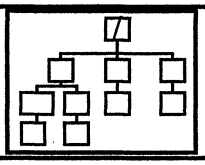
Data General

DG/UX Technical Brief

Number 7
July 31, 1991

Information About AViiON® Systems from Data General's UNIX™ Development Group



In This Issue:

The DG/UX™ 5.4 File System

Contents

How does the DG/UX 5.4 File System work? What are its features and benefits? How does the new demand-paged file I/O model work? How does the File System store files? What are logical disks, DARs, and inodes? What can I do to maximize File System performance?

After you've read this technical brief, you should have an understanding of how the DG/UX 5.4 File System interacts with application programs, the Virtual Memory Manager, the Logical Disk Manager, and disk systems. This new technical brief replaces technical brief number 3, which discussed the features of the older DG/UX 4.3x File System.

The section "Understanding the File System" provides background information about the DG/UX File System and Logical Disk Manager. We talk about basic concepts and review some terminology. We talk first about how the File System works with physical disks, then talk about logical disks. The discussions point out terminology differences between the DG/UX implementation of the File System and other UNIX implementations.

In the "Tips for Tuning a File System" section, we talk about File System parameters that you can change. We also provide some benchmark data to show how applications react to changes in File System parameters. The first FYI section describes some of the unique features that the DG/UX 5.4 operating system provides to support commercial applications. The second FYI section provides more information about the operating system's new demand-paged file I/O model.

Produced on a Data General
AViiON AV4000 with FrameMaker® 2.1X.

012-004054-00

Introducing the DG/UX 5.4 File System

Figure 1 is a high-level view of the DG/UX 5.4 File System. The figure shows how the File System connects to the disk I/O system and where file addresses (in italics) are translated from pathnames to locations on a physical disk.

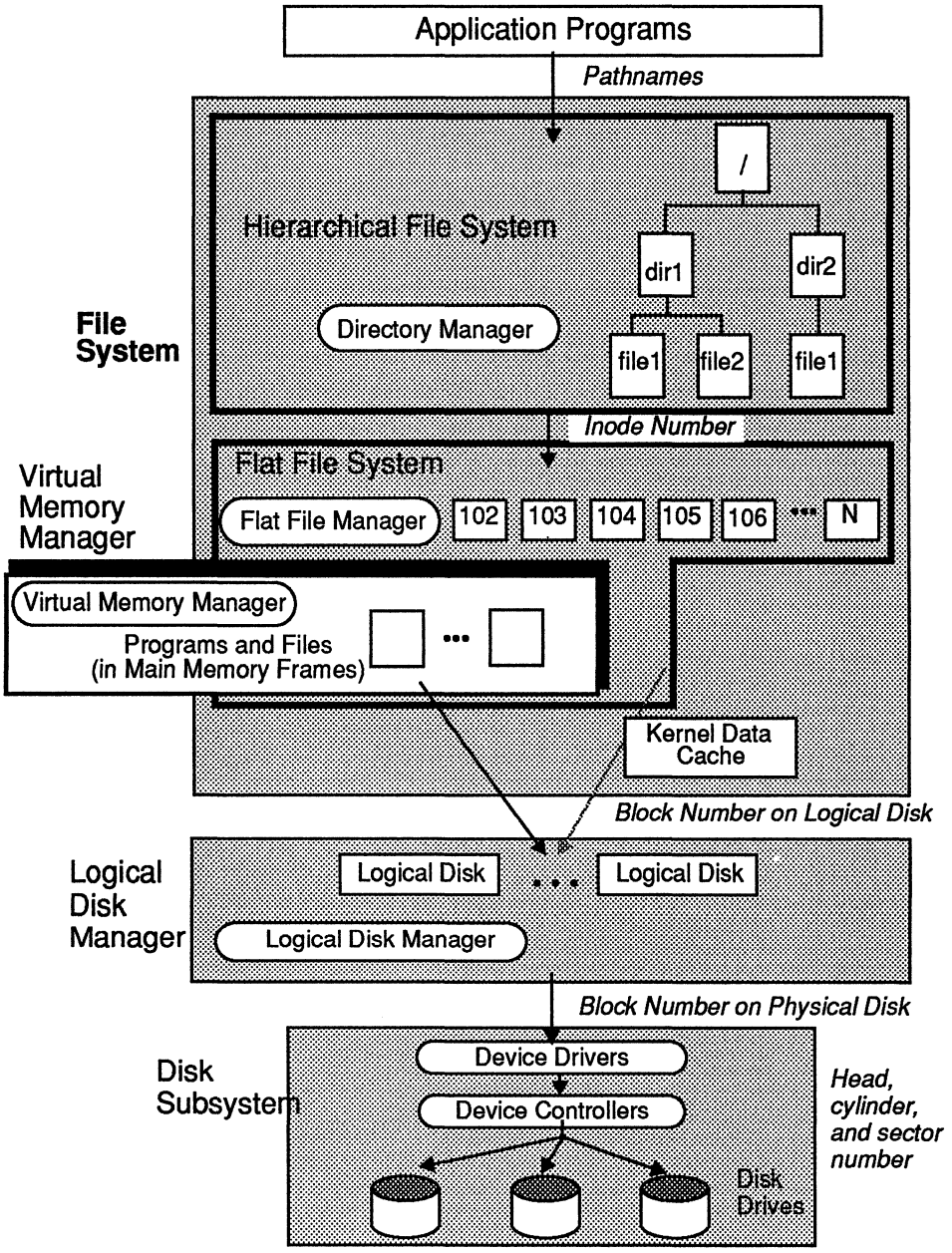


Figure 1 Overview of the DG/UX 5.4 File System and Virtual Memory Manager

Key Points

The DG/UX 5.4 File System supports the features of the classic UNIX file system, the Berkeley (BSD) Fast File System, and the DG/UX 4.3x file system. However, the DG/UX 5.4 File System is more generalized than the BSD file system and provides enhancements to work with the DG/UX multiprocessor architecture.

In the DG/UX 5.4 operating system, application programs and files as well as kernel programs are demand paged in main memory by the Virtual Memory Manager. In this model, virtually all of a computer's memory is treated as a cache for programs and files—unlike other UNIX implementations that use Virtual Memory for programs and a buffer cache for files. The DG/UX 5.4 operating system uses the kernel data cache only for file system metadata (Figure 1). File system metadata is "data about data"—data that the File System uses to describe and locate files.

Using the DG/UX 5.4 operating system's Virtual Memory Manager to handle both programs and files provides for more efficient use of a computer's main memory resource. The Virtual Memory Manager can adjust dynamically the memory resources to reflect the demand of a mix of applications. The result is improved File System performance without the need for trying to determine the set of system parameters, such as buffer cache size, that provides the best performance for a range of applications.

Some of the other key points about the File System are listed below. It's important to note that these features are transparent to applications that use the File System.

- You can create file systems on either logical disks or on physical disks. You will almost always want to create file systems on logical disks. Logical disks essentially isolate a file system from the implementation details of the underlying physical disk subsystems and provide flexibility in how you use a computer system's physical disk resources.
- To an application, a logical disk looks just like a physical disk. However, a logical disk can be as large as 2 Tbytes (2 Terabytes) and can span as many as 32 physical disks.
- The sizes of file systems are limited only by the size of the logical disk or physical disk on which a file system is created. A single file in a file system can be as large as 2 Gbytes.
- The File System supports mirroring of logical disks as well as data striping (interleaving). Striping distributes data from large files among two or more physical disks so that programs can access multiple physical disks concurrently.
- The File System divides disk space into Disk Allocation Regions (DARs). DARs enhance File System performance by keeping a file's data blocks and the pointers to its blocks close together, to minimize disk latency.

- The File System is responsible for translating file pathnames into addresses that a disk drive understands (block numbers).
- When you create file systems, you'll find that the default parameters work well with a range of applications. However, you have the option of changing several kernel and File System parameters to maximize the performance of your file system.
- You can move the contents of file systems to and from other UNIX systems by using the standard transfer utilities, such as **tar**. In addition, you can mount NFS file systems, DOS file systems, High Sierra, and ISO-9660 (CD-ROM) file systems.

Terminology

Here are some terms that we use in this technical brief.

Disk sectors
: The granularity at which disk drives and controllers work. Disk sectors are a fixed size; in AViiON systems a disk sector is 512 bytes. Also called disk blocks. (Sectors for other random access devices, such as optical disks, can have different sector sizes.)

Data block
: If we're talking about a block of data that is stored in a logical disk, we use the term *data block*. Data blocks are 512 bytes, and are a multiple of the underlying physical disk's sector size.

Data elements
: The logical granularity at which the DG/UX File System transfers files' data. You can set the size of a file's data elements from 512 bytes to several megabytes. The default data element size on DG/UX systems is 8 Kbytes, which is sixteen 512 byte disk sectors.

Frame
: A memory container that holds a page or part of a page. In AViiON computers, the frame size is 4 Kbytes.

Page
: The logical granularity at which the DG/UX Virtual Memory system allocates and transfers data. Page size must be a multiple of frame size. In DG/UX 5.4, the page size is 4 Kbytes (1 X frame size).

One other comment about terminology—in this technical brief, we use the term *File System* (with capital letters), when we're talking about the general concepts of the DG/UX 5.4 File System. We use the term *file system* (without capital letters), when we're talking about a specific instance of a file system—one that you've created.

Understanding the File System

The DG/UX File System is responsible for managing all files that are stored on disk—application program files as well as the operating system's files. This responsibility includes managing files' characteristics, such as access privileges, and translating file pathnames to addresses in physical or logical disks.

You typically create file systems on logical disks. When you create a file system on a logical disk, the Logical Disk Manager translates logical disk addresses (block numbers in a logical disk) to physical disk addresses (block numbers in a physical disk) and passes the physical disk addresses to the disk I/O system. The Logical Disk Manager is a pseudo-device driver—part of the DG/UX kernel. If you create a file system on a physical disk, the services of the Logical Disk Manager aren't used, and the file system passes its addresses directly to the disk I/O system.

We'll describe first how file systems work with physical disks, because the same concepts apply to file systems that you create on logical disks. Then, we'll talk about the unique aspects of logical disks. We'll examine the different aspects of the File System and Logical Disk Manager in the following order:

- Files and file systems
- How files are stored
- Accessing files
- Logical disks
- The File System and the Virtual Memory Manager
- Access modes for files
- Block I/O and character I/O

An Application's View of Files and File Systems

To a user program, a file has a name, a size, and other properties such as access flags and date/time stamps. Files can contain data, executable programs, shell scripts, or a list of information about other files (a directory). Programs access files with operating system calls such as **creat, open, read,** and **write.**

Files are stored in file systems. You can mount file systems to enable programs to access files within those file systems.

Within a file system, files are organized in a hierarchical, inverted tree structure. The top of the inverted tree is the root, which is represented by the slash character (/) (Figure 2).

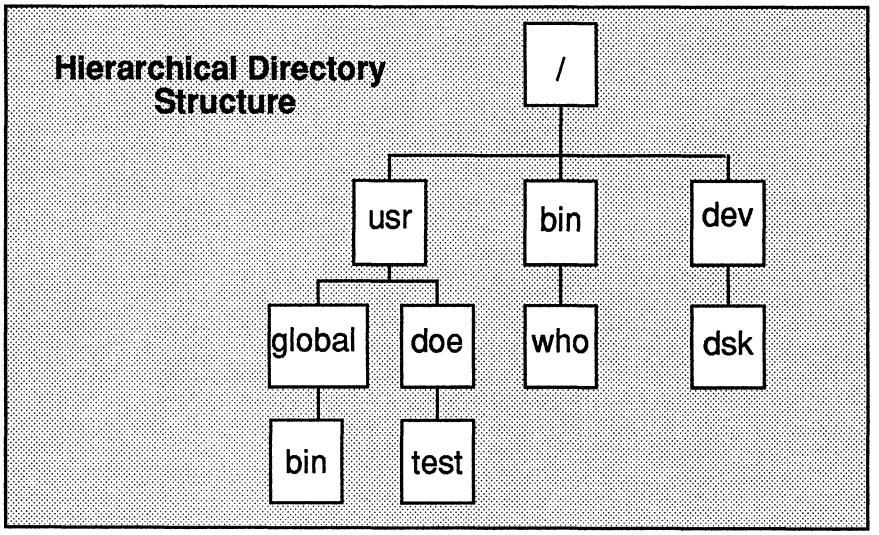


Figure 2 Files in the DG/UX Hierarchical File System

User programs refer to files by pathname, starting with the root. For example: **/usr/doe/test**. Intermediate names in a path, such as **usr** and **doe** are directories (nodes in the tree). Directories are files that contain a list of the files that are stored under them in the tree. Because directories themselves can contain other directories, a tree can contain many levels.

Inside the File System

The DG/UX kernel's Directory Manager creates and manages the user-visible hierarchical file structure. However, the File System doesn't work directly with the hierarchical structure—the File System works with a logical flat file structure, where all of a file system's files are in one "directory." This flat file structure (Figure 3) is invisible to application programs.

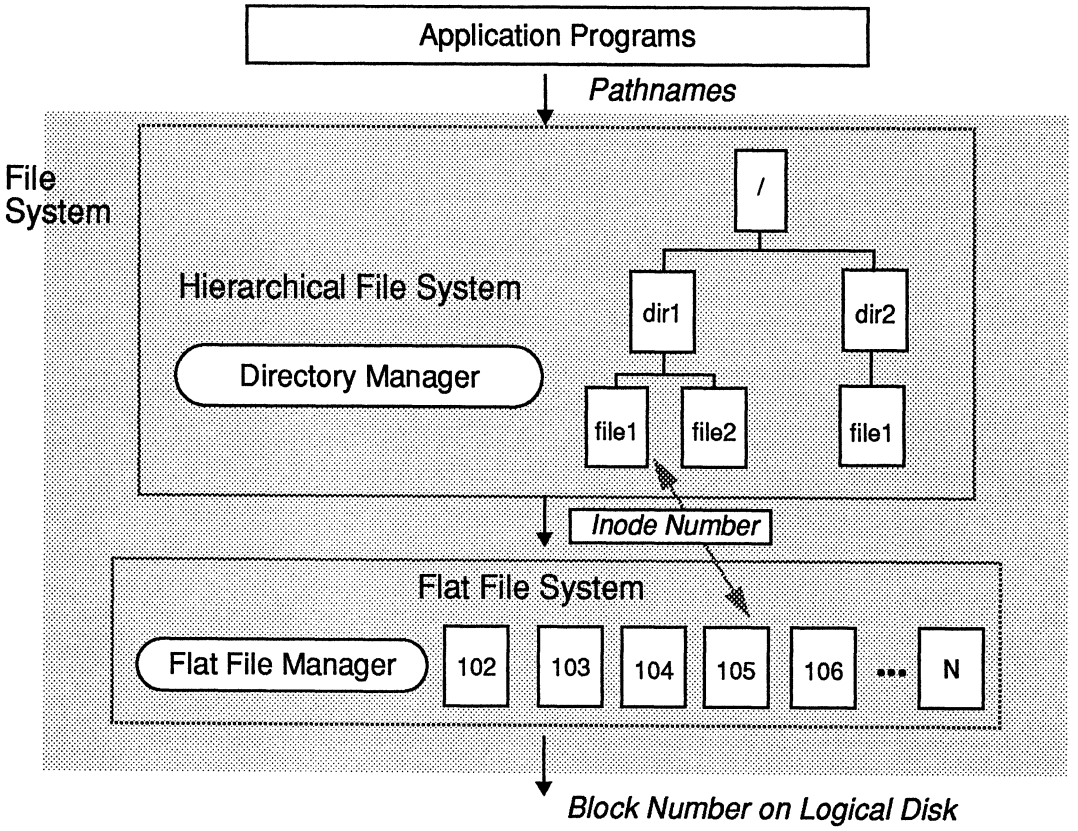


Figure 3 Directory Manager and Flat File Manager

The Flat File Manager is responsible for providing and managing the flat file model. In a file system's flat file structure, each file is identified by a unique index node (*inode*) number.

When the Directory Manager receives a user program's request to access a file, the Directory Manager converts the file's pathname into an inode number and passes the inode number to the Flat File Manager. For example, in Figure 3, **file1**, which is under **dir1**, might be stored as a file with inode number 105 in the flat file system.

Within each inode are pointers to the location of a file's data blocks. (We talk more later about how inodes point to a file's data blocks.)

The File System's View of Disk Storage

To the File System, disk storage, whether logical or physical, appears as a contiguous set of disk blocks. Superimposed on this contiguous set of disk blocks (Figure 4) are a File Management Information Area (FMIA), Disk Allocation Regions (DARs), a DAR Entry Table, and a backup copy of the FMIA.

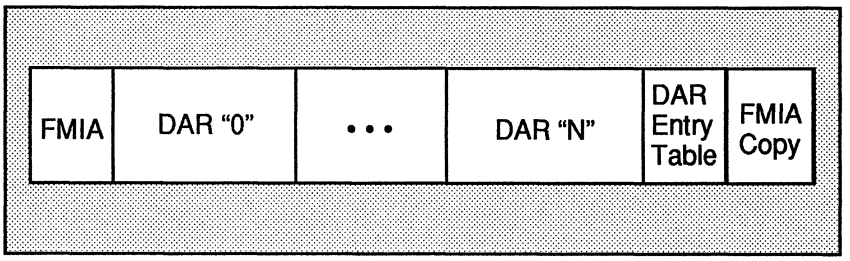


Figure 4 Layout of a File System

The FMIA, which is equivalent to the "superblock" in BSD and AT&T UNIX systems, contains information about the file system, including the DAR size, the number of inodes per DAR, and the default data element size for files and directories. The FMIA is stored at the beginning of each file system. A backup copy of the FMIA is stored at the end of a file system, along with a DAR Entry Table. The DAR Entry Table contains information about each DAR, including how many inodes and data blocks a DAR is using.

To increase file system performance, the rest of a file system is divided into Disk Allocation Regions (DARs). To access a file, the File System alternately reads a file's inode (to find where the file's blocks are stored) and the blocks themselves. By using DARs, a file system can keep a file's data blocks and inodes physically close together, which minimizes physical disk mechanical latency (seek time).

Figure 5 shows a file system with an exploded view of a DAR, which consists of:

- a bit map, which tells the Flat File Manager how the blocks in a DAR are being used. The bit map has an entry for each block in a DAR, including itself, the inode blocks, and the data blocks. For example, as files are created and deleted or grow and shrink, the data blocks in a DAR are used to store new blocks, or will become unused. The DAR's bit map reflects this changing usage.
- a table of inode slots—one inode slot is used for each file that is referenced from the DAR.
- blocks that contain files' data and index elements (discussed later).

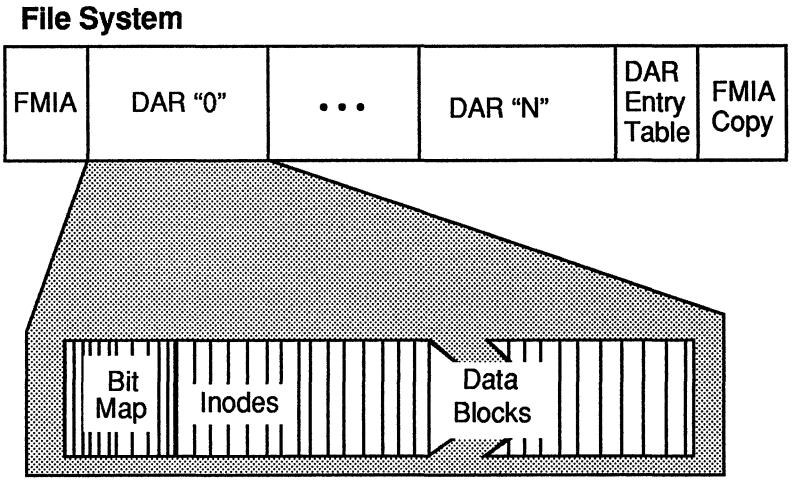


Figure 5 Inside a Disk Allocation Region

When you create a file system, you have the option of specifying the file system's DAR size (how many disk blocks will be in the file system's DARs). You can also specify the number of inodes in a file system, which establishes the number of files that a DAR can contain. The default is one inode for every 3,500 usable data bytes in a file system.

If you know that an application is going to create files with large contiguous allocations, you can create fewer DARs with larger sizes. Conversely, if your application is going to create mostly small files, you can create more (and smaller) DARs. The smaller DARs will keep the small files' data blocks and inodes close together.

Note that the last DAR in a disk will probably be smaller than the other DARs. For example, if you create a 1 Gbyte logical disk and set the DAR size to 10,000 disk blocks (5 Mbytes), the logical disk is divided into 200 DARs. The last DAR will be slightly smaller than 5 Mbytes to make room for the backup FMIA and the DAR Entry Table.

What happens if a file won't fit into a DAR? It's possible that a file can grow too large to fit within its DAR, or that a newly created file won't fit within one DAR. In these cases, the File System can automatically use parts of other DARs to store some or all of a file. The algorithm that the File System uses to store files in other DARs is based on a DAR's free space and a file's "anniversary sizes," which we describe later in the "Tips for Tuning a File System" section.

Accessing Files

In this section, we'll answer two questions that relate to how files are accessed in the DG/UX File System:

1) What are data elements?

2) How are inodes used to access files?

About Data Elements and Transfer Sizes

The DG/UX File System works with a granularity of 512 byte blocks, which is the same size as disk sectors on physical disks. However, it's inefficient for the File System to read and write data to the disk in these small blocks. On the average, it takes about 25 milliseconds for a disk to position its heads over a disk sector. Once a disk's heads are positioned, the File System might as well read more data from that area of a disk—anticipating that a program will be asking for the data next. Therefore, files are allocated using an aggregate of contiguous 512 byte data blocks called a file system *data element*. By default, a data element is 8 Kbytes (16 disk blocks). On a per-file basis, you can choose a data element size of 512 bytes to just less than the size of a DAR.

A data element is stored in contiguous disk blocks on a logical disk. However, data elements need not be stored contiguously within a DAR. Figure 6 shows an example of how a 32 Kbyte file might be stored in a DAR. We've assumed that the data element size is 8K bytes. Therefore, the file is stored in four data elements and each data element has in it 16 contiguous data blocks.

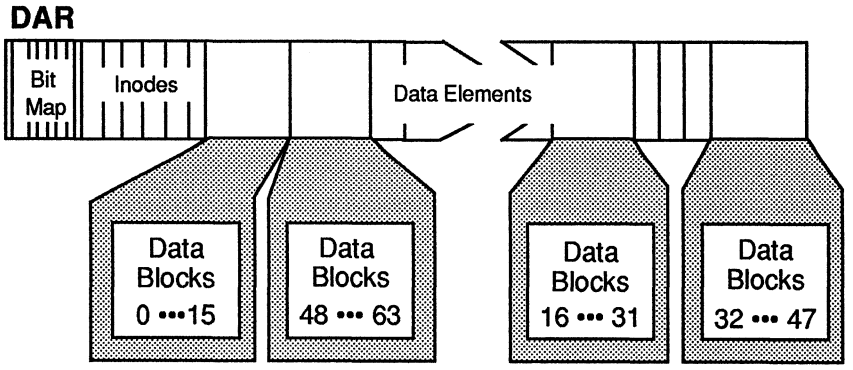


Figure 6 Data Blocks in Data Elements

For data element sizes up to 16 Kbytes, the File System will read, write, and cache data using the data element size. However, for data elements greater than 16 Kbytes, the read operations, write operations, and Virtual Memory caching are performed at the File System's 16 Kbyte *transfer size.*

For example, if a file's data element size is 64 Kbytes, each data element in that file contains 128 contiguous data blocks, while transfers are carried out in 16 Kbyte chunks.

When you create a file system, you can specify the file system's default data element size, which is used when files are created. When you create a file, you can specify a data element size different from the default. However, you can't change a file's data element size once the file has been created. We talk about the effects of changing data element size in the "Tips for Tuning a File System" section.

The File System also provides a facility to manage small files. Many files that the operating system and application programs use are smaller than the default 8 Kbyte data element size. When a program is modifying relatively small files, it's expensive to store on disk large data elements—it wastes I/O bandwidth and disk space. To conserve bandwidth and disk space, the File System uses *fragments* as a way of storing small files on a disk.

There are five fragment sizes (512 bytes, 1, 2, 4, and 8 Kbytes). Whether the File System stores a fragment instead of a data element depends on the size of the data element and the size of the file. The algorithm says "if a file is smaller or equal to half the size of the data element, the file is transferred and stored in the smallest possible fragment size, up to the maximum fragment size of 8 Kbytes."

Here are some examples. If the data element size is 8 Kbytes and an application wants to write a 1.5 Kbyte file, the File System will store the file in a 2 Kbyte fragment. If the data element size is 4 Kbytes and an application creates a 2.2 Kbyte file, the File System stores the file in 4 Kbytes. If the data element size is 32 Kbytes and an application wants to write a 12 Kbyte file, the file is stored in 32 Kbytes.

About Inodes

In the DG/UX File System, the key to accessing files is the inode. Pointers in an inode tell the Flat File Manager where a file's data elements are stored.

There is exactly one inode for each file in a file system. In addition to element pointers, an inode contains all of the other information that the File System needs to know about a file. An inode includes information about a file's data element size, access privileges, and last-modified times. When you use the **ls -l** command, the information that is displayed about the files in a directory, except the files' names, is coming from the files' inodes.

The inodes in a DAR (on disk) are a fixed size—124 bytes—and are stored four to an inode table block. Because inodes are small, they can't point directly to all of the data elements in a large file. (Remember that a DG/UX file can be as large as 2 Gbytes.) In fact, only the first ten data elements of a file are pointed to directly from an inode (Figure 7).

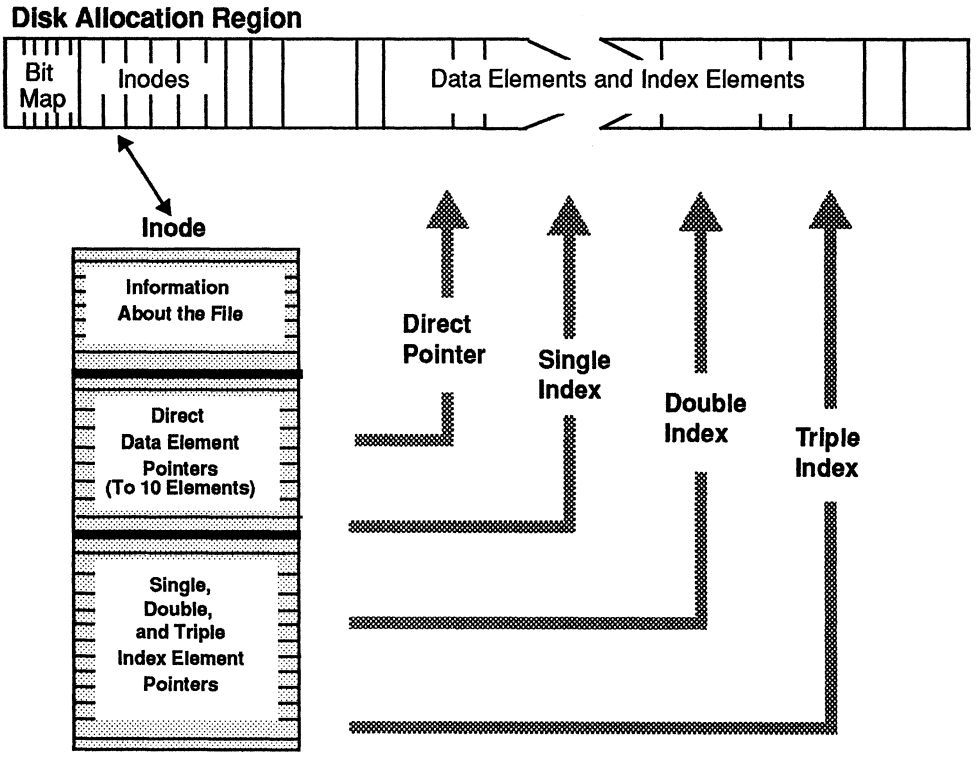


Figure 7 Locating a File's Data Blocks With an Inode

Both the DG/UX operating system and DG/UX applications often use small files as temporary files. The File System can locate quickly the data elements of small files by using inodes' direct pointers.

The remaining entries in an inode are pointers to *index elements*. Index elements, which are stored in the DAR along with data elements, contain pointers to the data elements in a file that aren't referenced with direct pointers. By using index elements, which can contain single, double, and triple indirect pointers, a small inode can reference all of a large file's data elements. For example, to reference a data element through a single indirect pointer, the Flat File Manager reads from disk the inode, then the index element, then the data element.

When you create a file system, you can change the default size of its data elements and index elements. We talk about that in the "Tips for Tuning a File System" section.

The File System caches inodes in virtual memory, which speeds access to files that are used often (such as the login password file **/etc/passwd**). Memory-resident inodes are 512 bytes long. When the File System reads an inode from the data cache, it calculates addressing information and puts the information into the expanded memory-resident inode.

The inode cache contains all of the inodes for files that are in use, and some number of recently used inodes. The size of the inode cache is controlled by a kernel parameter called the *free inode ratio*, which is the number of inodes that are in use, divided by the number of recently used inodes that are kept in the cache. The default free inode ratio is four. However, the minimum number of free inodes is the number of inodes that will fit in 2% of main memory.

For example, a computer with 16 Mbytes of memory will have, as a minimum, 640 free inodes ((2% * 16 Mbyte)/512 bytes per inode). For file systems with larger inode usage, say 10,000 inodes, there will be a minimum of 2,500 free inodes (using the free inode ratio of 4-to-1, which is larger than the 2% minimum).

By the way, to maintain compatibility with other UNIX operating systems, inode numbers in DG/UX file systems start at 2, which is always a root directory for the file system—there are no inodes 0 or 1.

File Systems and Logical Disks

The DG/UX operating system's *logical disks* are software abstractions that enable the File System to manage files the same way, regardless of how the files are stored physically. To the operating system, a logical disk is a collection of contiguous 512 byte disk blocks.

In the DG/UX operating system, you can create a logical disk from as many as 32 pieces. A *piece* is some part of a physical disk, which can be as large as a physical disk (currently 1 Gbyte for single-spindle disks and 30 Gbytes for the High Availability Disk Array (HADA) disk systems. A logical disk can use more than one piece from one physical disk, or pieces from 32 physical disks (Figure 8).

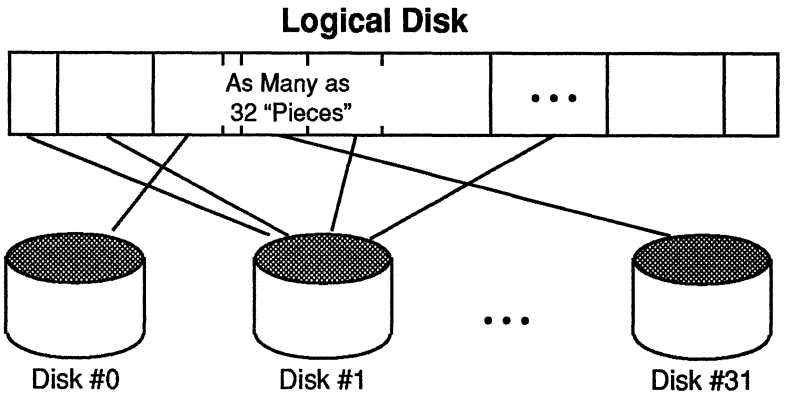


Figure 8 A Logical Disk, Built on Pieces of Physical Disks

Here is a summary of the rules that govern file systems and logical disks:

- A logical disk can contain only one file system.
- If a logical disk contains a file system, the file system will occupy the entire logical disk.
- A logical disk can use pieces from as many as 32 physical disks.
- A piece must reside on one physical disk—it cannot span physical disks.
- A physical disk can contain many pieces of many logical disks.
- Logical disks can be configured to support data striping (for increased read performance) and mirroring (for increased data availability).

With these configuration rules and 1 Gbyte disks, an AViiON computer could be configured with 2 30-Gbyte HADA disk systems and 28 1-Gbyte SCSI disks for a total of 88 Gbytes of storage. The theoretical addressing limit for logical disks is 2 Tbytes (2^{41} bytes), which exceeds the physical storage capacity of most other UNIX implementations.

Striping Logical Disks Across Physical Disks

In the DG/UX 5.4 and DG/UX 4.3x File Systems, you can arrange pieces of physical disks so that large files are distributed across the disks. This enables you to distribute accesses among the physical disks, instead of accessing all of a large file's data from one disk—which can become a bottleneck. In other words, you can place data blocks "1 to m" of a logical disk on one physical disk and "m+1 to n" on another physical disk, and so on.

The DG/UX 5.4 File System adds support for data striping, also called interleaving. Striping is another, more finely granular, way of distributing a logical disk's data blocks among different physical disks. Data striping can provide several benefits, which are listed below.

- Striping enables you to balance the I/O load across several physical disks.
- Striping can help processes avoid disk bottlenecks.
- Striping can improve the I/O performance of a single process by enabling the File System to perform read operations on one disk while it performs read ahead operations from other disks.

Figure 9 shows a simple example of striping with four physical disks. In the example, we've shown a *stripe unit* size of 64 blocks (32 Kbytes). Suppose that you were reading a large file that was stored in this striping configuration, with a data element size of 16 data blocks (8 Kbytes). In this example, there are 4 data elements stored in each stripe unit. Therefore, the first 4 data elements of the file might be stored on physical disk 1, the next 4 data elements on disk 2, and so on.

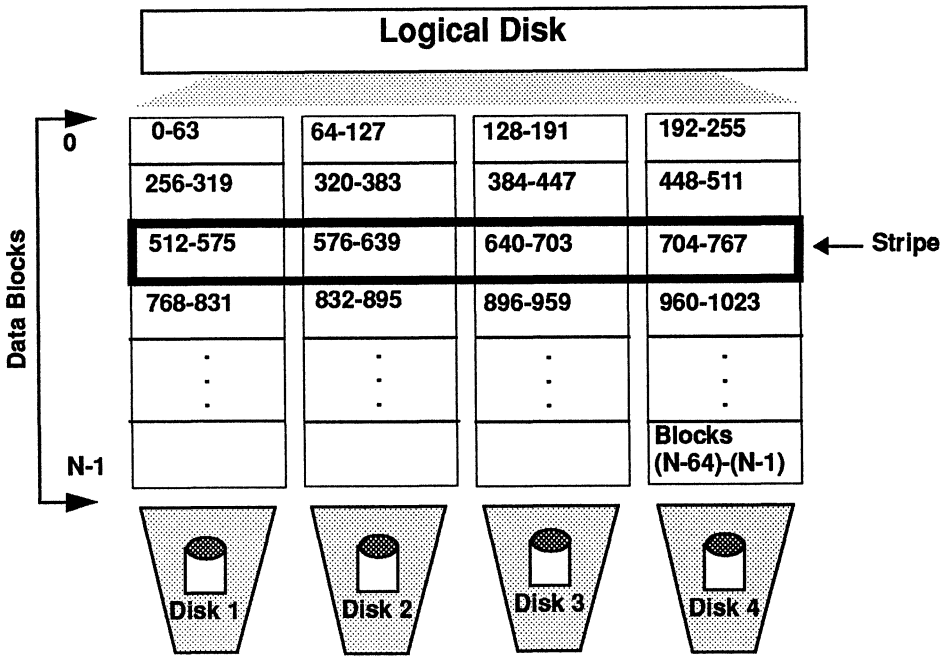
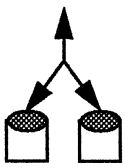


Figure 9 Data Striping Across Physical Disks

Configuration Notes for Striping.

- Striping may not automatically improve performance in multi-process or multi-file application mixes. If the applications are already performing relatively random file accesses, or if I/O operations are already balanced among physical disks, striping will probably offer no advantage.
- The Data General RAID 5 high availability disk systems (HADA and CSS2) support striping within groups of disk modules in an array of disks. The HADA and CSS2 disk systems use striping automatically as a way of increasing read performance and providing access to data if a single disk module in a group of disks fails. If you have a high availability disk system, you should not configure striping at the logical disk level. (RAID 5 disk systems are described in DG/UX Technical Brief Number 6, "A Look at High Availability Disk Systems" (012-004035)).

Mirroring Logical Disks on Physical Disks



Logical disk mirroring is a software technique that maintains identical copies of logical disks on separate physical disks. Disk mirroring is important for applications that can't afford to lose the ability to access data if a disk fails.

By mirroring a logical disk with pieces from different physical disks, you're able to access the logical disk's data, *without interruption*, if one of the mirrored physical disks fails. If a disk fails, the failure is reported to the operating system, which sends a message to the console and to the operating system error log. An application using the data is not affected at all.

Logical disk mirroring dynamically remaps bad blocks among mirror copies. If a bad block develops on one physical disk in a mirrored pair, the mirroring software reads the bad block's data from the other disk and maps it to a good block. See the "Dynamically Remapping Bad Blocks" section at the end of this brief for more information about bad block remapping.

The DG/UX 5.4 operating system performs software mirroring at the Logical Disk Manager level, and you can have up to three mirror copies. Two copies is the most common case—you have a copy of your data in case a block goes bad or if one of the mirrored physical disks fails.

With three-copy mirroring, you can split off one mirror copy—in effect taking a snapshot of your file system. You can make a tape backup of the snapshot while applications continue to run, without losing the security of mirroring during the backup operation. (Before you split off the mirror copy, you should be sure that all of the file system's data has been flushed from memory.)

Logical disk mirroring is independent of the kinds of underlying physical disks. At the logical disk level, you can create (with the **diskman** utility) mirrored disks on different kinds and sizes of physical disks. And, you can create mirrored logical disks of exactly the size that you need to store some critical data.

Configuration Notes for Mirroring

As with data striping, the DG/UX 5.4 operating system supports disk mirroring at the hardware level with the HADA and CSS2 high availability disk systems. If your system has a high availability disk system, you'll probably want to use its facilities for mirroring. An exception could be if you need to mirror only small amounts of data—software mirroring enables you to specify the size of the mirrored pieces; hardware mirroring requires that you mirror identical physical disks.

The DG/UX 5.4 Demand-Paged File I/O Model

One of the most significant features of the DG/UX 5.4 operating system is its use of a demand-paged file I/O model to handle programs' I/O requests, instead of the file buffer cache that is used in traditional UNIX operating systems.

Comparing the 4.3x and 5.4 File Systems

In the DG/UX 4.3x operating system (the left side of Figure 10), the file buffer cache stored both file data *and* kernel metadata (such as inodes, index elements, and directory information). The size of the buffer cache memory was a fixed size that you specified when you built the kernel. Memory allocation for programs was managed separately, by the Virtual Memory Manager.

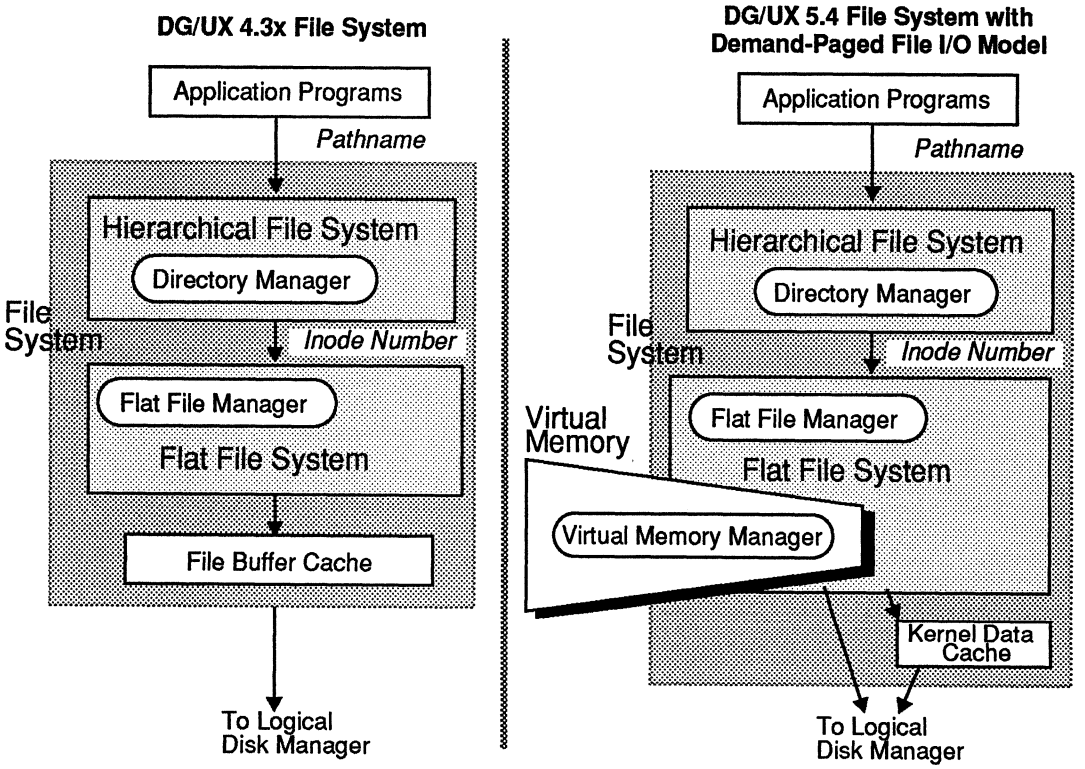


Figure 10 Comparing the DG/UX 4.3x and DG/UX 5.4 File Systems

The most significant change in the DG/UX 5.4 operating system's I/O model (the righthand part of Figure 10), is that files are no longer stored in the buffer cache. Instead, the Flat File System and Virtual Memory Manager work together to access and cache files as well as user and kernel programs.

In DG/UX 5.4, the kernel's Virtual Memory Manager treats all memory the same, whether it is used for programs, files, or stacks. In this model, essentially all of a machine's main memory is viewed as a cache for demand paging.

Note that a buffer cache and Buffer Manager are still present in the demand-paged file I/O model. Unlike application programs, the kernel doesn't pre-allocate memory for its data structures, so the Buffer Manager is still responsible for managing the small amount of pooled memory (disk buffers) that the kernel uses to perform I/O operations on file metadata. However, the kernel data cache is very small, because it is used only to cache kernel metadata (unlike the 4.3x buffer cache, which used 20% of main memory by default).

Advantages of Demand-Paged File I/O

The principle of Virtual Memory is that it presents to programs the illusion of almost unlimited memory. The Virtual Memory Manager creates this illusion by mapping, on demand, virtual (logical) memory addresses into a machine's main memory.

The Virtual Memory Manager looks at main memory as a group of 4 Kbyte frames. When a program accesses some data (with a **read** or **write** system call), the Virtual Memory Manager (VMM) checks to see if the logical pages that contain the data are already resident in main memory frames. If the pages aren't in memory, the VMM reads them into memory.

From this description, you can see the similarities between how Virtual Memory works and how traditional UNIX File Systems use a buffer cache. However, the demand-paged file I/O model provides for more efficient and flexible use of a computer's main memory resource by:

- Supporting "dynamic file caching" in memory, and
- Enabling application programs to use the DG/UX 5.4 memory mapping system call (**mmap**).

Dynamic File Caching

With a traditional fixed-size buffer cache, there are always situations when there is no "right" cache size for a mix of application programs. In the DG/UX 5.4 operating system, the Virtual Memory Manager dynamically balances the ratio of resident pages that contain files to resident pages that contain programs. The more a program uses a file, the more of the file's data will reside in main memory.

The demand-paged file I/O model enables a File System to maintain high transfer rates for repeated accesses to relatively large files. Figure 11 shows relative I/O transfer rates versus file size for a File System that uses a buffer cache and for the DG/UX 5.4 demand-paged file I/O model.

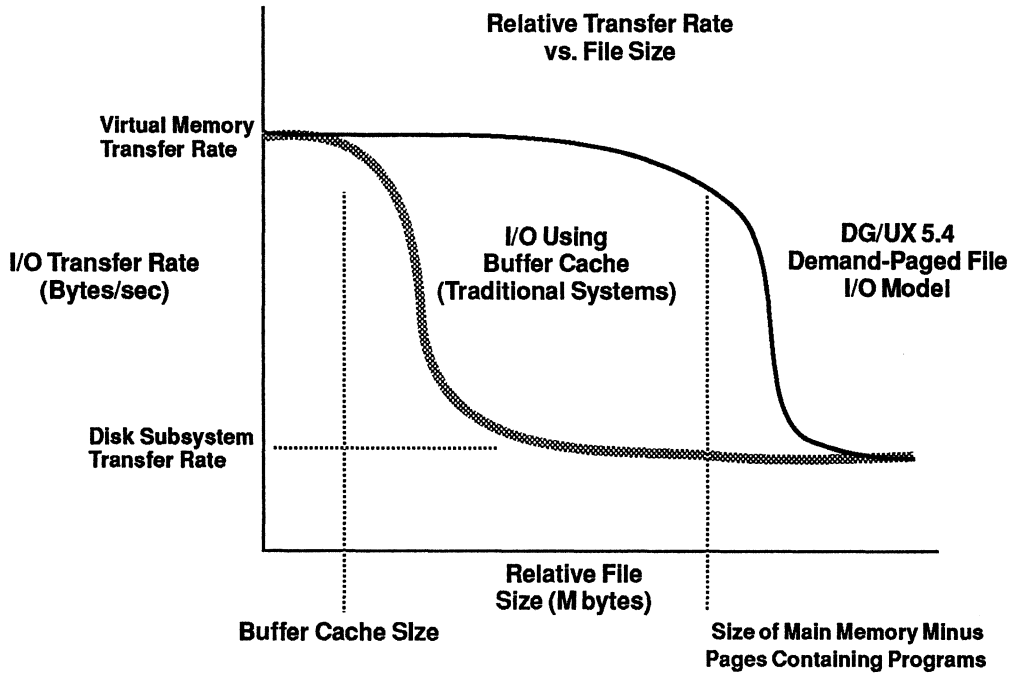


Figure 11 Comparing Transfer Rates

In a File System that uses a file buffer cache, the I/O transfer rate starts to fall off as a file's size approaches the size of the buffer cache. Because the demand-paged file I/O model uses as a cache essentially all of main memory, a high transfer rate can be maintained for much larger files.

Memory Mapping System Call

The **mmap** call eliminates the overhead of copying data from memory into a program's data space (which occurs when programs use the **read** system call). With the **mmap** call, application programs can map the virtual addresses of a file into their address spaces, rather than making copies of the data.

The **mmap** system call enables an application program to treat a region (or all) of a file as an array. A call to **mmap** maps the part of a file that you specify into virtual memory. You can access that part of the file directly; without using traditional **read** and **write** system calls.

In addition, the **mmap** system call:

- Supports sharing of data, which makes efficient use of memory. Two or more programs can reference the same data page for read operations.
- Optionally supports "copy on write" behavior. Copy on write means that two or more programs can share the same data page in main memory, up to the point that one of the programs makes a request to write to the page. At that time, the Virtual Memory Manager gives to the writing program a separate copy of the page.

The FYI section "More About the Demand-Paged File I/O Model" on page 39 talks more about how the **mmap** system call works.

Note that application programs that were written for DG/UX 4.3x will run on DG/UX 5.4, without source code changes or recompilation. If you're writing new programs, you can take advantage of the **mmap** system call.

Special Access Modes for Files

The File System normally accesses files asynchronously with Virtual Memory operations. That means that a program can continue its work even though data that a program has changed may not yet have moved from main memory onto a physical disk. However, the DG/UX File System provides system call support for two other file access modes: *synchronous* and *unbuffered*.

Synchronous Access Mode

You may want an application program to access files synchronously; for example, to suspend a program while a write operation's data is copied from main memory to disk. Many database programs use this technique to maintain transaction logs so that the database can be recovered if it is damaged.

When you open a file, you can specify that you want to access it synchronously. Synchronous mode affects only write operations. For writes, the system call doesn't return control to your program (the program doesn't run again) until modified data is written to the disk. If you've changed the size of the file, the file's inode and index elements (if used) are also written to disk.

Unbuffered Access Mode

You can use special system calls to bypass the buffer cache altogether and allow a program to read and write data directly to and from its own buffers. Sophisticated transaction processing programs, for example, may use

unbuffered access mode. Because these applications provide and manage their own buffering, they don't require the services of the File System's buffer cache.

The DG/UX 4.3x system calls that perform unbuffered I/O operations are available in the DG/UX 5.4 operating system. However, you can use the **mmap** system call to perform some unbuffered I/O operations more efficiently.

About Block I/O and Character I/O

The DG/UX operating system supports two kinds of disk I/O operations: block and character. Although these two operations don't "belong" to the File System, it's useful to examine how they interact with the File System.

As shown in Figure 12, *block* I/O transfers go through Virtual Memory. The transfer size is fixed at 16 Kbytes. Even if a program asks for a one byte record from a file, the File System reads 16 Kbytes from disk. Block I/O operations can be either asynchronous or synchronous. Block I/O is used only with disks, and a disk that is being accessed in block mode is called a *block disk*.

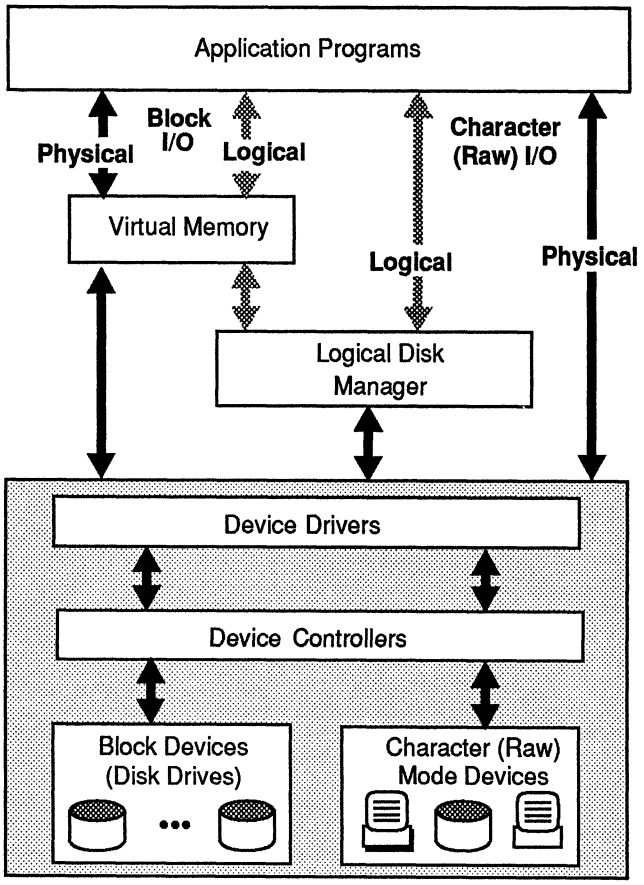


Figure 12 Block and Character I/O

Character I/O, also called "raw" I/O, is used with devices that transfer one character at a time, such as terminals and printers. Character I/O can also be used with disks, allowing unbuffered transfers of an arbitrary number of disk sectors. Character I/O operations are synchronous. Applications can bypass Virtual Memory by performing character I/O operations on a disk. For example, a database management program might use character I/O to manage its own disk transfers. A disk that is being accessed in character mode is called a *raw disk*.

The DG/UX operating system's ability to create file systems on both logical and physical disks, combined with the block and character access modes, gives application programs a great deal of flexibility. For example, a program can use character special I/O on a physical disk to interact directly with a disk—the I/O operations bypass both the File System (and Virtual Memory) and the Logical Disk Manager.

The following table summarizes how DG/UX handles the different combinations of disk types and access modes.

Disk Type	I/O Mode	Directory Pathname	Uses Virtual Memory?	Uses Logical Disk Manager?
Logical	Block	/dev/dsk	Yes	Yes
	Character	/dev/rdsk	No	Yes
Physical	Block	/dev/pdsk	Yes	No
	Character	/dev/rpdsk	No	No

You can see the different access modes of logical and physical disks by looking at the files in the **/dev** directory. For example, the access privileges for **/dev/dsk** are "brw," where the "b" stands for block special. In contrast, the access privileges for **/dev/rdsk** are "crw," where the "c" stands for character special.

Creating Logical Disks and File Systems

The DG/UX operating system provides utilities that enable you to create logical disks, and create and tune file systems.

If you're creating a file system on a logical disk, you use the **diskman** utility to create the logical disk. After you've created a logical disk, the **diskman** utility invokes **mkfs,** which you use to create a file system on a logical disk or physical disk. You can modify a file system's parameters with **tunefs,** which we describe in the next section.

The DG/UX **diskman** utility enables you to create and arrange logical disks while a system is running. If there's a logical disk that you're not using, you can delete it or make a copy of it. If there's a piece of disk space that is free, you can turn it into a logical disk. Note that the **diskman** utility is not part of the BSD or AT&T operating systems.

Here's a summary of the relationships and sizes of various File System data structures.

Data Structure	Size
Logical disk	Tunable—2 Tbytes maximum, 32 piece maximum.
File system	Limited by size of logical disk (2 Tbytes) and number of disk pieces (maximum of 32). If created on a physical disk, limited by disk's size.
DAR	Tunable—limited by size of file system.
Single file	Application dependent—can be larger than a DAR, up to a maximum of 2 Gbytes.
Data element	Tunable—512 bytes to several hundred Mbytes (just less than DAR size), in powers of 2. Default is 8 Kbytes.
Index element	Tunable—512 bytes to 64 Mbytes, in powers of 2. Default is 512 bytes.
Data block	Fixed—512 bytes for disks.
Disk sector	Fixed by disk system— 512 bytes for most disks.

Tips for Tuning a File System

Before you begin a tuning exercise, you should have some specific performance goals in mind. You should also have an understanding of the demands your application or mix of applications places on the system.

For example, are your applications memory intensive, CPU intensive, or I/O intensive? If your system has multiple physical disks, is the system's I/O load balanced among the disks? You should also understand that the tuning parameters interact among themselves. A change that you make to speed up one application might slow down a different application. Your goal should be to find the best balance of system parameters for your mix of applications.

After you've created a file system with **diskman** and **mkfs**, you can tune the file system. A simple way of optimizing a file system's performance is to create the file system on a fast disk. For example, if your system has a fast disk drive, you might use it for file systems that support your performance-sensitive applications. If you have multiple disks on your system, you may be able to increase performance by putting pieces of a logical disk on different physical disks, or by using data striping to enable the File System to access data from several physical disks concurrently.

In addition to moving a file system to a faster disk, you can tune a file system with the **tunefs** utility. You can also change the values of kernel parameters (in the kernel configuration file), which affects all of the file systems.

You can use the **tunefs** utility to change the defaults for a specific file system without affecting the parameters of other file systems. Often, this is easier than using system call parameters on a file-by-file basis. Changing a file system's defaults is also useful if you don't have access to a program's source code—for example, you can set data element size this way.

There are several file system parameters that you can change. Some parameters, such as Virtual Memory cleaning time, are set in the kernel configuration file and are used by all file systems. Other parameters, such as default data element size, are set when you create a file system with the **mkfs** utility and can be different for each file system. We'll look at some of the parameters in the general order that they affect performance:

- Size of main memory
- Virtual Memory cleaning time (kernel configuration file)
- Default data element sizes (**mkfs**, **tunefs**, and system calls)
- Default index element sizes (**mkfs**, **tunefs**, and system calls)
- Anniversary sizes (**mkfs** and **tunefs**)

Notice that we don't talk about making changes to the kernel data cache parameters of size and aging time. These parameters have relatively small effects on performance because the kernel data cache is only used to store small amounts of kernel metadata.

Adding Main Memory

In the technical brief that talked about disks on AViiON systems, we mentioned that the way to speed up a disk is to avoid using it. That's really not a paradox—you can significantly reduce disk I/O operations by installing as much memory in your system as your budget allows. This enables the Virtual Memory Manager to cache more program and file pages in main memory. Generally, increasing the size of a computer's memory directly reduces the number of disk I/O operations. During normal system operation, many files are short lived—with adequate memory, they can be created and deleted without requiring any disk I/O.

We'll look at the effect that adding memory has on a typical AViiON computer by running an I/O benchmark program.

The test program starts by creating an 8 Mbyte file. The program reads data records from the file sequentially, and then writes data records back to the file sequentially. The reads and writes continue until all of the records have been read and written. The test program then repeats the read and write operations using random accesses. The sequential and random operations are repeated five times, to transfer 40 Mbytes of data.

The test system was an AViiON AV310 with a model 6554 662-Mbyte SCSI disk. To show the effect of different memory sizes, we ran the benchmark program with memory sizes of 8 Mbytes and 16 Mbytes. Figure 13 shows the results of these tests.

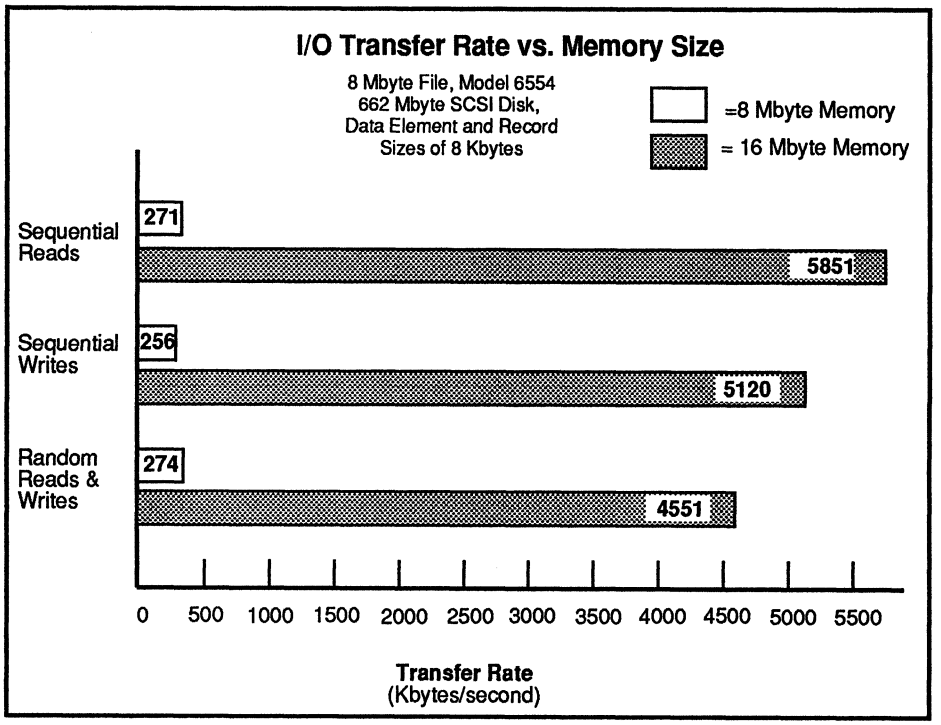


Figure 13 Effect of Different Main Memory Sizes

With 8 Mbytes of memory, only part of the 8 Mbyte test file can be cached. (Memory is shared by the data file, the operating system, and the test program.) Doubling the amount of memory (to 16 Mbytes) has significant effects on the I/O transfer rate because all of the 8 Mbyte test file can fit into memory. With the 16 Mbyte memory, the throughput for sequential read operations was 21.5 times better than with 8 Mbytes of memory, and write throughput was 20 times better. The throughput for random read and write operations was more than 16 times greater with the 16 Mbyte main memory.

Changing the Page Cleaning Time

The Virtual Memory Manager normally cleans pages (copies modified pages to disk) within 60 seconds after a page has been modified. The purpose of this policy is to maintain data integrity by copying modified pages to non-volatile storage (disk). Note that the modified pages are only copied to disk—the pages remain in memory until the Virtual Memory Manager swaps them out to make room for other pages.

You can change the cleaning time by changing the Virtual Memory cleaning time parameter (**MAXBUFAGE**) in the kernel configuration file and rebuilding the kernel. (In DG/UX 4.3x, this parameter controlled the buffer cache flush time.)

Turning the **MAXBUFAGE** value up—cleaning modified pages at longer intervals—can make many application mixes perform better, but reduces data integrity. By increasing the cleaning time, you're increasing the risk that recently modified data in memory will be lost if a system failure occurs.

Keep in mind that turning the **MAXBUFAGE** parameter down is in opposition to the general policy of keeping file and program pages in memory for as long as possible. This policy saves disk I/O operations by increasing the probability that a program will finish working with a file before the file is written to disk. If a temporary file is not written to disk before an application finishes using it, the file can be created in memory, used to store some intermediate data, then deleted—without requiring any disk I/O operations. For example, a compiler generates temporary files between the different phases of a compilation. Turning down the cleaning time can cause these temporary files to be written to disk, which results in unnecessary I/O operations.

Turning down the cleaning time also decreases the number of times that a program can modify data in a resident page, between the time that the data is first modified and the time that the data is written to disk. As always, you should experiment to find the best configuration for your particular mix of applications.

Figure 14 shows the relative results of changing times from the default of 60 seconds to 10 seconds. We used the same benchmark program as before, but with a file size of 16 Mbytes and main memory size of 16 Mbytes.

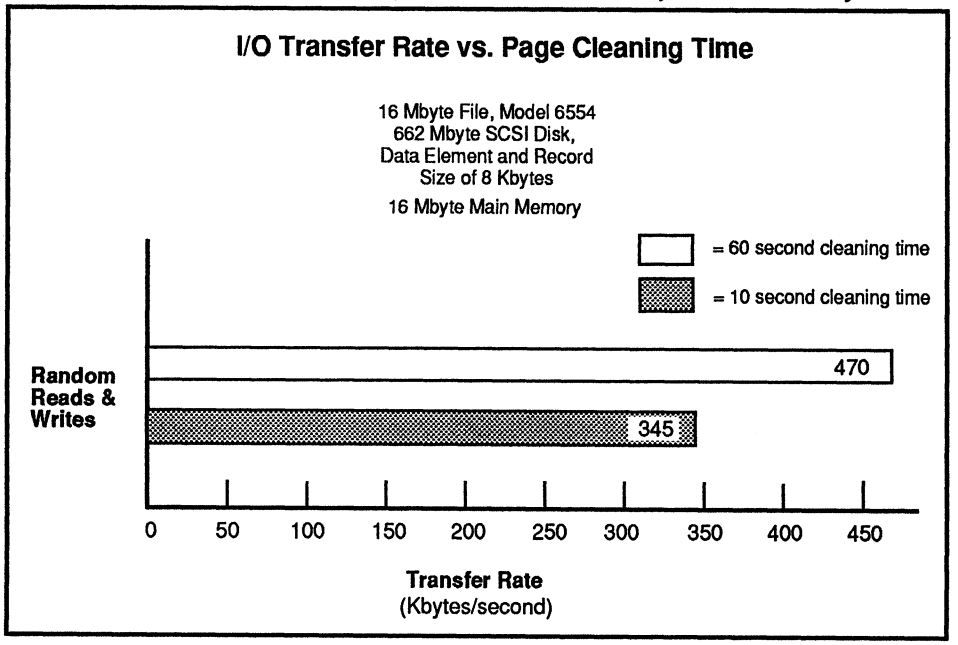


Figure 14 Effect of Virtual Memory Cleaning Time

Why is the throughput 1.3 times better with the page cleaning time at 60 seconds? Because the longer cleaning interval reduces the number of I/O operations to disk. With the default 60 second cleaning time, many write operations are made to already modified resident pages—before an I/O operation is used to clean the pages to disk. The longer cleaning time also enables the Virtual Memory Manager to take advantage of its algorithms that "batch" disk I/O operations. When the cleaning time is turned down to 10 seconds, more of the I/O system's resources are used to clean pages, and the program cannot move as much data.

Changing the Data Element Size

You can sometimes achieve performance gains by changing a file's data element size to match the record size of your application. You can make the data element size as small as one disk block (512 bytes) or as large as several hundred megabytes (just less than the size of a DAR).

If you don't take special action when you create a file system with **mkfs**, all files in a file system will have the default data element size of 8 Kbytes. With **tunefs**, you can change the default data element size for an existing file system. You can use the **dg_mknod** system call to create a new file and

set the file's data element size or index element size to something other than the file system's default values. You can use the **fez** command to display the size of a file's data elements and index elements.

In general, you want to match a file system's default data element size to the record size that your applications use. However, the effects on performance can vary, depending on whether an application is using mostly random I/O or mostly sequential I/O. Let's look at both cases.

Random I/O—Try Smaller Data Element Sizes

If you're working with an application that uses truly random I/O operations, you might consider using files with data elements that are smaller than the 8 Kbyte default size.

If a file's data element size is larger than an application's data record size and the application is using truly random I/O operations, the overhead of reading and caching the extra data can be significant. By "extra data," we mean the data in the data element that is not part of the data record. For example, if an application is accessing 2 Kbyte records and the data element size is 8 Kbytes, the File System is moving and caching 6 Kbytes of extra data for each I/O operation.

By decreasing the data element size so that it more closely matches the size of the records that a program is reading, you can reduce the amount of extra data that is transferred. If the chances are small that an application will make a Virtual Memory page "hit" on a data record from this extra data, it makes no sense to pay the cost of transferring and caching the data. Not only does it take longer to transfer this extra data, you're using memory that could be used to store files or programs from other applications.

Notice that we used the words "truly random I/O operations" at the beginning of this section. Be aware that many "random" data accesses really aren't truly random. The data often has some structure, which implies locality of reference. In these cases, you might want to make the data element size somewhat larger than the data record size.

The trade-off when you decrease the data element size is that small data element sizes can adversely effect sequential I/O operations. As you'll see in the next section, sequential operations benefit from having data element sizes that are larger than record sizes. Since data element size can be set on a per file basis, the same file system can accommodate files that have different data element sizes for different patterns of I/O.

For applications running on the DG/UX 5.4 operating system, you probably don't want to make the data element size any smaller than the Virtual Memory page size (4 Kbytes). If you do, it's likely that the Virtual Memory Manager will have to perform more than one I/O operation for each page,

and performance will suffer. This is a change from DG/UX 4.3x, which sometimes provided performance gains with data element sizes as small as 512 bytes.

Sequential I/O—Try Larger Data Element Sizes

If you're working with an application that uses sequential I/O operations, you might consider using files with data elements that are larger than the 8 Kbyte default size.

As you might suspect from reading the discussion of data elements and random I/O in the previous section, one reason for increasing the data element size for applications that use mostly sequential I/O is to increase the amount of localized data that is read into Virtual Memory. By increasing the size of a file's data element to more than an application's record size, you increase the probability that a follow-on request will find that the data that it wants is already resident in memory. The File System itself performs read ahead operations on sequential operations, by reading in the next data element. If the data elements are large, the probability increases that sequentially accessed data will be in memory when a program asks for the data.

There is essentially no limit on the size of a data element, except that it must be slightly smaller than the DAR size (the DAR size minus the space used by the DAR's inodes and bit map). A data element can easily be as large as several hundred megabytes.

Remember that a file's data blocks are allocated contiguously within a data element. If a file fits completely within a data element, you can approximate the behavior of file systems that support contiguous allocation. Or, you can use large data elements to reduce disk seek times. With a large data element, you'll be doing few seeks instead of potentially one seek for every 8 Kbytes with the default data element size.

For example, if you have a 5 Mbyte file and set the data element size to 1 Mbyte, the first megabyte of the file will be contiguous. The File System can read the file with a minimal number of seeks—the initial seek to a disk cylinder, a read of that cylinder, and a seek to the next cylinder.

One trade-off of using a large data element size is that you can adversely affect random I/O operations. Another trade-off is that you can fragment a disk by making the data element size too big. There may be plenty of disk space available, but the File System won't be able to find contiguous blocks to store large data elements. In this case, the File System produces an "out of space" error. Therefore, you don't want to make the data element size too big unless you really need the performance (or you have a very large disk).

Changing Index Element Size

By changing a file system's index element size, you control how many disk blocks are used to store index elements. The index element size does not generally have a significant effect on overall performance, but you can use it for fine tuning a file system.

If you're working with large files, you can use larger index elements to access more of a file's data elements with single index pointers, and reduce or eliminate the use of double and triple index pointers. Note that you can't change the number of direct pointers in an inode—that number is fixed at ten. However, you can affect the amount of data that is pointed to by the direct inodes by increasing the default data element size. So, for large files, you might consider increasing both the data element size and the index element size.

The trade-off of using large index element sizes is that they use up disk space and memory. This is a consideration with small files—it makes no sense to increase the index element size for files that can be referenced completely by an inode's direct pointers.

Changing Anniversary Sizes

As we mentioned earlier, the point of a DAR is to keep a file's inode, index elements, and data elements close together in a disk. The File System always tries to put all of a file's pieces into one DAR. However, the File System also tries to balance the allocation of data within a file system by keeping DARs equally full.

As a file grows in size, the File System allocates more and more blocks out of the DAR that contains the file, until the file reaches its first *anniversary size*. The first anniversary size is a limit that, when reached by a file, tells the File System to start allocating a file's data into another DAR. The anniversary size limit protects against having any one DAR too heavily subscribed.

The default anniversary size for files is 1 Mbyte. You can set a different size when you create a file system with **mkfs**, or afterwards with **tunefs**. You might want to use a larger anniversary size if you're working with large DARs. With large DARs, a larger anniversary size will keep the File System from moving parts of files to other DARs too quickly.

There's also a second anniversary size limit that tells the File System when to stop allocating data blocks from a secondary DAR and move data into yet another DAR (or DARs).

On a related note, how full should a file system be allowed to be? Generally, a file system should be less than 90% full—if a file system is more than 90% full, it becomes too difficult for the File System to find space on the

disk, and accesses to the disk slow down. You can set a limit on the "fullness" of a file system when you create it with the **mkfs** utility, or afterwards with the **tunefs** or **cpd** utilities.

You can control the size of directories by using Control Point Directories (CPDs). We describe CPDs in the "FYI" section on page 36.

FYI—Other Features of the DG/UX File System

The DG/UX 5.4 File System provides many advanced features that distinguish it from many other UNIX File Systems. Taken together, these features contribute to the File System's functionality, performance, and reliability in commercial applications. Although *some* other UNIX implementations offer *some* of these features, the DG/UX 5.4 operating system is unique because it provides a combination of features.

Record Locking

The DG/UX file system supports both advisory record locking and mandatory record locking. Advisory record locking means that you lock a range of bytes and other users cannot lock on that range of bytes. However, users can choose to ignore the locks. Mandatory record locking means that no one else can access the data that is governed by the locks. With a mandatory lock, you're saying in effect that "I'm the only one who can access these bytes." There are different kinds of locks for different file operations, such as reading and writing.

The DG/UX locking implementation is performed in the kernel. Access to record locking is via a system call. Note that BSD file systems don't generally support mandatory record locking—it's a System V feature. In some other UNIX operating systems, record locking is supported externally by a lock daemon.

Forward Progress Checks

The DG/UX kernel uses a "forward progress" algorithm to keep a process from "starving," waiting for record locks. The operating system has a data structure that keeps track of which users are waiting for which locks.

For example, suppose that user "A" locks bytes 1-5 in a data record and user "B" tries to lock bytes 5-7. User B will wait on user A's lock because of the overlap on byte 5. Now suppose that user "C" requests a lock on bytes 7-9. Bytes 7-9 aren't locked. On most UNIX systems, user C could lock these bytes. If user C gets the lock on bytes 7-9, user B may "starve"—may never get a lock on bytes 5-7. The forward progress algorithm looks to see if anyone else is waiting for a lock. In this case, the forward progress algorithm would place user C's request behind user B's request.

Consumers of record locks include database applications, COBOL applications, and Business BASIC applications. COBOL programs, for example, use locks to protect their I/O operations.

Adjustable File System Size

You can use the **diskman** utility to grow or shrink an unmounted file system. This enables you, for example, to increase the size of a file system that has become too full. You can also grow or shrink a logical disk that contains no file system.

You grow a file system by adding new disk pieces or by making the last piece bigger. In both cases, the file system will have more DARs available to store data. Shrinking a file system may cause logical disk pieces to be removed from the logical disk, and may cause the last piece to be reduced in size. System and user data will be compacted and copied into the smaller file system. Note that you cannot change the size of a logical disk that is being striped across physical disks.

Duplicate Data Structures

The File System creates duplicates of each physical disk's Physical Disk Information Table (PDIT) and each file system's File Management Information Area (FMIA). The File System also maintains copies of the File System's bad block remap table and logical disk piece table. If one of these data structures is corrupted, the File System automatically reads the backup copy.

For example, when you format a physical disk, the PDITs are placed on good disk sectors—they can't be moved unless you reformat the disk. If the operating system isn't able to read a PDIT, you'll receive a "loss of fault tolerance" message, and the operating system will automatically read the backup PDIT.

Dynamically Remapped Bad Blocks

Nearly all disk drives have some media defects. When you first format a disk, the low level formatting routines mark as "bad" any sector that has a defect. However, over time, new defects can appear, which can make a sector unreadable.

The File System assumes that it is working with disks that have no physical (media) defects. However, at a lower level, the Logical Disk Manager looks for and manages bad disk sectors (bad blocks). If the Logical Disk Manager detects errors on a disk sector, the manager tries to copy the data from the bad sector to a good sector. (The Logical Disk Manager maintains

a pool of good blocks that it can substitute for bad blocks.) This "bad block remapping" operation is done automatically—the File System doesn't know that the data has moved to a different sector.

Control Point Directories

Control Point Directories (CPDs) enable you to put size limits on directories so that you can control the size of individual subtrees within a file system. The concept is similar to that used in Data General's AOS/VS operating system.

When you create a directory as a CPD, you tell the File System how many blocks the directory can hold. This limit includes all of the directory's files, descendant directories, and their files. If you exceed this limit, the operating system will report "out of space" errors. Note that they are caused by a lack of space in your Control Point Directory—not necessarily a lack of space in a file system.

The BSD operating system has a concept of quota, which controls how many blocks can be on a file system for a given user ID. However, BSD quotas are harder to administer than CPDs, because:

- One quota per user ID per file system is less intuitive than one limit per CPD.
- Quotas must be administered by the superuser.
- Using a file's user ID for quota purposes makes potentially conflicting use of the user ID attribute, whose primary purpose is access control.

Read Only and Sealed File Systems

As much as people dislike talking about it, it's possible that a file system can be corrupted. Corruption, if it does occur, is usually caused by a hardware problem—for example, someone mistakenly disconnected a disk cable or a disk experienced a media failure. Occasionally a software problem could corrupt a file system.

If a file system becomes corrupted while you're using it, the File System changes the file system's access privileges to read only. If the read-only file system continues to generate corruption errors, the File System "seals" the file system so that you can't access it. If you continued using the file system, it's likely that you could make the problem worse. This is different from other UNIX file systems which, if they detect a corrupt file system, will panic or continue to allow access.

If a file system is sealed, you need to find the cause of the problem. Then, you can unmount the file system, run the **fsck** utility to fix the file system, and remount it. The DG/UX operating system enables you to do this without bringing the system down. You cannot run the **fsck** utility on the root

file system while a system is running, because the root can't be unmounted. However, the kernel automatically checks the root file system during a system boot.

System Shutdown

The DG/UX operating system provides the **halt** command, which performs an orderly system shutdown. The **halt** utility updates the file systems, unmounts them, and halts the processors. Other UNIX systems require that you use the **sync** command several times to ensure that a file system's information is stored safely on disk.

Fast Recovery Option for File Systems

In DG/UX 5.4, a fast recovery option for files has been added to the **mount** command. This option provides significant improvements in how fast a file system can be recovered with the **fsck** utility.

The fast recovery option tells the File System to maintain a log of what's being done to a file system and which files have had their buffers written to disk. When the file system is recovered following a crash, the File System looks at the log to see how to recover the file system.

The recovery time without a recovery log is proportional to the size of the file system. When you create a fast recovery log with the **mount** command, the recovery time is proportional to the size of the log, not to the size of the file system. As an example, the time to recover a 200,000 block file system without a recovery log is 28 seconds. The time to recover the same file system using the fast recovery option is 5 seconds (using an eight-block recovery log). If you increase the file system size, the standard **fsck** recovery time will increase proportionally, but a **fsck** recovery using the fast recovery option will remain at 5 seconds.

The fast recovery option requires a small amount of overhead to maintain the recovery log. Some operations, such as extending a file, will be slightly slower. However, some operations will actually be faster. For example, when the recovery log is "on," disk update commands can be written to the recovery log while the disk update itself is postponed. If the system were to crash between the time that the log is written and the disk update is complete, the recovery program reads the update request from the log and completes the operation.

Quiescent File Systems

If a file system is not in use (is mounted but has not been accessed for 5 minutes), the File System marks it as quiescent. When you reboot, any file system that was not mounted or was marked quiescent need not be checked by the **fsck** utility. This feature can save significant time when you have to reboot a system after a power failure or other system failure.

Memory File Systems

On personal computers, utilities are available that enable you to create a RAM disk, which uses a region of main memory to emulate a physical disk. Accesses to a RAM disk are extremely fast because they are made at the nanosecond speeds of memory instead of the millisecond speeds of a mechanical disk.

In the DG/UX operating system, there's a similar concept, called a *memory file system*. You create a memory file system with a **mount** command option.

If you have enough main memory, you can mount a memory file system that programs such as compilers and linkers can use for their temporary files.
A memory file system, because it is mounted from main memory, is much faster than a file system that is mounted from a disk. Note that you would not want to use a memory file system for an editor's temporary files because the files are needed for crash recovery.

Another use for a memory file system is to speed access to files on diskless workstations. A good example is a Network Information Service binding file, which contains information about a client's server. By maintaining this file in a memory file system, the DG/UX operating system can lock and access the file much faster than if the file had to be accessed over a network.

The disadvantages of a memory file system are that the file system uses up main memory resources such as swap space. Also, a memory file system is volatile—if the system loses power, the data in the memory file system is lost.

FYI—More About the Demand-Paged File I/O Model

Figure 15 shows more details of how the DG/UX 5.4 demand-paged file I/O model and the File System work together. We've annotated the figure to show how some different kinds of system calls are handled, where addresses are converted, how Virtual Memory is used for file data, and how the kernel data cache is used to cache kernel metadata.

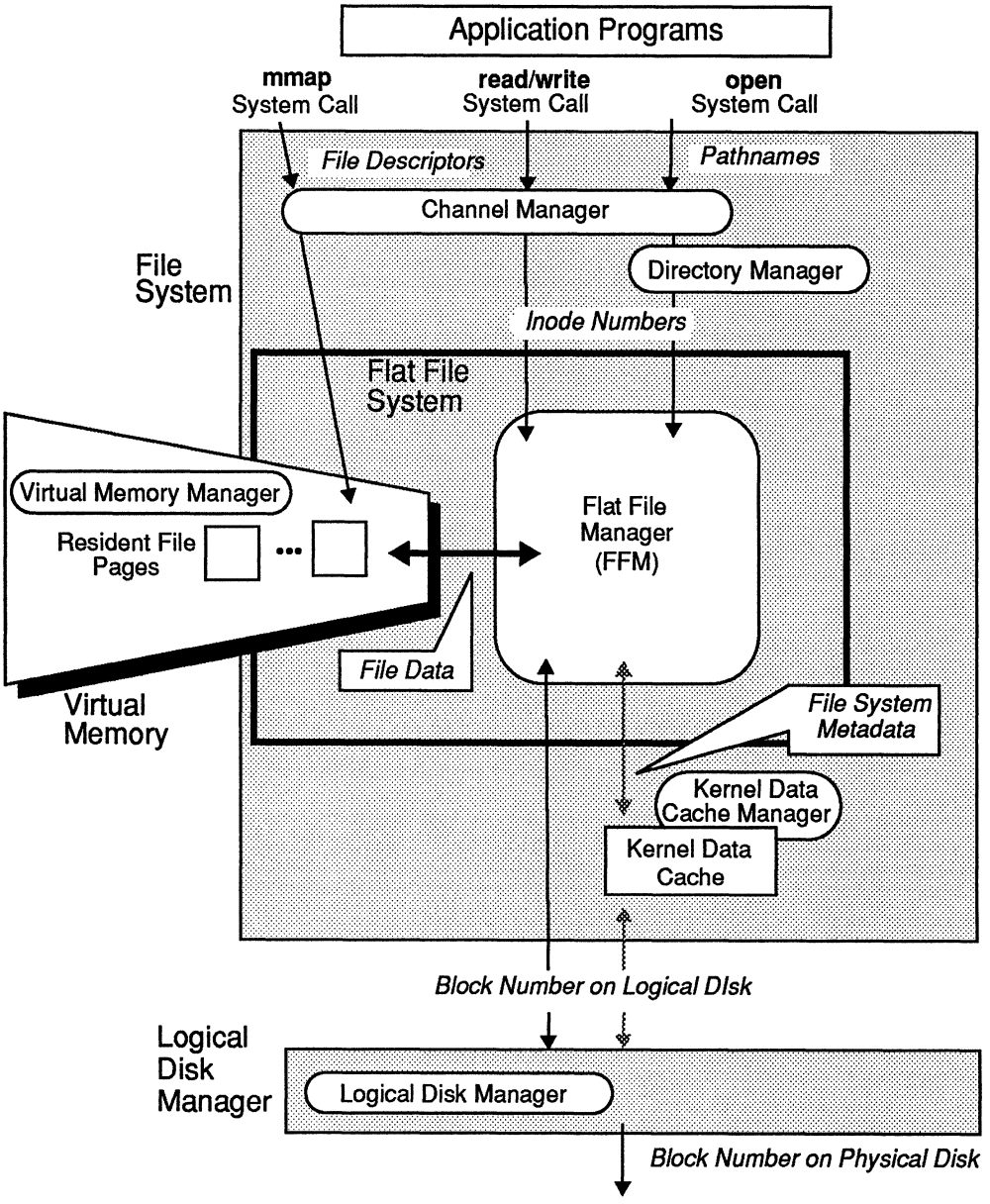


Figure 15 Virtual Memory and the File System

Channel Manager

In Figure 15, we've shown the File System's *Channel Manager.* The Channel Manager is a generic part of the File System that provides I/O access paths between application programs and files. These access paths, called channels, are independent of the type of file with which a program works.

An application program accesses a channel by referring to the channel's *file descriptor.* Within the File System, a file descriptor identifies a channel, which references an open file. More than one channel can reference the same file, so a file can be accessed via several channels. To an application program, a file descriptor is an integer that system calls, such as **read, write,** and **mmap,** use to identify a file.

FFM and the Virtual Memory Manager

The Virtual Memory Manager receives mapping requests directly (from **mmap** system calls) and indirectly (from **read, write,** and **open** system calls.)

When the FFM wants to read or write data for an open file, it obtains from the VMM a mapping for each referenced file page and performs the requested I/O operations to or from those pages. If the FFM requests a mapping for a file page that is not resident in memory, VMM performs a "page in" operation, by requesting the FFM to fill a frame (or frames) of main memory with the data from the appropriate file elements. The FFM translates the file offset to a disk address (block number on logical disk) and sends the I/O request and address to the Logical Disk Manager.

Similarly, when dirty (modified) file pages need to be written to disk (for either data integrity or page replacement), VMM performs a "page out" operation by requesting FFM to write file data from main memory frames to the appropriate data elements on disk.

Handling System Calls

As Figure 15 shows, the File System and Virtual Memory Manager handle application programs' file-related system calls in different ways. Because the **mmap** system call is new in DG/UX 5.4, let's summarize how **open, read** and **mmap** system calls are handled.

Open System Calls

When you use the **open** system call to open a file, the call returns a file descriptor. Subsequent system calls that reference the open file, such as **read, write,** and **mmap,** use the descriptor to identify the open file.

One argument to the **open** system call is the pathname of the file that you want to open. The pathname can contain the names of many subdirectories, which the Directory Manager and FFM resolve recursively to inode numbers and then to inodes. When the Directory Manager finds the directory entry for the last path in the pathname, it reads the target file's inode number from the directory entry and passes the inode number to the FFM.

The FFM converts the inode number into an inode and passes the inode back to the Channel Manager. The channel then has the target file's inode, and the file descriptor is returned to the application program by the **open** system call.

Read System Calls

The **read** system call takes the following arguments:

- a file descriptor,
- the number of bytes to read from the file, and
- the address of a program buffer that will get the requested data.

When the Channel Manager receives the file descriptor, it gets the referenced file's inode from the file descriptor's channel. The file's inode is already in the kernel data cache as a result of the system call to open the file.

Using the addressing information in the file's inode, the FFM performs an internal mapping calculation, and tells the VMM what pages that it wants mapped into memory. If the pages are not already in memory, the VMM gets them from disk. After the pages are in memory, the requested data is copied to the program's buffer. Therefore, the **read** system call uses two data-copy operations—one operation to copy data from disk to memory and another operation to copy data from memory to a program's buffer.

Memory Mapped System Calls

mmap system calls go directly to the Virtual Memory subsystem. This enables application programs to "touch" pages in memory by using virtual addresses—without the **read** system call's second step of copying data to the program's buffer.

The **mmap** call uses the Virtual Memory Manager's ability to manipulate addresses and causes the VMM to set up mappings between file pages and pages in virtual memory. It's more efficient to manipulate these mappings (in a page table) instead of actually copying data. After you have memory mapped a file (or part of a file), your program can use virtual memory addresses in its address space to access the file's data. Note that a page table maintains mappings for both resident and non-resident pages.

Memory mapping is performed on page boundaries and at page granularity. If the piece of a file that you want to access is smaller than a page, a page-sized piece of the file is mapped. If the piece of the file is larger than a page, multiple pages are mapped into memory.

The **mmap** system call includes the following arguments:

- a file descriptor,
- the mapping address (the program address at which to map the region),
- offset in the file of the region to map, and
- number of bytes of the region to map.

Figure 16 shows a simple example of how the **mmap** system call works. Assume that a process uses the **mmap** system call to map pages 0, 1, and 2 of the file "mydata" into pages 3, 4, and 5 of the process' memory space.

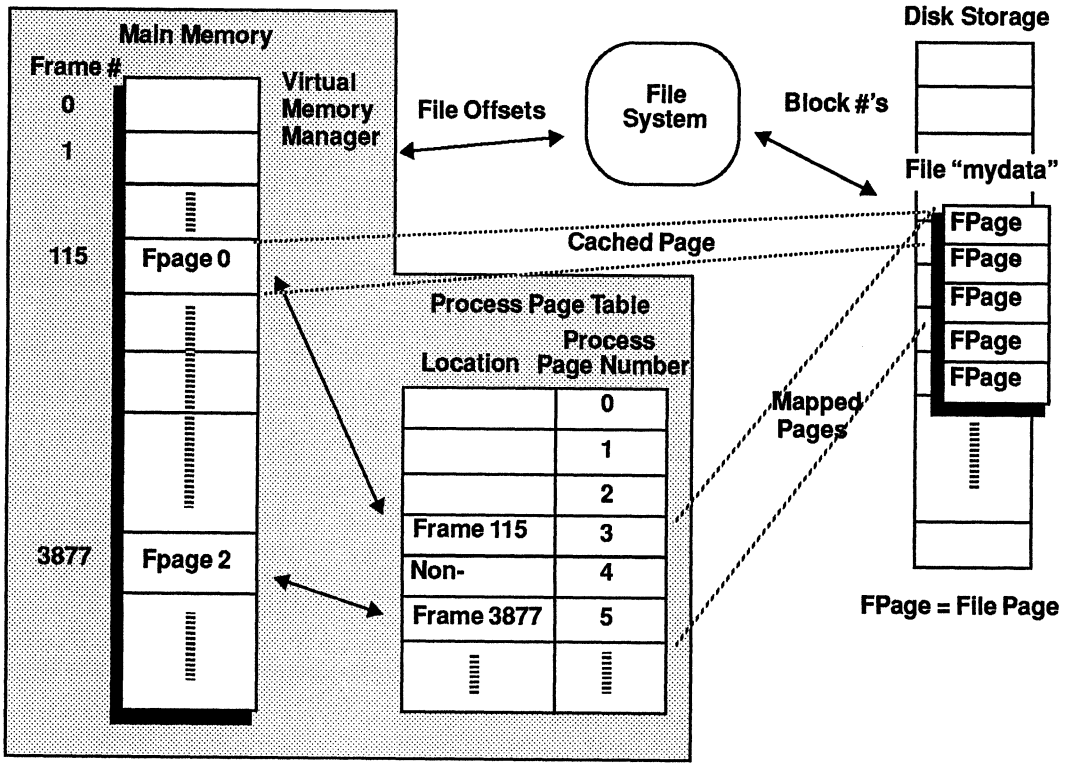


*Figure 16 Using the **mmap** System Call to Memory Map a File*

When the VMM receives the **mmap** system call from the process, the VMM sets up in the process' page table the mappings of virtual memory pages to file pages. The VMM does not read the pages from disk—it only sets up the mappings. As the process references particular pages, hardware page faults cause the VMM to perform "page in" operations to read the pages from disk.

In the example, file page 0 (FPage 0) resides in main memory frame 115, and so on. When the process references its page 3, that reference is mapped automatically by the VMM to main memory frame 115, which contains file page 0. Notice that the page table contains entries for non-resident pages. The locations for these pages are filled in when the VMM gets a page from disk.

The VMM manages the page table, independent of the process. If the process doesn't touch one of the pages for a period of time, the page will "age," and the VMM will swap the page back to disk when the memory is needed for other pages. Also notice that a page or pages may already be resident in memory if another process has accessed them. In that case, the processes can share memory pages.

More Reading

Installing the DG/UX™ System (093-701087)

Managing the DG/UX™ System (093-701088)

Customizing the DG/UX™ System (093-701101)

System Manager's Reference for the DG/UX™ System (093-701050)

Programmer's Reference for the DG/UX™ System (093-701055, 093-701056, and 093-701102)

DG/UX™ Technical Brief Number 6: A Look at High Availability Disk Systems (012-004035-00)