Storage organization and management in TENEX

by DANIEL L. MURPHY

Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

INTRODUCTION

In early 1969, BBN began an effort aimed at developing a new time-shared operating system.* It was felt at the time that none of the commercially available systems could meet the needs of the research planned and in progress at BBN. The foremost requirement of the desired operating system was that it support a directly addressed process memory in which large list-processing computations could be performed. The cost of core storage prohibited the acquisition of sufficient memory for even one such process, and the problems of swapping such very large processes in a time-sharing environment made that solution technically infeasible as well.

Paging was therefore the logical alternative, and our study and experience with list processing systems[1,2] led us to believe that using a demand-paged virtual memory system for such computations was a feasible approach.

With demand paged process virtual memory added to our requirements, we found no existing system which could adequately meet our needs. Our approach was to take an existing system which was otherwise appropriate and add the necessary hardware to support paging. The system chosen was the DEC PDP-10,[3] which, although not paged, was available with a time-shared operating system and substantial support software.

Consideration was given to modifying the existing PDP-10 operating system to support demand paging, but that approach was rejected because of the substantial amount of work which would be required, because of the inherent constraints imbedded in the architecture of any large system, and because development of a new operating system would allow the inclusion of a great many other features and facilities which were judged desirable. Among these were a multi-process job structure with software program interrupt capabilities, an interactive and well human-engineered command language, and advanced file handling capabilities.

Reports of some of the other operating system development in progress at the time suggested that considerable advantages were obtained by generalizing the concept of file storage and integrating process memory with it. Earlier systems had taken the view that files were sequential streams of bytes or words, perhaps with a facility for limited random accessing built on top.

In these earlier systems, process memory was viewed as the equivalent of the physical core memory that a program would see when running stand-alone on a dedicated processor. Time- and core-sharing facilities provided a means for several independent processes to use core and processor concurrently, but the basic concepts still required, for example, a file to be "read in" byte-by-byte or block-by-block into process memory.

The file-process memory integration achieved by MULTICS[4,5] provided an entirely different view of these concepts, and opened up many new possibilities for improved throughput, enhanced ease of programming, etc. The MULTICS segmentation concepts however, would have required substantial modification of the address computation logic of the processor and in other ways seemed to require a level of effort inappropriate to the scale of system we could support. Therefore, we began to examine the ways by which some of these same goals could be achieved in a system which had only paging hardware.

It was known from that outset that our system would contain multi-level storage components. A high speed, rapid access drum would obviously be needed as the swapping facility to support demand paging, and a larger and slower disk storage device (at least 50 million words) was planned for permanent storage. We were

* The work reported here was supported in part by the Advanced Research Projects Agency of the DOD, and in part by BBN.

already using a system, the XDS-940[6] which provided a means of "naming" process storage, and swapping on the basis of the named elements in a process memory. Although the file system was not integrated into this process memory naming scheme, certain basic concepts, e.g., a process memory map into which named elements could be placed, were present.

Thus, having determined that we would build a new monitor system to achieve certain specific objectives, we decided to adopt a more advanced architecture and obtain many other useful features. In particular, we realized that very little if any additional complexity was necessary in the design of the paging hardware in order to provide the base on which a monitor with integrated file and process memory could be built.

The system which resulted from this development effort is called TENEX, and this paper describes the facilities for naming memory and dealing with named memory which were developed and implemented in TENEX. Implementation details of the system are given, including the operation of the three levels of storage, and the flow of data between them.

NAMED MEMORY

TENEX terms and conventions

The discussion which follows will require knowledge of a few of the terms and conventions used in TENEX. The operating system provides a job structure which may contain multiple processes. By a job, we mean a set of active resources normally under control of a single user. That set may in principle be empty, but in practice will always contain at least one process.

In TENEX, each process is provided with an independent process address space, and is capable of performing computation in parallel with other processes. That is, TENEX processes are independent virtual machines with all necessary storage for holding the state of a computation. Various means are, of course, provided for allowing communication and control between processes.

File storage naming

The first and most obvious memory "name" in TENEX is the file name. A powerful and versatile directory and file naming facility is provided in which a particular file is identified by a fixed-depth path which includes device, directory name, file name, extension, and version.

The identifiers in each field (except for device and version) are strings of up to 39 characters. All permanent storage resides in files, so the first step in identifying any particular element of storage is to specify the path name.

It would be both cumbersome and inefficient to require that the file name be used for each operation on a file, even though TENEX provides default conventions which usually allow the user to specify only the name portion of the path. We therefore provide a means of associating the full path name with a small integer called a Job File Number (JFN) which will serve to identify the file over some limited period of time.

The JFN is an important concept in TENEX and deserves some further explanation. The first step in doing any operation on a file is to execute a monitor call giving as an argument the string representing the path name of the desired file.

Various conditions and default options are specified at that time. If the path name correctly identifies a single file, the monitor will return a JFN, and the association of that JFN with the file will remain in effect until the user program explicitly "releases" the JFN (or the job is logged out). JFN's are 18-bit numbers arbitrarily selected by the system, commonly but not necessarily assigned sequentially upward from 0. The domain of a JFN is the job in which it was assigned; therefore it may potentially be used by any process in the job (subject to various protection mechanisms). The system will always know what JFN's are in use in each job and so can assign at any time one known to be unique. It is possible for the same file to be associated with two or more JFN's within the same job (and with JFN's in other jobs), and this often happens when two processes are performing concurrent operations on the same file.

Once the initial association of JFN and file has been established, the JFN is used for all ensuing operations on the file, including sequential reading and writing, opening, closing, etc. The 18-bit JFN is a PDP-10 half-word, and so is conveniently manipulated by the system and user programs. Because the monitor system chooses JFN's to be indexes into system tables holding

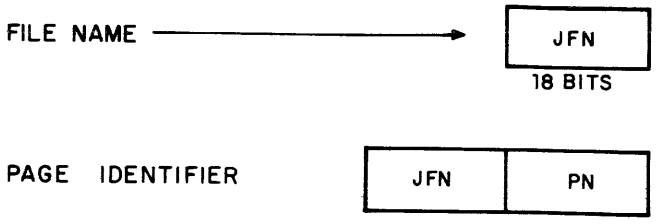

Figure 1

information about the relevant file, the lookup time on individual file function calls is very short and requires only a range test to reject invalid arguments.

Having once identified a particular file and obtained a JFN, a process need only identify the element within the file and the naming process will be complete. On a word-oriented machine such as the PDP-10, the most basic element in a file is obviously the word, but since we are operating in a paged environment, we will want to identify pages. Therefore, our complete identifier is constructed from the JFN of a file, and the page number (PN) within that file, as shown in Figure 1. The paging facilities will allow us then to reference any word within that page as described below.

File-to-process mapping

With the naming of our file memory specified, we next explain how this may be integrated with the address space of processes. As stated earlier, each TENEX process has an independent virtual memory of 256K words, a size fixed by the 18-bit addressing capability of the processor. With the TENEX page size of 512 words, each process virtual memory therefore consists of 512 pages. But these pages are not fixed storage. Rather, each page of the process virtual memory is actually a window through which one can look at a page of "real" storage.

To specify the contents (possibly null) of these windows, TENEX provides a virtual memory map, with one entry for each page of the virtual memory. Each map location is identified by a *map handle* which consists of two items, the process handle (provided by the system when the process was created), and the page number of the desired slot (Figure 2). It is important to understand that the map handle identifies a map slot and does not represent the contents (if any) of that slot. The monitor provides two basic operations for which the map handle is necessary, obtaining the identifier of the present contents of the slot, and placing an identified page into the slot.

This brings us to the basic facility for file/process memory integration. We have constructed a file system in which each page can be named with a convenient (one word) identifier, and we have specified a paged

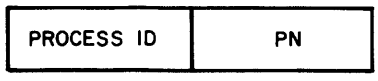

Figure 2

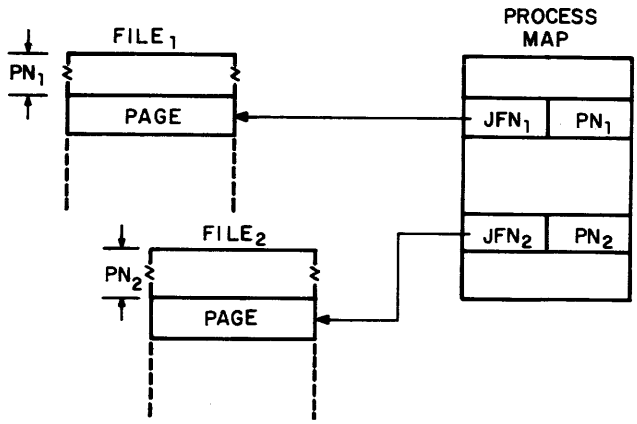

Figure 3—File-to-process mapping

process address space represented by a map into which page identifiers can be put. Figure 3 shows this graphically. The process address space contains pages from two files, indicated by identifiers in the process map which act as pointers to the file pages.

There is some additional information in the map slots not included in these page identifiers, and that is the *access* permission. The TENEX paging hardware provides independent read, write, and execute access control on each page, so when a process places a file page identifier in its map, it must specify which of these accesses (each represented by a bit) is allowed. The system may further restrict the access according to arguments given when the file was opened, which in turn are limited to combinations permitted by the general protection mechanisms associated with file names. Thus the access actually permitted to a mapped page is the logical *AND* of the specific case access request (specified by the process) and the general access permitted to the file (specified by information residing in the file directory).

Sharing named storage

Since the file path names identify files over the domain of all jobs in the system, it is evident that our naming and mapping procedures readily provide a means for sharing storage. Using the appropriate path names (including legality checks), processes in two or more different jobs can identify the same file, and each can obtain a JFN for it. Nothing in the mapping procedures specified above requires that either process be aware of the other's access, and so each process constructs an identifier and places it in its process map (Figure 4). Remember that the JFN is associated with

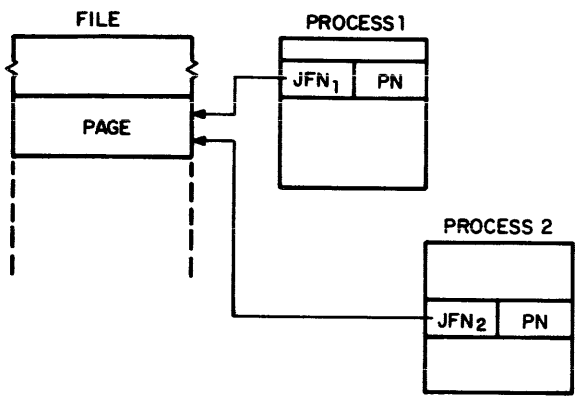

Figure 4—Shared file page

a file only within the domain of a job, and so the two JFN's shown are probably not the same small number. The page number (PN) shown is an absolute address within the file and will appear as the same number in both process maps. Thus two or more processes in the same or different jobs can identify and map the same page of physical storage. The mechanism by which this is implemented is described below.

Along with this basic sharing mechanism, TENEX provides a convention to help ensure that the access to shared or potentially shared information is logically consistent. We identify two cases:

1. A file contains information which must be in a consistent state to be used, e.g. a symbolic text file. Such a file may be read concurrently by several processes, but one process modifying the file precludes any other processes reading or modifying it.
2. A file contains information which, by agreement of the processes involved, can be simultaneously modified and used by several processes, e.g., a common data base or a file used for interprocess communication.

When a process opens a file, it must specify which of these two cases applies. The system will not permit any file to be open both ways at the same time on the grounds that such a situation can only result from disagreement among the processes on how the file is to be used, and is therefore a logical programming error. The monitor will permit any number of simultaneous case 2 openings of a file (which we call *thawed* access), and will allow any of the three types of access legal for the file to be used for each opening. The consistency and integrity of the data in the file is the responsibility of the processes using it.

The monitor will permit any number of case 1 openings of a file (which we call *frozen* access) providing all processes request only read and/or execute access. One or more openings of any type will preclude a new opening for write, and one write opening will preclude any new openings of any type. Thus the system guarantees the integrity of file data by prohibiting potentially conflicting access.

Copy-on-write access

One other important TENEX feature which facilitates sharing is a type of page access called copy-on-write. To our knowledge, this facility was first developed and used on the BBN-LISP system for the XDS-940[7]. It was developed as the result of two common observations:

1. Some programs, particularly older ones, are not quite reentrant. That is, they were coded without observing reentrant coding practices with the result that some code or initial data areas may be modified. Because of the architecture of the PDP-10, we in fact find many programs with completely reentrant code (even lazy programmers usually use the stack-oriented subroutine call and return instructions of the machine), but with local temporaries, data areas, etc., sprinkled arbitrarily through the program.
2. Some programs use large initial data bases which are common to all users, but which may be modified by some users in some specific cases. The principal example of this is the BBN-LISP system which initially contains over 100,000 words of compiled function code (reentrant), and some common list structure. It is however, necessary and legal for some users and some functions to modify portions of this base for local operations. In fact, none of this original base can be guaranteed immune from modification. For example, a list may be appended to, or a compiled function may have a "break point" temporarily inserted.

In TENEX, a process may specify this copy-on-write access whenever a file page is mapped into a process. Copy-on-write is legal even if write access is not. A page mapped in this way will remain shared so long as the process only does read or execute references. A write reference to the page will be trapped by the monitor, whereupon a private copy of the page will be made, and the process map changed so that it points to the copy rather than the original. Write access is then permitted to the copy, and the process' original write reference is completed.

All of this is invisible to the process, except that it may read its memory map and discover a different identifier and access than was initially used to map the page. This facility thus provides a means for allowing sharing wherever possible without penalizing unavoidable modifications or requiring the user program to handle them explicitly.

Examples of use of named memory

Let us consider the most common example of how file/process memory integration and sharing is used in TENEX, i.e., a file containing a commonly used program. We will identify this file as PROGRAM.SAV (the extension SAV by convention implies a core-image file). The file contains a number of pages of code and some mapping information as shown in Figure 5. The mapping information specifies where the code and data pages are to be placed in a process map to produce an image of the program. A monitor routine interprets the mapping information and performs the mapping. As shown in the figure, the code and data pages are arranged contiguously in the file, but may be put anywhere in the process map. In fact, the mapping shown is a common one, with data and temporary storage assigned to low addresses, and reentrant code assigned to addresses in the upper half of the process address space.

One might suggest that instead of placing pointers to the file in the process map, the file map itself be used as the process map. This would be analogous to running in a particular segment in the MULTICS-type segmentation scheme. But without the full power of general segment addressing, inter-segment references are not possible, and our procedure offers the following advantages.

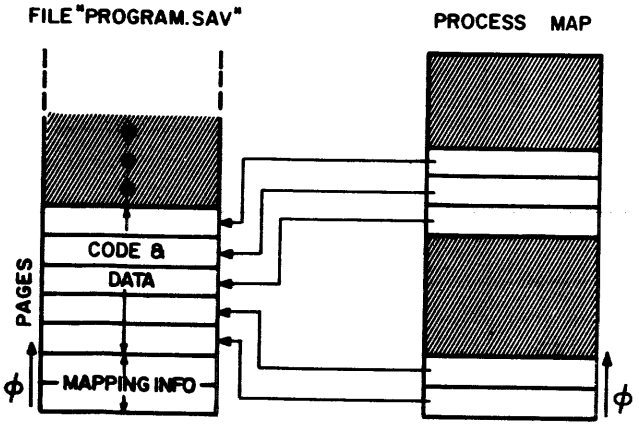

Figure 5

1. A process map may contain pages from several different files. In our scheme, individual pages or groups of pages may be viewed as mini-segments, and used in similar ways.
2. Different processes may have different access permissions to the same file page. In particular, when a write reference is done to a copy-on-write page, only one entry of the process map is changed to address the copy.

Sequential file access

While mapping operations are readily suggested in the case of program core images, it must be noted that the only basic type of file access permitted under TENEX is page mapping. TENEX provides a number of monitor facilities for other types of file access, the most common of which is sequential. To implement the file sequential monitor calls (e.g., byte-in, byte-out) the monitor maintains a number of "window" pages in a separate map invisible to the user process. For each file with sequential operations in progress, the monitor maps the file page which is to receive or provide the next byte. Each call from the user causes one or more bytes to be loaded from or stored into this page, and a count updated to determine if a new page should be mapped. Movement through the file is accomplished by mapping successive pages, and the sequential access module does not have to be aware of the physical device on which the page resides nor interface with I/O driver modules to read or write it. This modularity is very satisfying from an operating system design point of view.

As a final example, we note that processes may use shared file pages for interproccess communication. In this case, a particular file and set of pages within the file are agreed upon by several processes, and the pages are mapped into the address space of each of the processes. The actual map slots chosen by the processes need not be the same, i.e., the shared pages may be put in different places in the various process address spaces. Since the same physical storage is seen by all processes, any of a number of common techniques may be used to pass information in any direction, e.g., flags, ring buffers, etc.

In itself, this procedure does not provide any direct means for processes to signal one another, so for asynchronous events the processes are required to periodically test flag words in one of the shared pages.

IMPLEMENTATION

Pager

As stated above, paging hardware was designed and built as part of the TENEX development, and a few of

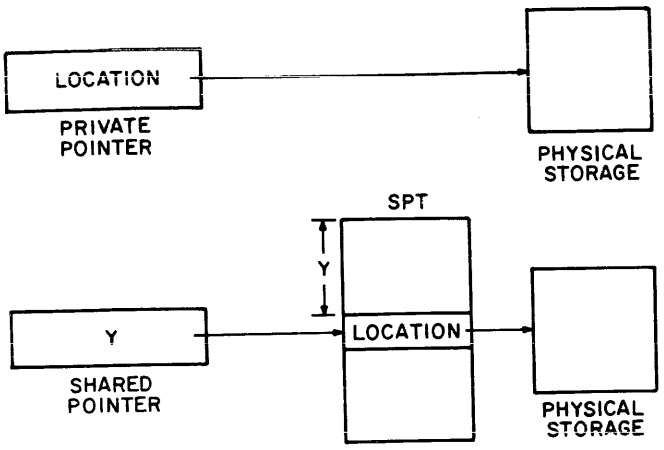

Figure 6—Pointer types

the characteristics of the BBN Pager are particularly relevant to this discussion. The pager is placed logically between the processor and the core memories and translates each memory address received from the processor into a physical core address which is sent to the memories. Control signals allow the pager to know what type of access the processor is making (read, write, or execute), and allow the pager to signal the processor when for some reason a reference cannot be completed (e.g., the page is not in core). The virtual addresses received from the processor are 18-bits, and the page size is 512 words, so the pager is in fact translating the high-order 9 bits of address, and passing the low-order 9 bits through unchanged.

The pager uses a set of associative registers to hold some number of recent virtual/physical address associations, but the source of this information is always a "page table" in core memory. Page tables contain (or point to) the physical storage address, if any, of each page of a virtual memory. Thus, each process virtual memory is represented by one page table. Page table entries are one word, hence a page table for a 256K virtual memory is 512 words, or exactly one page.

The pager references the relevant page table, using the 9 high-order virtual address bits as an index, whenever the associative registers fail to contain the requested virtual address. It is capable of interpreting three types of page table entries of which two are of interest here. The first is called a "private" pointer and contains a physical storage address. If this is a core address, the pager will load an associative register with the information and complete the requested reference. If it is any other address, the pager will initiate a trap to the monitor for appropriate action. The second type of page table entry is called a "shared" pointer, and contains an index into a system table at a fixed location. This "shared pages table" (SPT) contains the physical storage address, and the details of its function are described below.

These two pointer types are shown in Figure 6. The third type of page table entry is the "indirect" pointer described in Reference 8, but it is not relevant to this discussion.

One other fixed table, called the Core Status Table (CST), is used by the pager. For each page of physical core, this table contains information about recent references and notes if the page has been modified.

Hierarchical storage considerations

In any system using hierarchical storage, one is concerned with the movement of data between the various levels, with knowing where the current "up-to-date" copy is, with updating lower levels, etc. It is usually considered essential that the address of the currently valid copy of an item of storage reside in one and only one place. This tends to conflict with the goal of sharing which says that items of storage should be made available to many processes simultaneously. Replication of addresses would appear to admit the possibility of unresolvable phase errors, and the updating problem by itself would introduce undesirable complexity in the software.

One quite elegant solution to this problem is the hash table scheme which is shown in Figure 7. In this scheme, storage addresses reside in only one place, the storage hash table. Processes using an element of storage are given the "home" (and presumably invariant) address of the element, and the current location at any time may be found by performing a hash lookup into the table. Using this scheme, storage elements may be moved from place to place at any time, and only the table entry need be changed. Also, the table entry itself may be deleted when the element is moved back to its home address even though one or more processes are still using it. In this case the hash lookup will fail, and the monitor will have to re-create the entry.

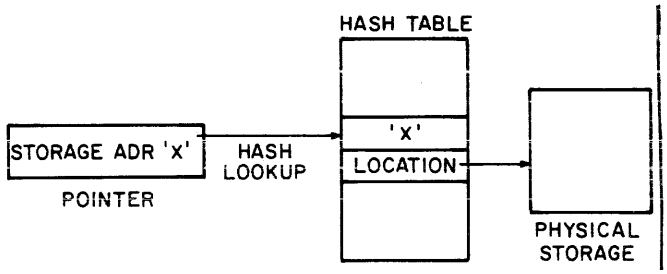

Figure 7—Hash table scheme

A second solution to the basic storage management problem is the shared pages table scheme used in TENEX and shown in Figure 6. In this scheme, storage addresses (for shared elements) again reside in only one place, a fixed table called the shared pages table. Processes using an element of storage are given a fixed index 'Y' which identifies the SPT entry holding the current address. Here, also, storage elements may be moved from place to place by changing only one address, but unlike the hash table scheme, an entry cannot be deleted from the SPT so long as pointers exist which use it. Therefore a share count is required for each entry to record the number of pointers to it which have been created.

We considered both of these schemes and a number of variations for TENEX before choosing the second of the above approaches. An exhaustive justification of this decision cannot be given here, but the decision was based primarily on our judgment that:

1. The cost of hardware to implement the hash table scheme was somewhat higher in terms of design effort and overall size and complexity.
2. Additional (time) overhead would be incurred in making the one or more probes into the hash table for each associative register reload.
3. The resident storage requirement of the hash table scheme would be greater.

TENEX implementation—mapping

We are now ready to show exactly how TENEX implements the file mapping operations discussed in the previous section, and how data flows between the several levels of storage. The TENEX storage hierarchy consists of three levels, core, swapping, and file. In practice, the swapping device is a fixed head drum with high transfer rate and fairly short latency time (e.g., less than 30 ms.). The file storage device is usually a movable head disk with substantially greater capacity, but reduced transfer and latency speeds.

As described in the previous section, named memory

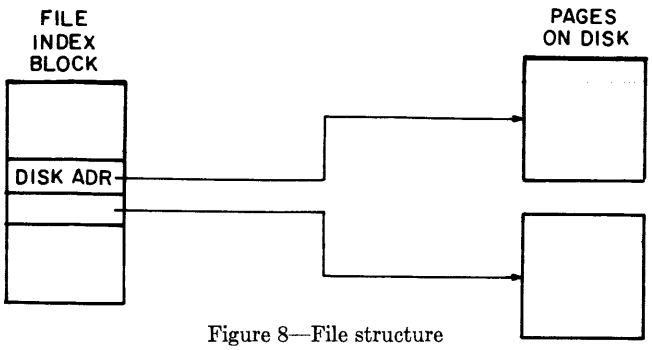

Figure 8—File structure

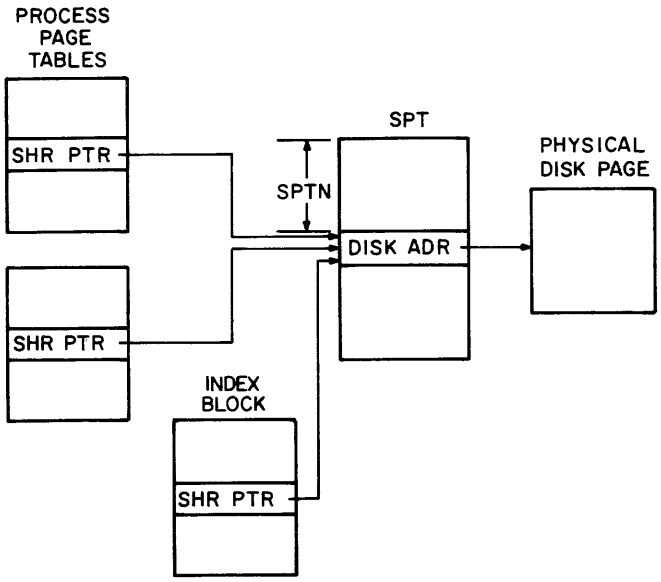

Figure 9—Two processes map a file page

consists of pages within files, so we start with an example file and two of its pages as shown in Figure 8. The basic structure of the file is an index block containing the storage addresses of all of the data pages. This index block is in fact a page table, initially containing private pointers. We assume a starting point where none of the file pages are mapped in any process, so the "one and only one" place for the storage address of each of these file pages is logically and properly the index block of the file which owns them.

Next, a process requests that one of these file pages be mapped into its address space. The monitor uses the JFN portion of the identifier to locate the file index block, and the PN (page number) portion to select the appropriate entry within it. Although our aim here is to have just one process using the page, we see that in fact the page must become shared at this point, that is, shared between the file and the process. Therefore, the monitor will assign a slot in the SPT and place in it the disk address obtained from the file index block. Simultaneously, it creates a shared pointer which points to that SPT slot and places a copy in both the file index block and the process page table. The share count for the SPT slot is set to reflect the fact that the page is in use twice, once by the file, and once by a process. A second process wishing to use the page proceeds in the same manner, but now it is only necessary to create another copy of the shared pointer and increment the share count. This situation is shown in Figure 9. The subsequent reduction of the share count to 1 (when all processes unmap the page) will indicate that the SPT entry may be reclaimed.

Some additional bookkeeping is necessary in order to keep track of the owner of the page, and the fact

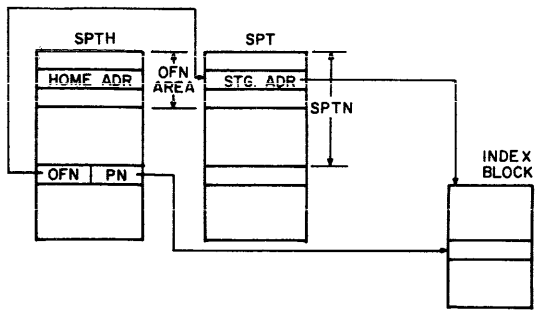

Figure 10—Ownership back pointers

that the file index block is in use. This is shown in Figure 10. The table labeled SPTH is a table parallel to and the same length as the SPT. For our example file page which was assigned slot 'SPTN', the parallel entry in the SPTH records the owning page table of the page. This is shown as OFN and PN. The OFN (open file number) is the monitor internal equivalent of the user's JFN, except that it identifies open files over the domain of all jobs in the system. The OFN is actually an index into a portion of the SPT which is reserved for index blocks, and the PN is the page number supplied by the user. The OFN portion of the SPTH holds the home addresses of the currently in use index blocks. The monitor must always open files on the basis of the storage address of the index block as obtained from the file directory, and a search of this part of the SPTH is necessary to determine if the file is already open.

Inter-level data flow

Next we show what happens when one of the processes references the file page which has been mapped. This is shown in Figure 11. The pager interprets the shared pointer found in the process map, and references the SPT. It finds, however, that the page is not in core and traps to the monitor. The monitor in turn selects

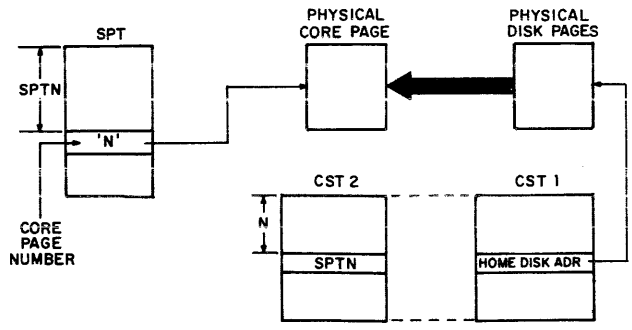

Figure 11—Page is referenced and brought into core

a page of real core and initiates a read of the to disk bring in the page. The SPT slot is then changed to indicate that the page is in core.

For completeness, we must note the function of two tables which record the state of physical core. These are the Core Status Tables (CST1 and CST2). For each page of physical core, CST1 holds the physical address of the next lower level of storage for the page. In our current example, this is a disk address because the page is just being read from the disk. CST2 records the name of the page table holding the pointer to that core page, which in this case is an SPT index. One additional bit (not shown) is used to record whether the page has been modified with respect to the next lower level of storage.

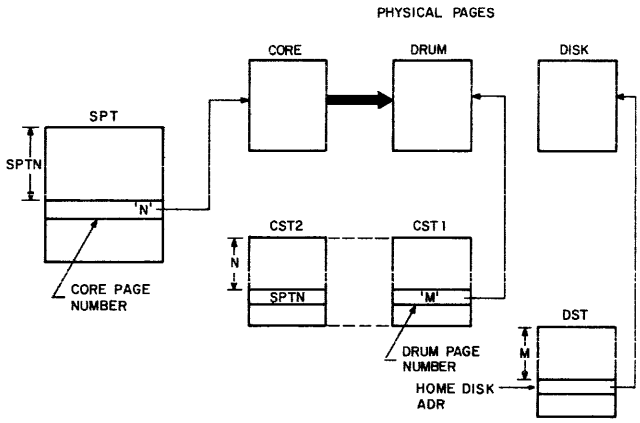

Figure 12—Page is swapped onto drum

Next we consider what is necessary for the monitor to swap the page onto the drum. It is important to note that during the course of the drum write (including latency) and for a period of time thereafter, the core page still contains a current copy of the data, and so we may properly leave the SPT slot pointing to it. This will prove useful in the event that a process makes another reference to the page during this time because the page will not have to be read into core again. Thus to begin the swapout, the monitor selects a free drum page, initiates the drum-write operation, and updates CST1 to reflect the fact that the next lower level of storage is now the drum.

However, we can't discard the home address of the page, so one other table is required. The DST (drum status table) serves a function for the swapping level of storage equivalent to that of the CST for core. That is, for each page in use on the drum, the DST holds the address of the next lower level of storage. It also records whether the copy on the drum has been modified with respect to the copy on the disk so that the monitor will

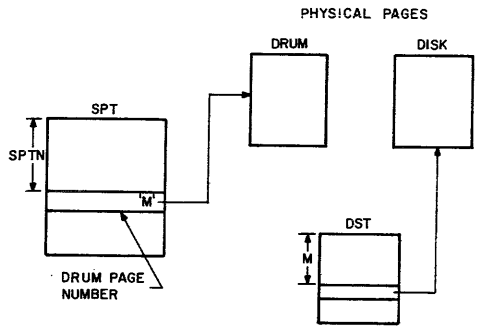

Figure 13—Core page is released

know whether a write is necessary at some time to update the disk copy. Our picture of a file page with copies on all levels of storage is now complete (Figure 12).

One final step is shown in Figure 13. If the page remains unreferenced for some period of time, the monitor will want to use the core page for some other purpose. To do this, the monitor will move the drum address from CST1 of the page being reclaimed to the SPT slot, and succeeding attempts to reference the page will discover that it is no longer in core.

Updating lower levels

So long as the page remains mapped by one or more processes, the share count will keep the SPT slot in use, and our convention is that the page will be moved between the drum and core as needed. This suggests that some procedure may be necessary to periodically update the home (disk) copy of pages. This is necessary both to guard against loss due to system crash, and because some files are mapped when the system starts up and are never unmapped (e.g., the disk assignment bit table). In TENEX, a special system process takes this responsibility. It periodically scans the open files, finding pages which have been changed since being read from the disk. File pages are backed up to the disk by setting a request bit in the CST which causes the swapper to move the page to the disk instead of the drum. File index blocks must also be updated but require a different procedure. For these, the backup process constructs an image of the index block as it would appear with no pages shared. That is, it finds the home address of each page and puts it in the index block in the form of a private pointer. This copy is then written on the disk. This procedure is a compromise of the goal or having only one copy of a storage address, but a simple interlock mechanism prevents any phase errors during the updating.

Dynamic storage management

One of the most important and difficult aspects of storage management in TENEX is the dynamic control of core and flow between levels of storage. The pager provides information on the frequency and type (read/write) of references made to pages in core. It also provides information on which of the processes sharing a page (i.e., having it mapped) have actually referenced it. A detailed description of these facilities and the algorithms which have been developed to handle dynamic storage management is beyond the scope of this paper.

SUMMARY AND CONCLUSIONS

This discussion has shown how named memory can be incorporated in an operating system having only paging facilities, and how some of the advantages of segmentation are thereby obtained. Although there are limitations to this approach, it does have the advantage of considerably less complex hardware and software. To date, we have not found a way to use mapping to provide dynamically linked library subroutines, one advantage which segmentation does provide. One possible solution may be to build a library of self-relocating subroutines and provide a convention for mapping them in a portion of the address space which the calling process is not using. Unfortunately, the PDP-10 processor does not provide a convenient facility for self-relocating code.

We have found that the process memory map is an extremely useful facility for a number of purposes. It is true that the 256K virtual memory eliminates the need for overlaying procedures in most programs, but where this technique is still required, it is easily implemented simply by remapping groups of pages.

The implementation of a three-level storage hierarchy used in TENEX has proved to be workable in over two years of actual operation. The software complexity required for the maintenance of the various tables is perhaps greater than would be required had we adopted the hash-table approach, but it has nonetheless been a manageable and programmable system.

ACKNOWLEDGMENTS

In addition to the author, T. R. Strollo, R. S. Tomlinson, J. D. Burchfiel, and E. R. Fiala actively

participated in the design of this implementation strategy. R. S. Tomlinson and J. D. Burchfiel did the logic design and checkout of the Pager. Appreciation is also due in large measure to J. I. Elkind and D. G. Bobrow whose inspiration, leadership, and support made the TENEX project possible.

REFERENCES

1 D G BOBROW D L MURPHY
The structure of a LISP system using two-level storage
Communications of the ACM Vol 10 No 3 March 1967

2 ———
A note on the efficiency of a LISP computation in a paged machine
Communications of the ACM Vol 11 No 8 Aug 1968

3 DIGITAL EQUIPMENT CORP
PDP-10 reference handbook Dec 1971

4 V A VYSSOTSKY F J CORBATO R M GRAHAM
Structure of the MULTICS supervisor
Proceedings AFIPS 1965 FJCC Vol 27 Pt 1 Spartan Books New York

5 R C DALEY P G NEUMANN
A general purpose file system for secondary storage
Proceedings AFIPS 1965 FJCC Vol 27 Pt 1 Spartan Books New York

6 B LAMPSON et al
A user machine in a time sharing system
Proceedings IEEE 54 12 Dec 1966

7 D G BOBROW D L MURPHY W TEITELMAN
The BBN-LISP system reference manual
BBN April 1969 pp 3.8–3.9

8 D G BOBROW J D BURCHFIEL D L MURPHY R S TOMLINSON
TENEX, a paged time sharing system for the PDP-10
Communications of the ACM Vol 15 No 3 March 1972