JSYS Traps—A TENEX mechanism for encapsulation of user processes*

by ROBERT H. THOMAS
Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

INTRODUCTION

The JSYS Trap mechanism is an extension to the TENEX operating system[1,2] which enables a process** to define and control the virtual machine seen by other processes. Using the mechanism, a process can control the execution environment of other processes by providing them with a virtual machine that enlarges, restricts or completely redefines the "standard" virtual machine provided by TENEX.

The controlling process does this by declaring that it wishes to "monitor" (trap) selected system calls (JSYS's) when executed by other (inferior) processes. When a monitored process attempts to execute one of the calls specified it is suspended and the monitoring process is notified. After gaining control, the monitoring process may take whatever action it finds necessary. For example, it may choose to perform the call itself on behalf of the trapped process. Alternatively, it may allow the trapped process to perform the call itself, or it may first modify the call parameters and then allow the trapped process to resume normal execution of the system call.

Mechanisms similar to JSYS traps have been proposed in the context of TENEX and elsewhere.[3,4] The motivating forces that transformed the JSYS trap mechanism from an idea to a design and implementation for TENEX were the requirements placed on TENEX by the Resource Sharing Executive (RSEXEC) system.[5] The RSEXEC system is being developed as part of a research project in distributed computation.

One of the goals of RSEXEC is to enable the various TENEX Host computers† on the ARPA Computer Network[6,7] to function together as a single, multi-host TENEX system. RSEXEC provides an environment within which the resources available to a user are enlarged to include those beyond the boundaries of his local TENEX Host. It does this in a way that removes the logical distinction between resources which are "local" and those which are "remote". This applies to both the user at the "command language" level and his programs at the "executing process" level.

An important part of the RSEXEC environment is a distributed, multi-Host file system which allows files to be referenced without requiring Host specification. That is, a process need not be aware of the location within the network of files it uses in order to access them. Whenever a process attempts an operation involving a non-local file, the operation is dispatched across the network to a cooperating process running on the appropriate remote Host. As a result, existing "subsystems", such as text editors, assemblers and compilers, need not be rewritten to operate in the multi-Host environment.

We initially conceived of RSEXEC as an evolutionary system whose development would require considerable experimentation. Consequently, we decided that, at least initially, the RSEXEC environment would not be implemented as part of the normal TENEX operating system. Rather it would be provided by "ordinary user" processes which would act on behalf of processes attempting to access non-local resources. To provide the environment in this way it is necessary that ordinary user processes be able to intercept system calls made by other processes before the operating system itself acts upon them. JSYS traps were implemented to provide such an encapsulation mechanism.

Although the JSYS trap mechanism was strongly motivated by the RSEXEC application, it represents an important and powerful addition to the TENEX operating system which is useful in a general manner in applications requiring a controlled execution environment. This paper describes the trapping mechanism and records design and implementation decisions that were made in adding it to the existing TENEX operating system. In doing so, the paper describes some aspects of TENEX not previously reported.

The next section is a brief sketch of the TENEX virtual machine. Following that, the JSYS trap mechanism is described in more detail. First, it is described in terms of the properties we wanted it to exhibit and the constraints that consistency with the existing TENEX virtual machine placed upon it. Next, we describe the user's view of

* This work was supported by the Advanced Research Projects Agency of the Department of Defense under Contract No. DAHC15-71-C-0088.

** Almost any of the common definitions for the term process is adequate for the needs of this section. TENEX supports the concept of a tree structured process hierarchy described more fully in subsequent sections. With the exception of when it interacts with other processes, a TENEX process proceeds asynchronously as if executing on its own machine.

† There were thirteen TEXEX Hosts on the ARPANET as of October 1974.

JSYS traps. Finally, we view its implementation. The paper concludes by comparing the trapping mechanism with similar features in other operating systems.

THE TENEX VIRTUAL MACHINE

TENEX is a time-shared operating system developed by BBN to run on the DEC PDP-10 processor augmented with paging hardware. TENEX provides a multi-process job structure with software program interrupt capabilities, advanced file handling features and an interactive and carefully human-engineered command language. At present (October 1974) there are fourteen TENEX systems. This section focuses on the system call and multi-process facilities of the TENEX virtual machine. Readers interested in other aspects of TENEX are referred to the literature.[1,2,8,9,10]

A user process running under TENEX executes on a virtual machine similar to a PDP-10 processor[11] with 256 K words of virtual memory. The direct input/output instructions of the PDP-10 are not available to user processes. Rather, the virtual machine provides input/output facilities which are considerably more powerful and sophisticated.

All of the virtual machine facilities* are accessed via a system call machine instruction, JSYS,** which was added to the PDP-10 processor for TENEX. The JSYS instruction accomplishes a transfer of control from a user process to the monitor routine that implements a particular system call in a single instruction time. The hardware interprets the address field of the JSYS instruction as an index into a transfer vector called the JSYS dispatch vector. The JSYS dispatch vector occupies exactly 1 page (512 words) in the monitor address space.*** TENEX users have come to regard the different system calls supported by the JSYS instruction as separate instructions. Thus, one speaks of the "Byte In" JSYS and the "Open File" JSYS, etc. This convention is used throughout the remainder of this paper.

When a user logs into TENEX a job consisting of a single process is created for him. By using appropriate system calls that process may create other processes which themselves may create further processes, etc. TENEX provides a separate virtual machine with its own address space for each such process. Each process has exactly one immediate superior (its creator) and may have any number of immediate inferiors (processes it has created). Thus the process hierarchy is tree-structured; the root of the tree being the process created at login time.

TENEX currently provides three mechanisms for inter-process communication:

1. Communication by direct process control whereby one process modifies the state of another.
   The state of a process includes its execution status (i.e., running, suspended by another process, blocked for input/output, etc.), program counter (PC), active registers (ACs) and the contents of its address space. A process can modify the PC and ACs of other processes and can start, stop and destroy them. The capability for direct process control is defined by the process hierarchy; processes may directly control only their inferiors.
2. Communication by pseudo-interrupt whereby one process transmits an interrupt signal to another.
   The signalling process specifies the target process and an interrupt channel. To receive the signal properly, the receiving process must have previously "armed" the specified interrupt channel by activating it, assigning it a priority, and specifying a routine to be executed whenever a signal for the channel occurs. The identity of the signalling process is not conveyed as part of the interrupt signal. In addition to other processes, a process may receive pseudo-interrupt signals from devices such as terminals and as a result of its own execution (e.g., arithmetic register overflow).
3. Communication through shared memory whereby communicating processes read and write from the same memory.
   The paging hardware partitions memory into pages of 512 words each. Each process sees a linearly addressable virtual memory of 512 pages which is defined by a memory map with an entry for each page. Each map entry describes a page in the process address space: an indication of whether the page exists, its physical location (i.e., current location in core or secondary storage) and the type of access the process has to the page. Processes can arrange to share portions of their address spaces by system calls that manipulate memory maps. For example, process A can share page 3 of its address space with page 5 of process B; any change to the shared page made by either process will be seen by both.

The multi-process features play an important role in standard TENEX operation. The process created for the user at login time runs the TENEX command language interpreter (EXEC). When a user invokes a subsystem (e.g., text editor) or a program of his own, the EXEC creates a process, whose initial virtual memory contains the program. After starting it, the EXEC blocks until the process terminates (see Figure 1). One interesting subsystem is IDDT, an interactive, "invisible" debugger,[12] which runs in a process inferior to the EXEC and superior to the process(es) being debugged. IDDT uses TENEX facilities for memory sharing and direct process control to enable a user to monitor (examine registers, address space,

* With the exception of the pager trapping facilities that implement the virtual memory and which are invisible to user processes.
** pronounced JAY-sys
*** For addresses greater than 511, the address is interpreted as an index into the user process address space and the process is dispatched to a routine in its own address space.

etc.) and control (start, stop, place "breakpoints", modify address space, etc.) the execution of a process in a manner that is transparent to the process.

DESIGN CONSIDERATIONS

Our goal was to add to TENEX a facility enabling one process to control the execution environment of another. The mechanism we chose was one in which the process being controlled is suspended whenever it attempts to execute JSYS's (system calls) previously specified by the controlling process to which control is then passed. The major constraint in designing the trapping mechanism was that it be done within the context of the existing operating system. Specifically, the mechanism had to be compatible with the TENEX virtual machine and its implementation could not require radical departure from the approach taken to the rest of TENEX. Of course, its implementation should not require excessive per process storage, should involve minimal overhead to processes not using it, and should not be excessively costly to those that do use it.

The following summarizes the major considerations that influenced definition of the JSYS trap mechanism:

1. The capability of a process for setting JSYS traps should be limited to processes inferior to it in order to provide a measure of protection that is consistent with the TENEX process hierarchy.
2. A process setting traps for another should be able to specify an arbitrary subset of JSYS's to be trapped rather than being required to specify only all or no JSYS's. This enables the inferior process to execute efficiently, incurring the overhead of being trapped only for those system calls the superior is interested in intercepting. In addition, it provides a measure of convenience for the trapping process; it need be programmed only to handle those JSYS's it is interested in. Furthermore, to allow for generality and flexibility a process should be able to dynamically remove traps it has set. It should not, of course, be able to remove traps set by other processes.

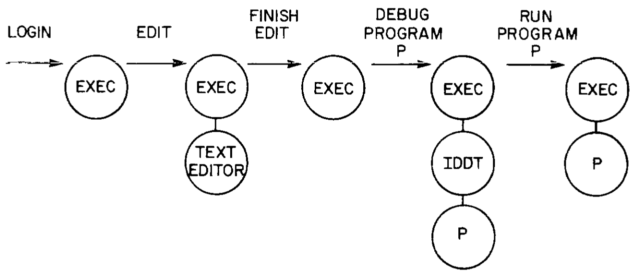

Figure 1—The process structure for a user job changes throughout the course of a TENEX session. The TENEX EXEC (command language interpreter), which resides in the top process in the job process hierarchy, creates and manages other processes as the user's requests dictate

3. The traps set for a process should be inherited by its inferiors. That is, when a process is created, it should be subject to the same traps as its creator. Additionally, when traps are set for a process, they should also be set for all existing processes inferior to it. This ability to set traps *indirectly* allows a process to control the virtual machine seen by all its inferiors without requiring that it know the details of the inferiors' process structure. In addition, it prevents a trapped process from using inferior processes to execute (trapped) system calls on its behalf in order to bypass the trapping mechanism.
4. A trapping process should be able to allow a process that has been suspended as a result of executing a trapped JSYS to resume "normal" execution of the JSYS that caused the trap. This is useful in situa~~tions in which one process is monitoring another~~. For example, a process which may not be completely trustworthy could be encapsulated by a monitoring process which would trap operations that are potential security violations in order to prevent it from writing "private" data to a "public" or non-secure "area". The monitoring process would allow the inferior to resume execution of such an operation only after checking the call parameters to ascertain that the operation is "safe".
5. Control should propagate up the process hierarchy from controlling process to controlling process. When a process attempting to execute a particular JSYS is suspended, control should be passed to the nearest superior in the hierarchy that requested to trap that JSYS. If that process resumes the suspended process without changing its PC and execution status, control should pass to the next superior in the hierarchy handling that JSYS. Should each controlling process in the hierarchy resume the trapped process without resetting its PC, "normal" execution of the JSYS should be resumed. This insures that each process in the hierarchy wishing to trap the JSYS has a chance to handle it. Additionally, it prevents a process from bypassing the trapping mechanism by creating inferiors and then trapping and immediately resuming their calls executed on its behalf.
6. The trapping mechanism should be transparent to the trapped process. In particular, the execution of a given JSYS should appear to be the "same" to the executing process whether or not the JSYS is trapped by a superior process. This permits existing programs to run in a trapping environment without requiring that they be rewritten.
7. To allow for flexibility and generality, a process should be able to use JSYS traps to control its various inferior processes differently. That is, it should be able to specify a different set of JSYS's to be trapped for each inferior.

The transfer of control from the trapped process to the trapping process involves suspension of the former and no-

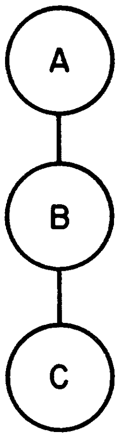

Figure 2—Process A may not directly set traps for Process C. However, process C inherits any traps set by process A for process B

tification of the latter. We chose to have notification of the trapping process occur via a pseudo-interrupt signal. Two other approaches suggested themselves:

1. The trapping process could use a periodic polling procedure to look for processes suspended as the result of traps it had set; or
2. The trapping process could execute a system call causing it to block until the "next" trap occurred.

The first alternative was rejected immediately on efficiency grounds because it requires a "busy wait" by the trapping process. The second was judged to be less flexible than the pseudo-interrupt approach because it requires the trapped process to relinquish control and therefore to remain idle while awaiting the next trap. If this effect is desired, a user can achieve it in a straightforward way using existing system calls in conjunction with the trap pseudo-interrupt. Furthermore, since the implementation would be virtually identical for both the pseudo-interrupt and blocking approaches, we selected the more flexible pseudo-interrupt approach.

As an implementation consideration, we restricted the ability of a process to set traps beyond that suggested in consideration (1) above. A process can *directly* set and remove traps only for processes that are *immediately* inferior to it. To allow a process to set traps for non-immediate inferiors would not violate the TENEX process hierarchy. However, it would require a considerably more complex implementation, particularly in terms of maintaining the data base required to describe the trapping situation (see "Implementation" Section below). Because traps are inherited by inferiors in the process hierarchy (consideration (3) above) this is not a severe restriction. For example, for the situation in Figure 2, process A may directly set traps for process B but not for process C; however, any traps set for B are inherited by C. This restriction also prevents a process from trapping its own execution of JSYS's. While there are situations in which this would be useful, we felt that the additional implementation complexity required to support the capability was unjustified.

USERS VIEW OF THE TRAPPING MECHANISM

The trapping mechanism was made available to user processes by augmenting the virtual machine with several new JSYS's (system calls). The basic calls are summarized below in an informal notation that conveys their meaning while avoiding the details of TENEX programming. Values returned by a call are indicated on the left side of an "=" sign.

To set or remove JSYS traps the following calls are used.

set-traps (proc, trap-spec)
remove-traps (proc, trap-spec)

Proc is the process ID of an immediately inferior process and *trap-spec* is the address of a table specifying the JSYS's for which traps are to be set or removed. A process can declare the channel on which it wishes to receive pseudo-interrupt signals resulting from JSYS traps by the call:

set-trap-channel (chn)

where *chn* is a channel number for the interrupt channel.

The call to determine the source of a trap pseudo-interrupt is:

proc, call=trap-data ()

Proc is the ID of the process that was suspended attempting to execute the JSYS *call.* Before such a pseudo-interrupt can occur, the trapping process must have previously set a trap for the *call* JSYS in *proc* and declared a channel for trap interrupts. To respond to a trap, the trapping process may use any of the operations normally available for direct process control: it can read the parameters supplied by the trapped process, set the value (if any) to be returned to the trapped process, change its PC, modify its address space, change its execution status, etc. After the trap has been handled, the trapped process may be allowed to resume execution using the call:

resume-trapped-proc (proc)

where *proc* is the process to be resumed.

A process may use the call

t=test-trap ()

where the value returned is either true or false, to determine whether traps have been set for it by a superior process. Since all JSYS's including those for managing traps may be trapped by superiors, this call need not violate the transparency property (6 above) desired for the trapping mechanism: A process could prevent inferiors from determining whether they are being trapped by trapping their execution of test-trap and always returning the value false.

An example should clarify how the trapping mechanism can be used. Consider the simple task of generating a frequency histogram of system calls made by an arbitrary program. A process Q can do this by creating another process P to run the program and then trapping and recording the JSYS's P executes. The following annotated program fragment describes Q:

```
    .
    .
    .
(enable pseudo-interrupt-system)
set-trap-channel(n)          //Assign n as channel for trap
                             //pseudo-interrupts.
P=create-proc(file)          //Create P initializing its address
                             //space to the program stored as
                             //file
set-traps(P,ALL)             //Set traps for ALL JSYS's
                             //executed by P.
start-proc(P)                //Start program execution by P.
wait-proc(P)                 //Wait until P terminates.
output(Histogram)            //Output the Histogram array.
    .
    .
    .
PSI-Handler-n:               //JSYS trap interrupt handling
                             //routine
  R,i=trap-data()            //Read trap data.
  Histogram(i)=Histogram(i)+1  //Account for JSYS in Histogram
                             //array.
  resume-trapped-proc(R)     //Allow P to resume normal
                             //execution of JSYS i.
  break                      //Break from the trap interrupt.
```

Use of the trapping mechanism in this example is relatively simple: process *Q* merely resumes *P* after recording the trap. In the distributed file system application described earlier, the process running RSEXEC uses trapping to extend the TENEX virtual machine to support access to files remote from the local TENEX Host. It traps file operations made by processes inferior to it (e.g., text editors, compilers, etc.). Whenever a file operation is initiated that requires access to a remote file, RSEXEC sends a request across the network to a cooperating "service" process at the proper Host instructing it to execute the operation on behalf of the inferior process (See Figure 3). Operations that can be handled locally are passed directly to the local operating system by RSEXEC. After the operation has been performed RSEXEC resumes the inferior process, by properly incrementing its PC and providing return parameters (if any). Because the trapping activity is transparent, the inferior process can uniformly access all files, both local and remote, without regard for their location within the network.

Other applications for the trapping mechanism readily suggest themselves. JSYS traps have proven to be a powerful debugging aid. For example, a complex program, which was believed to have been debugged and which is run continuously on TENEX as a service "demon" process, began to malfunction by closing a critical data file for no apparent reason on the order of once a day. After unsuccessfully studying program listings and using conventional debugging techniques for several days, the programmer built a simple process to trap and examine all operations that could possibly result in closing the file. He then ran the malfunctioning service process as an inferior to the trapping process and was able to intercept the operation that caused the malfunction the first time it occurred (approximately ten hours after the program was placed in execution). We plan to add this debugging technique to the repertoire of IDDT, the invisible debugger, such that a user can cause a program being debugged to "break" on certain system calls. This technique would enable the user to gain control on, for example, all file output operations without requiring that he remember and specify the program location of each. When the program breaks he could inspect the parameters, perhaps request IDDT to execute the call and inspect the result, and then allow his program to proceed to the next breakpoint.

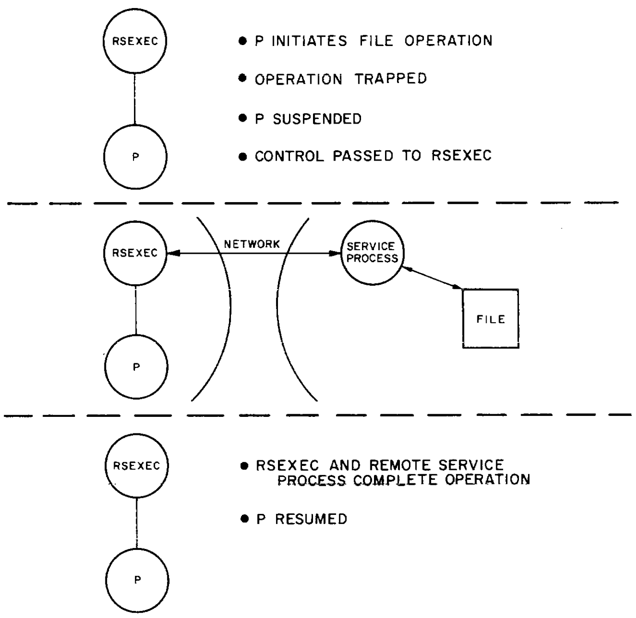
RSEXEC
P
• P INITIATES FILE OPERATION
• OPERATION TRAPPED
• P SUSPENDED
• CONTROL PASSED TO RSEXEC
RSEXEC
NETWORK
SERVICE PROCESS
FILE
P
RSEXEC
• RSEXEC AND REMOTE SERVICE PROCESS COMPLETE OPERATION
• P RESUMED
P

Figure 3—RSEXEC uses the JSYS trap mechanism to support uniform access by a user program (process P) to local and remote files. Access to remote files is accomplished by interacting with a remote service process

A somewhat different use of the trap mechanism would enable a user to use programs written by others with the assurance that doing so would not compromise the security of his data. For example, he could encapsulate such programs in a controlled environment which selectively inhibits output operations by trapping them and allowing only those directed to "legitimate" destinations to continue. He could even intercept and prevent use of more subtle techniques for leaking information such as those recently noted by Lampson.[13]

IMPLEMENTATION

The implementation of the JSYS trap mechanism is described in this section with the focus on approach rather than detail. The result is a simplified sketch of the implementation. First, it is necessary to present as background

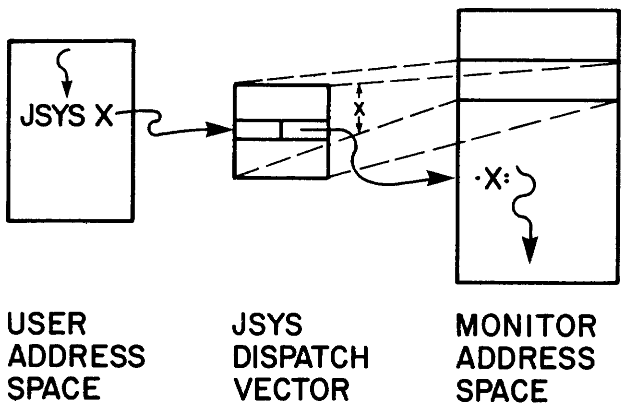

Figure 4—The JSYS instruction uses the JSYS dispatch vector to accomplish a transfer from user to monitor address space

some facts about the JSYS instruction and the structure of TENEX.

A TENEX process has two address spaces: A user address space where the process executes the instructions of the user program; and, a monitor address space that is invisible to the user program where the process executes monitor routines in response to system calls initiated by execution in the user address space. Transfer from user to monitor address space is accomplished by execution of a JSYS instruction with an effective address of less than 512 (see Figure 4). The processor enters "monitor mode" and uses the effective address as an index into the JSYS dispatch vector to fetch two pointers; it stores the user process PC and processor flags through one of the pointers and resumes execution in monitor address space at the location specified by the other.

Like the user address space, the monitor space for a process is a paged, 256K word linearly addressable space. Unlike the user space, the monitor space is partitioned into several areas (see Figure 5). These areas are:

1. a process private area that holds process state information (PC, ACS, user address space map, etc.);
2. a job private area, shared by all processes in a job, used for job wide data (the structure of the job process hierarchy, data about files open by processes in the job, etc.);
3. a "public" area, shared by every process in the system, that contains monitor routines and various system wide data bases.

The public area is further subdivided into in core resident and swappable regions. Both the process and job private areas are swappable.

The hardware paging device is aware of the organization of the monitor address space and, depending upon the area being referenced, takes different actions to complete a memory reference. References to the resident area are generally direct and bypass the page mapping operation although the pager can be instructed to "map the resident monitor" (see below); references to the public swappable area are mapped via a resident monitor page map; references to the job and process private areas are mapped via a page map for the process private area.

Implementation of the trapping mechanism specified in the previous section was feasible for TENEX because only a finite number of system calls (512) are possible* and all system calls "pass through" a single point in the system: the JSYS dispatch vector.

To implement JSYS traps, the dispatch vector, formerly a page shared by all processes in the system, was made process private. Trapped processes have a modified dispatch vector. Entries corresponding to trapped JSYS's point to a "trap and interrupt" routine and those for untrapped JSYS's point to the standard monitor routines for those calls. We saw two ways to make the dispatch vector private:

1. Make a minor (hardware) modification to the JSYS instruction so that it uses as its dispatch vector, a page in the process private area rather than one in the public area.
2. Leave the JSYS instruction unmodified. When a trapped process is running, set up the monitor map entry for the JSYS dispatch vector to point to a page in the per process area (the modified dispatch vector) and cause the pager to map references to the resident monitor (see above).

The primary advantage of the second approach is that it allows the trapping software to run on the existing hardware at all TENEX installations. Its disadvantages are two: slightly slower execution for trapped processes resulting from mapping each reference to the resident monitor; and severe constraints on the software for (planned) dual processor configurations resulting from limitations of both the paging device and PDP-10 processor. The current implementation provides for both alternatives, allowing each installation to choose one at "system generation time".

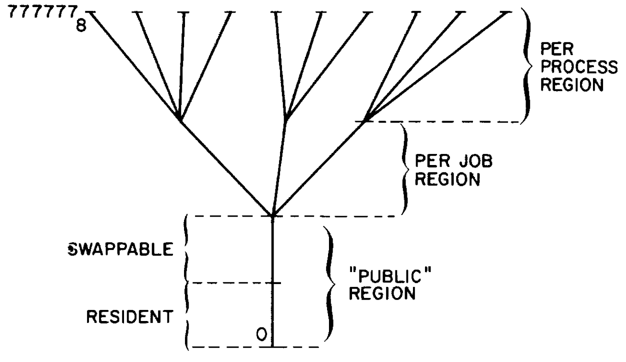

Figure 5—Schematic of the TENEX monitor address space

* The TENEX operating system, of course, can (and does) support more than 512 system operations. This is accomplished by multiplexing similar operations on a given JSYS by using call parameters to indicate the desired operation.

The "trap and interrupt" routine executed when a JSYS is trapped, suspends the trapped process and interrupts the proper process in the hierarchy. Should the suspended process be resumed without modification to its PC, the routine continues by searching the hierarchy for additional processes to interrupt. The situation in Figure 6 illustrates that the *immediate monitor* of a trapped process (i.e., the nearest superior process trapping its JSYS's) is not necessarily the correct process to interrupt. Consider the proper sequence of events for execution of JSYS's 1,2,3 and 4 by process E, assuming that all processes respond to the trap interrupt merely by resuming E:

JSYS 1: trap to A
(note bypass of immediate monitor D and intermediate monitor C)
JSYS 2: trap to D, then trap to C, then trap to A
(all monitors in hierarchy receive interrupt)
JSYS 3: trap to C, then trap to A
(note bypass of immediate monitor D)
JSYS 4: trap to D, then trap to A
(note bypass of intermediate monitor C)

The point here is that the immediate monitor may not have set the trap for the particular JSYS executed. The modified dispatch vector (JDVEC) is sufficient to initiate trap action but is insufficient, by itself, to specify the proper process(es) to be notified. To enable the "trap and interrupt" routine to complete the trap action properly, additional information is associated with each process:

1. The name of the process which is its immediate monitor (IM);
2. A list of the JSYS's for which its immediate monitor has set traps (TTBL).

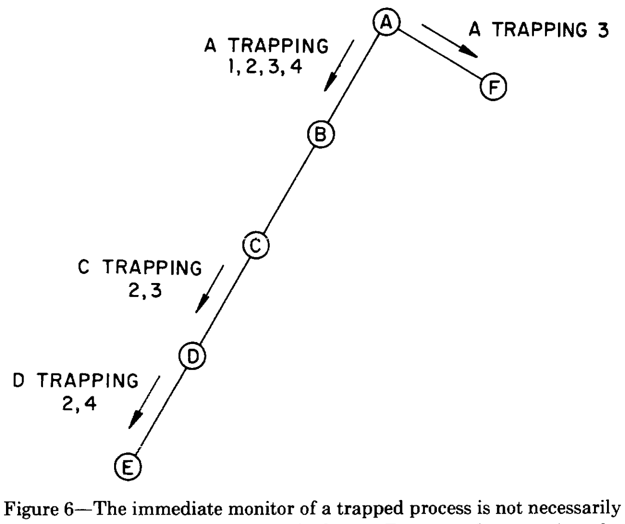

Figure 6—The immediate monitor of a trapped process is not necessarily the correct process to handle a JSYS trap. For example, execution of JSYS 1 by process E should initiate a trap to process A, bypassing intermediate monitors D and C

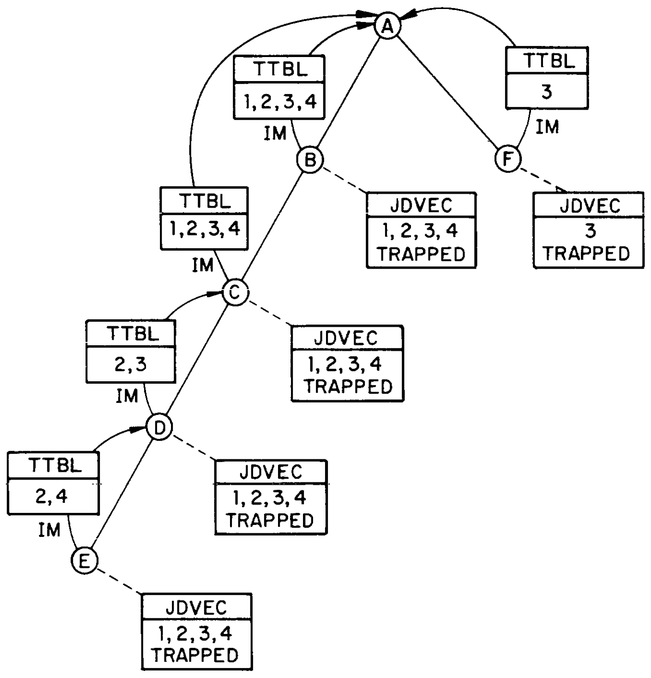

Figure 7—When a process attempts to execute a trapped JSYS, a "trap and interrupt" routine makes use of the modified JSYS dispatch vector (JDVEC), the name of the immediate monitor of the process (IM) and a list of JSYS's being trapped by the process (TTBL) to determine which process to interrupt

The situation in Figure 6 is redrawn in Figure 7 to illustrate how JDVEC, IM and TTBL are used when a process executes a trapped JSYS. Assume process E initiates JSYS 3 and, as before, assume that all processes receiving trap interrupts respond by resuming E. The following sequence of events occurs (refer to Figure 7):

1. E is dispatched through its JDVEC to the "trap and interrupt" routine.
2. IM of E is process D; D is not trapping E's execution of JSYS 3 (from E's TTBL); bypass D.
3. IM of D is C; C is trapping JSYS 3; interrupt C and suspend E;
4. C resumes E;
5. IM of C is A; A is trapping JSYS 3; interrupt A and suspend E;
6. A resumes E;
7. IM of A is null; dispatch E to standard monitor routine for JSYS.

Setting and removing traps for a process is accomplished by a recursive routine that "walks" over the process hierarchy appropriately updating JDVEC, IM and TTBL for the process and its inferiors. Reconsider the situation of Figure 7; the effect of

set-traps (C, JSYS-2-and-5)

executed by process B is shown in Figure 8. Removing traps is the trickier of the operations, as care must be

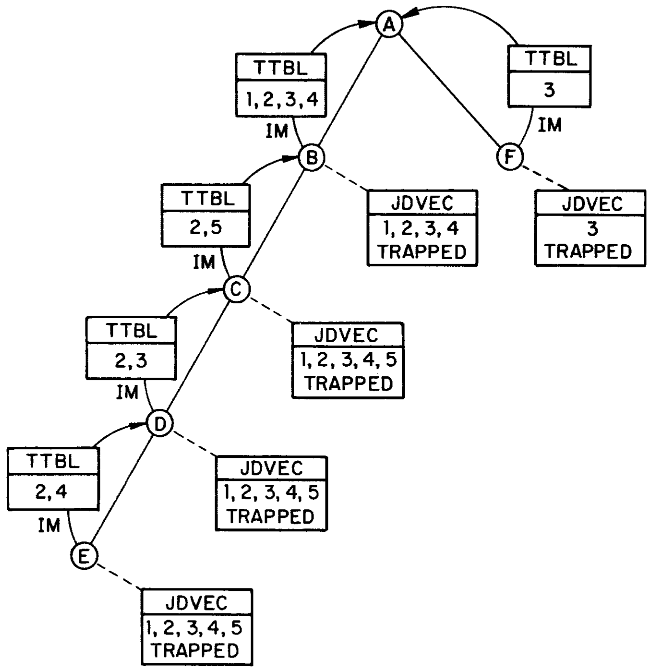

Figure 8—Note the changes in IM, JDVEC, and TTBL for processes C, D, and E resulting from process B setting traps for JSYS's 2 and 5 for process C

taken to restore to JDVEC the normal dispatches for only those JSYS's no longer trapped by any process in the hierarchy. The following cases (refer to Figure 7) illustrate the nature of the problem:

1. remove-traps (B, JSYS-1) executed by A:
   Restore the normal dispatch for JSYS 1 to JDVEC of B,C,D,E;
   Remove JSYS 1 from TTBL of B and C
2. remove-traps (B, JSYS-2) executed by A:
   Restore the normal dispatch for JSYS 2 to JDVEC for B and C;
   C and D trap JSYS 2, hence do not modify JDVEC of D and E;
   Remove JSYS 2 from TTBL of B and C
3. remove-traps (E, JSYS-2) executed by D:
   E's execution of JSYS 2 is trapped by A and C, hence do not modify JDVEC C of E;
   Remove JSYS 2 from TTBL of E.

Moving the JSYS dispatch vector from a resident page shared by all processes to a swappable page in the process private region adds to system memory management overhead by:

a. increasing paging activity—modified dispatch vectors must be swapped into core as the corresponding trapped processes execute; and
b. placing increased demand for storage on the swapping medium to hold the numerous dispatch vectors.

To reduce this overhead, the implementation minimizes the number of dispatch vectors the system must maintain by sharing them among processes whenever possible.* All untrapped processes share the same modified dispatch vector which is a page in the resident region. Furthermore, the routine that sets and removes traps is careful to insure that a process and all of its inferiors having the same immediate monitor share the same dispatch vector. Figure 9 shows how dispatch vector sharing relations change as traps are set and removed.

As we have described the implementation, a process whose execution of certain JSYS's is trapped can execute untrapped JSYS's without incurring overhead due to the trapping mechanism. Because the overhead resulting from execution of a trapped JSYS strongly depends upon the situation,** it is impossible to give a single, simple measure of how expensive trapping is. However, a comparison of the times required for a process to execute a given (nontrapped) JSYS and to execute the same JSYS for a well-defined, "best case" trapping situation represents a useful measure of the trapping overhead.

The time required to execute JSYS X is:

2 μsec (=time to accomplish transfer from user to monitor space; usually insignificant)
+CPU time to execute system routines that implement call X

The time required when JSYS X is trapped by only a

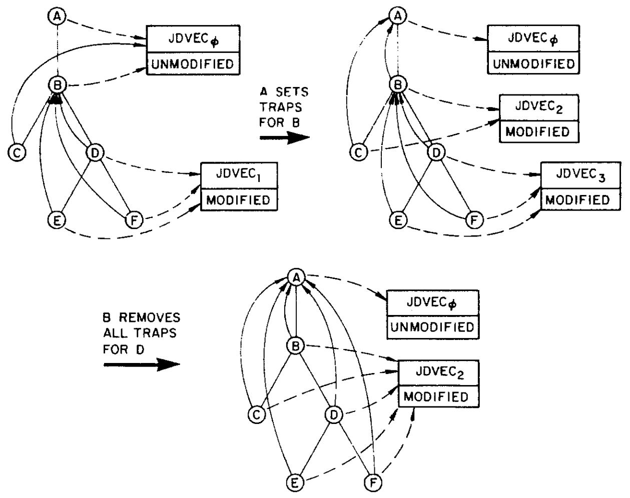

Figure 9—To minimize the number of JSYS dispatch vectors that must be maintained, TENEX insures that processes share dispatch vectors whenever possible

* Two processes can be made to share a page by setting the corresponding map entries in their memory maps to point to the same physical page of memory.

** i.e., the number of superior processes trapping the JSYS, the actions they take in response to the trap interrupt, and "unrelated" factors such as how heavily loaded the system is when the trap occurs.

single process that eventually returns control to the trapped process is:

2 μsec (insignificant here)
+CPU time to execute system routines that implement call X
+1.7 msec (=CPU time required to pass control from trapped to trapping process and back; determined by counting instructions making "best case" assumptions)
+CPU time trapping process uses in response to trap
+2 process wakeups*

It is clear that the (percentage) overhead incurred by trapping JSYS X is a strong function of the complexity of JSYS X. Measurements made for the "Byte IN" JSYS (a moderately "quick" call that reads the "next" byte from an open file) for the situation in which the trapping process immediately resumes the trapped process shows the untrapped operation to be 3 to 10 times faster than the trapped one.** By using highly tuned, tightly coded routines rather than the existing, "general purpose" monitor routines to transfer control among the processes, we estimate that the 1.7 msec figure could be halved. This would result in halving the overhead in trapping the "Byte IN" operation. Our experience with the trapping mechanism has shown that the delay resulting from the two process wakeups is the most significant component of the overhead. This component is largely due to the current TENEX scheduling algorithms which treat the two processes as independent, whereas, in reality, they operate in a tightly coupled, coroutine-like fashion. We feel that a significant reduction in trapping overhead would result by modifying the TENEX scheduler to support coroutine-like transfer of control between processes whereby one process could relinquish the processor (i.e., its remaining CPU quantum) and its memory resources (used to hold its working set) to another process without invoking the "normal" processor and memory management operations.

DISCUSSION

The features we are familiar with in other systems that most closely approximate the JSYS trap mechanism are "dynamic linking" in MULTICS[14,15] and "facility calls" in the operating system being developed by Project SUE.[16,17] The intended uses of these features are somewhat different from those of JSYS traps and of each other; thus the capabilities they provide, while similar in some respects, exhibit significant differences.

In MULTICS all "system calls" are made by invoking subroutines. The "linkage" to an "external" subroutine, such as one implementing a MULTICS "system function" is not established until the routine is called for the first time during program execution. At the first call a "fault" occurs that activates a dynamic linking procedure. As a result, various "system" and "user" file directories are searched for the routine. When (if) the routine is found, the linkage is made such that subsequent calls of the routine do not cause a "fault" and then normal execution of the subroutine call is resumed. Dynamic linking is motivated largely by the desire to support and encourage modular programming.

The dynamic linking mechanism of MULTICS can be used to substitute "non-standard" routines for the "standard system" routines by placing such routines in the file directories that are searched and (or) by specifying alternative directory "search paths". Use of dynamic linking in this way could approximate some capabilities JSYS traps offer. However, there are significant differences that should be noted. First, the "set-up" procedure is different and, in practice, would probably deter use of linking in this way. The set up involves storing a file for each call to be intercepted in an appropriate "user" file directory. The MULTICS system provides many ways of accomplishing a given function. As a result, in order to provide an execution environment for a given program it is necessary to know which of the many possible routines the program uses for each of the functions to be controlled. This, together with the potential for file naming conflicts, suggests that implementation of software to provide a controlled environment within which arbitrary users may run programs of their choice would be decidedly non-trivial.

Secondly, the linking activity takes place within the context of a single process. The multiple process nature of the trapping mechanism makes hierarchies of execution environments relatively easy to implement. Consider, as an example, the situation of a debugging environment (such as IDDT provides) existing within a multi-host file system environment (such as RSEXEC provides); system calls would be interpreted first by the process implementing the debugging environment and then by the process implementing the file system environment. It is difficult to see how dynamic linking could be used to implement such a hierarchy of execution environments. Finally, the linking mechanism is designed to serve as a subroutine linkage mechanism. The kind of "controlling" actions a "substitute" routine can take are constrained to be consistent with the subroutine discipline. Furthermore, once a link is established it generally exists for the duration of the computation. In this sense, JSYS traps are more "dynamic" in that they may be set and removed repeatedly.

Project SUE at the University of Toronto is developing an operating system in which system resources and services are provided by dedicated "system" processes. The "facility call" concept was developed to meet the interprocess communication requirements presented by such a system organization. To request a service a process

* Time for TENEX to "notice" that the trapping process has been interrupted, should be awakened and given the CPU and, later, that the trapped process has been resumed and should be awakened.

** The large variance is due to the fact that it is sometimes necessary to read a file page from secondary storage into the monitor file buffer to complete the read operation.

issues a facility call. As a result it is blocked until the process responsible for the service completes the request. Thus, the virtual machine seen by a process is defined by the processes which respond to the facility calls it makes. Because facility calls can be used by all processes for interprocess communication the distinction between "user" and "system" processes is a weak one. That is, a so called "user" process could provide service for another user process in much the same way a TENEX process can use JSYS traps to provide services for another process that are not directly supported by the operating system. Unlike JSYS traps, facility calls (as described in References 16 and 17) cannot be used to redefine existing system calls. Therefore the approach one would take to provide a controlled execution environment would be somewhat different than that using JSYS traps. It would involve reprogramming the processes that provide the functions to be controlled. The extent to which this is feasible would depend upon where the functions of interest are provided: at a low level by "standard" system processes, or at a high level by user processes. As noted earlier, a requirement of the JSYS trap mechanism was that it enable all standard system functions to be intercepted without modifying the way the operating system itself provides them. Because there is no obvious analogy in facility calls to the way a trap for a particular JSYS can be passed from process to process, hierarchies of controlled environments would be difficult to implement.

CONCLUDING REMARKS

Our experience with using the trapping mechanism has, to date, been somewhat limited. However, we feel that it represents an extremely powerful operating system facility. Although we have discussed trapping in the context of its realization in the TENEX operating system, we feel that the trapping concept is a general one which is consistent with a variety of operating system philosophies and should appear in some form in every "general purpose" operating system. We recommend that system designers seriously consider providing similar, user accessible mechanisms for program encapsulation in future operating systems.

ACKNOWLEDGMENTS

The author wishes to thank a number of colleagues for their contributions to the work presented above, especially R. S. Tomlinson and J. D. Burchfiel who constructively commented on the JSYS trap design and on the implementation approach; D. C. Allen and R. S. Tomlinson whose knowledge of the TENEX monitor was invaluable during the debugging phase; and R. E. Schantz, T. A. Standish, and W. R. Sutherland who constructively commented on the presentation of this paper.

REFERENCES

1. Bobrow, D. G., J. D. Burchfiel, D. L. Murphy and R. S. Tomlinson, "TENEX, a Paged Time-Sharing System for the PDP-10," *Communications of the ACM* 15, 3, pp. 135-143, March 1972.
2. Murphy, D. L., "Storage Organization and Management in TENEX," *AFIPS Conference Proceedings,* Vol. 41, 1972 AFIPS Press, Montvale, New Jersey, pp. 23-32.
3. Thomas, R. H., *A Model for Process Representation and Synthesis,* Ph.D. Thesis, Department of Electrical Engineering, M.I.T., June 1971. (Also available as Project MAC Technical Report TR-87.)
4. Bernstein, A. J. and P. Siegel, *Hardware for Level Structure Operating Systems,* Technical Report 21, State University of New York at Stony Brook, Department of Computer Science, October 1973.
5. Thomas, R. H., "A Resource Sharing Executive for the ARPANET," *AFIPS Conference Proceedings,* Vol. 42, 1973, AFIPS Press, Montvale, New Jersey, pp. 155-163.
6. Roberts, L. G. and B. D. Wessler, "Computer Network Development to Achieve Resource Sharing," *AFIPS Conference Proceedings,* Vol. 36, 1970, pp. 543-549.
7. Heart, F. E., R. E. Kahn, S. M. Ornstein, W. R. Crowther and D. C. Walden, "The Interface Message Processor for the ARPA Computer Network," AFIPS Conference Proceedings, Vol. 36, 1970.
8. TENEX JSYS Manual—A Manual of TENEX Monitor Calls, BBN—Computer Science Division, BBN, Cambridge, Massachusetts, September 1973.
9. Myer, T. H., J. R. Barnaby and W. W. Plummer, *TENEX Executive Language Manual for Users,* BBN Computer Science Division, BBN, Cambridge, Massachusetts, April 1973.
10. *TENEX User's Guide,* BBN Computer Science Division, BBN, Cambridge, Massachusetts, January 1973.
11. Digital Equipment Corporation, PDP-10 Reference Handbook, December 1971.
12. Plummer, W. W., IDDT User Manual, BBN Computer Science Division, BBN, Cambridge, Massachusetts, 1973.
13. Lampson, B. W., "A Note on the Confinement Problem," *Communications of the ACM,* 16, 10, October 1973, pp. 613-615.
14. Organick, E. I., *The MULTICS System: An Examination of its Structure,* M.I.T. Press, 1972.
15. Vyssotsky, V. A., F. J. Corbato and R. M. Graham, "Structure of the MULTICS Supervisor," *AFIPS Conference Proceedings,* Vol. 27, 1965.
16. Sevcik, J. W., J. W. Atwood, M. S. Grushcow, R. C. Holt, J. J. Horning, and D. Tsichritzis, "Project SUE as a Learning Experience," *AFIPS Conference Proceedings,* Vol. 41, 1972, pp. 331-338.
17. Holt, R. C. and M. S. Gruschow, "A Short Discussion of Interprocess Communication in the SUE/360/370 Operating System," Proceedings ACM SIGPLAN/SIGOPS Interface Meeting, April 1973.