A VIRTUAL MEMORY DISTRIBUTED FILE SYSTEM

Daniel L. Murphy

Digital Equipment Corp.

9-March-1989

Present address:

Open Software Foundation Research Institute

Eleven Cambridge Center

Cambridge, MA 02142

dlm@osf.org

A VIRTUAL MEMORY DISTRIBUTED FILE SYSTEM

ABSTRACT

This paper describes the practical extension of an existing operating system into a multiprocessor configuration using a high-speed serial interconnect. A major part of this effort was to distribute the management of the file system such that all file objects would be accessible from all processors. The original file system used virtual memory mapping with extensive sharing capabilities as its basic file access abstraction, and the multiprocessor extensions were required to maintain and support this abstraction.

A VIRTUAL MEMORY DISTRIBUTED FILE SYSTEM

1 INTRODUCTION - INTERCONNECT TECHNOLOGIES

During the early 1980's, innovations in the area of high speed interconnects led to new topologies for operating systems. One of these, known as "Clusters", is implemented by two systems from Digital Equipment Corp. -- VAX/VMS and DECSYSTEM-20. These two systems share much of the cluster technology at the hardware level, and some common techniques are also used in several areas of the operating system. In one particular area however, the original architectures of the systems were quite different, and this led to rather different solutions in the extension of the architectures to clusters.

The design and implementation of Clusters on VAX/VMS systems has been extensively discussed elsewhere[3, 4]. This paper will discuss those aspects of operating system design which are unique to the DECSYSTEM-20 implementation, i.e. the file system and virtual memory architecture.

Both systems are based on a "closely-coupled" structure using a high speed, serial, message-oriented interconnect known as the "CI". Relative to the Ethernet, the CI has the following characteristics:

1. Faster (70 mbit/sec)

2. Shorter (100 meters)

3. Intended for homogeneous configurations of cooperating operating systems and servers; more tightly coupled than typical network objects.

4. Small number of nodes (originally 16).

Unlike the multi-drop design of Ethernet, the CI uses a central coupler with cables radiating out to each node.

The CI presented the opportunity to configure system which would be more closely connected than on the typical wide- or local-area network, yet not so closely connected as with a shared-memory multiprocessor. Hence, the term "closely-coupled" system is used.

In addition to connecting CPUs, the CI also serves to connect mass storage disk and tape servers to the CPUs. The following shows a basic CI-linked configuration.

2 DISTRIBUTIBILITY OF SYSTEM COMPONENTS

The TOPS-20 operating system includes a rich set of system services, including a number of services which explicitly or implicitly provide interprocess communication, and such services are the focus for extending the system as a whole into a clustered environment. A basic entity of TOPS-20 is the "job", where each job is owned by a particular user and has access capabilities dependent on that user. A job has multiple processes organized in a hierarchy, and each process has a unique address space and a unique state represented by a PC, processor registers, and other software state information.

In a shared-memory multiprocessor systems, these resources would be distributed among the processors almost automatically inasmuch as they are represented by structure in memory, and the memory is accessed in common by the processors. Without shared memory, an active process context cannot exist simultaneously on two or more processors. It could be moved by copying, but such a move would take a prohibitively long time for larger processes, even at CI speeds. Hence, we concluded early in our design effort that dynamic movement of processes among processors would not be a goal of the cluster implementation.

We also considered whether the multiple processes of one job could be distributed among the processors of the cluster. There is a sizable job-common database which the processes use, and, although it appears possible to implement a distributed form of this database and operate it with acceptable performance characteristics, that was a larger effort than we wanted to attempt at the time.

Hence, we decided to assume that any particular job would exist entirely on one processor, and that assignment would be static for the life of the job.

With each job residing on a particular processor, any intra-job communication works as it does on single-processor systems. Inter-job communication services (implicit or explicit) include:

1. IPCF - Message oriented interprocess communication facility.

2. ENQ/DEQ - Queueing and interlocking oriented to files and data bases.

3. DECNET - Networking services to processes on the local system as well as foreign systems.

4. FILE SYSTEM - The general disk-based file system which provides dynamic sharing of file data as well as shared directories.

5. MISCELLANEOUS SYSTEM SERVICES - Job information tables, etc. accessible to users.

IO peripherals such as line printers and card readers might also be considered in this category, but they were already allocated to and controlled by dedicated jobs, so further distributed access was unnecessary.

Of the above services, the file system is clearly the most important. Information is communicated and shared through the file system continuously anytime the system is in use. Users access a large number of common directories for system programs, documentation, project data bases, etc. The mail system is merely a form of communication through the file system, with senders writing into receiver's mailbox files through a system server. Further, the file system provides simultaneous access to databases by cooperating processes, possibly in different jobs. A primary objective of cluster support was to make the file system as effective for jobs on different processors as it was for jobs on the same processor.

DECNET networking was already an inherently distributed facility. The other facilities were not distributed in the existing TOPS-20 design, and have been considered separately.

The capability of using a single logical file system from multiple CI-linked processors is called the Common File System (CFS) in TOPS-20, although it is also fair to think of it as a file system distributed among several closely cooperating nodes. Design and implementation of CFS is discussed in the balance of this paper.

3 VIRTUAL MEMORY FILE ABSTRACTION

The architecture for the disk file system in TOPS-20 is rather different from that of many other operating systems, including VMS. A salient difference is that the most basic form of file IO is not read/write transfer primitives but rather mapping of files into process address space. TOPS-20 does provide system services for read/write transfer IO, record IO, etc., but all of these are layered on top of the basic mapped file abstraction.

The mapped file abstraction implemented by TOPS-20 provides some characteristics that are not implicit in transfer type file access. Foremost among these is that the mapping of a file gives the appearance of allowing the program to directly reference the actual current file data, not a copy of the data. That is, the semantics of the operation of mapping a portion of a file into an area of process address space do not imply a copying of file data into local memory; rather the process address space becomes a window through which the process "sees" the actual information currently in the file. Given appropriate access privileges, two or more processes may map the same file data. Should one process modify the data via a write reference, the modification is immediately visible to the others.

Of course, the operating system implements this abstraction by copying (reading) data from the disk into memory. However, a single copy of a page of file data in memory is sufficient for all processes that have that page mapped, and the structure of page tables allows the operating system to move the page between memory, swapping storage, and home disk location transparently to all processes using it. Simultaneous write access presents no difficulty in a single-processor system, since all processes are referencing the same page of physical main memory.

The shared-file capabilities are used by the operating system itself, e.g. to access file directories.

As is typical of many systems, a directory is an ordinary file containing name information and some kind of link to a physical file structure. A TOPS-20 file directory consists of linked nodes organized to provide efficient lookup of three independent file name fields (name, extension, generation number) including recognition and on-demand completion of flexible abbreviations. Storage for file nodes is allocated from a heap in each directory, and each node contains several dozen items of information about the file. Any directory of interest is mapped into the operating system's address space whenever some access to the directory is required.

Disk allocation tables are also organized as files on the disk and mapped into system address space when needed.

Other capabilities visible to user programs include "copy-on-write" access and some basic file access interlock mechanisms. "Copy-on-write" is a mapping mode which is initially equivalent to shared-read, but if a write or modify reference is made to any such page, a copy is automatically created in place. This is used extensively for initialized variables in shared programs, and it also has some application in data base management. The file interlock mechanisms allow common utility programs to obtain consistent access to files by detecting and preventing inadvertent simultaneous write.

Also, TOPS-20 implements the concept of a "file structure" which is a named unit consisting of multiple directories with their files. A running system typically has several structures available, and individual structures may be added to or removed from the running system. A structure may reside on a physically removable medium such as a disk pack, but is more often just a logical entity on a fixed-media disk device.

The goal of the Common File System design was to extend the underlying page handling mechanisms so that these same abstractions would be maintained. The major problem is the fact that all processes using a particular file page are not necessarily on the same processor and therefore not referencing the same physical page.

By contrast, a read/write transfer based file system begins with the assumption that the user program sees a copy of the data, and therefore the primary requirement is just to get the bits to the desired place. In VMS, for example, the IO primitives provide no guarantees regarding concurrent reads and writes; rather it is a layered locking mechanism that controls concurrent access.

4 PAGE AND FILE LOCK MANAGEMENT

As noted, a page can not actually be written simultaneously by two processors. If two processors are sharing a page for read-only access, there may be a copy in the memory of each. However, if one is modified, the other is immediately rendered stale. Our initial thinking about this problem was based on a "write token" which would be passed from processor to processor to indicate that writing to the page would be allowed. If a processor did not hold the write token, no process on that processor could write the page, and the paging mechanisms would have to trap and prevent any such access. Upon further thought, it became evident that the "write token" model was insufficient to control access. A finite state machine model was developed, shown below in fig. 1, which uses several state variables.

We had certain desires with regard to the distributed file system data base which this design seemed to support. We felt that a symmetric distributed data base, with no master or central control point, was desirable in keeping with the general symmetry of the CI configuration. With the limited number of hosts in CI-linked configurations and the high speed and reliability of the CI, this seemed a realizable goal. Such a structure would also have advantages in reliability and failsoft characteristics.

The 512-word[1] page, being the unit of protection and mapping granularity, might be considered the obvious unit for interlocking and access control. However, we quickly realized that this would involve excessive data and communication overhead. Rather, the next larger unit, the 512-page "file segment", appeared to provide the right balance

1 - The DECSYSTEM-20 architecture is based on 36-bit words. Some might consider this archaic. We consider it merely out of current fashion.

between granularity and efficiency. In the TOPS-20 file structure, the 512-page file segment is a basic entity handled by the paging mechanisms. Files larger than 512 pages are supported by one additional level in the tree (providing 512 segments of 512 pages each), but this is realized in the file system logic above the level of the paging mechanisms. Thus, the 512-page file segment has a resident database which is accessible to low-level communication code and could serve as the base for distributed locking and access control.

Some analysis of statistics and experience further indicated that little would be gained by interlocking at the page level. Only a small fraction of file access in TOPS-20 involves active simultaneous write, and of these, only cases where different pages are being written within the segment would benefit from per-page interlocking. In cases where all processes accessing a file are read or copy-on-write, separate copies can exist on the different processors and no access interlocking is needed.

The implications of locking at the segment level are that the state and possibly the location of a number of pages may have to change as the result of a reference by this or one of the other processors. The cases are:

1 . If this processor wants to read one or more pages of a file segment, other processors may continue to read it, but none may write it.

2. If this processor wants to write a page of a file segment, other processors may not continue reading any of the pages of it.

3. For other processors to resume reading pages of a file segment after one has written it, three things must happen:

a. The reading processors must discard any pages of the file segment that may be in memory or on swapping storage as they may be stale.

b. The writing processor must write any modified pages back to their home location on the disk.

c. The reading processors may then read new copies of pages on demand from their home location on the disk.

This incorporates the usual readers/writers interlock, but goes beyond that by allowing the operating system to make dynamic changes of state.

The model generated by this approach is that the data base on each processor indicates whether a particular kind of access (read, write) can be made locally. If a desired access cannot be made, a processor will communicate with other processors to request the access. Because the data base is distributed and there is no master, this implies a "broadcast" type of request. The CI does not support broadcast directly, but each host maintains a list of all hosts currently in operation, so a broadcast can be easily simulated. A response of some sort must be received from all other hosts in order for a decision to be made.

Some hosts may have no local knowledge of the file segment in question because no process on that host is using it. Those hosts will respond with approval of the request and will otherwise ignore it. If a host is using a file segment and the request does not conflict with continued use, the host will also respond with approval. Otherwise, the host will respond in a way that indicates that the request will be approved but some cleanup action must be completed first, such as writing modified pages to disk or invalidating local copies.

Processes on different processors may be making conflicting simultaneous requests. In this case, timers are used to control the changes of state so that each processor gets the access it needs for some fraction of the time. The time interval is set large enough to prevent thrashing and excess CI traffic. Again, the processors are assumed to be cooperating, and, in fact, are running the same code, so consistent decisions are made and no central arbitrating authority is needed.

Simultaneous access, either read or write, may be prevented by the mode in which the file is opened. In this case, subsequent open requests on the same or other processors will be checked and refused if inconsistent. Here, no timeouts are needed because the restriction is set by the user program logic, not the underlying operating system mechanics.

The foregoing covers most of the logic necessary for proper management of pages to maintain the virtual memory abstraction and its assumptions. To support this, the low level paging mechanisms must provide:

1. Read-only access per page;

2. Read/write access per page;

3. Cancel write access for physical page;

4. Cancel all access for physical page;

5. Note if page modified;

The existing paging mechanisms provided all of these except #3 which was added by a microcode change to the processor during development.

5 OTHER SYSTEM FACILITIES AND SERVICES

Once virtual memory file mapping works transparently across the cluster, the operating system facilities that use it, including directory logic and disk allocation table management, can also be distributed without major restructuring. As one might expect, those facilities already have locking mechanisms separate from the virtual memory logic to maintain consistent access to the respective structures. That is, each file directory has a lock (mutex) that is set while a process is performing a file lookup or create operation so that other processes will not see the directory in an inconsistent state. Similarly, the allocation tables are locked while the allocator code is selecting a page to allocate.

To extend these locks across the cluster, much of the same logic is used as that which supports the page locking and control. A general interface was implemented for the management of locks in a manner required by the various higher level facilities. The locks represent resources in the system, such as a directory, a file, an allocation table, etc., but the lock manager need not be aware of the mapping of locks to resources. Again, the mechanism is based on a polling of other nodes whenever a lock needs to be upgraded, and one 'no' response means that another node has the resource locked.

To summarize the procedures used to maintain the distributed file and lock data base:

1. The paging mechanisms trap any reference to a mapped file page which does not already allow that type of reference. The kernel then checks the local data base to determine if that access is consistent with the known state of the resource, and if so, the page is made accessible and the program continues. System services needing to lock a resource first attempt to obtain the lock locally. If successful, they continue.

2. If the local state is insufficient, the kernel sends a polling request to all other known nodes on the CI. Each is guaranteed to respond within a few milliseconds under normal circumstances. The response will be in effect, "yes", "no", or "conditional yes". The last means that some housekeeping must be completed before a "yes" can be acted on, and the "conditional yes" allows the requesting processor to know that the request will soon be granted. If a "no" is received, then an interlock is being held by the other processor indefinitely (i.e. subject to higher level code). When all responses have been received, and all conditional "yes" responses cleared, the requesting processor may proceed as in #1. The requesting processor will have updated its local data base so that a subsequent trap to a different page in the same file segment will not have to repeat the polling procedure.

3. Processors may receive polling requests at any time. They are received at interrupt level and handled immediately so that latency does not accumulate. If the request is for a resource which is unknown to this processor, a "yes" response is generated, meaning "whatever you want, it's ok with me". This means that processors do not have to maintain resources in their data base that they are not using.

4. If the resource is in use locally, it will be known. Its local state and the incoming request are compared, and an appropriate response is generated. If the "conditional yes" response is generated, the appropriate housekeeping action is also queued for a local process to perform. At the completion of that, the local process will send a complementary message which removes the condition.

5. Resources are not necessarily removed from the data base when local use is completed, rather

they are left in an inactive state. This allows for a subsequent local request for the resource to be granted immediately rather than requiring a polling cycle. This saves much time in the rather frequent case that a particular resource is used repeatedly on one host and is not being requested by other hosts.

6. Downgrading or releasing a lock does not require communication with other nodes. The local data base is adjusted, and subsequent requests received from other nodes will generate a response based on the new state.

6 RELIABILITY, FAILSOFT

A primary reason for choosing a symmetric distributed data base is its failsoft characteristics. There is no information unique to a node which is lost if that node fails. Of course, failure of a node may cause loss of recently modified pages, directories, etc., but this must be handled by higher levels of software in single-processor systems also. With regard to the locking mechanisms:

1. A node may fail and not respond while a polling cycle is in progress. The rules are that all nodes must respond, so the requesting node will not proceed. The lower level CI service will detect a node going offline (through periodic polling) and notify all clients. The lock manager considers this as one of several events that require the polling to be restarted. Note that the request/response rules result in idempotent requests -- any request may be repeated without causing a problem at the responding nodes.

2. A node may fail while holding any number of resources. When another node issues a polling request for one of those resources, the failed node will not be polled, having already been noted as down by the lower-level CI service. Hence, no "no" votes will be received, and the polling node will assume the resource.

The foregoing means that no special operation is necessary to update data bases when a node goes on or off line, other than updating the list of known nodes. If the node holding a lock fails, then by definition the node can no longer be using the resource. Upon failure, a node will not respond to requests or will not be polled, and other nodes will see no impediment to acquiring the resource. In practice, there is usually no noticable impact on the remaining nodes when one node fails. Particular users may notice a pause in some response if the operation happens to require a resource that was held by the failing system. It may take a number of seconds for the timers to act and remove the failing node from the poll list.

This design requires a strict discipline in bringing nodes on line. In particular, once a node has failed, it cannot be brought back on line except by being reinitialized. Normal startup of a node implies that it holds no locks or resources, and its local database so indicates. This is consistent with any possible state of the nodes already in operation, so the new node can come on line and begin polling for any resources it needs. A failed node has implicitly lost some or all of the resources it held, so it cannot be brought back on line without completely resetting its local data base, including aborting any processes which were using the resources. Considerable interest was expressed in the possibility of supporting some sort of rejoin capability as a matter of principle. In practice, failure of a node is usually fatal and there is no reason for it to rejoin without reinitializing.

Another failure mode which must be considered is loss of CI communication not involving the failure of any node. Because of the redundant and highly reliable design of the CI, this is unlikely. It must be considered, however, if for no other reason than that service personnel might sometime disconnect the wrong cable. To handle this possibility, we implemented an algorithm whereby each node can determine if it has been separated from the rest of the cluster. Specifically, a node which can "see" the central star connector (through a hardware loopback message) will consider itself still part of the cluster, and a node which cannot see the star will consider itself segregated. A segregated node will immediately cease using any physical resources (e.g. disks) that have been used by other nodes, and will refuse to rejoin the cluster even if the connection should be restored. Its action upon seeing the connection restored is to crash and reinitialize.

Several potential race conditions can arise where two or more nodes are coming on line or requesting the same resource at the same time. The symmetric nature of all algorithms would tend to result in multiple connections being opened, a deadlock over resources, etc. These conditions can all be detected and are resolved based on processor serial number. The processor serial number is unique to each processor and, by favoring the higher-numbered processor, the symmetry is broken just enough to avoid the deadlock. The frequency of such occurrances is too low to raise any fairness issues with this solution.

It is a useful design principle that infrequently exercised software logic will tend to develop bugs, and where such logic is involved in failure handling, cascading failures are likely to result. We believe the above approach minimizes the amount of logic that is invoked only on rare failures. Although the CI segregation logic is in fact rarely exercised, it is very simple and makes conservative decisions. That is, it will crash the local node if there is any doubt rather than risk corruption of system file structures through lock mismanagement. Most importantly, it does not attempt complex recovery strategies; its purpose is to detect certain problems and prevent blundering ahead.

7 PERFORMANCE ISSUES

The performance of this system depends on the speed of the lock communication and on the frequency of lock collisions among different nodes. Before implementation began, we considered the sources of possible lock collisions on file data. Statistics gathered over many years indicated that simultaneous shared write was used a very small fraction of the time by user programs. Certain application were known to use it, such as DBMS, but these applications generally had other characteristics and interlocking mechanisms which would also have to be modified for effective cluster operation. On the other hand, simultaneous write was used by the system itself for file directories and disk allocation tables, and we needed to understand more about that use.

To develop this information, we instrumented the system to gather statistics and traces of directory locking. For worst case analysis, we needed to know how often a directory would be used by one job shortly before or after being used by another. While the jobs may often be on the same node in practice, a worst case assumption would be that successive accesses would be from different nodes. From this experiment, we found that most directories had an insignificant frequency of conflict. Certain directories showed a high collision rate -- those that held the commonly used system utility programs and files. Fortunately, most accesses to these directories are read only, and, as we have seen, read accesses do not incur latency or bus traffic. Hence, we were confident that system use of directories would not incur unacceptable performance penalities.

We also gave special consideration to the handling of disk allocation tables. We determined that there would be a much higher frequency of conflict because there is only one such resource for each disk structure, and because most accesses do modify the data. We could see no way to reduce the conflict rate in this case without some additional logic. We chose to add some caching of disk blocks so as to reduce the number of accesses to the actual allocation tables. Rather than allocating one page at a time for a file, a set of pages is pre-allocated. Subsequent growth of the file is taken from the pre-allocated pages.

8 IMPLEMENTATION EXPERIENCE

Of the set of interjob communication facilities listed earlier which were not already distributed, only the file system and part of ENQ/DEQ have been extended to support the distributed cluster environment. Hence, some utilities and services at higher layers which use the unextended facilities also do not support the environment. In many cases, the layered services are being modified to use DECNET in place of some other service, reflecting what we believe is a more extensible facility. Despite the lack of these other facilities, the Common File System facilities alone have proved enormously useful within our development environment.

The processors of a cluster are separate nodes as far as the external network (DECNET, Ethernet) is concerned since each processor has its own connection to the network. However, the users of the cluster prefer to view it as a single computing resource and usually don't care to make any distinction among the processors. A typical step we took to make a cluster look more like a single system was to set up mail files on a common file structure and modify the mail programs to use the same file for any particular user regardless of the node on which the mail arrives. Hence, users can read and send mail while on any node. Over a relatively short time, most user files migrated to common structures and, as a result, users can log into any node and find the same environment.

As is typical of multiprocessor development, debugging the common file system code presented some challenges. Fortunately, TOPS-20 has long had fairly good symbolic debugging facilities, even at the lowest levels of the kernel, including breakpoints, tracing, patching, etc. Of course, hitting breakpoints runs afoul of the various timeouts in the lock managers described above and tends to cause the breaking node to be summarily exorcised from the cluster. To handle this, a piece of protocol is used to declare debugging mode to the other nodes. This effectively extends the timeouts to infinity, so the other nodes will wait as long as necessary for a poll response.

9 SUMMARY

The measure of an operating system is in the abstractions it provides. Do they enhance programming effectiveness? Do they foster effective use of the hardware resources? One test of an abstraction can only be made over time: does it remain viable in new environments not contemplated in the original design? Does it provide transparency and, thereby, portability to layered software when the underlying configuration changes?

Virtual memory mapping as the basic file access abstraction is not present in many operating systems, and some people believe it to be an obsolete technique not compatible with distributed system environments. Our experience with TOPS-20 is that the file system has been extended into a distributed configuration with complete transparency and with entirely acceptable performance. There is literally no assumption built into user programs regarding the file system which is not supported in CFS; therefore, user programs have not had to change to take significant advantage of the new configurations. We believe that the cluster implementation is not limited, relative to a single system, in any way except with respect to simultaneous shared write. And even here, the basic assumptions still work, although at a reduced bandwidth.

We understand however, that the cluster is based on certain assumptions not necessarily true in other distributed environments. A very high speed interconnect and a homogeneous operating system environment are key elements of this design. The virtual memory abstraction is well suited to a high bandwidth, high performance, multi-process environment, and in such environments, it provides a great deal of power to the clients. It also "keeps secrets" about the gritty details of particular interconnects, disk controller characteristics, queueing requirements, memory management, etc. An interconnect of typical wide-area network speed, or a heterogeneous environment involving different file formats, data formats, and access semantics would not be a good basis for a virtual memory abstraction. The power implicitly provided to the client would not in fact be available, and performance would fall short of expectations.

On the other hand, it seems likely that the 10 Mbit ethernet would be a wholly satisfactory interconnect for this purpose, but only between compatible and cooperating operating system types.

10 ACKNOWLEDGEMENTS

TOPS-20 was originally based on the architecture of TENEX[1], and started with a version of TENEX circa 1973. The base virtual memory architecture discussed herein is much as it was in TENEX and is further described in [2].

If the locking rules for CFS seem relatively simple and straightforward in retrospect, the total effort to implement clusters certainly involved many other areas. Design and implementation of protocols for basic CI service, for mass storage data transfer, port hardware, etc. occupied dozens of engineers in Digital. Within the DECSYSTEM-20 group, Peter Hurley proposed many of the original Loosely Coupled Systems concepts. Arnold Miller did the original implementation of the CFS lock manager. Many other members of the TOPS-20 group were

involved in completing, testing, and polishing the implementation, including Clair Grant, Judy Hall, Ron McLean, Dave Lomatire, Tom Moser, and others.

11 REFERENCES

1. "TENEX, A Paged Timesharing System for the PDP-10" - D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, R. S. Tomlinson. CACM Vol 15 No 3, March 1972.

2. "Storage Organization and management in TENEX" - Daniel L. Murphy. AFIPS Conference Proceedings Vol. 41, 1972.

3. "VAXclusters: A Closely-Coupled Distributed System" - Nancy P. Kornenberg, et al. ACM TOCS Vol 4 No 2, May 1986.

4. Digital Technical Journal, Number 5, September 1987. Digital Equipment Corp.