TENEX, a Paged Time Sharing System for the PDP-10

Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S. Tomlinson
Bolt Beranek and Newman Inc.*

TENEX is a new time sharing system implemented on a DEC PDP-10 augmented by special paging hardware developed at BBN. This report specifies a set of goals which are important for any time sharing system. It describes how the TENEX design and implementation achieve these goals. These include specifications for a powerful multiprocess large memory virtual machine, intimate terminal interaction, comprehensive uniform file and I/O capabilities, and clean flexible system structure. Although the implementation described here required some compromise to achieve a system operational within six months of hardware checkout, TENEX has met its major goals and provided reliable service at several sites and through the ARPA network.

Key Words and Phrases: TENEX, paging, virtual machines, time sharing system, scheduling algorithm, process structure, PDP-10

CR Categories: 2.44, 4.32, 4.39, 4.42

Presented at the Third Annual Symposium on Operating Systems Principles, Palo Alto, California, October 18–20, 1971.

1. Introduction

TENEX is a new time sharing operating system implemented on the DEC PDP-10. It was developed because no existing system of the appropriate size and cost could meet the requirements of the research projects at BBN. During the development phase of TENEX we formulated a set of goals which we hoped would produce a workable and well-integrated system, as well as help us to realize our specific requirements. Our background included knowledge and use of a number of other systems including the DEC PDP-1 systems designed at BBN[1], the Berkeley system for the SDS 940[10], MIT CTSS[3], the DEC 10/50 system[5], and MULTICS[4]. We used good design ideas from each of these systems, and tried to avoid what we felt were problems of operation and implementation which we saw in these systems. The scale of the system and available resources imposed certain constraints on the implementation, including (1) that minimal change be made to the PDP-10 processor and none to the basic address computation, and (2) that the system had to be in service for users within six months of the operation of the hardware and at that time dominate our previous service, the DEC 10/50 and the SDS 940 systems.

Our design goals, presented below, are generally understood. Some, however, are often overlooked and usually not emphasized. We considered all of these important in the design of TENEX. Our design goals

* Computer Science Division, 50 Moulton Street, Cambridge, MA 02138. The work reported here was supported in part by the Advanced Research Projects Agency of the DOD, and in part by BBN.

fall into three broad categories with several specific objectives under each.

I. State of the art virtual machine.

a. Paged virtual address space equal to or greater than the addressing capability of the processor with full provision for protection and sharing.

b. Multiple process capability in virtual machine with appropriate communication facilities.

c. File system integrated into virtual address space, built on multilevel symbolic directory structure with protection, and providing consistent access to all external I/O devices and data streams.

d. Extended instruction repertoire making available many common operations as single instructions.

II. Good human engineering throughout system.

a. An executive command language interpreter which provides direct access to a large variety of small, commonly used system functions, and access to and control over all other subsystems and user programs. Command language forms should be extremely versatile, adapting to the skill and experience of the user.

b. Terminal interface design which facilitates intimate interaction between program and user, provides extensive interrupt capability, and full ASCII character set.

c. Virtual machine functions which provide all necessary options, with reasonable default values simplifying common cases, and require no system-created objects to be placed in the user address space.

d. The system should encourage and facilitate cooperation among users as well as provide protection against undesired interaction.

III. The system must be implementable, maintainable, and modifiable.

a. Software must be modular with well-defined interfaces and with provision for adding or changing modules clearly considered.

b. Software must be debuggable and reliable, allowing use of available debugging aids and including internal redundancy checks.

c. System should run efficiently, allow dynamic manual adjustment of service if desired, and allow extensive reconfiguration without reassembly.

d. System should contain instrumentation to clearly indicate performance.

2. Hardware Development for TENEX

Hardware development and modification for TENEX was limited to that necessary to achieve the goals specified above. This effort included an address mapping (paging) device implemented with then current DEC modules and some changes to the PDP-10 processor. A hard limit on the latter resulted from the lack of physical room available in the processor. Both processor changes and paging facilities had to be designed to allow the standard DEC time sharing software and diagnostics to run.

2.1 The BBN Pager

The BBN pager is an interface between the PDP-10 processor and the memory bus. It provides individual mapping (relocation) of each page (512 words) of both user and monitor address spaces using separate maps for each. The pager uses "associative registers" and core memory tables to store the mapping information. On each memory request from the processor, the 9 high-order bits of the address and the request-type level (read, write, execute) are compared in parallel with the contents of each associative register. If a match is found, the register containing the match also contains 11 high-order address bits to reference up to 1,048,576 words of physical core.

If no match is found, reference is made to a 512 word "page table" in physical core memory. The word selected in this page table is determined by a dispatch based on the original 9 high-order address bits. In the simple case of a private page which is in core, the 11 high-order address bits and protection bits are found in this word and are automatically loaded into an associative register by the pager.

There are three other cases:

1. The page is not in core, is protected from the requested type of access, or is nonexistent; in this case a page fault (trap) will occur.
2. The page is shared; in this case the map contains a "shared" pointer to a system table which contains the location information for the page.
3. The page belongs to another process; in this case, the entry contains an "indirect" pointer to an entry in another page table from which the location information is obtained.

The goal of program (code and data) sharing was given extensive consideration in the design of the BBN pager. The indirect and shared pointer mechanism allows pages to be actively shared (be represented in more than one address space) but still have the current address (core or secondary storage) stored in only one place. This allows memory control tables and data structures to be kept simple and the memory management software to move pages without extensive computational overhead. The pager permits individual pages to be shared for write as well as read references, so two or more processes may communicate by sharing

a common page into which any or all may write. Rather than enforce a discipline of pure procedures, with private data in another segment, a unique "copy-on-write" facility allows users to share large portions of an address space containing procedures and data, and to obtain private copies of only those pages which are changed. This is implemented through an independent per-page status bit, available to users, which will produce a trap on a write reference, and a system procedure by which a private copy is then created. For example, this permits shared programs to be prepared with preconstructed data areas, which will be kept shared if not modified and will be put in private storage *if changed.*

One final, unique feature is that the pager maintains a record of the activity of the pages in core memory in a "core status" table. The pager notes when a page has been referenced, which processes have used that page, and whether the page has been written into. This information is used by the memory management software to be described.

2.2 Processor Modifications

Except for the pager trapping facilities, all of the TENEX virtual machine facilities (monitor calls) are reached via a new system call instruction, JSYS, added to the PDP-10 processor. JSYS provides an independent transfer mechanism into the monitor which does not conflict with "UUO" system calls used by DEC software. It accomplishes a transfer from the user program to the specified monitor routine in one instruction time via a block of cells called the JSYS transfer vector. The JSYS address provides the index to the proper transfer vector entry. The state of the processor, including the return address, is stored in a location specified by the transfer vector, usually in a separate data area, so that a JSYS call is suitable for reentrant code. The JSYS transfer vector occupies exactly one page in the monitor space and could be mapped independently for each process, but this is not done in the current system.

There exists a context, either user or monitor, for each instruction execution. The JSYS system call may be executed in either context, with the "callee" operating in monitor context. One hardware modification to the PDP-10 adds a bit to the state word of the processor to record the context of the "caller." Two ways of accessing the caller memory context were added to the PDP-10. One is an execute instruction which allows current or previous context to be specified for each memory reference of the object instruction. The second access instruction group moves data between the AC's (general registers) of the current context and memory of the previous context. AC's from the previous context are accessed using a pager " AC base register" which specifies the location of the stored AC's.

3. The TENEX Virtual Machine

A user process running under TENEX operates on a virtual machine similar to a PDP-10 arithmetic processor with 256K of virtual memory. The paging hardware traps processor references to any data not in core, and a core manager performs the necessary I/O to make the referenced page available. Such traps are invisible to the user process.

The virtual processor does not make available to the user the direct I/O instructions of the PDP-10. But through instructions which call monitor routines, the virtual machine provides facilities that are considerably more powerful and sophisticated than typical hardware configurations used directly.

3.1 Virtual Memory Structure

The TENEX virtual memory may be viewed as a linear address space of 256K words, and programs may use it in this fashion. However, the existence of the paging hardware means that the monitor must deal with memory in pages of 512 words, and some of the power which the mapping hardware provides is accessible to a user program.

The contents of the virtual memory are specified by the *virtual memory map* of 512 slots which the user may read or write via monitor calls. The contents of each slot specify the page in that position in the virtual address space, and the type of access allowable (read and/or write and/or execute) for that page. In the simplest case, a map slot may contain (a pointer to) a private page, i.e. a page shared with no other processes in the system. A private page is automatically created whenever a process makes a reference to a page for which the map slot is empty. A slot may also contain an indirect pointer to a page in this or some other process. A memory reference to a location in such a page will be executed just as though the instruction had directly addressed the page pointed to. Any change made to the page by either process will be seen by both processes. If the owner of the page changes the contents of his memory map, then the process with the indirect pointer will see the change. A virtual memory slot may also contain a pointer to a page from a file in the file system, as discussed later.

3.2 Job Structure

A job is a set of one or more hierarchically related processes, and it has the following attributes.

1. The name of user who initiated the job.
2. An account number to charge costs associated with use of system resources.
3. Some open files.
4. A hierarchy of running and/or suspended processes.

A job may also have one or more terminal or other devices assigned and attached.

3.2.1 Process Hierarchy. TENEX permits each job to have multiple simultaneously runnable processes.

The relationships among them are defined by a structure which looks like an inverted tree defined by the capability for direct control and killing. A process always has exactly one immediately superior process and may have one or more inferior processes. Two processes are said to be parallel if they have the same immediate superior. In TENEX, a process may create processes inferior, but *not* parallel or superior in the structure. A process can communicate with other members of the structure by (a) sharing memory, (b) direct control (superior to inferior only), or (c) pseudo (software simulated) interrupts as described in Section 3.3.

Although not completely general, a tree structure process heirarchy implicitly provides the protection and reference facilities that are wanted in most applications. These include referencing inferior processes as a class for freezing, killing, and resuming; fielding of interrupts and special conditions by a superior process; and protection of the superior process from inferiors.

Currently in TENEX, multiple processes are used:

1. To enable the EXEC to run user programs, handling faults, and servicing user requested interrupts.
2. To allow programs to block for an arbitrary set of events; one process waits for each event and signals the main process when it occurs.
3. To implement an invisible debugging program, completely protected from malfunction of the program under test.

3.3 Pseudo Interrupt System

TENEX provides a facility for a process to receive asynchronous signals from other processes or from terminals or as the result of its own execution. The various processes in a job may explicitly direct interrupts to each other for purposes of communication. A process may enable an interrupt which will occur whenever the user hits a particular key on the controlling terminal. Finally, a process may use the pseudo interrupt system to detect any of a set of unusual conditions, including illegal references to memory, processor overflow conditions, end-of-file, and data errors.

3.4 Other Monitor Functions

Other functions which form a part of the virtual machine include:

1. Functions which provide information to the program about the state of the system or job (time of day, runtime used, name of user, etc.).
2. Functions which save and restore the computational environment of a process to allow restarting of a suspended program.
3. Functions which provide frequently needed forms of I/O conversions, such as fixed or floating point number input and output, and date and time to string conversions.

3.5 Backward Compatibility (DEC 10/50 Monitors)

Since TENEX was being implemented on a machine for which a large useful program library existed, mostly for use under the DEC 10/50 time sharing monitor, we felt it was highly desirable to be able to run such programs under the new monitor system. We felt it should be possible to run *binary* images of old programs, i.e. without reassembling.

Toward this end, all of the TENEX monitor calls were implemented with the JSYS instruction, reserving all old monitor calls for their previous use. Secondly, routines were designed which implemented all of the existing 10/50 monitor calls in terms of the available TENEX monitor calls. This set of routines implements all of the functions available in the 10/50 monitor except those specifically intended for the maintenance of the system. Assembled together as a compatability package, they occupy slightly less than 2.5K of core. The package is kept as a core image file and is never seen by programs which use only TENEX monitor calls. However, the functions are automatically made available to 10/50 type programs by the monitor. When a program makes its first 10/50 type monitor call, the TENEX monitor maps the compatability package into a remote portion of the process address space, an area not usually available on a 10/50 system. Subsequent 10/50 type monitor calls cause a transfer to the compatability package which then interprets the call.

The compatibility routines are placed in the user space for several reasons: (a) regular use can be made of the pseudo interrupt system; (b) the compatability package (which requires constant maintenance) can be maintained as a separate module, totally independent from the monitor; and (c) the monitor is protected from malfunction by the compatibility routines.

4. User Interaction with TENEX

4.1 Terminal Interaction Capabilities

The terminal service module of TENEX was designed to provide any type of interactive behavior a program might find useful. Many programs, especially the command language interpreter described below, benefit by having many short interactions with the user, often one or a few characters. Full-duplex terminals are preferred for use with TENEX for these reasons and for the reason that the user can in fact anticipate the machine's responses and begin typing input before output is completed. Algorithms for echoing typein ensure that the typescript is an accurate record of the dialog. Half-duplex terminals may be used but at some cost in convenience.

4.2. Executive Command Language

Users at terminals communicate and work with TENEX primarily through a command language interpreter called the TENEX Executive, or EXEC. The EXEC

is an interactive, well human-engineered program which can accept commands from a user's teletype or from a file. It is implemented as a reentrant, shared program which runs in user mode, usually as the top level process in the structure.

The EXEC provides the user with a multitude of facilities which are activated by simple, easy-to-learn commands. These facilities provide access to the system (e.g. LOGIN); utility operations on files and file directories; initiation of private programs and subsystems; limited debugging aids; initiation of batch (detached operations); printout of user information and system statistics; and system maintenance.

The EXEC was designed with two primary objectives—ease of learning and ease of use. To ease the learning process, all commands are English words which are descriptive of the facility being activated (e.g. COPY to copy information from one file to another, STATISTICS to obtain a listing of current system statistics). Each command begins with a keyword. Depending on the command, the initial keyword may be followed by arguments, such as file names, numbers, and additional keywords, and/or "noise words" to make the command more readable. The noise words are enclosed in parentheses to distinguish them from the arguments.

In order to help novice users, two special assistance features were incorporated. First, when the EXEC requires input from the user during a command interaction (for instance, to collect arguments of that command), a cue is typed to indicate to the user what is expected. For example, an interaction which renames a file might be

@ **REN$AME** (EXISTING FILE) **ALPHA$.MAC** (TO BE) **BETA**

The user's input is bold face. The $ indicates a typed ESC (ASCII escape, code 33_8) which invokes the EXEC's verbose cueing responses in parentheses. In this example the user typed only three letters REN followed by ESC which invoked command completion by the EXEC, a feature which makes the language particularly easy to use. An ESC after any initial substring of a command or argument (such as a file name) invokes completion. If the substring is insufficient for unique identification of the intended input, the EXEC rings the teletype's bell and awaits additional characters. If the initial substring cannot be recognized the EXEC types "?" to ask the user to retype that input. If the novice user still doesn't understand what is expected in his response, he may invoke the second special assistance feature by typing the character "?". This causes the EXEC first to type out a list of all options available to the user at that point and then request a response.

To summarize, three general styles of input may be used, distinguished by syntactic structure, and including special input terminators. Hence the styles do not require different input modes, and thus may be intermixed freely within a session or even within a statement, adapting to the state of knowledge and verbosity of the user. The input styles allow:

1. *Complete input.* A complete command may be typed in, with all keywords and noise words given in their entirety and without use of any nonprinting characters.
2. *Abbreviations.* The user may shorten a command in two ways: he can omit noise words completely and he can shorten keywords. Any keyword may be abbreviated with any initial substring (terminated with space) long enough to distinguish it from the other keywords acceptable in that context.
3. *Completion.* The user types the same characters as in abbreviated input, except he terminates each field (keyword or argument) with the ESC key. This produces a print-out of the complete command—each ESC causes the rest of the field (if an abbreviated keyword or file name) and any following noise words (with enclosing parentheses) to be printed.

The EXEC also provides editing characters to permit the user to correct typing errors in his input. These editing characters permit the user to delete the last character of his typed input, the last word, or all of it. He can also ask for his edited input to be retyped for clarity.

4.3 Interrupt and Escape Characters

ASCII Control-C is the EXEC's attention character. When typed by the user, it causes any running program to be stopped and control to be given to the EXEC via the pseudo interrupt system. The user may then continue his program or take any other action.

Another terminal interrupt character, Control-T, is serviced by the EXEC. It interrupts a user's EXEC process to type out the total CPU and console time used and the status of the process being run under the EXEC; this lower user process continues.

5. The Tenex File System

The TENEX file system provides a general mechanism for obtaining information from and sending data to external devices attached to the TENEX system [12]. Write-only and read-only devices are included in the file system so that all TENEX I/O may be handled uniformly. The first major function of the TENEX file system is to provide symbolic file name management. This includes two separate but related activities. The first involves translation of a symbolic name into an internal "file descriptor block" pointer associated with that name; the second involves checking information concerned with (1) the file status, e.g. whether it

exists, access rights, etc.; and (2) the process requesting access to the file. This second activity, known as File Access Protection, determines if this process should be allowed to know about the existence of this file, and if so, what access it is allowed.

A symbolic name for TENEX files consists of up to five fields and thus conceptually represents a tree of maximum depth five. Not all nodes of this tree go down to maximum depth. This scheme was chosen rather than a full tree to simplify the problem of compatibility with existing DEC PDP-10 software and name lookup and recognition. We are currently considering the feasibility of implementing a full tree directory structure similar to MULTICS [11]. At each level there would be a set of information, which is related to access rights, and media dependence of the data access for this node. Each node would represent a collection of related information, with the terminal nodes being files.

The fundamental unit of storage in a TENEX file is a byte, which may be from 1 to 36 bits in length. A stream of bytes constitutes a file, which is the basic named element in the file system. Programs may reference files byte by byte in a sequential manner or, if the device permits, at random. String (multiple byte) transfers can also be made. No structure other than bytes and files is imposed on the user, and byte and string input and output are the basic operations. Of course, additional structure and other operations may be implemented by the user programs.

5.1 File Names

A TENEX file is named by a file descriptor composed of five fields some of which are omitted for certain devices. The five fields are device name, directory name, file name, extension, and version number.

The file name field is intended to designate a class of files which are related in some way. This convention is not enforced, but most users of TENEX follow the convention since it facilitates management of a user's files. The extension field is intended to designate variously processed forms of the same information. A file's extension is frequently specified by a program. For example, PROG.MAC, PROG.REL, and PROG.SAV would be used to indicate the MACRO assembly code source, relocatable file, and binary image of a single program.

The version number of a file enumerates successive versions of a file. Normally each time a file is written a new version is automatically created by making its version number be one greater than the highest existing version. This protects a user from loss if he accidentally writes on the wrong file. Excess versions may be deleted by the user, or automatically by the system, when they have been put on a backup storage medium.

Any of the fields of a file description may be abbreviated except for device and version. The appearance of an ESC in the file descriptor causes the portion of the field before the ESC to be looked up, and the system will supply the omitted characters and/or fields. Abbreviation without this output is not provided in order to insure that the typescript reflects exactly what was done. The system provides default values for each field except the file name.

A default value is used for a field if the user omits any input for that field, e.g. the device and directory. This simplifies references to files in most common cases.

5.2 File Access Protection

Because TENEX must service a diverse user community, it is essential that access to files be protected in a general way. Generally, access to a file depends on two things: the kind of access desired and the relation of the program making the access to the owner of the file. Presently, a simple protection scheme is implemented in which the only possible relationships a program may bear to the file's owner are:

1. The directory attached to the job under which the program is running is the same as the owning directory.
2. The directory attached to the job under which the program is running is in the same group as the owning directory.
3. Neither 1 nor 2.

Five kinds of access are distinguished for a file: directory listing, read, write, execute, and append. The above three relationships and five protection types are related by 15 bits (a 3×5 binary matrix) in which a one indicates that a particular access is permitted for a particular relationship. If directory listing access is not permitted, the process requesting access is given an error return which is indistinguishable from the error for nonexistent file. This is important if the information that a file exists should not be generally available, as is the case for secure systems. Other access restrictions cause errors only when an attempt is made to open a file, as described below.

For purposes of determining group access, a 36 bit word is administratively associated with each directory and each user. If the bitwise "and" of the user group word of the accessor and of the directory group word of the accessee is nonzero, the group access permission is used.

Provision has been made for a more general file protection system in which more general access relationships may be expressed in a special file protection language. For example, access may be allowed only to an explicitly named set of users.

5.3 File Operations

Using a file in TENEX is basically a four step process: first a correspondence is established between a file name and a Job File Number (JFN), which is a small index into a job table for files; next the file is opened, establishing the mode and access permission and setting up monitor tables to permit the data of the file to be accessed; third, data is transferred to or from the file;

and finally, the file is closed, fixing up the directory information and releasing the space occupied in system tables for the file.

For purposes of file sharing, all instances of opening a particular file should reference the same data. Data written in a file will be immediately seen by readers of the file. To protect against confusion resulting from multiple uncooperating simultaneous writers and readers of a file, a file can be opened with what we call *thawed* or *unthawed* access. With thawed access, a file may have any number of thawed writers and/or thawed readers, but no provision is made to guarantee that information is in a consistent (frozen) state. With unthawed access, a file may have any number of unthawed readers, or exactly one unthawed writer; this prevents any potentially conflicting operations. Simultaneous accessors of a file must be all thawed or all unthawed.

6. The Monitor

6.1 Scheduler

The TENEX scheduler is designed to meet a set of potentially conflicting requirements. The first is to provide an equitable distribution of CPU service, which we define as at least $1/N$ of real time where there are N jobs on the system. Secondly, because TENEX is designed to be a good interactive system, the scheduler must identify and give prompt service to jobs making interactive requests. Thirdly, because use of the CPU is intimately tied to the allocation of core memory, it must make efficient use of core memory to maximize CPU usage. Finally, the scheduler should have provision for administratively controlling the allocation of resources so as to obtain other than equal distribution if desired.

6.1.1 Balance Set Scheduling. For the scheduler, we want a coherent policy which obeys Denning's "working set principle" [6]. A priority rating, as described below, is given to each runnable process in the system, and an estimate is made of the working set size of each process. The jobs with highest priority whose total working sets will fit in core are called the balance set and may be run concurrently. When any process in the set page faults one of the others in the balance set is given CPU service. Denning [7, 8] has shown that such balance set scheduling minimizes thrashing and tends to maximize system efficiency. Periodic monitoring of the entire set of runnable processes for changes in priorities and in working set sizes allows adjustment of balance set membership.

6.1.2 Setting Process Priorities. To implement the basic scheduling function, a scheduling algorithm was chosen which groups processes together on a number of separate queues each with an associated runtime quantum, similar to algorithms described by Corbato [3] and BBN [1]. Lower queues in general have lower priorities but longer runtimes. A common problem with many schedulers of this type is that processes are placed on the highest priority queue after any interaction. Under conditions of heavy load or with poorly behaved interactive processes, it may happen that the interactive processes succeed in using all of the available time and so lock out the compute-bound processes which have fallen to the lower queues.

In TENEX, priority is based on a long term average ratio of CPU use to real time, and a process's priority after an interaction is determined by its priority before the interaction and the length of the interaction. Specifically, a process's priority is decreased while running at a constant rate, C, and increased while blocked at a rate of C/N, where N is the number of runnable processes in the system. This ensures that equitable service is given both to compute-bound and interactive jobs.

To improve response characteristics, an interactive "escape clause" is included in the scheduling algorithm. After a block wait of greater than minimum time, a process is given a short quantum at maximum priority. Priority and queue position after this burst are determined by the long term average. The effect of this provision is to ensure quick service to very short interactions, even when requested immediately after a long computation.

6.1.3 Resource Guarantees and Limitations. In some cases CPU resource guarantees independent of load are desired, e.g. a demonstration which requires significant CPU time during a period of medium or heavy load, or a user who is willing to pay extra for premium service which does not degrade as the load on the machine increases. A facility is implemented in TENEX to handle these situations.

A person with appropriate administrative access can assign to any job on the system a fraction, F, of guaranteed CPU service. For any job so designated, the scheduler will attempt to ensure that

$$C/T > F,$$

where C is the CPU seconds used by the process, and T is the real time since the process last unblocked. For example, if the parameter is set to 30 percent, the scheduler will provide at least 18 seconds of CPU service to the specified job during each minute of real time.

This parameter acts as a ceiling as well as a floor for CPU service. That is, if there are other runnable processes on the system which are not declared special, then the scheduler will ensure that the special process receives no *more* than the stated fraction of CPU service. We have found that a process with this sort of resource guarantee displays very consistent interactive behavior despite widely varying loads on the time sharing machine.

6.2 Core Management

The information provided in the core status table by the paging hardware is essential to the proper management of core memory in TENEX to avoid thrashing and other forms of inefficient operation. Paging is done on demand. No ordinary pages are preloaded before a process is run, and in general, a process will not have all the pages of its virtual memory in core at once.

When a process references a page which is not in core, a pager trap occurs and a core management routine is invoked. The run time since the last page fault is used for a running average of page fault times, i.e. interfault intervals in process time. A process is considered to have enough of its working set if its average page fault time equals PAV, a system parameter currently set to 67ms (or 2 drum revolutions). If the process is faulting more often than PAV, it is considered below its working set size, and a swap to bring in the requested page initiated. Control returns to the scheduler so that it can run another process until the swap is complete. If the process is faulting less often than PAV, a core management routine is invoked to reduce the size of the process, i.e. remove some of its pages from core if space is needed. We have found this algorithm is stable and works well; Denning and Schwartz [9] show formally that this is a good idea.

To reduce the size of a process working set, a least recently used algorithm is used. The age of each page of a process is determined by a 9 bit logical age field stored by the pager in the core status table when a reference to that page causes a pager reload. Since collecting pages is costly, all sufficiently old pages are removed on a working set reduction. A quick reference however can catch a page before it leaves physical core.

6.3 System Measurements

In order to observe and improve the performance of TENEX in regular service, various measuring functions were built into monitor routines. Some measures serve to indicate the efficiency of scheduling, the core/CPU balance, and the nature of the various processes running on the system. The scheduler maintains a set of integrals over time which give (as a fraction of real time):

IDLE, time when no processes are requesting CPU service.

WAIT, time when all runnable processes are waiting for completion of page fault.

CORE, overhead time spent in core management.

TRAP, time spent handling pager traps.

Two relationships among these are:

S = IDLE + WAIT + CORE = total time in scheduler.

REALTIME − S − TRAP = time spent running user processes.

Also maintained by the scheduler is an integral over time of the number of processes in the balance set, the number of transfers between core and secondary storage, and the number of terminal interactions.

One measure is of interest on a recurring basis to all users of the system. The scheduler maintains three exponential averages

$$A(T+t) = A(T)* \exp(-t/c) + N*(1-\exp(-t/c)),$$

with time constants c equal to 1, 5, 15 minutes and N equal to the number of runnable processes on the system. This indicates the true current load on the system better than the number of jobs logged in. Users often choose on the basis of these load figures what they do on the system at a particular time. Three figures are better than just the last one, because from these the user can predict the trend as well as note local variation.

6.4 Debugging Aids

Certain debugging procedures and aids used in the development of the system contributed greatly to the speed of development and integrity of the system. Our principal debugging aid is a program called DDT, available in several forms in the system. DDT is a program which allows memory locations to be examined and modified, and breakpoints (return of control to DDT) to be placed in a running program. All interactions with DDT are symbolic, using the symbols defined in the source program and obtained from the assembler.

The form of DDT first used and still necessary for debugging basic level code is a stand-alone version which resides in core memory along with monitor. It is used for debugging the scheduler, portions of the core manager, and other basic routines. The second form of DDT was added as soon as the basic monitor could support demand paging and create a virtual memory. This DDT exists in the monitor map and may be used as an ordinary program at a system terminal. It is capable of examining and changing the running monitor and all of the associated tables and other contents of the monitor virtual memory. Use of this form of DDT actually allows several persons to work on debugging portions of the monitor simultaneously. A third form of DDT is used with user programs and is cognizant of the access status (execute or write protection, etc.) of pages of the user program.

Debugging the system was further facilitated by the use of a considerable number of internal redundancy and consistency checks which call one of two recovery routines upon detecting any malfunction. If the system is attended by system personnel, these routines enter a DDT breakpoint and the state of the monitor can be examined to determine what has gone wrong. This has enabled us to find and correct the obscure or infrequent faults in the software. If the system is unattended, then, depending on which routine was called and the severity of the fault, one of three things happens: (1) operation is continued with no interruption; (2) one process is crashed; or (3) the

system is automatically reloaded and restarted. This usually provides continued operation in the event of unexpected hardware or software malfunction. Both routines cause the source of the inconsistency and a message describing the problem to be logged for further action by system personnel.

7. Conclusion

One of the most valuable results of our work was the knowledge we gained of how to organize a hardware/software project of this size. Virtually all of the work on TENEX from initial inception to a useable system was done over a two year period. There were a total of six people principally involved in the design and implementation. An 18 month part-time study, hardware design and implementation culminated in a series of documents which describe in considerable detail each of the important modules of the system. These documents were carefully and closely followed during the actual coding of the system. The first stage of coding was completed in 6 months; at this point the system was operating and capable of sustaining use by nonsystem users for work on their individual projects. The key design document, the JSYS Manual (extended machine code), was kept updated by a person who devoted full time to insuring its consistency and coherence; and in retrospect, it is our judgment that this contributed significantly to the overall integrity of the system.

We felt it was extremely important to optimize the size of the tasks and the number of people working on the project. We felt that too many people working on a particular task or too great an overlap of people on separate tasks would result in serious inefficiency. Therefore, tasks given to each person were as large as could reasonably be handled by that person, and insofar as possible, tasks were independent or related in ways that were well defined and documented. We believe that this procedure was a major factor in the demonstrated integrity of the system as well as in the speed with which it was implemented.

Acknowledgments. In addition to the work done by the authors, significant contributions to the design and implementation of TENEX were made by T.R. Strollo, who led the technical staff, and J.R. Barnaby, who implemented the EXEC subsystem. Others who contributed are T. Myer, E. Fiala, D. Wallace, and J. Elkind.

References

1. BBN Medical Information Technology Department. The hospital computer project time sharing executive system, BBN Rep. No. 1673, Apr. 1968.
2. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time sharing system for the PDP-10. BBN Rep. No. 2180, Aug. 1971.
3. Corbató, F.J., et al. An experimental time-sharing system. Proc. AFIPS 1962 SJCC, Vol. 21 Spartan Books, New York, pp. 335–344.
4. Corbató, F.J., et al. An introduction and overview of the Multics system. Proc. AFIPS 1965 FJCC, Vol. 27 Pt. 1, Spartan Books, New York, pp. 185–196.
5. Digital Equipment Corp. PDP-10 Reference Handbook. Dec. 1971.
6. Denning, P. The working set model for program behavior. *Comm. ACM 11*, 5 (May 1968), 323–333.
7. Denning, P. Thrashing, it's causes and prevention. Proc. AFIPS 1968 FJCC, Vol. 33 Pt. 1, AFIPS Press, Montvale, N.J., pp. 915–922.
8. Denning, P. Equipment configuration in balanced computer systems. *IEEE Trans. Comput. C-18*, 11 (Nov. 1969), 1008–1012.
9. Denning, P., and Schwartz, S.C. Properties of the working set model. *Comm. ACM 15*, 3 (Mar. 1972), 189–196.
10. Lampson, B., et al. A user machine in a time sharing system. *Proc. IEEE 54*, 12, (Dec. 1966), 1766–1774.
11. Spier, M.J., and Organick, E. The Multics interprocess communication facility. Proc. Sec. Symp. Oper. Sys. Princ., Oct. 1969, ACM, New York, pp. 83–91.
12. Wilkes, M. *Time Sharing Computer Systems*. American Elsevier, New York, 1968.