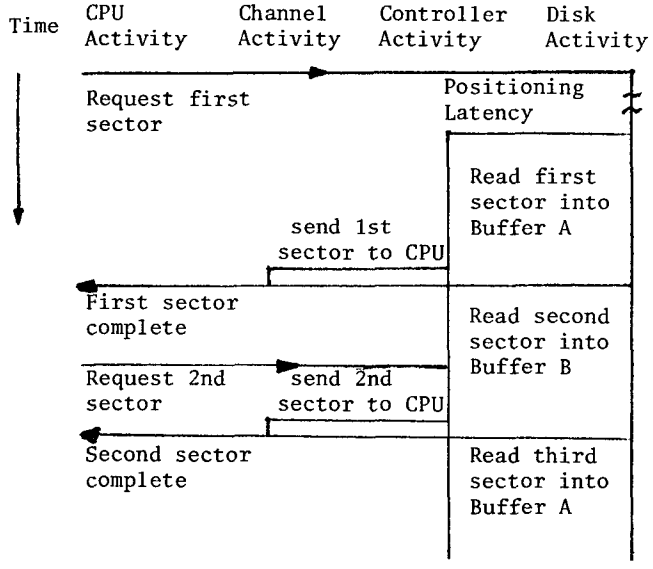


Figure 3: Operation of the Disk Controller

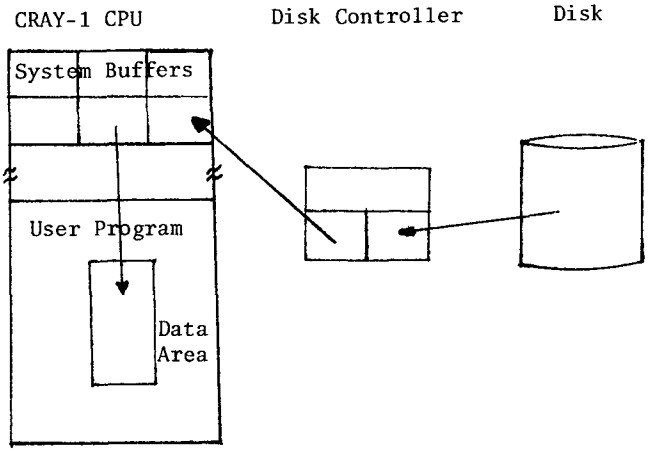


Figure 4: Two Level Buffering by Hardware and File System

In the case of many utility programs (for example, copy or translate a file), C is close to zero. Therefore, this buffering is sufficient to allow high transfer rates for this class of program. In the case of programs which perform non-trivial calculations on the data in each sector, two problems may arise. The first problem is that, although the average computation for each sector may be sufficiently small, the computation for some sector is too long. In spite of the buffering in the controller, this would mean that a revolution is missed whenever the compute time for a sector was too large. The second problem is that the average compute time per sector may be too large, and few consecutive sectors could be read without missing a revolution. These problems can often be alleviated by introducing buffering in main memory.

## 5.2. File system buffering

In the previous section, it was noted that programs with sufficiently small amounts of processing time for each sector could sustain very high data rates using the controller buffering. Since the data rates in the observed systems do not approach these high values even when the CPU is busy, it appears that frequently the average computation required for each sector is too great. In order to achieve high data rates in this situation, additional buffering is required. Figure 4 shows how this buffering works.

Data is first tranferred from the disk into a controller buffer. From there it is sent to a buffer in the system area until requested by the user. When the user performs a read operation requesting the data, it is moved from the system buffer into his memory field. After the system buffer has been emptied, it may be reused for buffering other sectors.

It is useful to know the minimum number of buffers required to achieve a given level of system performance. Suppose a program behaves in the following manner. Computation of C seconds is performed on each sector, C > S. There are N buffers which may be used to buffer data to the program. Suppose sector I is the last sector read from the disk. If sector I+1 is passing under the read/write heads and there is a buffer available, sector I+1 will be read into that buffer. Thus the system tries to keep the N buffers full with useful data. Figure 5 shows N versus CPU utilization for several values of C/S (using a Pareto distribution with infinite variance for C). The C/S ratio varies from 0.5 (the bottom curve) to 4.5 (the top curve). It is encouraging to note that the knees in the curve all occur below 8 buffers, indicating that a small number of buffers would be sufficient for reasonable performance.

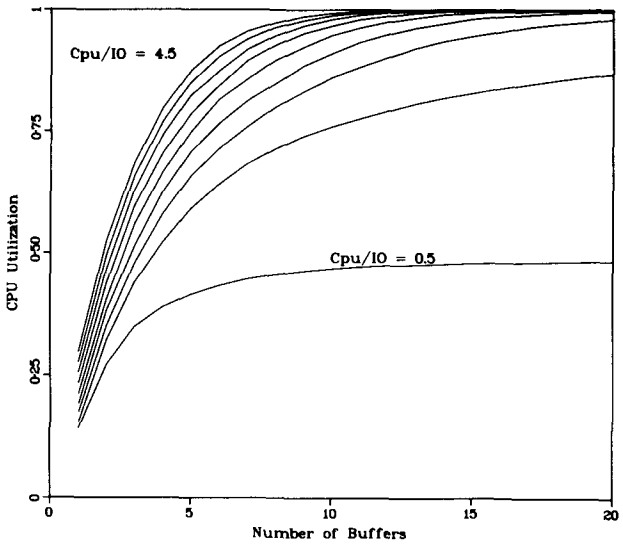


Figure 5. BUFFERS VERSUS CPU UTILIZATION (Pareto Cpu Requests)

In figure 6 are shown plots of C/S versus N at 90 percent CPU utilization. All three of the curves peak near 1. Below 1, the computation is limited by the speed of the disk, and increasing the number of buffers will not improve things. Far above 1, it takes more time for the program to empty the buffers than for the system to fill them, so it is not hard for the system to keep ahead of the program with only a few buffers. Figure 7 shows the same curves at 70 percent CPU utilization. The shape is essentially

the same, but the buffer requirements are much smaller. Although the higher CPU utilization may be required when only one job is running, lower values are probably adequate when multiprogramming. Of course, when only one job is running, proportionally more system buffers may be devoted to its files than when several jobs are in the system.

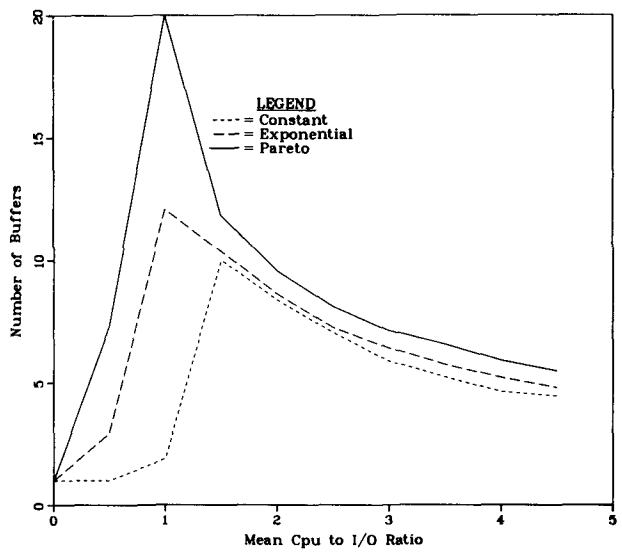


Figure 6. BUFFERS VERSUS CPU/IO RATIO
(Cpu Utilization = 0.9)

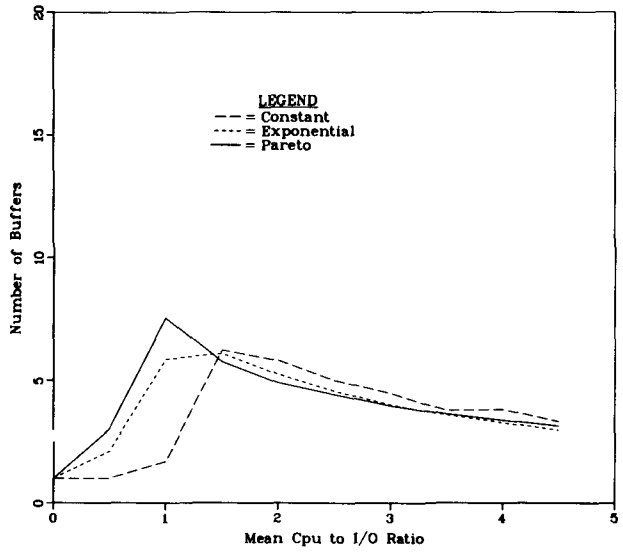


Figure 7. BUFFERS VERSUS CPU/IO RATIO
(Cpu Utilization = 0.7)

## 5.3. Disk space allocation

The above discussion has assumed that the data in a file is stored in consecutive sectors on the disk. To approach this ideal situation, it is necessary that the file system routines be able to allocate consecutive sectors to a file. This section describes how this may be accomplished in a flexible fashion which is transparent to the user.

The management of disk space can be critical to the performance of a file system. File space should be allocated and released efficiently. The most active files in the system are frequently the temporary work files which are created, used and destroyed in a short period of time. It is therefore necessary to perform the allocation quickly, yet allocate the file so that I/O may be done efficiently. When the file is destroyed, the space it occupies should be available for reuse as soon as possible. Disk space should be used effectively. Small files should not take up large amounts of space, and the disk space used by the file system routines to keep track of files and free space should be minimal.

In order that small files be economical, the allocation increment must be relatively small. Based on the file size data sited above, a disk track (18 sectors or 73,728 bytes) was deemed too large. Therefore a 4096-byte sector or block is used as the allocation unit. The block is the standard unit of data manipulated by the file system routines.

The information about how many and which blocks belong to a given file is stored in a file descriptor. To keep track of blocks which do not belong to any file, a bit map is kept with a bit corresponding to each block. If the bit is on, the block is in use; otherwise, the block is not currently allocated to some file.

Use of a bit map makes possible efficient decisions in allocating blocks to files. The most common criterion for selecting a block for a file is how close it is to other blocks in the file. Whereas keeping a list of free blocks would require searching the list in order to find a block which is close to the others, a bit map may easily be tested for free blocks among the set of good choices. Because of the controller buffering on the CRAY-1 disks, some blocks are definitely better candidates for allocation than others.

Most programs in this system manipulate files in one of two ways: either by reading the file sequentially or by writing it sequentially. Therefore, when a block is re-

quested by a program, the probability that it will shortly request the next block of the file is high. If the next block is stored in the sector following the requested block, the disk controller will automatically read it into its buffer. If that block can be transferred to main memory and saved until the program requests it, there will be no latency associated with the request for that block. To try to make use of this feature, the following allocation scheme is used for file blocks.

Suppose that data is being written on the end of a file, that is, that new blocks must be allocated to the file to hold the data. The choices for a sector to hold the new block according to the basic allocation strategy are as follows:

1) the sector immediately following the sector containing the previous block on the same track and cylinder as the previous block,

2) the sector which is two sectors after the sector containing the previous block on any track on that cylinder,

3) subsequent sectors on any track on that cylinder.

If no suitable sector can be found on that cylinder, then the file must be extended onto a new cylinder. The new cylinder is selected based on how close it is to the current cylinder, and how many sectors are available on the cylinder.

There are two possible problems with this basic allocation strategy. The first is that if more than one file were being written by a program at the same time, the blocks of the files would tend to be interleaved. The second problem is that in order to make the file system reliable, the file descriptors and allocation maps would have to be written to the disk each time a block is allocated. In order to alleviate these two problems, the basic allocation mechanism is augmented by a pre-allocation strategy.

The pre-allocation strategy is based on predicting future program behavior from its past behavior. As file blocks are allocated according to the basic allocation strategy, an indication is recorded of how fast blocks are being allocated to the file. Each allocation decision is matched by a pre-allocation decision. The pre-allocation decision is whether or not to pre-allocate a number of blocks to the file, and how many to pre-allocate. When blocks are pre-allocated, they are marked as being used in the allocation bit map, and added to the list in the file descriptor. However, in the file descriptor they are marked as being pre-allocated. The basic allocation strategy will always prefer a pre-allocated block over others. If the pre-allocation routine pre-allocates blocks frequently enough, the basic allocation strategy would never have to perform actual block allocation. Moreover, the increments of pre-allocation can be large enough to minimize the possibility of interleaving the blocks of two files.

## 5.4. Strategy routines

The term strategy routine is used for a subroutine or task which is not necessary for the correct operation of the system. In particular, if the strategy routines did nothing (other than terminate in a finite amount of time without errors) the system should still operate and give correct results. The purpose of a strategy routine is to improve the performance of the system, by monitoring system activity and "recommending" actions on the part of other routines.

Two examples of strategy routines are found in the DEMOS file system. The first is the read-ahead routine. Its function is to cause blocks which are likely to be used in the near future to be brought into memory. The second is the pre-allocation routine. Its purpose is to pre-allocate groups of blocks to files shortly before they are needed. The two routines operate in similar fashions. Control is passed to the strategy routine when certain system events occur. The strategy routine records such information as is appropriate and based on the new information as well as previously recorded information, it performs some action. That action usually will have an effect on the system. If it does not, because of some error or because the system is too busy, the correct operation of the system should not be affected.

### 5.4.1. Read-ahead strategy

The read-ahead routine receives control from the request interpreter each time a read request is received. The read-ahead routine examines the size of the current read request, the rate at which read requests are occurring for that file, and indicators of overall system I/O activity. Based on this information, it may select additional blocks of the file to be read, and request that the buffer manager obtain those buffers. If the buffer manager is short on buffers, or if the blocks do not exist in the file, then

the requests are merely discarded.

The read-ahead routine attempts to track the activity on a file and keep sufficient buffers full of data so that the user program rarely waits for I/O to complete. The metric of file activity is a sum of file requests which decays over time. As discussed earlier, the number of buffers required to sustain a given I/O rate may vary widely. The job of the read-ahead routine is not just to read-ahead, but to read-ahead an appropriate amount.

However, the bias of the read-ahead routine is to read too much. The reason for this is that the cost of reading too much is small. Assuming that files are generally contiguous, the additional sector-time to transfer an extra block is only 925 microseconds. Compared to the minimum seek time of the disk(15 milliseconds), this amount is almost negligible. The cost of occupying a buffer with extra data is also small, since "stale data" (data which was read but has not been used for a while) is easily discarded by the buffer manager.

### 5.4.2. Pre-allocation strategy

The pre-allocation routine receives control from the disk driver each time a new block is requested for a file. The pre-allocation routine examines the size of the current allocation request, the rate at which new blocks are being allocated to the file, indicators of overall system I/O activity and the amount of disk space available on the disk where the file is located. Based on this information, it may request the pre-allocation of some number of blocks to the file. If there is insufficient disk space for the file, the request is merely discarded.

The pre-allocation routine will receive some assistance from outside the file system. The file space on the CRAY-1 will be divided into several pools. One pool will be for resident user files, another for resident system files, one for swap space, and another for temporary work files. In this manner, the highest activity files will be located away from the typically small user files which tend to fragment the disk. Files in the temporary work file pool will have a short lifetime, and a relatively high percentage of free space will be maintained in that pool. Therefore, it should be easier to find large contiguous groups of blocks. The ability to migrate files back to the Common File System will provide another mechanism for assuring sufficient free space to avoid fragmentation and to allow the pre-allocation routine to allocate contiguous blocks. For files resident for longer periods of time, the disks will be re-organized during periods of low activity to reduce the fragmentation of files.

## 6. Conclusion

Adding strategy routines to a file system is like adding a cache to a memory. It is desirable to have a general-purpose file system or a large memory, but it is necessary to have the performance of a special-purpose file system or a small memory. With both, it is possible to come close to optimal performance by using some property of the request distribution to develop an *ad hoc* solution which works well a large percentage of the time. In this case, that property is the sequential nature of most file operations, as well as the ordered arrangement of the data on the secondary storage device.

DEMOS is being developed in an evolutionary manner from the vendor-supplied operating system for the CRAY-1. The file system in that operating system is similar in its lack of sophistication to the file systems mentioned earlier. It offers a useful benchmark against which the performance of the DEMOS file system will inevitably be compared.

It is hoped that this kind of implementation will make it possible to have more of the general-purpose capabilities found in other systems to be included in high-performance operating systems.

## 7. Acknowledgements

I must express my gratitude to Forest Baskett, J. C. Browne, John H. Howard, and John T. Montague, whose well-considered suggestions and criticisms improved not only the presentation of the ideas in this paper, but also the likelihood that they will work.

## 8. References


Amdahl, G. M., "Storage and I/O Parameters and Systems Potential", Proceedings of the IEEE Computer Group Conference, June 16-18, 1970, Washington, D. C., pp 371-372.

Bobrow, D. G., Burchfiel, J. D., Murphy, D. L., and Tomlinson, R. S., "TENEX, A Paged Time Sharing System for the PDP-10", CACM 15, 3 (March 1972), 135-143.

Collins, M. W., LASL CCF Mass Storage Requirements. Los Alamos Scientific Laboratory internal memo. (December 1975).

Cray Research, Inc., CRAY-1 Computer System Reference Manual, Publication 2240004, 1977.

Feiertag, R. J. and Organick, E. I., "The Multics Input-Output System", Proceedings of the Third Symposium on Operating Systems Principles, Oct 18-20, 1971, ACM, New York, 35-41.

Keller, T. W., Measurement data taken at Los Alamos Scientific Laboratory, (March 1977).

Keller, T. W., CRAY-1 Evaluation Final Report. Los Alamos Scientific Laboratory. Informal report. LA-6456-MS (December 1976).

Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", CACM 17, 7 (July 1974), 365-375.

Sloane, L., Measurement data obtained at Lawrence Livermore Laboratory. (February 1977).