Structure of the ELF operating system*

by DAVID L. RETZ
Stanford Research Institute
Menlo Park, California

and

BRUCE W. SCHAFER
University of California
Santa Barbara, California

ABSTRACT

This paper describes the ELF operating system structure and discusses a number of considerations which influenced its design. Several applications supported by the system are presented in relation to that structure. The software tools used for constructing and maintaining the system are described, and several implementation problems which were encountered are presented.

INTRODUCTION

ELF is a multiprogrammed operating system which provides a set of programming tools for a computer network environment. The system is implemented for the DEC PDP-11 series computers and is being used to support a number of research applications at various sites in the ARPA network.[1,2]

ELF development was started in early 1973 at the Speech Communications Research Laboratory in Santa Barbara, California. An initial version of the system provided multi-user terminal support capability for network access.[3] In early 1974 that system was tested at several network sites and a decision was made to expand the system structure to provide a more general operating system environment. An initial version of the expanded system was operational at the beginning of 1975, and is currently being used at 30 network sites.

Development of the ELF** system was motivated by the need for a flexible network/user interface. The requirements for such an interface included multi-user terminal access to remote interactive systems, peripheral support for transfer of locally-stored files, real-time data acquisition facilities, and a test-bed for research problems related to computer networks. It was required that these functions be implementable with a variety of hardware and software configurations and be maintainable using the communication facilities provided by the network.

Design of the ELF system draws upon a number of techniques used in other operating systems described in the literature. For example, ELF enables sharing of hardware resources (processor, memory, and I/O devices) by providing a multiprogrammed virtual memory environment; ELF implements a tree structure of processes similar to that found in the TENEX,[4] UNIX,[5] and XDS 940[6] systems.

The ELF system has been structured in a hierarchical fashion in order to simplify its specification and testing, and to provide flexibility of system configuration. Real-time constraints imposed on the system force a compartmentalization of critical paths which disable response to interrupts; limits are placed on the maximum duration of these paths.

A robust inter-process communication facility is incorporated in the system in order to facilitate communication with remote programs (processes) in the network. Programs which utilize ARPA network protocols[7,8] to access remote resources (e.g., files) utilize this inter-process communication facility extensively.

A command language interpreter allows the system to accept requests for service from user terminals, and provides functions for controlling individual user programs in the virtual memory environment provided by the system. A library of these user programs provides capabilities for terminal access to remote systems, Input/Output of digitally-sampled data, transfer of files, or the debugging of new system facilities.

This paper describes the ELF operating system structure and discusses a number of considerations

* This work was supported by the Advanced Research Projects Agency, through Contract No. N00014-73-C-0221 at the Speech Communications Research Laboratory, Santa Barbara, California, and through Contract No. F30602-75-C-0320 at Stanford Research Institute.

** The name ELF is German for "eleven," and is somewhat germane to the naming of IMPs in the ARPA network.

which influenced its design. Several applications supported by the system are presented in relation to that structure. The software tools used for constructing and maintaining the system are described, and several implementation problems which were encountered are presented.

SYSTEM ORGANIZATION

At the center of the ELF system exists a set of modules, collectively referred to as the Kernel, which provides a set of primitive functions for outer-level procedures, and performs tasks which allow the system's processor, memory, and I/O devices to be shared among a set of processes. Kernel primitives perform functions such as dynamic creation of processes, process synchronization, allocation of virtual storage, scheduling of I/O requests, and sharing of an interval timer.[9]

The selection of procedures which reside above the Kernel varies somewhat and is determined by the range of applications to be supported by a system. In ELF systems which provide multi-user terminal support, a set of procedures known as the EXEC provides a mechanism for allocating system resources to users. The notion of a "Job" is utilized at this level to provide an encapsulation of the resources associated with a given user: a tree structure of processes, a collection of open files, allocated virtual storage, and so forth. EXEC procedures utilize Kernel facilities to create Jobs, interpret user commands, run user processes, and maintain the system file structure. User processes which are run under the EXEC provide a variety of functions, including terminal access to remote interactive systems on the network. An illustration of the layered system structure is shown in Figure 1.

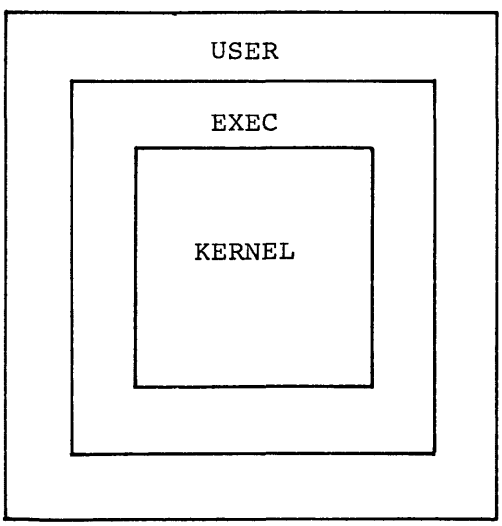

Figure 1—An example of the layered ELF system structure

The Network Control Program (NCP) is another module which may be selected. The NCP uses facilities provided by the Kernel to support a communication mechanism between local and remote processes in the ARPA network. It consists of a set of procedures which establish and control data flow on a set of connections using the ARPA network protocols [Ref].

THE ELF KERNEL

The ELF Kernel concerns itself with three primary areas. The first of these, Processor Management, controls distribution of the processor among a number of asynchronous processes and provides for process synchronization and mutual exclusion. The second major portion of the Kernel is Storage Management, which handles the allocation of physical and virtual storage available to processes in the system. The third portion, I/O management, controls the interaction between processes and external devices, and additionally provides a facility for inter-process communication.

Processor management techniques

The Kernel provides a set of system calls which allow processes to be created, compete for processor service according to priority, inter-communicate, or be terminated. A process is characterized by a virtual program counter, a set of general-purpose registers, a stack, and a queue of elements which are called event messages. Processes are created in the ready state, and remain ready until they are blocked by calling a system primitive for synchronization or mutual exclusion. A scheduling program in the Kernel maintains a list of processes which are in the ready state and transfers control to the highest priority ready process.

Process synchronization

A process synchronization mechanism similar to the message buffer scheme described by Brinch-Hansen is implemented.[10] Each process owns a queue of event messages sent to it by other processes. A process can block itself until a message is added to its event queue by issuing a system primitive called WAIT, which places the process in the "waiting" state if its event message queue is empty. If a process issues the WAIT primitive and elements exist on its event message queue, the process is left in the "ready" state, the event message at the head of its queue is removed and is returned as a parameter.

Event messages are placed on a process' event queue by another process which invokes the SIGNAL primitive, specifying a destination process name and event message. When a process awakens, it receives the 24-bit message in addition to the name of the process

which signalled it. In general, the 24-bit message field is interpreted by ELF system processes as an 8-bit "event code" and a 16-bit "data" field. While this assignment of bits is a convention for system processes, higher-level (user) processes which choose to synchronize using SIGNAL and WAIT may use the 24-bit event message field arbitrarily. A system primitive exists to allow a process to wait for a specific event-code.

Examples of synchronization primitives are shown below:

WAIT→⟨PNAME, EVENT, DATA⟩
SIGNAL ⟨PNAME, EVENT, DATA⟩
WAIT-SPECIFIC ⟨EVENT⟩→⟨PNAME, EVENT, DATA⟩

Processor scheduling

Every process that is in the ready state resides in a priority queue that corresponds to a priority level that it was assigned when it was created. The ability of a process to gain control of the processor is a function of the priority queue in which it runs and its position in that queue.

The position of a process within a given queue is determined according to its behavior. A daemon process receives control at regular intervals (currently, every 250 ms.). When it receives control it re-orders the processes within each queue according to their utilization of the processor during the preceding time interval. A process' composite priority is a combination of the priority queue in which it resides and its position in that queue; the process in control of the processor at any point in time is the process at the head of the queue with the highest priority value. This scheme effects a "round-robin" scheduling technique for compute-bound processes while assigning a higher composite priority to processes that are not compute-bound.

Protection mechanisms

Because processes rely on the validity of messages received on their event queues, a protection scheme is required to prevent processes from receiving messages from other non-authorized processes. This mechanism is implemented by means of a capability value which is assigned to a process when the process is created, and may change as the process makes calls to various primitives in the operating system. The access rights of a process executing at a given capability level are a subset of the rights allowed a process with a lower capability value.

A process may be assigned a "branch name" which identifies the processes and all sub-processes in the tree below it. (The assignment of branch-names to processes is utilized by the ELF EXEC in identifying a process as a member of a particular Job.) Processes which have the maximum capability value can SIGNAL any process in the system; processes which are running with lower capability values are limited to the set of processes having identical branch-names.

Mutual exclusion techniques

Processes may request mutual exclusion by means of binary semaphores. Kernel primitives allow dynamic assignment of semaphore names, and provide Dijkstra's P and V operations on those semaphores.[11] Entries may continue to be added to a process' event message queue while the process is waiting for exclusive access to a resource.

Process creation and termination

A process is created by issuing a CREATE-PROCESS system call, specifying a starting address, set of registers, capability value, priority level, and an event code to be used to SIGNAL the creating process when the created process halts itself or encounters a system error (e.g., invalid parameter specification to a Kernel primitive). The system maintains a relationship between each process and its creator, and keeps a list of all processes created by a given process.

A process may be frozen by means of the FREEZE-PROCESS primitive. This causes the process to be blocked and its creator to be signalled with an event message containing the name of the process which was frozen. At this point, the process' registers may be examined or modified, the process may be "thawed," or the process may be terminated. Entries may continue to be added to the event message queue while a process is frozen.

A process may only be terminated by its creator. When a process is terminated, all the processes it has created are also terminated and any resources associated with it (such as outstanding event messages) are released.

Storage management techniques

The Storage Management portion of the Kernel provides a mechanism for creation of a number of virtual address spaces and controls their mapping into physical memory. An address map defines the relation between a user's virtual storage and physical storage addresses. A specific address map becomes associated with a process when the process is created. Any number of address maps may be defined for processes running in user mode; there is a single address map defined in kernel mode, and this is utilized for system (Kernel) primitives. The processor switches from kernel mode to user mode when it gives control to a process; it switches from user mode to kernel mode

when a user process makes a system call or is interrupted.

Storage management data structures

There are two data structures used to maintain the relationship between virtual and physical storage. Each virtual address space is mapped as a set of 4096-word pages. A Virtual Storage Map (VSM) is an array which describes the state of each page in a virtual address space. A given page may be undefined (a "hole" in the address space), defined but non-resident, or defined and resident. Each entry in the VSM which describes a resident page contains a pointer to the page in physical storage. A virtual page may also be "read-only," in which case the hardware memory mapping facilities are utilized to prevent modification of shared or protected data. A Virtual Storage Map exists for each virtual address space. Kernel primitives exist for the creation (allocation) of a new virtual address space, and the creation of pages within an address space.

The Physical Storage Table (PST) indicates the relationship between physical pages and the (virtual) pages which occupy them. A mechanism for sharing of virtual pages is provided by the system; in the case of shared pages, the physical storage table points to the head of a linked list of VSM entries. The data structure used to implement this mechanism is shown in Figure 2.

A Virtual Storage Map identifier is included in the system state information for each process. This value uniquely identifies the user address space in which a process is running, and is a logical extension of the process' program counter, as shown in Figure 3. Hardware storage mapping register values are derived from the Virtual Storage Map when the scheduler transfers control to a process.

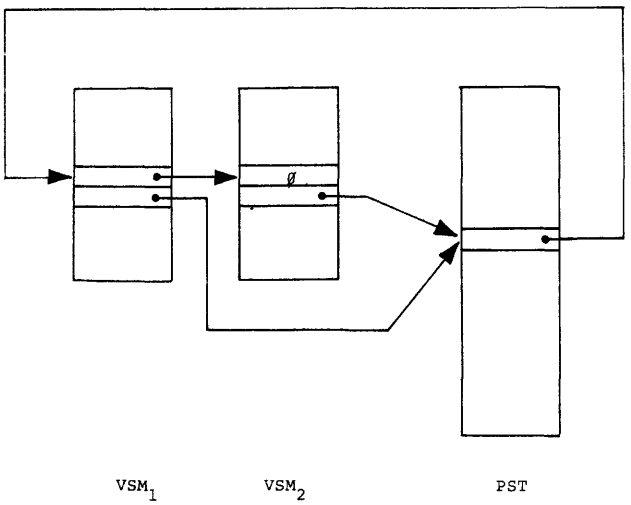

Figure 2—Storage management data structures

Storage management primitives

A set of Storage Management primitives enables a process to allocate a new virtual address space, to allocate pages within an address space, to cause pages to be shared between address spaces, or to block transfer data between address spaces. A new address space is created using the primitive CREATE-VSM, which returns an 8-bit VSM identifier for the new address space. The address space (and all physical storage associated with it) is released when a process issues a DELETE-VSM primitive or when the process which created the address space is terminated.

Creation of a new address space does not allocate any pages within it. A process may define (allocate) new pages within an address space by issuing a DEFINE-PAGE primitive, which allocates a new page in virtual storage and defines the specified page. The caller specifies the Read-Write/Read-Only status of the page. The process may optionally cause the page to be mapped into a page in another address space, causing the pages to be shared between the address spaces. For certain applications (such as system debugging) it is necessary to map a user page into an arbitrary physical page of memory. A privileged (i.e., capability-restricted) primitive allows a process to perform this function.

System primitives require the ability to access parameters which are passed as arguments by a calling process. A system call exists for transfer of a block of data between two address spaces. A primitive may request the identifier of the previous address space in order to obtain or return parameters to its caller.

Hierarchical procedures

ELF allows the establishment of a hierarchical structure of system primitives by means of a run-time binding mechanism. This permits a modular extension of system facilities, such as the addition of file system functions, without requiring modification of the Kernel. The mechanism enables a system initialization process to specify a correspondence between a set of system primitives and the procedures in a user address space which implement them.

When a process makes a call to a system primitive

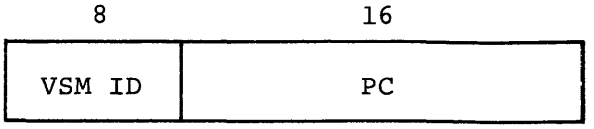

Figure 3—A process' virtual program counter

the system performs some task as an extension of that process. Control may be transferred from the calling process to a primitive in a different address space. In this case the system reloads the hardware mapping registers from the new virtual storage map and transfers control to the primitive in the new address space. The identifier of the previous address space is saved on the Kernel stack to allow restoration of mapping information when the primitive returns.

A process which runs in a user address space may utilize special instructions which "trap" through well-known address vectors on a stand-alone processor; the instructions trap through the corresponding locations in the active user space. A process which is being debugged and reaches a breakpoint, for example, causes control to be transferred to the address contained in the break-point vector (Location 14) of the user address space. This has facilitated the adaption of several debugging packages which run on a stand-alone PDP-11. It is also possible for a process to call a Kernel primitive which requests a signal in the event that a specified process reaches a break-point, and causes the break-pointed process to be placed in the frozen state. A process may thus be responsible for the debugging of a number of processes in the system. The application of this mechanism will be described later.

Input/output management

The Input/Output portion of the Kernel provides a set of system primitives which enable processes to schedule physical I/O requests to devices and to utilize the system's hardware clock. The I/O system queues requests to each device, performs a mapping from virtual to physical storage, provides an inter-process communication facility, and allows the system to be tailored for a particular hardware configuration by selection of a set of device driver modules.

Processes cause the initiation of physical I/O transfers by calling a system primitive which places their request on a queue for a particular device. An I/O process schedules I/O activity on each device, performs storage mapping functions, and causes physical I/O requests to be initiated by calls to device driver procedures.

Input/output primitives

Processes cause requests to be placed on a queue for a particular device by issuing the system primitive START-IO and specifying a device name, a buffer address, a byte count, a function code, and an optional device address. When the requested I/O operation completes, the user process is signalled by the I/O system with the requested event code. Function codes are defined for performing Read, Write, or Device-Specific (e.g., rewind tape) operations. I/O primitives exist for allocation of devices and their optional assignment on a per-process basis. When a sending process issues a write request, it is signalled when all of the bytes it has requested to be written have been transferred. When a process issues a read request, it may specify a data transfer mode (in the I/O function code). A "record" mode read request signals the process (and satisfies its request) when one or more bytes have been read; the reader is informed of the number of bytes actually transferred. A "stream" mode read request signals the reader when the number of bytes requested have been transferred.

INPUT/OUTPUT DEVICE CLASSES

There are three types of I/O devices supported by the system. The first of these classes handles character-oriented devices, such as terminals or line printers, which are used in an interrupt-driven fashion. In the case of character-oriented devices, the system moves the block of data being transferred between the Kernel and the requester's address space. This is necessary to allow interrupt routines to directly address the buffer used for data transfer.

Two utility procedures are made available to device drivers which support terminals. One of these procedures manages a ring buffer to allow type-ahead in the absence of an I/O request from user process. A second utility procedure performs terminal-specific output functions, such as output of padding (ASCII NUL) or expansion of ASCII Horizontal TAB characters. Selection of these two options is made possible by an I/O system primitive for setting device characteristics.

A second class of devices which are supported by the system transfer data on a direct memory access basis and require no processor intervention during the data transfer. The virtual storage structure of the system forces the I/O system to perform a mapping of a user process' I/O buffer into its location in physical storage before calling the device driver which initiates the transfer. (I/O devices which directly access memory do not utilize the processor's memory mapping hardware.) Additionally, the system must determine whether a buffer which is specified in an I/O request crosses page boundaries, and take special action if the associated physical pages are non-contiguous. When this is the case, the I/O system carries out the operation in a piecemeal fashion, initiating separate data transfers for the portions of the buffer residing in each individual page. User processes may avoid this inefficiency by appropriate allocation of buffer space.

A third class of devices provides a mechanism for inter-process communication in a fashion which appears identical to data transfer to a physical I/O device. A set of pseudo-devices, called "Inter-Process Ports" (IPP's) may be used for data transfer between processes. User processes may take advantage of this

facility in order to allow flexible assignment of their Input/Output streams; specifically, a process may accept input from either a physically-connected terminal or a remote process on a network. The utilization of Inter-Process Ports for network communication will be described later.

An Inter-Process Port is effectively a mailbox[12] which may be read or written by a pair of processes. A separate read and write request queue is maintained for each Inter-Process Port. Whenever the I/O system receives a matching pair (read, write) of requests, data is transferred from writer to reader. The writer and reader processes are signalled if their respective requests are satisfied. It is possible for the sender to issue an "end-of-stream" indication, which unconditionally satisfies the request of a receiver which is reading in stream mode; the receiver is signalled as usual with the number of bytes actually transferred. Inter-Process Ports may be assigned to a specific pair of processes.

Timer primitives

Timer primitives enable a process to set a number of interval timers and to receive an event message when a timer expires. Additionally, the process may stop a running timer by setting its interval to 0 or may get the current time remaining before a timer expires.

The Kernel maintains a 32-bit Time-of-Day, which is kept in 40-microsecond ticks. Kernel primitives allow user processes to get (or set) the Time-of-Day value. Conversion routines exist outside of the Kernel for obtaining the Time-of-Day as a character string.

THE ELF EXEC

The EXEC is utilized in ELF systems which require a flexible user programming environment. The EXEC interprets user commands and provides a framework for user program support. The command language provides functions such as user identification, display of system status, and initiation of user processes. Design of the ELF command language is patterned after the executive language of the TENEX operating system because of the user-oriented characteristics of that language.[4]

The EXEC support structure provides a set of primitives which are called by user processes below it and includes a terminal control mechanism for interrupting those processes. EXEC primitives allow logical data paths to be established for terminal control and for access to a system file structure; they also perform a number of utility functions, such as data conversion (e.g., time-of-day to string).

The EXEC is implemented as a set of re-entrant procedures which utilize Kernel primitives to support an inverted tree structure of processes. A process called the Logger issues I/O requests to a set of terminals or Inter-Process Ports and "listens" for the arrival of an attention character (control-C). When this occurs, the logger creates an EXEC process which receives characters from the device and interprets user commands. Additionally, it allocates and formats a set of control tables which are used to maintain resources allocated to the newly-created "Job." A Job is identified by a branch-name which is assigned to the EXEC process and becomes associated with all processes in the inverted tree under it. The Job effects a policy for allocation of resources (storage, files, etc.) to each user.

A system primitive exists to enable the logger process to be signalled when a terminal port has been dynamically added to the system. This function allows the system to respond to remote requests for connection from the network and to support a number of virtual terminals. Once a new logger port is established, the logger process issues I/O requests in the same fashion as it would for local terminals; in this case, however, I/O takes place on a pair of Inter-Process Ports.

The EXEC maintains a directory of "sub-systems" which may be dynamically expanded while the system is running. A sub-system is a named collection of procedures which have been loaded into a virtual address space. The EXEC allows the user to initiate a subsystem by typing its name; the EXEC then creates a user process and enables that process to communicate with the controlling terminal by means of EXEC file primitives.

A number of user sub-systems have been written to perform a variety of functions. TELNET, for example, provides the function of terminal access to remote systems on the ARPA network. The TELNET program utilizes file primitives provided by the EXEC to interpret user commands and to establish or terminate connections to remote network Hosts. A cross-network loader, called USERLOADER, allows programs to be loaded into a user virtual address space from a remote file system by means of the file transfer protocol[8] used in the ARPA network.

Generally, a user process is connected to a controlling terminal until the process halts (by freezing itself) or the user interrupts it by typing the EXEC attention character (control-C). I/O to the controlling terminal is re-directed to the EXEC process until the user allows the interrupted sub-system to resume or runs another sub-system. The active tree of user processes is terminated, whenever a user requests to run a subsystem, explicitly issues a "reset" command, or logs out of the system.

Some user processes require the ability to suppress the interrupt facility provided by the EXEC; the user process may specify a "transparent mode" which causes the EXEC to ignore interrupt characters and places this responsibility on the sub-system. This is needed, for example, in the TELNET sub-system,

which must be able to transmit the interrupt character to a remote system on the network.

The association between a controlling terminal and a Job may be dissolved by means of a user command or as a result of an error received on a terminal port. The state of the Job and its user processes is saved, and a user may re-attach to it by an EXEC command.

Design of the ELF file system has been aimed at providing a mechanism for terminal control over a set of user processes and the support stream-oriented Input/Output for a variety of device classes. A table-driven mechanism is used to implement stream Input/Output for sequential and file-structured devices. The file structure is compatible with the FILES-11 structure utilized by the DEC RSX-11 operating systems.[13]

The file system provides a primitive for initializing a file path; the file system returns with a Job-unique identifier called a Job File Handle. This identifier may then be used to efficiently call primitives which perform file I/O. The file system supports I/O primitives which read or write relative blocks within a random-access file, and provide a device-independent mechanism for reading or writing a stream of bytes. Stream-oriented primitives exist for byte-input, byte output, string-input, and string-output.

NETWORK COMMUNICATION CONTROL

The Network Control Program (NCP) is an essential component in the ELF system which makes possible process-to-process communication in a network environment. The NCP provides a mechanism for establishing and breaking connections between ELF processes and processes distributed on the ARPA network.

The NCP includes a set of processes which receive messages from the network, interpret messages according to ARPA network standard protocols, format outgoing messages, and control the flow of data on connections by means of the Inter-Process Port facility provided by the ELF Kernel.

A set of network control primitives enables processes within ELF to request the establishment or termination of network connections. When a process calls an NCP primitive to open a connection, the primitive returns the name of an Inter-Process Port which is to be used for data transfer on the connection. When the connection is opened, the process may cause data transfers by calling the normal Kernel I/O primitives using the Port-name which was returned. A process may request that an Inter-Process Port supporting a connection be specifically assigned to the NCP and user processes in order to prevent malevolent processes in the system from performing unauthorized data transfers on the Port.

An important Kernel function which is used by the NCP in the control of data flow on a network connection is that of obtaining the number of bytes requested to be read or written on an Inter-Process Port. This function is necessary to enable the NCP to intelligently determine the amount of data which may be accepted from a remote process. ARPANET protocols utilize a flow control mechanism whereby a sending process must be informed of the buffering capability at the destination.

In addition to control of communication using standard ARPA network protocols, the NCP provides a simple dispatch function for development of experimental protocols. Processes which utilize this function must specify a field which uniquely identifies received messages. Messages are delivered to the user processes (via an Inter-Process Port) in the form in which they arrive from the network.

SYSTEM APPLICATIONS

The ELF system is being used in a variety of applications which require a set of operating system components for network communication. The ELF Kernel provides a base for a variety of software configurations which are tailored according to system requirements. The configuration of processes shown in Figure 4 is typical of ELF systems which provide terminal access to remote systems on the ARPA network. The Kernel, EXEC, and NCP modules support a number of users who access the network by running the TELNET sub-system of the EXEC.

Additional processes may be added to the system for peripheral support. For example, a simple process which uses the ELF NCP to await a remote request for connection may be included in the system. The process accepts a connection, receives a stream of data, and outputs it to a local line printer. A status indication from the NCP signals that a remote process has closed the connection, and the peripheral control process then returns to the "listening" state.

This process provides a simple mechanism for sharing of a peripheral device (in this case, a line printer) among a number of service sites in the network. A daemon process at each service site checks periodically for files to be listed. When they exist, it opens a connection and transmits each file. The ELF NCP queues remote requests for connection while the peripheral process is busy.

A more flexible mechanism for peripheral support is provided by the ARPA network file transfer protocol server process which may be included in the system. The file transfer server responds to remote requests and utilizes facilities provided by the ELF EXEC (file system) to carry out data transfer to a variety of peripherals.

The support of real-time data acquisition functions for speech research has been one of the goals in design of the ELF Kernel. A user process runs as an EXEC sub-system and performs functions of real-time data sampling and file I/O for digitization of speech wave-

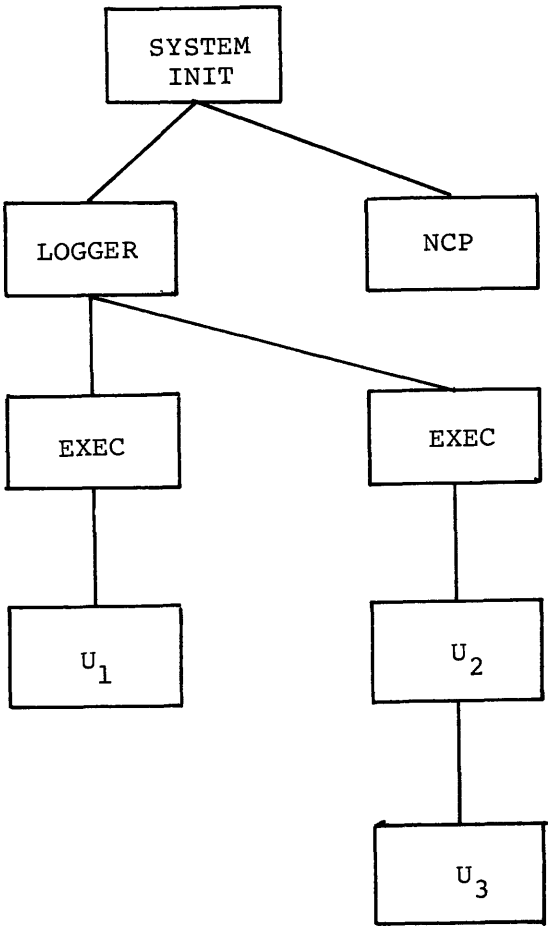

Figure 4—Example process tree

forms. The system may thus be used as a data acquisition station while simultaneously providing a terminal support function for access to remote systems. The sampled signal may be transferred to the remote system by the file transfer protocol process.

The ELF Kernel provides a support base for development of experimental networks and protocols. The packet radio network[14] for example, utilizes the environment provided by the Kernel to support a set of user processes which perform network routing control functions. Additionally, new protocols being tested in the ARPA network utilize the inter-process communication facility provided by the Kernel in the implementation of new network control processes.

DEVELOPMENT AND MAINTENANCE TOOLS

Development and maintenance of the ELF system has made extensive use of software tools provided within the framework of the ARPA network. The TENEX operating systems available on the network, for example, have provided a number of services for software development. ELF modules are edited, compiled, and linked together using a set of online subsystems in TENEX. When an ELF system is generated it may be bootstrapped into a target PDP-11 system by transfer of the binary file through the network.

The ELF Kernel, EXEC, and NCP are written in MACRO-11 assembly language. The virtual memory structure provided by the system allows user processes to be written in higher-level languages. Compilers for the BCPL and L1011 Languages are available under TENEX and generate code for the PDP-11. The files produced by these compilers may be transferred to an ELF file structure by means of the file transfer protocol or may be loaded by the ELF USERLOADER sub-system, which was described above.

A number of debugging tools have been developed. A low-level (stand-alone) debugger, called FLEA, was implemented to facilitate checkout of the virtual memory system. FLEA provides the capability of examining or breakpointing any location in physical memory.

Several debuggers which run on a stand-alone PDP-11 have been modified to run in an ELF user address space. Modification was required to make use of ELF I/O primitives for terminal control. Hardware I/O device registers are not available to processes in a user address space.

An additional debugging mechanism involves the interpretation of debug commands received from the network by a system debugging process. This approach minimizes space requirement in a processor being debugged while taking advantage of facilities available on large systems to provide a comfortable user language and a symbolic representation of addresses and instructions. A system debug process* which resides in ELF utilizes the breakpoint signal facility provided by the Kernel and is responsible for debugging of a number of other processes.

ELF system software is distributed as a set of source modules which are accessible in a directory at one of the network TENEX service sites. Users may access the source files by means of the network file transfer protocol. Release-notes and bug-reports (user-feedback) take place through the use of the network message system.

CONCLUSION

The foregoing discussion has presented a structural view of the ELF system as a mechanism for user access to remote resources. A number of design decisions have been made in supporting a range of user applications. At this point we critically examine several of

* Developed at Bolt, Beranek, and Newman, Inc., Cambridge, MA.

these decisions and discuss their relation to the resulting system structure.

The event message scheme which is utilized for process synchronization has proven to be a flexible mechanism which allows processes to wait for multiple events. This capability is required for reliable communication in a network environment: error conditions may arise and "time-outs" are sometimes needed in order to resynchronize a connection sequence. The synchronization mechanism used here is an alternative to the creation of a process for each event; that solution was considered too costly in terms of processor overhead and system table space requirements. Rather, the event message mechanism enables a single process to be multiplexed for several events. Two drawbacks were encountered with the ELF event mechanism, however. First, it is required that any procedure which performs an operation which specifies an event code (e.g., initiation of an I/O request) must receive as a parameter an event code which it may use. In the absence of a statically-allocated set of event codes, an arbitrary system procedure has no knowledge of the set of event codes currently being used by a calling process. A second drawback is the lack of a reliable mechanism for bounding the number of event messages which may be allocated by a signalling process. The strategy of blocking a signalling process when the system's supply of event messages has run out is susceptible to deadlock. For example, the signalling process may be executing an interrupt routine which signals completion of an I/O operation. However, if the process is itself the process being signalled and the supply of event messages runs out the process will deadlock. As a result of these problems, the current system implementation simply returns an error condition to the signalling process, indicating the lack of event message queue space in the system.

The Inter-Process Port mechanism which has been implemented in the system has been valuable in implementing network protocol processes and has provided an effective mechanism for performing local and network I/O in a transparent fashion. There are two disadvantages to that mechanism. First, a Port mechanism requires buffer space in each address space which communicates using the port. Second, processing time is consumed in the copying of data from the sender to the receiver's buffer space. It is believed that these disadvantages are outweighed, however, by the system flexibility introduced by a general inter-process communication mechanism.

The memory management scheme used in ELF has been designed to provide a simple mechanism for allocation of physical storage and to provide a means for sharing of user pages. A problem with the current storage allocation scheme is that of internal fragmentation which results when a number of small user procedures reside in a virtual address space or when a large number of user processes wish to share a small amount of data. A modification of the storage management portion of the Kernel to provide for variable-sized segments is currently under consideration.

The choice of a hierarchical structure of system functions has provided modularity and has facilitated checkout of the system. Its drawback is an occasional overhead introduced in the passing of parameters between levels. Provision of standard memory management functions which facilitate these tasks in hardware would alleviate this problem.

SUMMARY

This paper has presented a description of the ELF Operating system which provides a set of user access facilities for the ARPA computer network. We have attempted to describe the system from a structural point of view and point out operating system functions which are necessary in a network environment.

ACKNOWLEDGMENTS

In addition to the work done by the authors, significant contributions to the design and implementation of the system were made by Mr. Jon Miller, who was responsible for the I/O and Storage Management portions of the Kernel, and Mr. Jim McClurg, who participated in implementation of the file system and maintained system documentation. Contributions were also made by V. Strazisar and E. Mader at Bolt, Beranek, and Newman, Inc., and by Mr. P. Raveling of University of Southern California Information Sciences Institute.

REFERENCES

1. Roberts, L. G., B. D. Wessler, "Computer Network Development to Achieve Resource Sharing," *Proceedings of AFIPS SJCC* 1970, pp. 543-549.
2. Heart, F. E., R. E. Kahn, S. M. Ornstein, W. R. Crowther, D. C. Walden, "The Interface Message Processor for the ARPA Computer Network," *Proceedings of AFIPS SJCC* 1970, pp. 551-567.
3. Retz, D. L., "ELF—A System for Network Access," *Proceedings of IEEE Intercon*, New York City, April, 1975.
4. Bobrow, D. G., J. D. Burchfiel, D. L. Murphy, R. S. Tomlinson, "TENEX, a Paged Timesharing System for the PDP-10," *Communications of the ACM*, Vol. 15, No. 3, March, 1972, pp. 135-143.
5. Richie, D. M., K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM*, Vol. 17, No. 7, July, 1974, pp. 365-375.
6. Lampson, B., et al., "A User Machine in a Time-Sharing System," *Proceedings IEEE 54*, 12 December, 1966, pp. 1766-1774.
7. McKenzie, A. A., *HOST/HOST Protocol for the ARPA Network*, National Technical Information Service, AD757680.
8. Crocker, S. D., R. M. Metcalfe, J. B. Postel and J. F. Heafnet, "Function-Oriented Protocols for the ARPA Computer Network," *Proceedings of AFIPS SJCC* 1972, Vol. 40, pp. 271-279.
9. Retz, D., J. Miller, J. McClurg, B. Schafer, *ELF System*

Programmer's Guide," Speech Communications Research Laboratory, Santa Barbara, Calif., September, 1974.
10. Brinch-Hansen, P., *Operating System Principles,* Prentice-Hall, July, 1973.
11. Dijkstra, E. W., The Structure of the "THE"-Multiprogramming System," *Communications of the ACM*, Vol. 11, No. 5, May, 1968, pp. 341-346.
12. Spier, M., E. Organick, "The MULTICS Interprocess Communication Facility," *Proceedings of the ACM Second Symposium on Operating System Principles,* Princeton University, October 20-22, 1969, pp. 83-91.
13. Digital Equipment Corporation, *RSX-11 I/O Operations Manual,* Order No. DEC-11-OMFSA-B-D.
14. Burchfiel, J., R. Tomlinson, M. Beeler, "Functions and Structure of a Packet Radio Station," *Proceedings of AFIPS NCC* 1975, Vol. 44, pp. 245-251.