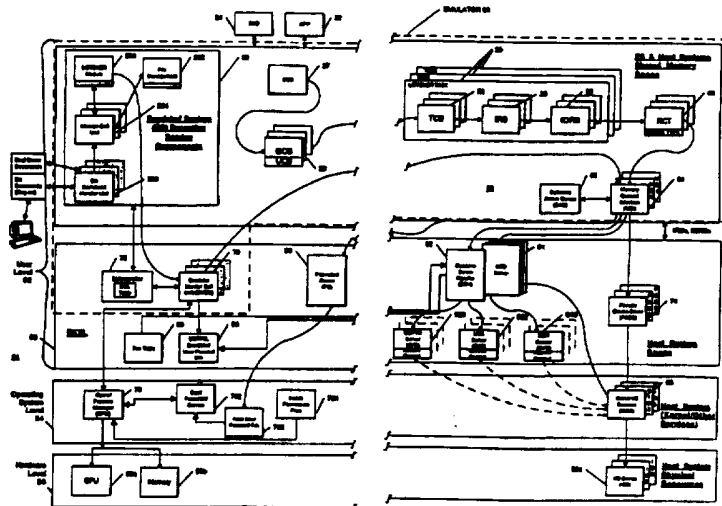




INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification ⁶ : G06F	A2	(11) International Publication Number: WO 96/10224 (43) International Publication Date: 4 April 1996 (04.04.96)
(21) International Application Number: PCT/US95/12354 (22) International Filing Date: 12 September 1995 (12.09.95) (30) Priority Data: 08/311,646 23 September 1994 (23.09.94) US (71) Applicant: BULL HN INFORMATION SYSTEMS INC. [US/US]; 300 Concord Road, Billerica, MA 01821-4186 (US). (72) Inventors: HIRSCH, Thomas, S.; 24 Fox Run Road, Bedford, MA 01730 (US). BIANCHI, Richard, S.; 4 Rodeo Circle, Billerica, MA 01821 (US). PERRY, Ron, B.; 63 Hillside Road, Wilton, NH 03086 (US). (74) Agent: SOLAKIAN, John, S.; Bull HN Information Systems Inc., Law Office MA30-883A, 200 Concord Road, Billerica, MA 01821-4186 (US).		(81) Designated States: European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE). Published <i>Without international search report and to be republished upon receipt of that report.</i>

(54) Title: A MECHANISM FOR LINKING TOGETHER THE FILES OF EMULATED AND HOST SYSTEM FOR ACCESS BY EMULATED SYSTEM USERS



(57) Abstract

A host data processing system which includes a plurality of input/output devices operates under the control of an enhanced version of the UNIX operating system. The host system includes an emulator which runs as an application process for executing user emulated system (ES) application programs. The emulator includes a number of emulated system executive service components operating in shared memory and an interpreter, an emulator monitor call unit (EMCU) and a number of server facilities operating in the host memory. The ES executive service command handler and file management components are extended to accommodate to allow creation and access to linked files within both host and emulated system files. The server facilities include mechanisms for performing trusted user level validation when a command invokes the access of such a linked file and access checking at file access time which ensures security relative to preventing both unauthorized user access and compromises in host file system data by improper use of linked files.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	GB	United Kingdom	MR	Mauritania
AU	Australia	GE	Georgia	MW	Malawi
BB	Barbados	GN	Guinea	NE	Niger
BE	Belgium	GR	Greece	NL	Netherlands
BF	Burkina Faso	HU	Hungary	NO	Norway
BG	Bulgaria	IE	Ireland	NZ	New Zealand
BJ	Benin	IT	Italy	PL	Poland
BR	Brazil	JP	Japan	PT	Portugal
BY	Belarus	KE	Kenya	RO	Romania
CA	Canada	KG	Kyrgyzstan	RU	Russian Federation
CF	Central African Republic	KP	Democratic People's Republic of Korea	SD	Sudan
CG	Congo	KR	Republic of Korea	SE	Sweden
CH	Switzerland	KZ	Kazakhstan	SI	Slovenia
CI	Côte d'Ivoire	LI	Liechtenstein	SK	Slovakia
CM	Cameroon	LK	Sri Lanka	SN	Senegal
CN	China	LU	Luxembourg	TD	Chad
CS	Czechoslovakia	LV	Latvia	TG	Togo
CZ	Czech Republic	MC	Monaco	TJ	Tajikistan
DE	Germany	MD	Republic of Moldova	TT	Trinidad and Tobago
DK	Denmark	MG	Madagascar	UA	Ukraine
ES	Spain	ML	Mali	US	United States of America
FI	Finland	MN	Mongolia	UZ	Uzbekistan
FR	France			VN	Viet Nam
GA	Gabon				

**A MECHANISM FOR LINKING TOGETHER THE FILES OF EMULATED AND HOST SYSTEM
FOR ACCESS BY EMULATED SYSTEM USERS**

This is a Continuation-in-Part of copending patent application serial no. 08/128,456, filed on September 28, 1993.

RELATED APPLICATION(S)

1. The patent application of Richard S. Bianchi, Thomas S. Hirsch and Ron B. Perry entitled, "A Dual Decor Capability for a Host System which Runs Emulated Application Programs to Enable Direct Access to Host Facilities for Executing Emulated System Operations," filed on _____, bearing serial number _____ which is assigned to the same assignee as this patent application.
2. The patent application of John L. Curley, Thomas S. Hirsch, James W. Stonier and Kin C. Yu entitled, "A Mechanism for Enabling Emulation System Users to Directly Invoke a Number of Host System Facilities for Executing Host Procedures Interactively or Non-Interactively," filed on _____, bearing serial number _____ which is assigned to the same assignee as this patent application.
3. The patent application of Thomas S. Hirsch, Richard S. Bianchi, Ron B. Perry and Kenneth S. Buck entitled, "A Mechanism for Viewing the Files of a Host System by an Emulated System," filed on _____, bearing serial number _____ which is assigned to the same assignee as this patent application.

-2-

BACKGROUND OF THE INVENTION

Field of Use

The present invention relates to the emulation of one computer system by another computer system, and more particularly to software emulation.

Prior Art

With the rapid evolution of higher speed computer systems and new microprocessor chip designs, users now have even a greater need to have their application programs written for a specific computer system run on different computer systems. Moreover, it becomes more important than ever to provide this capability without having to convert or port such application programs. Therefore, a number of vendors have developed both hardware and software approaches to achieve this result without having to convert or port such application programs.

One prior art system achieves the above by providing the capability of running a plurality of different operating systems. Each operating system is defined as having an interior decor and are utilized by an apparatus which isolates the operating systems and ensures that only the instructions of the currently active operating system are executed. This system is disclosed in U.S. Patent No. 4, 530,052 to James L. King entitled, "Apparatus and Method for a Data Processing Unit Sharing a Plurality of Operating Systems" which issued on July 16, 1985 and is assigned to the assignee named herein. This system has the limitation of not being able to run more than one operating system at a time.

Another prior art system provides a plurality of central processing units each of which tightly couples together and utilizes a different type of operating system. The central processing units in combination with their operating systems operate in a peer relationship in which all systems have access to all system resources. To ensure the integrity of the most secure of the operating systems (i.e., deemed the proprietary operating system), certain mechanisms are included in the system for enforcing certain restrictions on the other less secure operating system (i.e., deemed the non-proprietary operating systems). These restrictions are that a non-proprietary operating system can only access memory allocated by the proprietary operating system and can use only those channels and input/output instructions specifically assigned by the proprietary operating system. Stated differently, mechanisms are

provided to prevent the intentional or unintentional use of resources not assigned to the particular central processing unit and operating system combination.

The above system is disclosed in U.S. Patent Nos. 5,027,271 and 5,230,065 invented by John L. Curley, et al. entitled, "Apparatus and Method for Alterable Resource Partitioning Enforcement in a Data Processing System Having Central Processing Units using Different Operating Systems" and "Apparatus and Method for a Data Processing System Having a Peer Relationship Among a Plurality of Central Processing Units" which issued on June 25, 1991 and July 20, 1993, respectively, and are assigned to the same assignee as named herein. It will be noted that while this system enables the running of more than one operating system, it requires the use of separate independent central processing units and mechanisms for enforcing the use of only assigned system resources.

When running different operating systems within a system, it becomes desirable to provide a user with the capability of utilizing both systems in an efficient manner. In the prior art system of U.S. Patent No. 5,230,065, a user is allowed access to both operating systems through a user registration process performed on the proprietary operating system whose results are then transferred to the non-proprietary operating system. When a user has been registered on the proprietary operating system as having access to both systems, the user can issue a special switch command which switches the user's terminal over to the non-proprietary operating system. The switch command allows permanent access (i.e., access until logout from the current session) or temporary access (i.e., access to execute a single command) to the non-proprietary operating system. During this access, the user is able to issue commands while running on the non-proprietary operating system which may for example allow the transfer of data to the other operating system. For a further description of this system and operations, reference may be made to the publication entitled "GCOS6 HVS6 PLUS XC Processor Systems Basics" published by Bull HN Information Systems Inc., dated December, 1991, order number LH39-02.

While the above prior art arrangement allows a user access to both operating systems, such access takes place at the user level while both operating systems maintain their independent mode of operation. It will be noted that this type of access capability can only be invoked at a high level and still requires user interaction with both operating systems as independent entities. Hence, this capability was in effect built on top of both operating systems, relying heavily on a user's ability to issue the

-4-

proper sequence of commands to both systems for the execution of the appropriate procedures by each operating system. This arrangement is less efficient since it can be very cumbersome and time consuming to use.

Further, the above prior art arrangement does not provide for stringent security enforcement within both operating systems. That is, the proprietary operating system validates security as to unauthorized access for the entire system through the proprietary operating system's log-in mechanism. As to security for preventing compromises in data, the system includes mechanisms which enable the proprietary operating system to enforce resource restrictions on the non-proprietary system. Thus, this type of approach would not be effective in systems such as that of the present invention in the case where the application programs being run on the proprietary operating system are the application programs which are required to be emulated by the host system.

Furthermore, in the type of prior art proprietary operating system contemplated, information disclosing a substantial part of the organization of the operating system and its associated data structures historically have been made accessible to users. Therefore, it is likely that a knowledgeable user could intentionally or unintentionally alter the operating system data structures defining the user's identity and data access rights to gain access to other user system data files. Since this type of activity would go undetected in the prior art system and therefore unpreventable, this would result in significant compromises in system security.

Accordingly, it is a primary object of the present invention to provide access to host file system facilities for more efficient execution of application programs being emulated on a host system.

It is another object of the present invention to provide a method and system which enables application programs referencing linked files while running in an emulation environment on a host system to be efficiently executed in a highly secure manner.

It is still another object of the present invention to provide a method and system for executing application programs running in an emulation environment on a host system which requires no changes to the host system operating system facilities thereby facilitating software enhancement and support.

SUMMARY OF THE INVENTION

The above and other objects of the present invention are achieved in a preferred embodiment of the host data processing system of the present invention which includes a plurality of input/output devices and operates under the control of a non-proprietary operating system. In the preferred embodiment, the operating system is an enhanced version of the UNIX operating system known as the AIX™ operating system developed by IBM Corporation. The host system includes an emulator which runs as an application process for executing user emulated system (ES) application programs which in the preferred embodiment were originally written for use on the DPS6 and DPS6000 family of computers and the GCOS6™ and HVS operating systems developed by Bull HN Information Systems Inc.

The emulator includes a number of emulated system executive service components (e.g., HVS executive service components) operating in ES and host system shared memory environment and an interpreter, an emulator monitor call unit (EMCU) and a number of server facilities operating in the host system memory environment. The ES executive service command handler and file system components are extended and modified to accommodate links created between emulated system and host system files and links created to named pipes for establishing interprocess communications (IPC) between emulated and host systems processes.

The EMCU includes mechanisms for performing an initial level security validation operation which allows subsequent trusted verification of user identity when dual decor commands or functions are invoked. The server facilities include mechanisms for performing a two step security validation process for ensuring security relative to preventing both unauthorized user access and compromises in user data through improper use of dual decor commands or functions. The first step validates security at the user level and the second step validates security at the file level.

More specifically, first, the system validates that the user invoking the operation is in fact a trusted dual decor user. This is carried out by using a unique encrypted description entry of each dual decor user which was previously stored in a user table located in the host memory environment which is inaccessible to ES users. The storing of the entry is done as part of the login procedure in which the ES

system, upon verifying that the user is a dual decor user, issues a special monitor call to the EMCU. After validating that the special monitor call came from the proper source, the EMCU verifies that the user is an authorized host system user through the host encrypted password facilities. Once the user identity has been authenticated, the ES system creates the appropriate entry in the user table, which uniquely defines the personality of the user.

In accordance with the present invention, this encrypted description entry has three basic parts or components. These are: the emulated system user id itself which, during the log-in procedure, was established as matching the host user identification; the address of a first emulated system data structure defining the location of the user control block (UCB) and the address of a second emulated system data structure defining the location of the group control block (GCB). The addresses of each set of these data structures are uniquely assigned to a user by the appropriate ES system component during login and only a single copy for any given ES user exists in memory. By maintaining this set of uniquely assigned addresses of these data structures and associating them with the user id, the system of the present invention is now able to determine during the first step of the verification that the user is a trusted dual decor user at the point when a dual decor command or function is invoked by the user.

The user personality is validated by the EMCU confirming that original description or personality of the user has not been changed. That is, the user has not changed the user id so that it no longer is associated with the same set of UCB and GCB originally assigned addresses indicating that the user's personality has been changed.

If the user description is established as being valid, then the required type of server facilities are created or assigned and necessary connections are made between the server facilities and the ES component which initiated the dual decor link operation to the host file system.

If the system determines that the user invoking the dual decor link operation is not a legitimate dual decor user, the system takes no further action (i.e., does not establish the necessary connections) to execute the dual decor link function. Thus, there is no way for the user to have the function executed thus ensuring security. The user is notified of the violation by the particular ES component which initiated the operation.

-7-

The second step is performed by the server facilities at the file or directory level at file access time (open) using the built-in file protection mechanisms of the host system to compare the user permissions within the group to which the user belongs against the permissions associated with the data files to be accessed to verify that they are correct. If correct, the requested operation is allowed to proceed. If the user is determined as not having the required level of permission, the server facilities do not perform the requested operation and the user is notified of the access security violation.

In the preferred embodiment, a dynamic server handler (DSH) included as part of the server facilities performs the first step of the security validation operation for essentially all of the different types of dual decor commands and functions. The access check step is performed at a lower level by either a link file or pipe server assigned or created by the DSH unit to perform the particular dual decor operation. Access control checking is done at a lower level where it can be executed by the link file or pipe server designed to handle that operation and for more effective validation of each type of dual decor file access requests (e.g., open, read, write) for the particular type of host file involved. For example, this arrangement ensures security in those situations where it is not possible to determine the type of access being requested by a particular user before an initial phase of the operation is allowed to take place (i.e., pipe mechanisms).

In accordance with the teachings of the present invention, dual decor link commands or functions enabling access to the host file system via links are provided as part of the dual decor facilities. The link (LK) dual decor commands are designated as extensions to existing commands such as by the inclusion of uniquely designed arguments designating host system entities (e.g., host path or procedure names).

The link dual decor commands allow an emulated system user to establish links between files within the emulated system and host system. In the preferred embodiment, a user can create links between host files and emulated system. As indicated above, the files can include standard files and pipes. The pipes are host "named pipes" accessed for establishing interprocess communication between emulated system and host applications for cooperative concurrent processing. In the preferred embodiment, the host files are accessed as emulated system files in binary or text mode.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a block diagram of a host system which incorporates the dual decor link command mechanisms of the present invention.

Figurea 2a through 2f are flow diagrams used to explain the operation of the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENT

Figure 1 is a block diagram of a host system 54 which incorporates the dual decor components of the present invention. As shown, the system 54 includes a hardware platform 56 which contains the hardware elements such as a central processing unit 58a, a main memory 58b and a number of input/output peripheral/communications devices including a local area network for connecting system 54 to other processing systems via standard communication network facilities.

The central processing unit (CPU) represented by block 58a is a reduced instruction set (RISC) based processing unit which takes the form of the RS6000 microprocessor manufactured by IBM corporation. The hardware platform 56 may also take the form of the DPX/20 system marketed by Bull HN Information Systems Inc.

As seen from Figure 1, hardware platform including processing unit 58a operates under the control of an enhanced version of the UNIX™ operating system such as the AIX™ operating system. Portions of physical memory represented by MEM block 58b are illustrated in terms of the layered construction. As shown, memory is divided into two basic levels, a user level and a kernel level. The user level is divided into emulated system (ES) and host shared memory space and host or native memory space. The shared memory space contains the ES executive level 16 which includes a plurality of executive program tasks 30 spawned by ES executive services components of block 28 for executing ES application programs 22 and system administrator programs 24.

In the emulated system, each task 30 utilizes a plurality of data control structures, such as a task control block (TCB) structure 32, an indirect request block (IRB) structure 36, an input/output request block (IORB) structure 38 and a resource

control table (RCT) structure 40. The task control block (TCB) structure 32 contains information pertaining to the state of execution of the associated task as well as pointers to interrupt save areas for storing hardware parameters related to the task. The indirect request block (IRB) structure 36 contains information defining the operation requested by an associated task and includes pointers identifying the task and its associated task control block (TCB) and a pointer to the associated IORB structure.

The input/output request block (IORB) structure 38 is used as the standard means of requesting a physical I/O service. It contains information such as a logical resource number (LRN) that identifies the I/O device being addressed as well as the location and size of the buffer to be used for the transfer and the specific function (operation) requested. The resource control table (RCT) structure 40 contains information describing the resources, such as its characteristics or information regarding the tasks or requests being executed by a corresponding resource as well as pointers to its associated task control block (TCB) structure.

Additionally, two other structures depicted in Figure 1 utilized by the present invention are a group control block (GCB) structure and a user control block structure of block 29. The GCB structure contains information required to define and control the operations of a specific task group which defines a named set of one or more tasks with a common set of resources within which a user and system function must operate. Each group has a two character name (e.g., \$L, \$S) by which the group is uniquely known to the system. The GCB structure includes information identifying the lead task whose execution spawns all other tasks required for executing group programs. As indicated, the GCB structure includes a number of user control blocks (UCB), each of which contains information defining the user's personality such as user node identification, user group id within a node, user task id within group, user person id and pointer information to directories to which the user has access.

As shown, the emulated system utilizes a further data structure corresponding to system control block (SCB) structure 27. This data structure is created at system startup and contains information defining system resources and pointers to the different task groups established by the system represented by a corresponding number of group control blocks in the system. For further information regarding such structures and their relationships to each other, reference may be made to U.S.

-10-

Patent No. 5,111,384 and the publication entitled "HVS PLUS Systems Concepts" published by Bull HN Information Systems Inc., Order No. HE03-01.

As indicated in Figure 1, the shared memory space further includes a memory queued interface (MQI) represented by block 84 which provides a form of interprocess communication mechanism and a software active queue (SAQ) of block 88. SAQ block 88 represents a data structure used to provide the path by which the results of the operations performed by the kernel level components are passed back or returned by the host processes to the requesting emulated system user level tasks 30 being executed. Thus, it can be viewed as functioning as an output stage of MQI 84. This data structure is similar to data structures which are used by the emulated system operating system..

MQI block 84 is a semaphore data structure which takes the form of a single linked list controlled by semaphores through a set of routines which are executed by the various host processes operating within different levels or layers that want to communicate with each other. Its routines are used to manage queues within the pseudo device drivers 74 and the software active queue 88.

Executive Services Components 28

As seen in Figure 1, the executive services components 28 of executive layer 16 includes a plurality of components or facilities which are equivalent to those facilities normally included in emulated system. The facilities utilized by the present invention include a listener module 280, a file management facility 282, a monitor call unit 284 and a ES command handler unit 286 which are arranged as shown. The listener module 280 is responsible for monitoring the operations of terminals configured for login and for initiating user processes in response to user commands. As indicated in Figure 1, listener module 280 runs as a task 30 with its own set of unique data structures.

As described herein, the listener module 280 is able to consult a profiles file containing user specific registration information such as user id, login id and password requirements tabulated by the system administrator for all registered users. The listener module 280 checks the user profile when monitoring the privileges and/or restrictions given to each user. The file management facility 282 includes the conventional shared data structure and set of routines normally provided to perform functions that access such data structure to control the synchronization of concurrent

-11-

processes or tasks in addition to performing various system services or functions. That is, the facility responds to system service monitor calls identifying the types of services requested (e.g. creating or deleting files, reading or writing records or blocks in files) which result in the specified system services being executed by the emulated system on behalf of executing user application programs.

File Management Data Structures

The ES file management component 282 utilizes a number of data structures in conjunction with carrying out file system services as described in greater detail herein. Information about a ES disk device including the device name as well as the name of the logical volume currently mounted on that device is maintained in a volume descriptor block (VDB) structure. A chain of VDB's is provided off of the SCB. The emulated system file system provides a directory tree structure within each mounted disk volume. For each file, index and directory being accessed, information is maintained in memory in a file descriptor block (FDB) structure. FDB's are organized and chained in memory in a tree structure which emanates from VDB's similar to how it is organized on the disk. FDB's contain the name of the file, links to the superior and inferior directory FDB levels as well as a link to the next FDB on the same directory level. FDB's contain logical information such as record size, key locations and sizes which allow for logical record level I/O services.

In addition, an FDB describes where (at what physical sectors) the data of the file is located and also contains information necessary to request I/O services from the appropriate device driver in the system being emulated. An FDB is created in memory as a result of a user application monitor call which reserves the file for processing and it is deleted when the application removes the file.

Also, the ES file management component 282 creates a file control block (FCB) data structure which contains information relating to a specific use of the file such as a current record pointer (e.g., one instance of opening the file). The FCB structure is found by locating a logical file table (LFT) linked to the group control block (GCB) structure. The LFT is simply an array of pointers to FCBs. The logical file number (LFN) is assigned when the file is reserved and serves as an index into the array.

For non-disk devices such as tapes, printers and terminals, a device descriptor block (DDB) structure is maintained. A DDB contains the name of the device as well

-12-

as logical information such as record size. For tapes, a DDB contains the name of the volume currently mounted on the device and the name of the file currently being accessed. A DDB also contains the physical information necessary to request I/O services from the appropriate emulated system device driver. A chain of DDBs is provided off of the SCB and this chain as other chains are created during system configuration (startup).

The monitor call unit 284 receives monitor calls from the interpreter component 72 which are in turn to be executed interpretively using the ES executive service components of block 28. The command handler unit 286 contains the routines that respond to user commands entered via a terminal or program. In response to such commands, the unit 286 routines invoke the appropriate processes for executing such commands. As discussed herein in greater detail, each of these components have been augmented or extended to incorporate different dual decor aspects or features according to the teachings of the present invention.

Emulator Level Layer 68

As indicated in Figure 1, the next layer within the user level is the emulator executive level 68. This level includes certain components present in the emulated system which have been transformed into new mechanisms which appear to the remaining unchanged components to operate as the original unchanged components of the emulated system. At the same time, these new mechanisms appear to the components of the kernel level 64 as native components with which the host system is accustomed to operate. As shown, the components include the interpreter 72, an emulator monitor call unit (EMCU) 73, a plurality of servers 90, through 940, and a plurality of pseudo device drivers (PSDD) 74 arranged as shown.

Additionally, level 68 includes a data structure in the form of user table 96 which contains entries which uniquely define the personalities of the different dual decor users. This table is maintained to be able to determine that a given user has successfully gone through the login procedure. In accordance with the present invention, each such encrypted user description entry has three basic parts or components. These are: the emulated system user id itself which during the log-in procedure was previously established as matching the host user identification; the address of a first emulated system data structure defining the location of the user control block (UCB) and the address of a second emulated system data structure

defining the location of the user group control block (GCB). The Appendix shows the format of these entries in greater detail.

The interpreter 72 successively fetches the instructions of an emulated system application program, categorizes each instruction and executes it interpretively through sequences of RISC instructions which allows CPU 58a, MEM 58b and other elements of host system 54 to emulate the operations of corresponding elements of the emulated system. The interpreter 72 includes a monitor call (MCL) table containing information for each possible monitor call which it utilizes to determine whether to trap or send an ES monitor call to the ES executive services components 28 for execution of the instruction or to make an emulator call to EMCU 73 for execution of the instruction through the services of an appropriate C language routine (server). The EMCU 73 is responsible for acquiring from the host system 54, the necessary memory and other resources, for initializing the emulated system data structures and invoking interpreter 72 and the various server processes. Both the interpreter 72 and EMCU 73 run as host processes but not as root.

As viewed by the host system, the ES service components 28 and tasks 30 being executed on behalf of the application programs, the interpreter 72 and EMCU 73 are executed in the system 54 of Figure 1 as a single process 80 wherein such process corresponds to one or more user processes as defined by the conventions of the host operating system being run on host system 54. Thus, it is possible to have multiple instances of the emulated system concurrently emulated on host system 54.

The password server 90, the dynamic server handler (DSH) 92 and the network terminal driver (NTD) server 94 are created by EMCU 73 during initialization. Each of the servers 90, 92 and 94 communicate with emulated system processes through MQI 84 as indicated. The lower level group of servers 920, 922 and 940 are dynamically created by their respective higher level servers 92 and 94 for carrying dual decor operations according to the present invention. All of the servers operate as root and therefore have super user privileges with access to any file within the host system 54. The NTD server 94 is designed to contain the functionality required to handle different types of terminals such as the network terminal driver described in U.S. Patent No. 4,951,245 which issued on August 21, 1990.

As described in detail herein, both the servers 92 and 94 include mechanisms specifically designed for validating security at the user level in conjunction with the execution of dual decor commands and functions. As shown in Figure 1, the lower

-14-

level group of servers 920, 922 and 940, each include access control mechanisms used to validate security at the file level. In the case of server 940, host access control mechanisms are used.

As indicated in Figure 1, the EMCU 73, dynamic server handler 92 and NTD server 94, each have access to a data structure in the form of a user table (USRTBL) 96 which has been located in host space so as to be inaccessible to ES user application programs. As described herein in greater detail, the table 96 contains encrypted user descriptor information which is used for validating a user's identity according to the teachings of the present invention. The encryption ensures security in the event of a memory dump.

Also, the emulator level includes a file table structure 98 which is used for storing information pertaining to linked host system files being processed by the server facilities for which links to emulated system files were created as the result of having invoked dual decor link commands. Such information provides an appropriate reference information for processing successive reads or writes to and from a given host file as described herein.

As indicated in Figure 1, the emulator executive level 68 further includes a plurality of pseudo devices drivers (PSDD) 74 for each input/output device or type of input/output device which is required to be emulated by host system 54. For example, the pseudo device drivers 74 will include PSDDs for terminals, disk drivers, tape drivers, displays and for certain communication devices.

For a more detailed discussion of other aspects of the SAQ 88, MQI block 84, PSDD 74 and other emulator components, reference may be made to the related patent application.

Operating System/Kernel Level

The operating system/kernel level 64 includes the standard mechanisms and components normally included within the host operating system. As shown, level 64 includes a kernel process manager component 70, a host user password file 700, a host crypt password service facility 702, Inode permission files storage 704 and a number of host kernel I/O services (KIOS) processes 66 for each pseudo device driver (PSDD) 74 which is to be emulated by the host system and for each of the servers 920 through 940 assigned to execute dual decor commands and functions. Since the components of the kernel level 64 are well known, they are only briefly

described herein. The host user password file 700 is used for storing information identifying registered host users. This file is updated and maintained by the host operating system kernel process manager 70. A part of the information stored in the password file 700 is the user encrypted password generated using the one way crypt password service facility 702 invoked by the kernel process manager 70.

Additionally, in the preferred embodiment of host system 54, level 64 is assumed to contain the standard utility programs, shell, editors, compilers, etc. and libraries (e.g., I/O libraries, open, close) which are accessed in the host user mode. For further information regarding the use of such arrangements, reference may be made to publications of the IBM Corporation describing the AIX operating system.

Host System File System

The host file system consists of a sequence of homogeneous logical blocks, each containing a convenient multiple of 512 bytes. In the host file system, the internal representation of a file is given by an Inode which is the contraction of the term index mode. The mode contains a description of the disk layout of the file data and other information such as the file owner, access permissions and file times. Every file has one Inode, but the Inode may have several names or links, all of which map into the Inode. The kernel maintains a file table and a user file descriptor table. The file table is a global kernel structure and contains information for tracking where a user's next read or write starts and the access rights allowed to the opening process. The user file descriptor table is allocated per process and contains information identifying all open files for a process. When a process opens or creates a file, the kernel allocates an entry from each table corresponding to the file's Inode. The entries in the user file descriptor table, file table and Inode table maintain the state of the file and user's access to the file.

The Inode permission files storage contains entries listing the attributes and disk addresses of each file in the system. Each Inode entry contains information about the type, size, times, ownership and disk blocks in each Inode. The directory structure in the host system uses entries, each of which contains just a file name and its Inode number. The attributes of an entry include references to access control lists (ACLs) defining the file's base and any extended permissions. The base permissions correspond to the traditional file access modes (i.e., read, write and execute/search) assigned to a file owner, file group and other users. The extended permissions

-16-

designate how the base permissions have been modified in terms of permitting, denying or specifying access modes for specific individuals, groups, or user and group combinations designated by user and group ids. For the purpose of the present invention, this arrangement can be considered conventional in design. For further information regarding the use of such arrangements, reference may be made to publications of the IBM Corporation.

DESCRIPTION OF OPERATION

With reference to Figure 2, the dual decor link command mechanism incorporated into system 54 according to the teachings of the present invention will now be described. As diagrammatically illustrated in Figure 1, dual decor commands encompass new commands and extensions to existing emulated system commands. The dual decor link (LNK) file commands can be viewed as extensions to existing commands.

The format of the LNK command is:

LK GCOSNAME "X:xpath" [-BINARY | -TEXT] [-FILE | -PIPE].

The term "LK" specifies the link command code which is followed by a series of arguments. The arguments are used to define a subset of the large number of emulated system file types. The arguments essentially indicate what the host file type as viewed by the emulator system. The argument GCOSNAME specifies the emulated system name of the link.

The arguments -BINARY | -BIN | -B | -TEXT | -TXT | -T specify whether the pipe or link being created is a binary pipe or link or a text pipe or link. The xpath argument is a host pathname for the link. The pathname is case sensitive. It must be enclosed in quotes to protect lower case characters and start with the prefix x: or X:. When the pathname does not start with the prefix x: or X:, it is assumed to be an emulated system file. If the LNK command specifies linking to a file that does not exist, the file will be created if security access permissions are satisfied. The -FILE argument is used to create a link to a file and not used with -PIPE. The -PIPE argument is used to create a named pipe and is not used with -FILE. The LK command has the following defaults:

<u>Argument</u>	<u>Default</u>
LK	binary file

-17-

LK -FILE	text file
LK -PIPE	text pipe.

It will be assumed by way of example that a user registered as a dual decor user has successfully completed the login procedure. Since login has been successfully completed, the appropriate encrypted entry defining the user's personality will have been stored in user table 96 and, hence, is available for use during execution of file operations accessing links created between host and emulated system files as described herein.

The user entered LK command invokes a command processor (not shown) included within block 28. The command processor is a system software component that reads each command line entered by a user through a terminal or from a command file and then passes it to the appropriate command handler within block 28 for execution. Since the LK command is an extension to an existing command, the LK command handler has been modified and receives the command line arguments from the command processor via a pointer.

The LK command handler retrieves the arguments and parses them verifying that the arguments, such as pathname and file types are valid and supported file types. Any errors in pathname or file type are reported and processing of the LK command is terminated.

As indicated in Figure 2, the LK command handler, during its examination of the arguments, determines if the command is a dual decor link command which specifies the creation of a link to a host file or pipe (block 200). If it is not a dual decor link command, then the operation proceeds in a normal manner resulting in the creation of an emulated system directory link entry which is then written to the file system (block 202). If it is a dual decor link command as designated by the "X:", then the LK command handler creates the host file system pathname and a keyword designating the file type. The file type keyword is derived from the text, binary file and pipe arguments which provide a certain level of information about the characteristics of the particular host file for which the link is being created. That is, there are four possible file types which correspond to the four combinations: binary file, binary pipe, text file and text pipe. The four combinations are compressed into one key word value.

Next, the LK command handler issues a standard file system monitor call which is passed by monitor call unit 284 to file management component 282. As

-18-

indicated in Figure 2, the file system component 282 issues a request I/O (RQIO) for writing the appropriate directory link file entry on the emulated system disk like any other link entry. The request is handled like any other I/O request made by an emulated system user. Briefly, the request is passed through MQI84 to the appropriate pseudo device driver which initiates the write to the disk I/O device via kernel I/O services block 66.

As seen from Figure 2a, this completes the execution of the LK dual decor command. Nothing further is done regarding the created link until a user reserves the emulated system file associated with the earlier link prior to opening it by issuing a get file command or system monitor call. Such calls are used to control file concurrency.

The GET file command as in the emulated system results in a system call being issued by monitor call unit 284 to the file management component 282. As seen in Figure 2b, the file management component 282 during its processing of the call arguments determines if the link name contains an "X:" indicating access to host link files and is to be treated in a special fashion. If it does, then as part of resolving the link name, it stores the name away in an appropriate structure so that it can reference it later. Since information about the type file is already available, no file system calls are issued. Component 282 uses the keyword identifying the file as either text or binary or either pipe or file to set up internal indicators in a predetermined manner for designating the characteristics of the type of file to be processed (block 210). If the file management component 210 does not contain an "X:", it executes the operation in the normal manner (block 212).

As indicated in Figure 2b, the file management component 282 next allocates and initializes the FCB and FDB file control block (FCB) and file descriptor block (FDB) structures as required to execute the link operation. More specifically, indicators are set within the FCB to designate that this is not a normal link unit specifying host access requiring dual decor functionality. The FDB is set up to identify the emulated file system/host file system pathname.

Also, a buffer control block (BCB) structure is allocated and is used for setting up the Request I/O (IORB) structure which will be used for every operation on the file. The BCB structure is accessed by an address pointer contained in the FCB structure.

The IORB structure is extended to store certain types of information required for processing the dual decor link operation. This includes an address to the file table,

pointers to the current files, write seek value fields for enabling the emulated system to update the necessary file system data structures and the FCB address for the file as a convenient reference pointer. Another field corresponding to a link pipe | link file | CPX field is used on IORB CONNECTS and enables DSH server 92 to determine the type of CONNECT. Other fields such as FS failure and I/O failure are used to pass back different error code information. Also, the IORB includes fields for referencing the BCB. The IORB structure is described in greater detail in the Appendix.

As described herein, the file system 282 uses the IORB structure in a bidirectional fashion. It initializes it and sends it down to the server in the case of all requests. For the most part, the contents of the basic IORB fields are reset for each request by the server except for the contents of several fields such as the pathname field, a range of pathname field defining the number of bytes in the pathname, and other information which are passed across in a special manner only for the first request. Since the same IORB is used in subsequent requests, some of the fields are used by the server as working storage.

As indicated in Figure 2b, the file management component 282 initializes the BCB/IORB structure to store the pathname address and the server LRN value (block 214). Next, the file management component 282 obtains the LRN of the DSH92 server and then initiates a request I/O (RQIO) emulated system monitor call specifying a physical connect operation.

The request is received by the EMCU 73 together with the IRB containing the LRN value of DSH92 and other information for processing the request. EMCU 73 receives all such requests and has the responsibility of determining which server is to receive each request. It will be noted that in the case of linked file request, there will be a different LRN value for each file opened by the same user and that only one server will be created or assigned to process all requests for a given user. In the case of linked pipe requests, a different server will be created or assigned to process each such request. Each server has its own queue on which requests are placed.

Therefore, EMCU 73 includes a dispatching mechanism for determining which servers/queues are to receive the linked file and linked pipe requests using only the LRN values contained in the associated IRBs. The dispatching mechanism utilizes the RCTs, each containing a number of type fields and associated with the files and are in turn linked to the server queues. The EMCU dispatching mechanism references the

-20-

RCT for the particular file designated by the LRN value and obtains a pointer from the user table 96 which identifies the server queue and places the request on the queue. When a new server is created, a server request queue will also be created and linked off the user table entry for that server.

In the case of the RQIO to DSH92, EMCU 73 places the request on the MQI queue for the DSH server 92. When no server has been created or assigned to process this user's request, the RCT entry corresponding to the LRN is set to a value (i.e., ZERO) to indicate this fact (block 216).

As indicated in Figure 2, DSH server 92 performs a user security level check to validate that the command specifying access to the dual decor (special) link was issued by a dual decor user and that security is not being compromised (block 218). This is done by obtaining the encrypted user id and UCB and GCB address information from user table 94 and comparing against the current user id and UCB and GCB address information after encrypting it. Assuming that the current user personality is the same validating the user as a "trusted user", DSH92 next determines which server is required to handle this type of request. DSH92 makes this determination by obtaining the RCT entry using the LRN value contained in the IORB and examining the "device type field." If the "device type field" indicates that a server does not exist for this type of request, then DSH92 creates a server and passes the request to the assigned server via MQI. That is, DSH92 issues fork and exec system calls to the kernel manager 70 which spawns a new host process running LK file or LK pipe server code as root. DSH92 then enqueues the IORB onto the server's MQI queue (block 220). If the request is a LK file request, DSH92 creates a new server and queue pair if one does not already exist to handle the file requests for that user. If the request is a LK pipe request, DSH92 creates a new server and queue pair for each such request. That is, each time DSH92 creates a new server, it also creates a request queue within file table 96 and links it off the user table entry for that server.

Next, DSH92 establishes the connection between the file management component 282 and the assigned server so that all subsequent requests pertaining to that file go directly to the assigned server. That is, DSH92 establishes the proper link between the RCT associated with the request and the server queue for directing file requests made by a given user to the single common queue utilized by the assigned server and for directing each pipe request made by a given user to the queue of a different server. This is done by linking the server off of the RCT from the LRN.

Next, the next free queue entry is obtained from the file table 96 and linked into the user table entry for that user as described above (block 220).

As indicated in Figure 2b, either the pipe server or file server performs the required access control checks to determine that the user has the required permissions for maintaining security (blocks 222 and 224). Since the command is a get command, it is not known at this point exactly what type of operation the user intends to perform on the reserved file. Therefore, the access control check is performed at the directory level. If the user has access to the file being reserved and is able to open the file, then the server will deem that the user has the required access permission. Hence, no violation will be reported and the execution of the get command deemed completed.

Normally, the user next issues an open command after the get command. Referring to Figure 2c, it is seen that the file management component 282 executes its normal sequence of operations until it reaches a point where it determines that a special dual decor link file is involved. It then issues a Request I/O (IORB) for performing a logical CONNECT using the same IORB that was provided for the reserve file operation. Hence, the IORB is issued to the same server LRN that was given the last RQIO (block 230).

As indicated, EMCU 73 receives the request and dispatches the IORB to the queue of the appropriate server designated in the RCT for that particular file. EMCU 73 references the RCT, gets the user table address and puts the request on the appropriate queue (block 232). If the request is a LK file request, the file server performs a full access check at this time at which the file is being opened (block 234). Assuming that the user has the required permissions, the file server issues an open system call to kernel manager 70 which results in the opening of the host file (block 235).

If the request is a LK pipe request, the pipe server defers the complete access check (block 236). The pipe server then performs the open request by issuing an open system call to the kernel manager 70 which results in the opening of the host pipe (block 238).

The open command is then followed by a first read or write command and then further read or write commands as indicated in Figure 2d. These commands are processed in the same manner as the open command. That is, the operations of the

-22-

blocks 240 and 242 are the same as those of blocks 230 and 232, respectively, except for the type of IORB (Logical CONNECT vs. read/write).

If the request is a LK file request, the file server performs no further access control checks, since it established proper access previously. Next, the file server issues a read/write system call to kernel manager 70 for performing the requested operations on the specified host file.

If the request is a LK pipe request, the pipe server at the time of the first read or write is able to perform a complete access control check to determine that the request specifying the dual decor link pipe operation was invoked by a trusted user having the required permissions. Assuming that is the case, then the pipe server issues a read/write system call to kernel manager 70 for performing the requested operation (block 246).

As indicated in Figure 2d, the pipe and file servers both execute subsequent read/write commands without further access control checks. The servers process commands in the same manner until a host close command is issued. As indicated in Figure 2e, this command is also processed in the same manner as the open and read/write commands. The only different is that the pipe and file servers each perform the requested operation by issuing a close system call to kernel manager 70 (blocks 256 and 258). This results in the closing of the file or pipe by the host file system. Also, at that time, the server routes the IRB to DSH92 and frees up the File table entry. When DSH92 determines that all files/pipes associated with a server have been closed, then it kills the server and marks the RCT as available.

The last emulated system command issued is a remove command. As indicated in Figure 2f, the remove command causes the file manager component 282 to again process the command in a normal manner. Upon determining that a special dual decor link is involved, component 282 operates to remove the appropriate data structures.

From the above, it is seen how the dual decor link file/pipe mechanisms of the present invention are integrated within a host system to allow trusted users to access host files and pipes.

APPENDICES

These Appendices include source listings which are subject to copyright protection. The copyright owner, Bull HN Information Systems Inc., has no objection to the facsimile reproduction by any one of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights in such listings whatsoever.

APPENDIX A -Dual Decor Components	Pages A-1 - A-48
APPENDIX B - LN Command Examples	Pages B-1 - B-5

24

APPENDIX A DUAL DECOR COMPONENTS

 Relevant Emulated System Structure Definitions

FDB structure definition

```

$fdb    mac
#
[*]***  FILE DESCRIPTOR BLOCK (FDB)          ** Aug 10,1994 ** 20 ****
[*]
d_fdbp  equ    0          -> next FDB on same directory level      *
d_sbp   equ    $AF        -> immediately superior FDB             *
d_ibp   equ    d_sbp+$AF   -> inferior FDB chain                  *
d_ind   equ    d_ibp+$AF   indicators word                         *
d_ind2  equ    d_ind+1     second indicators word                 *
d_lrn   equ    d_ind2+1    logical resource number                 *
d_nu    equ    d_sem+$AF+1+4*$AF+5 concurrency: total number of users *

...
d_hxlp  equ    d_rxbp      -> HVX link object pathname            *
[*]
d_prim  equ    d_rxbp+$AF   start of primary directory record      *
d_nme   equ    d_prim      file name                               *
d_typ   equ    d_nme+6     file type                                *
d_lrsz  equ    d_typ+1     logical record size                     *
d_cisz  equ    d_lrsz+1    CI size for UFAS files and block size  *

...
d_hxls  equ    d_wrid      size of HVX link pathname in bytes     *
*
[*]***  FDB second indicators word (D_IND2) masks: *****
[*]
[*]      specific to network files:
[*]
md_hxl  equ    z'0020'     bit 10: 1=HVX file or pipe link        *
[*]
[*]***  FILE DESCRIPTION BLOCK (FDB) *****
endm
  
```

FCB structure definition

```

$fcf    mac
#
[*]***  FILE CONTROL BLOCK (FCB)          ** Aug 17,1994 ** 21 ****
[*]
c_fcbp  equ    0          -> next FCB in chain                    *
c_fdbp  equ    $AF        -> file descriptor block (FDB)          *
c_ncbp  equ    c_fdbp+$AF -> name control block (NCB)             *
c_ind   equ    c_ncbp+$AF FCB indicators word                     *
c_ind2  equ    c_ind+1    FCB indicators word 2                   *
  
```

25

```

c_ind3 equ c_ind2+1      FCB indicators word 3      *
c_ucnt equ c_slcd        number of current users of this *
...
[*] data/storage management specific work area      *
c_dmsm equ c_nbf+1      data/storage mgmt work area  *
c_bcbp equ c_dmsm        -> buffer control block (BCB) *
...
[*] HVX link specific fields (storage mngmt only):    *
c_bkno equ c_amsmp       next block number           *
c_bksz equ c_bkno+2      block size                  *
...
[*]
[*]*** FCB indicators word 3 masks *****
mc_hxl equ z'0080'       bit 8: 1=HVX file or pipe link *
[*]
[*]*** FILE CONTROL BLOCK (FCB) *****
endm

```

BCB structure definition

```

$bcbb mac
#
[*]*** BUFFER CONTROL BLOCK (BCB) ** Sep 09,1994 ** 14 ****
[*]
b_prvp equ 0             -> prior BCB of file          *
b_nxtp equ b_prvp+$AF     -> next BCB of file          *
b_ind equ b_nxtp+$AF      indicators word             *
b_cino equ b_ind+1        control interval or block number *
b_ucnt equ b_cino+2        number of current users of this BCB *
b_rc equ b_ucnt+1         return code from the last i/o request *
b_sem equ b_rc+1          semaphore to synchronize block requests *
[*]
b_trno equ b_sem+4*$AF+5   physical sector number within file *
b_trp equ b_trno+2        -> transfer area             *
b_trsz equ b_trp+$AF      transfer size               *
b_mxsz equ b_trno         maximum transfer size for device files *
b_sim equ b_trno+1        software simulation for device files *
[*]
b_irb equ b_trsz+1        IRB:                        *
b_iorb equ b_irb+i_len    IORB:                      *
b_lrn equ b_iorb          :LRN:                      *
b_lnk equ b_lrn+3         :-> IRB:                    *
[*]
:0:1:2:3:4:5:6:7:8:9:a:b:c:d:e:f: *
b_rct1 equ b_lnk+$AF      :rs      :t:w:u:d:retry:i: *
b_rct2 equ b_rct1+1      :escape lrm :b:rfu: funct : *
b_bad equ b_rct2+1       : -> buffer area : *
b_rng equ b_bad+$AF      :buffer range : *
b_dsw equ b_rng+1        :device specific word : *
b_rsr equ b_dsw+1        :residual range : *
b_st1 equ b_rsr+1        :hardware status : *
b_sz1 equ b_st1+1        size of BCB used for devices *
[*]

```

26

```

...
[*]
[*]*** EXTENDED BCB (for HVX links) *****
[*]
b_xuch equ b_bad :HVX -> UCB (for physical connect) *
[*]
b_xext equ b_stl+1 :Extension size *
b_xrc equ b_xext+1 :HVX return code *
b_xctl equ b_xrc+1 :HVX control indicators *
b_xfcp equ b_xctl+1 :HVX FCB address *
b_xflb equ b_xfcp+$AF :HVX elements RISC flabl address *
b_xpsn equ b_xflb+2 :HVX 32 bit record position *
b_xpap equ b_xpsn+2 :HVX -> pathname *
b_xpln equ b_xpap+$AF :HVX pathname length *
b_xtyp equ b_xpln+1 :HVX access type (for pipe or file) *
b_xrc2 equ b_xtyp+1 :HVX secondary return code value *
b_xpsb equ b_xrc2+1 :HVX PSB address (FIB | GIPSB | etc.) *
b_xmcf equ b_xpsb+$af :HVX mcl function code *
b_xrfu equ b_xmcf+1 :HVX space for growing more bright ideas*
b_sz4 equ b_xrfu+16 :HVX size of BCB used for HVX links *
...
[*]
[*]*** B_XCTL HVX indicator masks *****
[*]
mb_rnw equ z'8000' 1=opened in renew (empty) mode *
mb_app equ z'4000' 1=opened in append mode *
mb_cur equ z'3000' read/write currency: *
[*] (00=GCOS type double currency) *
[*] (10=UNIX type single currency) *
[*] (01=Fortran type single currency) *
mb_sgc equ z'2000' 1=opened in single currency mode *
[*] for C language type i/o *
mb_sgf equ z'1000' 1=opened in single currency mode for *
[*] Fortran language type i/o *
mb_mod equ z'0800' 1=text mode; 0=binary mode *
mb_psn equ z'0400' 1=position (seek) *
mb_eod equ z'0200' 1=position (seek) to End-Of-Data (EOD) *
mb_nwt equ z'0100' 1=nowait (HVX pipes only) *
mb_dir equ z'00c0' read/write direction: *
mb_dwr equ z'0080' 1=writer *
mb_drd equ z'0040' 1=reader *
mb_uwp equ z'0020' 1=break followed by UW or break caused *
* by software signal (trap #49: unwind) *
[*] 0=break followed by SR (MCL should be *
[*] retried) *
mb_con equ z'000f' file concurrency: *
mb_cwi equ z'0008' 1=inhibit other writers *
mb_cri equ z'0004' 1=inhibit other readers *
mb_cwr equ z'0002' 1=writer *
mb_crd equ z'0001' 1=reader *
[*]
[*]*** B_XTYP HVX indicator values *****
[*]
mb_pip equ z'0001' pipe *

```

27

```

mb_soc equ    z'0002'      socket      *
mb_fil equ    z'0003'      file         *
mb_cpx equ    z'0004'      cpx          *
[*]
[*]***  BUFFER CONTROL BLOCK (BCB) *****
endm

```

```

*****
Relevant Host System Structure Definitions
*****

```

```

struct TTBL

```

```

{
    struct LQHEAD tq; /* uses wtcB from connect as priority */
    unsigned short cnt; /* connect count */
    short type; /* server type (LK_FILE or LK_PIPE) */

    struct MQI tch;
    struct LQHEAD flist; /* queue of fbl entries this ttbl */
};

```

```

union FD

```

```

{
    int fdes; /* file descriptors */
    FILE *stream;
};

```

```

struct LRB

```

```

{
    struct neg_lrb
    {
        WORD lk_lrx; /* lrx extension (lrx in ct2 = FD) */
        ADDR lk_rrb[2]; /* return rb (r=1) */
    } neg;
    ADDR lk_rs[2]; /* return status */
    WORD lk_ct1; /* rs |t|w|u|s|a|r|d|l */
    WORD lk_ct2; /* lrx |i|b| |e| funct */
    ADDR lk_adr[2]; /* buffer address (UCB adr. on connect) */
    WORD lk_rng; /* range */
    WORD lk_dvs; /* device specific word */
    WORD lk_rsr; /* residual range */
    WORD lk_st; /* first status word */
    WORD lk_ext; /* extended rb length */

    WORD lk_fsfail; /* 02xx failure code */
    WORD lk_control; /* control bits */
    WORD lk_fcb[2];
    WORD lk_fbl[2]; /* elements RISC ftabl address */
    WORD lk_seek[2]; /* byte offset for seek */
    ADDR lk_pthp[2]; /* address of file/pipe name */
    WORD lk_prng; /* range of pthp */
    WORD lk_type; /* LK_PIPE | LK_FILE | LK_CPX */
    WORD lk_iofail; /* 01xx failure code */
}

```

28

};

struct FTBL

```
{
    struct LQHEAD fq;
    ulong      tskadr; /* TTBL entry for this file */
    union FD    fd; /* file descriptors for binary & text files */
    int        status;
    ulong      current_rseek; /* offset of current record for read */
    ulong      current_wseek; /* offset of current record for write */
    ulong      current_rwseek; /* offset for rewrite */
    ulong      fcb;
    short      cdvs;
    short      type;
    char pathname[MAX_PATH_RANGE+1];
};
```

29

```

*****
Dynamic Server Handler Code
*****

void
rtn_ttbl(struct LRB *, struct RCT *, struct IRB *);
void
exec_drvr(struct RCT *);

struct sigaction lnk_actions;
sigset_t lnk_sigs;
struct MQI *lnk_srvr;
struct LQHEAD *ptr_dd_freeq;
extern void srvrsegv();
extern struct LQHEAD *lnk_findq();
extern struct LQHEAD *lnk_dqh();
extern void
sigsak(), sigdanger1();

char person[14];
int mainpid, pid, signo, lrm, plog = 0;
unsigned int fdkey, msgaddr;
char *srvr_base();
struct TTBL *qhdt, *tskadr;
struct FTBL *qhdf, *fbl;
struct USRTBL *qhdu;
struct LQHEAD *lptr, *lnkq_frm, *prev, *fbl_head;
struct TCB *tcb;
struct UCB *ucb_ptr;

static union MATH math;
static char func_code;
static char *ptr;
static int i;
static unsigned long ucb, priority;
FILE *ipipe_stream, *opipe_stream; /* dummies for linker */

/* func_code deltas for specific servers - must be a multiple of 16 */

static char tree[MAXPATHLEN + 15];
char group_path[MAXPATHLEN + 1];
struct GCB *gcb;
struct LRT *lrt;
char *lnk_table;
int seq = 0;

/*****
* main *
*****/
void
main(argc, argv, envp)
int argc;
char *argv[];
char *envp[];
{

```

30

```

struct passwd  *pwent;
struct IRB     *irb;
struct RB      *iorb;

process = SERVER;
openlog(get_session_name(), LOG_PID, LOG_LOCAL0);
setlogmask(LOG_MASK(LOG_ERR)
           | LOG_MASK(LOG_WARNING)
           | LOG_MASK(LOG_NOTICE));
if (parse_options(argc, argv) != 0)
{
    syslog(LOG_ERR, "E588: LINK: Invalid command line options");
    exit(1);
}
close_fd();
mainpid = getppid();
if ((base = shmat(opt.shmid, (char *) NULL, 0)) == (char *) -1)
{
    syslog(LOG_ERR, "E589: LINK: shmat: %m");
    exit(1);
}
init_vm();
if (intrp_vm())
{
    lnk_cleanup();
    exit(1);
}
srvr_vm();
/* attach shared memory containing allow table */
if ((Msg = (struct MSG *) shmat(msg_shmid, (char *) MSG_ADDR, 0))
    == (struct MSG *) - 1)
{
    syslog(LOG_ERR, "E1146: LNK: shmat(): %m");
    lnk_cleanup();
    exit(1);
}
if ((unsigned int) Msg != msgaddr)
    syslog(LOG_ERR, "E1147: attached @ %x, should be %x", Msg, msgaddr);
lnk_actions.sa_handler = lnk_handler;
lnk_actions.sa_mask = lnk_sigs;
lnk_actions.sa_flags = 0;
for (signo = 0; signo < NSIG; signo++)
{
    switch (signo)
    {
        case SIGUSR2:
            sigaction(signo, &lnk_actions, NULL);
            break;
        case SIGCHLD:
        case SIGABRT:
        case SIGTERM:
            sigset(signo, SIG_DFL);
            break;
        case SIGSOUND:
    }
}

```

31

```

        init_rusage("DDECOR", lnk_srvr->isem.sid);
        break;
    case SIGSAK:
        sigset(SIGSAK, sigsak);
        break;
    case SIGDANGER:
        sigset(SIGDANGER, sigdanger1);
        break;
    case SIGSEGV:
        sigset(SIGSEGV, srvrsegv);
        break;
    default:
        sigset(signo, SIG_IGN);
        break;
    }
}
/* endloop */
get_acl_qhead();
lptr = (struct LQHEAD *) qhdu;
syslog(LOG_NOTICE, "N062: USRTBL:      sid = %x", lptr->prio1);
if (geteuid())
    syslog(LOG_ERR,
        "E855: hvx_lnk not run as root: DDCOR functions disabled!");
for (;;)
{
    irb = (struct IRB *) mqi_dqh(lnk_srvr);
    SRVR_RISC_ADDR(iorb, (struct RB *), irb->i_rb, irb);
    iorb = (struct RB *) ((char *) iorb - OFF_RB);
/* do io */
    lnk_io(iorb, irb);
}
/* for */
}
/* main */

/*****
 * lnk_handler * signal handler
 *****/
void
lnk_handler(int sigtype)
{
    switch (sigtype)
    {
        case SIGUSR2:
            lnk_cleanup();
            break;
    }
}
/* lnk_handler */

/*****
 * parse_options *
 *****/
int
parse_options(int argc, char **argv)
{
    unsigned int    lnk_queue;
    int             o;

```

32

```

extern char    *optarg;
char          cbuf[10];

opt.private = TRUE;
opt.rusage = FALSE;
while ((o = getopt(argc, argv, "PrVva:b:u:f:i:l:q:s:t:g:w:D:")) != -1)
{
    switch (o)
    {
        case 'D':
            sscanf(optarg, "%d", &plog);
            break;
        case 'a':
            sscanf(optarg, "%d", &msg_shmid);
            break;
        case 'b':
            sscanf(optarg, "%d", &msgaddr);
            break;
        case 'g':
            sscanf(optarg, "%s", &group_path);
            break;
        case 'P':
            opt.private = FALSE;
            break;
        case 'r':
            opt.rusage = TRUE;
            break;
        case 'V':
            hvx_verV(argv[0]);
            exit(0);
            break;
        case 'v':
            closelog();
            openlog(get_session_name(), LOG_PID, LOG_LOCAL0);
            setlogmask(LOG_MASK(LOG_ERR)
                | LOG_MASK(LOG_WARNING)
                | LOG_MASK(LOG_INFO)
                | LOG_MASK(LOG_NOTICE)
                | LOG_MASK(LOG_DEBUG));
            verbose = TRUE;
            break;
        case 's':
            sscanf(optarg, "%x", &opt.shmid);
            break;
        case 'f':
            sscanf(optarg, "%d", &fdkey);
            break;
        case 'l':
            sscanf(optarg, "%x", &lnk_queue);
            lrt = (struct LRT *) lnk_queue;
            break;
        case 'q':
            sscanf(optarg, "%x", &lnk_queue);
            lnk_srvr = (struct MQI *) lnk_queue;
    }
}

```

33

```

        break;
    case 'u':
        sscanf(optarg, "%x", &lnk_queue);
        ptr_dd_freeq = (struct LQHEAD *) lnk_queue;
        break;
    case 't':
        sscanf(optarg, "%x", &lnk_queue);
        lnk_table = (char *) lnk_queue;
        qhdt = (struct TTBL *) lnk_table;
        qhdf = (struct FTBL *) (lnk_table + (sizeof(struct LQHEAD)));
        break;
    case 'w':
        sscanf(optarg, "%d", &virt_view);
        break;
    default:
        break;
    }
    /* switch */
}
/* while */
if (argc == 1)
    return (1);
else
    return (0);
}
/* parse_options */

/*****
 * lnk_io *      Dual Decor server - execs all dynamic servers
 *****/
void
lnk_io(struct LRB * iorb, struct IRB * irb)
{
    struct RCT    *rct;
    /*
     * control server process waits with mqi_dqh for ES requests
     * and requests for assistance from the task servers (iorb function code =
     * original+FUNC_DELTA)
     */
    lrn = (iorb->lk_ct2 >> 8);
    if (lrn == 0xfd)
        lrn = iorb->neg.lk_lrx;
    RISC_ADDR(rct, (struct RCT *), (lrn->rt_rct + lrn * 2));
    rct = (struct RCT *) ((char *) rct - OFF_RCT);
    if (rct->qaddr.internal_id == LNK_ID)
    {
        switch (iorb->lk_type)
        {
        case LK_FILE:
            rct->qaddr.internal_id = LINKF_ID;
            break;
        case LK_PIPE:
            rct->qaddr.internal_id = LINKP_ID;
            break;
        case LK_ISAM:
            if (allowed("cisam"))
            {

```

34

```

        rct->qaddr.internal_id = LINKI_ID;
    }
    else
    {
        ret_ddcor_rct(rct);
        io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
        return;
    }
    break;
default:
    ret_ddcor_rct(rct);
    io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
    return;
    break;
}
/* switch */
}
func_code = iorb->lk_ct2 & 1_fc;
switch (rct->qaddr.internal_id)
{
case LINKP_ID:
    switch (func_code)
    {
        case CONN_FUNCTION:
        case CONN_FUNCTION + FUNC_DELTA:
        case DISC_FUNCTION:
        case DISC_FUNCTION + FUNC_DELTA:
        case BREAK_FUNCTION:
            break;
        default:
            ret_ddcor_rct(rct);
            io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
            return;
            break;
    }
    break;
case LINKI_ID:
case LINKF_ID:
    switch (func_code)
    {
        case CONN_FUNCTION:
        case CONN_FUNCTION + FUNC_DELTA:
        case DISC_FUNCTION:
        case DISC_FUNCTION + FUNC_DELTA:
            break;
        default:
            ret_ddcor_rct(rct);
            io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
            return;
            break;
    }
    break;
case CPX_ID:
    switch (func_code)
    {

```

35

```

    case CONN_FUNCTION:
    case CONN_FUNCTION + FUNC_DELTA:
    case DISC_FUNCTION:
    case DISC_FUNCTION + FUNC_DELTA:
        break;
    default:
        ret_ddcor_rct(rct);
        io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
        return;
        break;
    }
    break;
case NXSH_ID:
    switch (func_code)
    {
    case FC_SWITCH:
    case DISC_FUNCTION:
        break;
    default:
        ret_ddcor_rct(rct);
        io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
        return;
        break;
    }
    break;
/* bsc is not "dual decor" - there are no user checks - rcts are fixed */
case BSC_ID:
    switch (func_code)
    {
    case CONN_FUNCTION:
    case DISC_FUNCTION:
        func_code += BSC_DELTA;
        break;
    default:
        io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
        return;
        break;
    }
    break;
default:
    syslog(LOG_ERR, "E847: DDCOR Invalid internal_id=%d; lrn=%d",
        rct->qaddr.internal_id, lrn);
    if (rct->qaddr.internal_id != BSC_ID)
        ret_ddcor_rct(rct);
    io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
    return;
    break;
}
/* switch internal_id */
switch (func_code)
{
case CONN_FUNCTION:
    iorb->lk_iofail = iorb->lk_fsfail = 0;
    RISC_ADDR(tcb, (struct TCB *).irb->i_tcb);
    tcb = (struct TCB *) ((char *) tcb - OFF_TCB);

```

36

```

RISC_ADDR(ucb_ptr, (struct UCB *), tcb->t_ucbp);
memcpy(person, &ucb_ptr->u_pid, 12);
person[12] = '\0';
if (ptr = (strchr(person, ' ')))
    *ptr = '\0';
for (i = 0; i < strlen(person); i++)
{
    if (isupper(person[i]))
        person[i] = tolower(person[i]);
}
if ((geteuid()) || (!(get_user(irb, iorb, person))))
{
    if (geteuid())
    {
        syslog(LOG_ERR, "E856: hvx_lnk must run as root!");
    }
    iorb->lk_fsfail = ACCESS_CONTROL_VIOL;
    if (rct->qaddr.internal_id != CPX_ID)
        ret_ddcor_rct(rct);
    io_exit(rct, irb, SUCCESS_CODE);
    return;
}
if (((!iorb->lk_dvs & 1_hu)) || (rct->qaddr.internal_id == CPX_ID))
/* physical connect (lk_dvs bits different for CPX)*/
{
    if (rct->qaddr.internal_id == CPX_ID)
    {
        SRVR_RISC_ADDR(ptr, (char *), iorb->lk_adr, irb);
    }
    else
    {
        if (iorb->lk_prng > MAX_PATH_RANGE)
        {
            iorb->lk_fsfail = LONG_PATH_ERROR;
            ret_ddcor_rct(rct);
            io_exit(rct, irb, SUCCESS_CODE);
            break;
        }
        SRVR_RISC_ADDR(ptr, (char *), iorb->lk_pthp, irb);
    }
    if (ptr[0] != '/')
    {
        iorb->lk_fsfail = ILLEGAL_PATHNAME;
        if (rct->qaddr.internal_id != CPX_ID)
        {
            ret_ddcor_rct(rct);
        }
        io_exit(rct, irb, SUCCESS_CODE);
        break;
    }
}
else
{
    if (rct->qaddr.internal_id != CPX_ID)

```

37

```

        {
            ret_ddcor_rct(rct);
        }
        io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
        break;
    }
}

/*
 * for pipes & CPX each connect execs a task server
 * for files the connect execs a task server for each unique UCB
 */
SRVR_RISC_ADDR(ucb, (unsigned long), iorb->lk_addr, irb);
switch (rct->qaddr.internal_id)
{
case LINKI_ID:
case LINKF_ID: /* is the waiting task UCB in ltable? */
    lnkq_frm = lnk_findq(qhdt, ucb, &prev);
    tskadr = (struct TTBL *) lnkq_frm;
    if ((lnkq_frm == 0) || (tskadr->type != rct->qaddr.internal_id))
    {
case LINKP_ID:
        lnkq_frm = lnk_dqh(qhdt, AVAIL_Q);
        if (lnkq_frm == 0)
        {
            syslog(LOG_ERR, "E493: Logic error in lnk_io!");
            io_exit(rct, irb, UNAVL_CODE);
            break;
        }
        if (rct->qaddr.internal_id == LINKF_ID)
            priority = ucb;
        else
            priority = (unsigned long) rct;
        lnk_qoh(qhdt, lnkq_frm, priority);
        tskadr = (struct TTBL *) lnkq_frm;
        tskadr->type = rct->qaddr.internal_id;
        init_mqi(&tskadr->tch, 0); /* mqi address for task
                                   * server */
        tskadr->cnt = 0; /* connect count for this task server */
        exec_drvr(rct);
        tskadr = (struct TTBL *) lnkq_frm;
        if (pid < 0)
        {
            tskadr->tch.isem.pid = 0;
            rtn_ttbl(iorb, rct, irb);
            io_exit(rct, irb, UNAVL_CODE);
            break;
        }
        tskadr->tch.isem.pid = pid;
        syslog(LOG_DEBUG, "Created pid %d for LRN %d", pid, lrn);
    }
    syslog(LOG_INFO, "I543: LNK Connect to LRN %d", lrn);
}

/*
 * at this point either a new task server has been exec'ed or the TTBL entry
 * for an existing task has been located. In both cases the TTBL entry is in
 * lnkq_frm. Now get a FTBL entry for this connect.

```

38

```

*/
    tskadr = (struct TTBL *) lnkq_frm;
    rct->qaddr.pid = (int) tskadr;
    lnkq_frm = lnk_dqh(qhdf, AVAIL_Q);
    if (lnkq_frm == NULL)
    {
        syslog(LOG_ERR, "E506: Logic error in lnk_io!");
        io_exit(rct, irb, UNAVL_CODE);
        break;
    }

/*
* put FTBL element address (task id) in iorb for file system
* fill in FTBL fields for task server
* link FTBL entry to TTBL queue
* queue irb to task server
*/
    ftbl = (struct FTBL *) lnkq_frm;
    ftbl_head = (struct LQHEAD *) (&tskadr->flist.priol);
    lnk_qoh(ftbl_head, ftbl, (ulong) lnkq_frm);
    ftbl->tskadr = (ulong) tskadr;
    PUT4(iorb->lk_ftbl, (ulong) lnkq_frm);
    ftbl->cdvs = iorb->lk_dvs;
    ftbl->type = rct->qaddr.internal_id;
    GET4(ftbl->fcb, iorb->lk_fcb);
    ftbl->status = ftbl->current_rseek =
        ftbl->current_wseek = ftbl->current_rwseek = 0;
    SRVR_RISC_ADDR(ptr, (char *), iorb->lk_pthp, irb);
    memcpy(ftbl->pathname, ptr, iorb->lk_prng);
    ftbl->pathname[iorb->lk_prng] = '\0';
    mqi_qot(&tskadr->tch, irb);
    break;
case CPX_ID:
    init_mqi(&rct->qaddr, 0);          /* mqi address for task server */
    syslog(LOG_INFO, "I573: CPX Connect to LRN %d", lrn);
    exec_drvr(rct);
    rct->qaddr.isem.pid = pid;
    mqi_qot(&rct->qaddr, irb);
    break;
default:
    /* not LK_PIPE, LK_FILE */
    io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
    break;
}
/* lk type switch */
break;
case CONN_FUNCTION + FUNC_DELTA:
    /* error from child process */
    iorb->lk_ct2 -= FUNC_DELTA;
    rct->qaddr.pid = 0;
    rtn_ttbl(iorb, rct, irb);
    io_exit(rct, irb, iorb->lk_iofail);
    break;
case CONN_FUNCTION + BSC_DELTA:
    init_mqi(&rct->qaddr, 0); /* mqi address for task server */
    syslog(LOG_INFO, "I621: BSC Connect to LRN %d", lrn);
    exec_drvr(rct);
    rct->qaddr.isem.pid = pid;

```

39

```

        mqi_qot(&rct->qaddr, irb);
        break;
    case DISC_FUNCTION + FUNC_DELTA:
        iorb->lk_ct2 -= FUNC_DELTA;
    case DISC_FUNCTION:
    case DISC_FUNCTION + BSC_DELTA:
        rct->qaddr.pid = 0;
        switch (rct->qaddr.internal_id)
        {
            case LINKI_ID:
            case LINKF_ID:
            case LINKP_ID:
                if (iorb->lk_iofail == SUCCESS_CODE)
                    rtn_ttbl(iorb, rct, irb);
                break;
            case NXSH_ID:
                syslog(LOG_INFO, "I574: NXSH disconnect LRN %d", lrn);
                hvx_clnpla(&rct->qaddr.isem.pid, 0);
                ret_ddcor_rct(rct);
                break;
            case CPX_ID:
                syslog(LOG_INFO, "I575: CPX disconnect LRN %d", lrn);
                hvx_clnpla(&rct->qaddr.isem.pid, 0);
                ret_ddcor_rct(rct);
                break;
            case BSC_ID:
                syslog(LOG_INFO, "I622: BSC disconnect LRN %d", lrn);
                hvx_clnpla(&rct->qaddr.isem.pid, 0);
                break;
        }
        io_exit(rct, irb, iorb->lk_iofail);
        break;
    case BREAK_FUNCTION:
        iorb->lk_iofail = iorb->lk_fsfail = 0;
        GET4(math.adrs, iorb->lk_ftbl);
        lnkq_frm = (struct LQHEAD *) math.ptr;
        ftbl = (struct FTBL *) lnkq_frm;
        GET4(math.adrs, iorb->lk_ftbl);
        if (lnkq_frm->prio1 != math.adrs)
        {
            io_exit(rct, irb, ILLEGAL_FUNCTION_CODE); /* illegal task id */
            break;
        }
        kill(tskadr->tch.isem.pid, SIGTERM);
        io_exit(rct, irb, iorb->lk_iofail);
        break;
    case FC_SWITCH:
        RISC_ADDR(tcb, (struct TCB *), irb->i_tcb);
        tcb = (struct TCB *) ((char *) tcb - OFF_TCB);
        RISC_ADDR(ucb_ptr, (struct UCB *), tcb->t_ucbp);
        memcpy(person, &ucb_ptr->u_pid, 12);
        person[12] = '\0';
        if (ptr = (strchr(person, ' ')))
            *ptr = '\0';

```

40

```

    for (i = 0; i < strlen(person); i++)
    {
        if (isupper(person[i]))
            person[i] = tolower(person[i]);
    }
    if (!(get_user(irb, iorb, person)))
    {
        iorb->lk_fsfail = ACCESS_CONTROL_VIOL;
        io_exit(rct, irb, SUCCESS_CODE);
        return;
    }
    init_mqi(&rct->qaddr, 0); /* mqi address for task server */
    syslog(LOG_INFO, "I576: NXSH Connect to LRN %d", lrn);
    exec_drvr(rct);
    rct->qaddr.isem.pid = pid;
    mqi_qot(&rct->qaddr, irb);
    break;
default:
    io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
    break;
}
/* func code switch */
return;
/* lnk_io */
}

/*****
 * rtn_ttbl *
 *****/
/*
 * return an ftbl entry (sent by a child) to the available queue for FILE/PIPE.
 * if the last file for this child return the ttbl entry as well & kill child.
 */
void
rtn_ttbl(struct LRB * iorb, struct RCT * rct, struct IRB * irb)
{
    GET4(math.adrs, iorb->lk_ftbl);
    ftbl = (struct FTBL *) math.ptr;
    GET4(math.adrs, iorb->lk_ftbl);
    lnkq_frm = (struct LQHEAD *) math.ptr;
    tskadr = (struct TTBL *) ftbl->tskadr;
    ftbl_head = (struct LQHEAD *) (&tskadr->flist.pri1);
    ftbl_head = lnk_dqh(ftbl_head, (ulong) lnkq_frm);
    if (ftbl_head == NULL)
    {
        syslog(LOG_ERR, "E591: Logic error in lnk_io!");
        exit(1);
    }
    iorb->lk_ftbl[0] = 0;
    iorb->lk_ftbl[0] = 0;
    ftbl->fd.fdes = 0;
    ftbl->tskadr = 0;
    lnk_qoh(qhdf, lnkq_frm, AVAIL_Q); /* return to available queue */
    syslog(LOG_INFO, "I546: LNK disconnect LRN %d", lrn);
    ret_ddcor_rct(rct);
    if (!(tskadr->cnt))

```

41

```

    {
        hvx_clnpla(&tskadr->tch.isem.pid, 0);
        tskadr->type = 0;
        lnkq_frm = lnk_dqh(qhdt, tskadr->iq.priol);
        lnk_qoh(qhdt, lnkq_frm, AVAIL_Q);
    }
    return;
}
/* rtn_ttbl */

```

```

/*****

```

```

* exec_drvr *
*****/

```

```

void

```

```

exec_drvr(struct RCT * rct)

```

```

{
    int      i, j = 1;
    char      *newargv[100];
    char      shmid[14];
    char      aplog[14];
    char      emain_pid[14];
    char      rct_addr[14];
    char      queue_addr[14];
    char      lrt_addr[14];
    char      task_addr[14];
    char      virv[14];
    char      private[4];
    int      emupid, oldmask;
    static char  myname[MAXPATHLEN + 15];
    char      grp_path[MAXPATHLEN + 1];
    char      usr_name[20];

```

```

    if (!rct)

```

```

    {
        lnk_cleanup();
        exit(1);
    }

```

```

    switch (rct->qaddr.internal_id)

```

```

    {
        case LINKP_ID:
            ptr = _PLNK;
            build_path(S_PLNK, tree);
            break;
        case LINKF_ID:
            ptr = _FLNK;
            build_path(S_FLNK, tree);
            break;
        case LINKI_ID:
            ptr = _ILNK;
            build_path(S_ILNK, tree);
            break;
        case CPX_ID:
            ptr = _CPX;
            build_path(S_CPX, tree);
            break;
    }

```

42

```

case NXSH_ID:
    ptr = _XSH;
    build_path(S_XSH, tree);
    break;
case BSC_ID:
    ptr = _BSC;
    build_path(S_BSC, tree);
    break;
default:
    break;
}
newargv[0] = myname;
if (verbose)
    newargv[j++] = "-v";
sprintf(shmid, "-s%d", opt.shmid);
newargv[j++] = shmid;
sprintf(aplog, "-D%d", plog);
newargv[j++] = aplog;
sprintf(virv, "-w%d", virt_view);
newargv[j++] = virv;
if (rct->qaddr.internal_id == NXSH_ID)
{
    sprintf(emain_pid, "-p%x", mainpid);
    newargv[j++] = emain_pid;
}
if ((rct->qaddr.internal_id == LINKF_ID)
    || (rct->qaddr.internal_id == LINKI_ID)
    || (rct->qaddr.internal_id == NXSH_ID))
{
    sprintf(lrt_addr, "-l%x", lrt);
    newargv[j++] = lrt_addr;
}
if (rct->qaddr.internal_id != LINKF_ID)
{
    sprintf(rct_addr, "-R%x", rct);    /* LNK/CPX/XSH/BSC rct address */
    newargv[j++] = rct_addr;
}
sprintf(queue_addr, "-q%x", lnk_srvt); /* LNK server MQI address */
newargv[j++] = queue_addr;
if ((rct->qaddr.internal_id == LINKP_ID)
    || (rct->qaddr.internal_id == LINKI_ID)
    || (rct->qaddr.internal_id == LINKF_ID))
{
    sprintf(task_addr, "-t%x", tsaddr); /* LNK MQI address */
    newargv[j++] = task_addr;
}
if (!opt.private)
{
    sprintf(private, "-P");
    newargv[j++] = private;
}
if (rct->qaddr.internal_id != BSC_ID)
{
    sprintf(usr_name, "-u%s", person);

```

43

```

        newargv[j++] = usr_name;
    }
    newargv[j] = NULL;
    if ((pid = fork()) == 0)
    {
        setuid(getuid());
        close(0);
        close(1);
        close(2);
        open("/dev/null", O_RDWR);
        open("/dev/null", O_RDWR);
        open("/dev/null", O_RDWR);
        if (opt.private)
        {
            execvp(ptr, newargv);
            syslog(LOG_ERR, "E742: Exec of DDecor child failed %m %s", ptr);
        }
        else
        {
            execvp(tree, newargv);
            syslog(LOG_ERR, "E848: Exec of DDecor child failed %m %s", tree);
        }
        exit(1);
    }
    /* child process */
else
if (pid <= 0)
{
    syslog(LOG_ERR, "E743: fork: %m");
}
}

/*****
* lnk_cleanup *
*****/
void
lnk_cleanup()
{
    int      j;
    struct TTBL  *ttbl;

    ttbl = (struct TTBL *) (lnk_table + (2 * sizeof(struct LQHEAD)));
    for (j = 0; j < fdkey; j++, ttbl++)
    {
        hvx_clnp1a(&ttbl->tch.isem.pid, 0);
    }
    exit(0);
}
/* lnk_cleanup */

```

44

```

*****
  Linked Files Server Code
*****

int
parse_options(int, char **, struct EMU_OPTS *);
int
rtn_ftbl(struct IRB *);
void
lfs_file(struct LRB *, struct IRB *);

struct sigaction lnk_actions;
sigset_t lnk_sigs;
extern void srvrsegv();
extern void
sigsak(), sigdanger1();

extern int Uid;
char *username;
char person[14];
int mainpid, signo;
char *svr_base();
struct TTBL *tskadr;
struct FTBL *ftbl;

static union MATH math;
static char func_code;
static int bytes_tran, xfer_cnt;
static char *newlineptr, *ptr;
static int string_length;
static int status, i, pid;
static struct stat file_status;
FILE *ipipe_stream, *opipe_stream; /* dummies for linker */
struct MQI *lnk_svr;
struct LRT *lrt;

/*****
 * main *
*****/
void
main(argc, argv, envp)
int argc;
char *argv[];
char *envp[];
{
    struct passwd *pwent;
    struct IRB *irb;
    struct RB *iorb;

    process = SERVER;

    pwent = getpwuid(getuid());
    if (pwent == NULL)
    {
        fprintf(stderr, "Unable to determine user name\n");
    }
}

```

45

```

    }
    username = strdup(pwent->pw_name);
    openlog(username, LOG_PID, LOG_LOCAL0);
    setlogmask(LOG_MASK(LOG_ERR)
               | LOG_MASK(LOG_WARNING)
               | LOG_MASK(LOG_NOTICE));
    if (parse_options(argc, argv, &opt) != 0)
    {
        syslog(LOG_ERR, "E857: LINK: Invalid command line options");
    }
    close_fd();

    mainpid = getppid();
    if ((base = shmat(opt.shmid, (char *) NULL, 0))
        == (char *) -1)
    {
        syslog(LOG_ERR, "E858: LINK: shmat: %m");
        exit(1);
    }
    init_vm();
    if (intrp_vm())
    {
        exit(1);
    }
    svr_vm();

    lnk_actions.sa_mask = lnk_sigs;
    lnk_actions.sa_flags = 0;

    for (signo = 0; signo < NSIG; signo++)
    {
        switch (signo)
        {
            case SIGUSR2:
                sigaction(signo, &lnk_actions, NULL);
                break;

            case SIGSAK:
                sigset(SIGSAK, sigsak);
                break;

            case SIGDANGER:
                sigset(SIGDANGER, sigdanger1);
                break;

            case SIGSEGV:
                sigset(SIGSEGV, svrsegv);
                break;

            default:
                sigset(signo, SIG_IGN);
                break;
        }
    }
    /* endloop */

```

46

```

    lfs_file((struct LRB *) iorb, irb);
}                                     /* main */

/*****
 * parse_options *
 *****/
int
parse_options(int argc, char **argv, struct EMU_OPTS * opt)
{
    unsigned int    lnk_queue;
    int             o;
    extern char     *optarg;
    char            cbuff[10];
    opt->private = TRUE;
    opt->rusage = FALSE;

    while ((o = getopt(argc, argv, "PrVvw:l:s:t:R:q:u:")) != -1)
    {
        switch (o)
        {
            case 'P':
                opt->private = FALSE;
                break;

            case 'l':
                sscanf(optarg, "%x", &lnk_queue);
                lrt = (struct LRT *) lnk_queue;
                break;

            case 'V':
                hvx_verV(argv[0]);
                exit(0);
                break;

            case 'v':
                closelog();
                openlog(username, LOG_PID, LOG_LOCAL0);
                setlogmask(LOG_MASK(LOG_ERR)
                    | LOG_MASK(LOG_WARNING)
                    | LOG_MASK(LOG_INFO)
                    | LOG_MASK(LOG_NOTICE)
                    | LOG_MASK(LOG_DEBUG));
                verbose = TRUE;
                break;

            case 's':
                sscanf(optarg, "%d", &opt->shmid);
                break;

            case 'q':
                sscanf(optarg, "%x", &lnk_queue);
                lnk_srvr = (struct MQI *) lnk_queue;

```

47

```

        break;

    case 't':
        sscanf(optarg, "%x", &lnk_queue);
        tskadr = (struct TTBL *) lnk_queue;
        break;

    case 'u':
        sscanf(optarg, "%s", &person);
        break;

    case 'w':
        sscanf(optarg, "%d", &virt_view);
        break;

    default:
        break;
    }
    /* switch */
}
/* while */
if (argc == 1)
    return (1);
else
    return (0);
}
/* parse_options */

/*****
 * rtn_fibl *
 *****/
/*
 * clean up an fibl entry & queue the IRB to the server.
 */
int
rtn_fibl(struct IRB * irb)
{
    tskadr->cnt--;
    mqj_qot(lnk_srvr, irb);
    return tskadr->cnt;
}
/* rtn_fibl */

/*****
 * lk_killed *
 *****/
void
lk_killed()
{
    fibl->status |= Lk_SIG;
}
/* lk_killed */

/*****
 * lk_break *
 *****/
void
lk_break()
{

```

48

```

    ftbl->status |= Lk_BREAK;
}                                     /* lk_break */

/*****
 * lfs_file *
 *****/
void
lfs_file(struct LRB * iorb, struct IRB * irb)
{
    /*
     * task driver process for files
     *  waits for irb then sets up iorb and FTBL entry
     */
    int      access, access_flag, oflag, seek_stat, dupl, sav_stat;
    char      tempbuff[MAX_BLOCK_RANGE + 1];
    int      disc_detected = FALSE;
    struct FTBL *f_entry, *f_entry1;
    struct RCT *rct;
    int      lrn;

    while (!disc_detected)
    {
        irb = (struct IRB *) mqi_dqh(&tskadr->tch);
        SRVR_RISC_ADDR(iorb, (struct LRB *), irb->i_rb, irb);
        iorb = (struct LRB *) ((char *) iorb - OFF_LRB);
        GET4(math.adrs, iorb->lk_ftbl);
        ftbl = (struct FTBL *) math.ptr;
        func_code = iorb->lk_ct2 & 1_fc;
        iorb->lk_iofail = iorb->lk_fsfail = 0;
        /*
         syslog(LOG_DEBUG, "TS function = %x", func_code);*/
        lrn = (iorb->lk_ct2 >> 8);
        if (lrn == 0xfd)
            lrn = iorb->neg.lk_lrx;
        rct = (struct RCT *) sys_r0;
        if (lrn <= MAX_LRN)
            RISC_ADDR(rct, (struct RCT *), (lrn->rt_rct + lrn * 2));
        rct = (struct RCT *) ((char *) rct - OFF_RCT);

        switch (func_code)
        {
            case CONN_FUNCTION:
                get_acl_qhead();
                if (!(iorb->lk_dvs & 1_hu))    /* physical connect */
                {
                    if (ftbl->status & P_CONN)    /* reconnect */
                    /*
                     * file previously physically connected - reconnect done typically to change
                     * read/write access.  if new request settings don't fly restore old ones.
                     */
                    {
                        sav_stat = ftbl->status;
                        ftbl->status = 0;
                        do
                            seek_stat = stat(ftbl->pathname, &file_status);

```

49

```

while ((seek_stat < 0) && (errno == EINTR));
if (seek_stat == -1)
{
    if (errno != ENOENT)
    {
        syslog(LOG_ERR, "E592: LINK %s error on stat function: %m",
            fbl->pathname);
        iorb->lk_fsfail = UNAVL_CODE;
        fbl->status = sav_stat;
        io_exit(rct, irb, SUCCESS_CODE);
        break;
    }
    else
    {
        iorb->lk_fsfail = FILE_NOT_FOUND;
        fbl->status = sav_stat;
        io_exit(rct, irb, SUCCESS_CODE);
        break;
    }
}
if (file_status.st_type != VREG)
{
    iorb->lk_fsfail = ILLEGAL_FILE_TYPE;
    fbl->status = sav_stat;
    io_exit(rct, irb, SUCCESS_CODE);
    break;
}
access = get_access(&fbl->pathname, person, LK_PIPE);
if ((access == -1) || (!(access & EXEC_ACCL)))
{
    iorb->lk_fsfail = ACCESS_CONTROL_VIOL;
    fbl->status = sav_stat;
    io_exit(rct, irb, SUCCESS_CODE);
    break;
}
io_exit(rct, irb, SUCCESS_CODE);
break;
}
/* reconnect */
tskadr->cnt++;

access = get_access(&fbl->pathname, person, LK_PIPE);
if (geteuid())
{
    syslog(LOG_ERR, "E859: hvx_file must run as root!");
    access = -1;
}
do
    seek_stat = stat(fbl->pathname, &file_status);
while ((seek_stat < 0) && (errno == EINTR));
if ((seek_stat == -1) && (access != -1))
{
    if (errno != ENOENT)
    {
        syslog(LOG_ERR, "E574: LINK %s error on stat function: %m",

```

50

```

        ftbl->pathname);
iorb->lk_fsfail = UNAVL_CODE;
iorb->lk_ct2 += FUNC_DELTA;
rtn_ftbl(irb);
break;
    }
else
    {
/* file does not exist - if directory permissions are OK create it */
        if ((access & (EXEC_ACCL | WRITE_ACCL)) ==
            (EXEC_ACCL | WRITE_ACCL))
        {
            errno = 0;
            i = umask(0);
            do
                ftbl->fd.fdes = mknod(ftbl->pathname,
                                      (S_IFREG
                                       | S_IRUSR | S_IWUSR
                                       | S_IRGRP | S_IWGRP), 0);
            while ((ftbl->fd.fdes < 0) && (errno == EINTR));
            umask(i);
            if ((ftbl->fd.fdes < 0) && (errno != EEXIST))
            {
                syslog(LOG_ERR, "E849: : LINKF %s mknod failed: %m",
                      ftbl->pathname);
                iorb->lk_fsfail = UNAVL_CODE;
                iorb->lk_ct2 += FUNC_DELTA;
                rtn_ftbl(irb);
                break;
            }
            if (!(errno == EEXIST))
            {
                do
                    i = chown(ftbl->pathname, Uid, -1);
                while ((i < 0) && (errno == EINTR));
                if (i < 0)
                {
                    unlink(ftbl->pathname);
                    syslog(LOG_ERR, "E850: : LINKF %s chown failed: %m",
                          ftbl->pathname);
                    iorb->lk_fsfail = UNAVL_CODE;
                    iorb->lk_ct2 += FUNC_DELTA;
                    rtn_ftbl(irb);
                    break;
                }
            }
            ftbl->status |= P_CONN;
            io_exit(rct, irb, SUCCESS_CODE);
            break;
        }
        /* create file access */
    }
else
    {
        iorb->lk_fsfail = ACCESS_CONTROL_VIOL;
        iorb->lk_ct2 += FUNC_DELTA;
    }

```

51

```

        rtn_fibl(irb);
        break;
    }
}
}
if ((access == -1) || (!(access & EXEC_ACCL)))
{
    iorb->lk_fsfail = ACCESS_CONTROL_VIOL;
    iorb->lk_ct2 += FUNC_DELTA;
    rtn_fibl(irb);
    break;
}
if (file_status.st_type != VREG)
{
    iorb->lk_fsfail = ILLEGAL_FILE_TYPE;
    iorb->lk_ct2 += FUNC_DELTA;
    rtn_fibl(irb);
    break;
}
fibl->status |= P_CONN;
io_exit(rct, irb, SUCCESS_CODE);
break;
}
/* physical connect */

if (iorb->lk_control & l_dirr)
    fibl->status |= Lk_READ;
if (iorb->lk_control & l_dirw)
    fibl->status |= Lk_WRITE;
fibl->status |= Lk_CONN;
access_flag = oflag = 0;
if (iorb->lk_control & (l_dirr || l_dirw))
{
    if (iorb->lk_control & l_dirr)
    {
        if (iorb->lk_control & l_dirw)
        {
            access_flag = READ_ACCL + WRITE_ACCL;
            oflag |= O_RDWR;
        }
        else
        {
            access_flag = READ_ACCL;
            oflag |= O_RDONLY;
        }
    }
    else
    if (iorb->lk_control & l_dirw)
    {
        access_flag = WRITE_ACCL;
        oflag |= O_WRONLY;
    }
}
}
else
{

```

52

```

if (iorb->lk_control & l_cr)
{
    if (iorb->lk_control & l_cw)
    {
        access_flag = READ_ACCL + WRITE_ACCL;
        oflag |= O_RDWR;
    }
    else
    {
        access_flag = READ_ACCL;
        oflag |= O_RDONLY;
    }
}
else
if (iorb->lk_control & l_cw)
{
    access_flag = WRITE_ACCL;
    oflag |= O_WRONLY;
}
}

if (iorb->lk_control & l_cri)
    oflag |= O_NSHARE;
else
if (iorb->lk_control & l_cwi)
    oflag |= O_RSHARE;
if (iorb->lk_control & l_renew)
    oflag |= O_TRUNC;
if (iorb->lk_control & l_appnd)
{
    oflag |= O_APPEND;
    ftbl->status |= Lk_APPND;
}
dupl = 0;
if (tskadr->cnt > 1)
{
    /* check for duplicate open to same file */
    f_entry = (struct FTBL *) tskadr->flist.fwd;
    f_entry1 = (struct FTBL *) & tskadr->flist.priol;
    while (f_entry != f_entry1)
    {
        if (f_entry == ftbl)
        {
            f_entry = (struct FTBL *) f_entry->fq.fwd;
            continue;
        }
        if (strcmp(ftbl->pathname, (f_entry->pathname)) == 0)
        {
            if (f_entry->status & Lk_OPEN)
            {
                access = get_access(&ftbl->pathname,
                                    person, LK_FILE);
                if ((access == -1)
                    || ((access & access_flag) != access_flag))

```

53

```

        {
            iorb->lk_fsfail = ACCESS_CONTROL_VIOL;
            dupl = -1;
            break;
        }
        do
            ftbl->fd.fdes = dup(f_entry->fd.fdes);
            while ((ftbl->fd.fdes < 0) && (errno == EINTR));
            dupl = 1;
            break;
        }
        f_entry = (struct FTBL *) f_entry->fq.fwd;
    }
    /* duplicate */
    if (dupl == -1)
    {
        io_exit(rct, irb, SUCCESS_CODE);
        break;
    }
    if (!dupl)
    {
        access = get_access(&ftbl->pathname, person, LK_FILE);
        if ((access == -1) || ((access & access_flag)
                               != access_flag))
        {
            iorb->lk_fsfail = ACCESS_CONTROL_VIOL;
            ftbl->status &= ~Lk_CONN;
            io_exit(rct, irb, SUCCESS_CODE);
            break;
        }
        do
            ftbl->fd.fdes = open(ftbl->pathname, oflag);
            while ((ftbl->fd.fdes < 0) && (errno == EINTR));
    }
    if (ftbl->fd.fdes < 0)
    {
        if (errno == ETXTBSY)
            iorb->lk_fsfail = FILE_ALREADY_OPEN;
        else
            iorb->lk_fsfail = UNAVL_CODE;
        syslog(LOG_ERR,
            "E497: LINK %s open failed: %m", ftbl->pathname);
        ftbl->status &= ~Lk_CONN;
        io_exit(rct, irb, SUCCESS_CODE);
        break;
    }
    else
    {
        do
            seek_stat = lseek(ftbl->fd.fdes, 0, SEEK_END);
            while ((seek_stat < 0) && (errno == EINTR));
            ftbl->current_wseek = seek_stat;
        }
    }
}

```

54

```

    fbl->status |= Lk_OPEN;
    io_exit(rct, irb, SUCCESS_CODE);
    break;
case READ_FUNCTION:
    SRVR_RISC_ADDR(ptr, (char *), iorb->lk_adr, irb);
    if (iorb->lk_ct2 & 1_b) /* buffer starts in right byte */
        ptr++;
    if (iorb->lk_control & 1_seek)
    {
        GET4(fbl->current_rseek, iorb->lk_seek);
    }
    else
    {
        PUT4(iorb->lk_seek, fbl->current_rseek);
    }
    fbl->current_rwseek = fbl->current_rseek;
    do
        seek_stat = lseek(fbl->fd.fdes,
                           fbl->current_rseek, SEEK_SET);
    while ((seek_stat < fbl->current_rseek) && (errno == EINTR));
    if (seek_stat != fbl->current_rseek)
    {
        iorb->lk_fsfail = DATA_OUTSIDE_FILE;
        io_exit(rct, irb, SUCCESS_CODE);
        break;
    }
    if (iorb->lk_control & 1_mod)
    {
        if (iorb->lk_rng > MAX_BLOCK_RANGE)
        {
            iorb->lk_fsfail = RECORD_LENGTH_ERROR;
            io_exit(rct, irb, SUCCESS_CODE);
            break;
        }
        if (iorb->lk_rng == 0)
        {
            /* consider a zero range to be a seek only */
            iorb->lk_rsr = 0;
            io_exit(rct, irb, SUCCESS_CODE);
            break;
        }
        do
            bytes_tran = read(fbl->fd.fdes, tempbuff, iorb->lk_rng);
        while ((bytes_tran < 0) && (errno == EINTR));
        if (bytes_tran == -1)
        {
            syslog(LOG_ERR, "E575: LINK %s read failed: %m",
                   fbl->pathname);
            iorb->lk_rsr = iorb->lk_rng;
            io_exit(rct, irb, FAILURE_CODE);
            break;
        }
        string_length = getlin(ptr, bytes_tran, tempbuff);
        if (string_length == 0)

```

55

```

{
    iorb->lk_fsfail = END_OF_FILE;
    iorb->lk_rsr = iorb->lk_rng;
    io_exit(rct, irb, SUCCESS_CODE);
    break;
}
fbl->current_rseek += string_length;
if (ptr[string_length - 1] == '\n')
    string_length--;
if (iorb->lk_control & l_curnc)
    fbl->current_wseek = fbl->current_rseek;
iorb->lk_rsr = iorb->lk_rng - string_length;
io_exit(rct, irb, SUCCESS_CODE);
break;
}
/* text transfer */
else
{
    do
        bytes_tran = read(fbl->fd.fdes, ptr, iorb->lk_rng);
    while ((bytes_tran < 0) && (errno == EINTR));
    if (bytes_tran == -1)
    {
        syslog(LOG_ERR, "E576: LINK %s read failed: %m",
            fbl->pathname);
        iorb->lk_rsr = iorb->lk_rng;
        iorb->lk_fsfail = END_OF_FILE;
        io_exit(rct, irb, SUCCESS_CODE);
        break;
    }
    if (bytes_tran == 0)
    {
        iorb->lk_rsr = iorb->lk_rng;
        iorb->lk_fsfail = END_OF_FILE;
        io_exit(rct, irb, SUCCESS_CODE);
        break;
    }
    if (bytes_tran != iorb->lk_rng)
    {
        iorb->lk_rsr = iorb->lk_rng - bytes_tran;
        fbl->current_rseek += bytes_tran;
        if (iorb->lk_control & l_curnc)
            fbl->current_wseek = fbl->current_rseek;
        io_exit(rct, irb, SUCCESS_CODE);
        break;
    }
    iorb->lk_rsr = 0;
    fbl->current_rseek += bytes_tran;
    if (iorb->lk_control & l_curnc)
        fbl->current_wseek = fbl->current_rseek;
    io_exit(rct, irb, SUCCESS_CODE);
    break;
}
/* binary transfer */
break;

```

56

```

case WRITE_FUNCTION:
    SRVR_RISC_ADDR(ptr, (char *), iorb->lk_adr, irb);
    if (iorb->lk_ct2 & 1_b)/* buffer starts in right byte */
        ptr++;
    if (fbl->status & Lk_APPND)
        if (iorb->lk_control & 1_seek)
        {
            io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
            break;
        }
        else
        {
            do
                seek_stat = lseek(fbl->fd.fdes,
                                fbl->current_wseek, SEEK_END);
            while ((seek_stat < fbl->current_wseek) &&
                    (errno == EINTR));
            PUT4(iorb->lk_seek, fbl->current_wseek);
        }
    else
    {
        if (iorb->lk_control & 1_seek)
        {
            GET4(fbl->current_rseek, iorb->lk_seek);
        }
        else
        {
            PUT4(iorb->lk_seek, fbl->current_rseek);
        }
        do
            seek_stat = lseek(fbl->fd.fdes,
                            fbl->current_wseek, SEEK_SET);
        while ((seek_stat < fbl->current_wseek) && (errno == EINTR));
        if (seek_stat != fbl->current_wseek)
        {
            iorb->lk_fsfail = DATA_OUTSIDE_FILE;
            io_exit(rct, irb, SUCCESS_CODE);
            break;
        }
    }
    do
        bytes_tran = write(fbl->fd.fdes, ptr, iorb->lk_rng);
    while ((bytes_tran < 0) && (errno == EINTR));
    if (bytes_tran != iorb->lk_rng)
    {
        if (bytes_tran == -1)
            syslog(LOG_ERR, "E577: LINK %s write failed: %m",
                fbl->pathname);
        else
            syslog(LOG_ERR, "E578: LINK %s write failed",
                fbl->pathname);
        iorb->lk_rsr = iorb->lk_rng - bytes_tran;
        io_exit(rct, irb, FAILURE_CODE);
        break;
    }

```

57

```

    }
    /* write failure */
    ftbl->current_wseek += iorb->lk_rng;
    if (iorb->lk_control & l_mod)
    {
        do
        {
            bytes_tran = write(ftbl->fd.fdes, "\n", 1);
            while ((bytes_tran < 0) && (errno == EINTR));
            if (bytes_tran != 1)
            {
                if (bytes_tran == -1)
                    syslog(LOG_ERR, "E579: LINK %s write failed: %m",
                        ftbl->pathname);
                else
                    syslog(LOG_ERR, "E580: LINK %s write failed",
                        ftbl->pathname);
                iorb->lk_rsr = 0;
                io_exit(rct, irb, FAILURE_CODE);
                break;
            }
            ftbl->current_wseek++;
        }
        /* text transfer */

    if (iorb->lk_control & l_curnc)
        ftbl->current_rseek = ftbl->current_wseek;
    iorb->lk_st = 0;
    iorb->lk_rsr = 0;
    io_exit(rct, irb, SUCCESS_CODE);
    break;

case REWRITE_FUNCTION:
    if (ftbl->status & Lk_APPND)
    {
        io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
        break;
    }
    SRVR_RISC_ADDR(ptr, (char *), iorb->lk_adr, irb);
    if (iorb->lk_ct2 & l_b) /* buffer starts in right byte */
        ptr++;
    if (iorb->lk_control & l_seek)
        GET4(ftbl->current_rwseek, iorb->lk_seek);

    do
    {
        seek_stat = lseek(ftbl->fd.fdes,
            ftbl->current_rwseek, SEEK_SET);
        while ((seek_stat < ftbl->current_rwseek) && (errno == EINTR));
        if (seek_stat != ftbl->current_rwseek)
        {
            iorb->lk_fsfail = DATA_OUTSIDE_FILE;
            io_exit(rct, irb, SUCCESS_CODE);
            break;
        }
    }
    do
        bytes_tran = write(ftbl->fd.fdes, ptr, iorb->lk_rng);
    while ((bytes_tran < 0) && (errno == EINTR));

```

58

```

if (bytes_tran != iorb->lk_rng)
{
    if (bytes_tran == -1)
        syslog(LOG_ERR, "E581: LINK %s write failed: %m",
            ftbl->pathname);
    else
        syslog(LOG_ERR, "E582: LINK %s write failed",
            ftbl->pathname);
    iorb->lk_rsr = iorb->lk_rng - bytes_tran;
    io_exit(rct, irb, FAILURE_CODE);
    break;
}
/* write failure */
if (iorb->lk_control & l_mod)
{
    do
        bytes_tran = write(ftbl->fd.fdes, "\n", 1);
    while ((bytes_tran < 0) && (errno == EINTR));
    if (bytes_tran != 1)
    {
        if (bytes_tran == -1)
            syslog(LOG_ERR, "E583: LINK %s write failed: %m",
                ftbl->pathname);
        else
            syslog(LOG_ERR, "E584: LINK %s write failed",
                ftbl->pathname);
        iorb->lk_rsr = 0;
        io_exit(rct, irb, FAILURE_CODE);
        break;
    }
}
/* text transfer */

iorb->lk_st = 0;
iorb->lk_rsr = 0;
io_exit(rct, irb, SUCCESS_CODE);
break;

case DISC_FUNCTION:
    if (iorb->lk_dvs & l_hu)
    {
        if (!(ftbl->status & Lk_CONN))
        {
            io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
            break;
            /* already logically disconnected */
        }
        ftbl->status &= ~(Lk_CONN | Lk_APPND);
        ftbl->current_rseek = ftbl->current_wseek = 0;
        if (ftbl->status & Lk_OPEN)
        {
            do
                status = close(ftbl->fd.fdes);
            while ((status < 0) && (errno == EINTR));
            if (status == -1)
            {
                io_exit(rct, irb, FAILURE_CODE);
            }
        }
    }
}

```

59

```

        syslog(LOG_ERR, "E585: LINK close failed: %m");
        break;
    }
    else
    {
        ftbl->fd.fdes == -1;
        ftbl->status &= ~Lk_OPEN;
    }
}
io_exit(rct, irb, SUCCESS_CODE);
break;
}
/* logical disconnect */
iorb->lk_ct2 += FUNC_DELTA;
ftbl->status = 0;
if (rtn_ftbl(irb) == 0)
    disc_detected = TRUE;
break;

default:
    io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
    break;
}
/* function code switch */
/* for */
}
/* sleep forever, the dual decor server will kill us in a moment */
for (;;)
{
    sleep(1000);
}
/* lfs_file */
}

```

60

```

*****
Linked Pipes Server Code
*****

int
parse_options(int, char **, struct EMU_OPTS *);
int
rtn_fibl(struct IRB *);
void
lfs_pipe(struct RCT *, struct LRB *, struct IRB *);

struct RCT      *rrct;
struct sigaction lnk_actions;
sigset_t        lnk_sigs;
extern void      srvrsegv();
extern void
sigsak(), sigdanger1();

extern int      Uid;
char            *username;
char            person[14];
int             mainpid, signo;
char            *srvr_base();
struct TTBL     *tskadr;
struct FTBL     *fibl;

static char      func_code;
static int       bytes_tran, xfer_cnt;
static char      *newlineptr, *ptr;
static int       string_length;
static int       status, i, pid;
static struct stat file_status;
static union MATH math;
FILE             *ipipe_stream, *opipe_stream;      /* dummies for linker */
struct MQI       *lnk_srvr;

/*****
* main *
*****/
void
main(argc, argv, envp)
int      argc;
char     *argv[];
char     *envp[];
{
    struct passwd *pwent;
    struct IRB    *irb;
    struct RB     *iorb;

    process = SERVER;
    pwent = getpwuid(getuid());
    if (pwent == NULL)
    {
        fprintf(stderr, "Unable to determine user name\n");
        exit(1);
    }
}

```

61

```

    }
    username = strdup(pwent->pw_name);
    openlog(username, LOG_PID, LOG_LOCAL0);
    setlogmask(LOG_MASK(LOG_ERR)
               | LOG_MASK(LOG_WARNING)
               | LOG_MASK(LOG_NOTICE));
    if (parse_options(argc, argv, &opt) != 0)
    {
        syslog(LOG_ERR, "E942: LINK: Invalid command line options");
        exit(1);
    }
    close_fd();
    mainpid = getppid();
    if ((base = shmat(opt.shmid, (char *) NULL, 0))
        == (char *) -1)
    {
        syslog(LOG_ERR, "E943: LINK: shmat: %m");
        exit(1);
    }
    init_vm();
    if (intrp_vm())
    {
        exit(1);
    }
    svr_vm();
    lnk_actions.sa_mask = lnk_sigs;
    lnk_actions.sa_flags = 0;
    for (signo = 0; signo < NSIG; signo++)
    {
        switch (signo)
        {
            case SIGUSR2:
                sigaction(signo, &lnk_actions, NULL);
                break;
            case SIGCHLD:
            case SIGABRT:
            case SIGTERM:
                sigset(signo, SIG_DFL);
                break;
            case SIGSAK:
                sigset(SIGSAK, sigsak);
                break;
            case SIGDANGER:
                sigset(SIGDANGER, sigdanger1);
                break;
            case SIGSEGV:
                sigset(SIGSEGV, svrsegv);
                break;
            default:
                sigset(signo, SIG_IGN);
                break;
        }
    }
} /* endloop */
lfs_pipe(rrct, (struct LRB *) iorb, irb);

```

62

```

}                                /* main */

/*****
* parse_options *
*****/
int
parse_options(int argc, char **argv, struct EMU_OPTS * opt)
{
    unsigned int   lnk_rct, lnk_queue;
    int            o;
    extern char    *optarg;
    char          cbuff[10];

    opt->private = TRUE;
    opt->rusage = FALSE;
    while ((o = getopt(argc, argv, "PrVvw:s:t:R:q:u:")) != -1)
    {
        switch (o)
        {
            case 'P':
                opt->private = FALSE;
                break;
            case 'V':
                hvx_verV(argv[0]);
                exit(0);
                break;
            case 'v':
                closelog();
                openlog(username, LOG_PID, LOG_LOCAL0);
                setlogmask(LOG_MASK(LOG_ERR)
                    | LOG_MASK(LOG_WARNING)
                    | LOG_MASK(LOG_INFO)
                    | LOG_MASK(LOG_NOTICE)
                    | LOG_MASK(LOG_DEBUG));
                verbose = TRUE;
                break;
            case 's':
                sscanf(optarg, "%d", &opt->shmid);
                break;
            case 'R':
                /* RCT address this pipe */
                sscanf(optarg, "%x", &lnk_rct);
                rrct = (struct RCT *) lnk_rct;
                break;
            case 'q':
                sscanf(optarg, "%x", &lnk_queue);
                lnk_srvt = (struct MQI *) lnk_queue;
                break;
            case 't':
                sscanf(optarg, "%x", &lnk_queue);
                tskadr = (struct TTBL *) lnk_queue;
                break;
            case 'u':
                sscanf(optarg, "%s", &person);
                break;
        }
    }
}

```

63

```

        case 'w':
            sscanf(optarg, "%d", &virt_view);
            break;
        default:
            break;
    }
    /* switch */
}
/* while */
if (argc == 1)
    return (1);
else
    return (0);
}
/* parse_options */

/*****
 * rtn_ftbl * clean up an ftbl entry & queue the ttbl to the server.
 *****/
int
rtn_ftbl(struct IRB * irb)
{
    tskadr->cnt--;
    mqj_qot(lnk_srvr, irb);
    return tskadr->cnt;
}
/* rtn_ftbl */

/*****
 * lk_killed *
 *****/
void
lk_killed()
{
    ftbl->status |= Lk_SIG;
}
/* lk_killed */

/*****
 * lk_break *
 *****/
void
lk_break()
{
    ftbl->status |= Lk_BREAK;
}
/* lk_break */

/*****
 * lfs_pipe *
 *****/
void
lfs_pipe(struct RCT * rct, struct LRB * iorb, struct IRB * irb)
{
    /*
     * task server process for pipes
     * waits for irb then sets up iorb and FTBL entry
     * This process handles connects (physical and logical),
     * all read & write requests and both logical & physical disconnects.
     * Pipe opens are postponed until the first read or write is received.
    */

```

64

```

*/
int      access;
int      disc_detected = FALSE;
char     tempbuff[MAXPATHLEN + 1];

sigset(SIGTERM, lk_break);
sigset(SIGPIPE, lk_killed);
while (!disc_detected)
{
    irb = (struct IRB *) mqi_dqh(&tskadr->tch);
    SRVR_RISC_ADDR(iorb, (struct LRB *) irb->i_rb, irb);
    iorb = (struct LRB *) ((char *) iorb - OFF_LRB);
    GET4(math.adrs, iorb->lk_ftbl);
    ftbl = (struct FTBL *) math.ptr;
    func_code = iorb->lk_ct2 & 1_fc;
    iorb->lk_iofail = iorb->lk_fsfail = 0;
/*
    syslog(LOG_DEBUG, "TS function = %x", func_code);*/
    switch (func_code)
    {
        case CONN_FUNCTION:
            if (iorb->lk_dvs & 1_hu)      /* logical connect */
            {
                if (iorb->lk_control & 1_dirr)
                    ftbl->status |= Lk_READ;
                if (iorb->lk_control & 1_dirw)
                    ftbl->status |= Lk_WRITE;
                ftbl->status |= Lk_CONN;
                io_exit(rct, irb, SUCCESS_CODE);
                break;
            }
            /* logical connect */
            get_acl_qhead();
            if (geteuid())
            {
                syslog(LOG_ERR, "E860: hvx_pipe must run as root!");
                iorb->lk_fsfail = ACCESS_CONTROL_VIOL;
                iorb->lk_ct2 += FUNC_DELTA;
                rtn_ftbl(irb);
                break;
            }
            if (!tskadr->cnt)
                tskadr->cnt++;
            if (ftbl->status & Lk_SIG)
            {
                ftbl->status &= ~Lk_SIG;
                iorb->lk_iofail = 0;
                iorb->lk_fsfail = END_OF_FILE;
                iorb->lk_ct2 += FUNC_DELTA;
                rtn_ftbl(irb);
                break;
            }
            ftbl->status |= (iorb->lk_control & Lk_CONCUR);
            do
            {
                i = stat(ftbl->pathname, &file_status);
                while ((i < 0) && (errno == EINTR));
            }

```

65

```

if (i == -1)
{
    if (errno != ENOENT)
    {
        syslog(LOG_ERR, "E944: LINK %s error on stat function: %m",
            ftbl->pathname);
        iorb->lk_fsfail = UNAVL_CODE;
        iorb->lk_ct2 += FUNC_DELTA;
        rtn_ftbl(irb);
        break;
    }
}
else
{
    if (file_status.st_type != VFIFO)
    {
        iorb->lk_fsfail = ILLEGAL_FILE_TYPE;
        iorb->lk_ct2 += FUNC_DELTA;
        rtn_ftbl(irb);
        break;
    }
}
/* node exists */
/* check directory permissions, etc here if pipe does not exist */
if (li)
{
    /* can't check permissions til first I/O - won't know direction till then */
    ftbl->status |= P_CONN;
    io_exit(rct, irb, SUCCESS_CODE);
    break;
}
access = get_access(&ftbl->pathname, person, LK_PIPE);
if (access == -1)
{
    iorb->lk_fsfail = FILE_NOT_FOUND;
    iorb->lk_ct2 += FUNC_DELTA;
    rtn_ftbl(irb);
    break;
}
if ((access & (EXEC_ACCL | WRITE_ACCL)) != (EXEC_ACCL | WRITE_ACCL))
{
    iorb->lk_fsfail = ACCESS_CONTROL_VIOL;
    iorb->lk_ct2 += FUNC_DELTA;
    rtn_ftbl(irb);
    break;
}
errno = 0;
i = umask(0);
do
    ftbl->fd.fdes = mknod(ftbl->pathname,
        (S_IFIFO | S_IRUSR | S_IWUSR
         | S_IRGRP | S_IWGRP), 0);
while ((ftbl->fd.fdes < 0) && (errno == EINTR));
umask(i);
if ((ftbl->fd.fdes < 0) && (errno != EEXIST))

```

66

```

{
    syslog(LOG_ERR, "E945: LINK %s mknod failed: %m",
           ftbl->pathname);
    iorb->lk_fsfail = UNAVL_CODE;
    iorb->lk_ct2 += FUNC_DELTA;
    rtn_ftbl(irb);
    break;
}
if (!(errno == EEXIST))
{
    do
        i = chown(ftbl->pathname, Uid, -1);
    while ((i < 0) && (errno == EINTR));
    if (i < 0)
    {
        unlink(ftbl->pathname);
        syslog(LOG_ERR, "E946: LINK %s chown failed: %m",
               ftbl->pathname);
        iorb->lk_fsfail = UNAVL_CODE;
        iorb->lk_ct2 += FUNC_DELTA;
        rtn_ftbl(irb);
        break;
    }
    ftbl->status |= Lk_CRPIPE;
}
ftbl->status |= P_CONN;
io_exit(rct, irb, SUCCESS_CODE);
break;
case READ_FUNCTION:
    if (ftbl->status & Lk_SIG)
    {
        ftbl->status &= ~Lk_SIG;
        iorb->lk_fsfail = END_OF_FILE;
        io_exit(rct, irb, SUCCESS_CODE);
        break;
    }
    if (ftbl->status & Lk_WRITE)
    {
        /* read to a write pipe */
        iorb->lk_fsfail = ILLEGAL_ACCESS;
        io_exit(rct, irb, SUCCESS_CODE);
        break;
    }
    if (!(ftbl->status & Lk_OPEN))
    {
        access = get_access(&ftbl->pathname, person, LK_FILE);
        if ((access == -1) || !(access & READ_ACCL))
        {
            iorb->lk_fsfail = ACCESS_CONTROL_VIOL;
            io_exit(rct, irb, SUCCESS_CODE);
            break;
        }
        if (iorb->lk_control & l_mod)
            ftbl->status |= Lk_MOD;
        do

```

67

```

    if (fbl->status & Lk_MOD)
    {
        fbl->fd.stream =
            fopen(fbl->pathname, "r");
        if (fbl->fd.stream > (FILE *) NULL)
            setvbuf(fbl->fd.stream, NULL, _IONBF, 0);
    }
    else
        fbl->fd.fdes = open(fbl->pathname, O_RDONLY);
    while ((fbl->fd.fdes < 0)
        && ((errno == EINTR)
            && !(fbl->status & (Lk_BREAK | Lk_SIG))));
    if (fbl->status & Lk_BREAK)
    {
        fbl->status &= ~Lk_BREAK;
        iorb->lk_fsfail = END_OF_FILE;
        iorb->lk_st = l_prg;
        io_exit(rct, irb, SUCCESS_CODE);
        break;
    }
    if (fbl->fd.fdes < 0)
    {
        syslog(LOG_ERR, "E947: LINK %s open failed: %m",
            fbl->pathname);
        io_exit(rct, irb, UNAVL_CODE);
        break;
    }
    fbl->status |= (Lk_OPEN | Lk_READ);
}
/* if not open */
if (iorb->lk_control & l_seek)
{
    /* seek not allowed for pipes */
    iorb->lk_fsfail = ILLEGAL_ACCESS;
    io_exit(rct, irb, SUCCESS_CODE);
    break;
}
SRVR_RISC_ADDR(ptr, (char *), iorb->lk_adr, irb);
if (iorb->lk_ct2 & l_b)
    ptr++;
PUT4(iorb->lk_seek, fbl->current_rseek);
do
    if (fbl->status & Lk_MOD)
    {
        errno = 0;
        if (fgets(tempbuff, iorb->lk_rng + 1, fbl->fd.stream)
            == NULL)
        {
            if (errno) /* if error */
                xfer_cnt = bytes_tran = -1;
            else /* if eof */
                xfer_cnt = bytes_tran = 0;
        }
    }
    else
    {
        xfer_cnt = bytes_tran = strlen(tempbuff);
    }
}

```

68

```

        if (tempbuff[bytes_tran - 1] = '\n')
            bytes_tran--;
        memcpy(ptr, &tempbuff, bytes_tran);
    }
}
else
    xfer_cnt = bytes_tran = read(ftbl->fd.fdes,
                                ptr, iorb->lk_rng);
while ((bytes_tran < 0)
        && ((errno == EINTR)
            && !(ftbl->status & (Lk_BREAK | Lk_SIG))));
if (ftbl->status & Lk_BREAK)
{
    ftbl->status &= ~Lk_BREAK;
    iorb->lk_fsfail = END_OF_FILE;
    iorb->lk_st = l_prg;
    io_exit(rct, irb, SUCCESS_CODE);
    break;
}
if (bytes_tran == -1)
{
    syslog(LOG_ERR, "E948: LINK read failed: %m");
    io_exit(rct, irb, FAILURE_CODE);
    break;
}
ftbl->current_rseek += bytes_tran;      /* xfer_cnt???? */
if (iorb->lk_control & l_curnc)
    ftbl->current_wseek = ftbl->current_rseek;
iorb->lk_rsr = iorb->lk_rng - bytes_tran;
if (xfer_cnt > 0)
    io_exit(rct, irb, SUCCESS_CODE);
else
{
    iorb->lk_fsfail = END_OF_FILE;
    io_exit(rct, irb, SUCCESS_CODE);
}
break;
case WRITE_FUNCTION:
    if (ftbl->status & Lk_READ)
    {
        /* write to a read pipe */
        iorb->lk_fsfail = ILLEGAL_ACCESS;
        io_exit(rct, irb, SUCCESS_CODE);
        break;
    }
    if (!(ftbl->status & Lk_OPEN))
    {
        access = get_access(&ftbl->pathname, person, LK_FILE);
        if ((access == -1) || !(access & WRITE_ACCL))
        {
            iorb->lk_fsfail = ACCESS_CONTROL_VIOL;
            io_exit(rct, irb, SUCCESS_CODE);
            break;
        }
    }
    if (iorb->lk_control & l_mod)

```

69

```

        ftbl->status |= Lk_MOD;
    do
        if (ftbl->status & Lk_MOD)
        {
            ftbl->fd.stream = fopen(ftbl->pathname, "w");
/* in this mode the pipe must be unbuffered */
            if (ftbl->fd.stream > (FILE *) NULL)
                setvbuf(ftbl->fd.stream, NULL, _IONBF, 0);
        }
        else
            ftbl->fd.fdes = open(ftbl->pathname, O_WRONLY);
    while ((ftbl->fd.fdes < 0)
        && ((errno == EINTR)
        && !(ftbl->status & (Lk_BREAK | Lk_SIG))));
    if (ftbl->status & Lk_BREAK)
    {
        ftbl->status &= ~Lk_BREAK;
        iorb->lk_fsfail = END_OF_FILE;
        iorb->lk_st = l_prg;
        io_exit(rct, irb, SUCCESS_CODE);
        break;
    }
    if (ftbl->fd.fdes < 0)
    {
        syslog(LOG_ERR, "E949: LINK %s open failed: %m",
            ftbl->pathname);
        io_exit(rct, irb, UNAVL_CODE);
        break;
    }
    ftbl->status |= (Lk_OPEN | Lk_WRITE);
}
/* if not open */
if (iorb->lk_control & l_seek)
{
    /* seek not allowed for pipes */
    iorb->lk_fsfail = ILLEGAL_ACCESS;
    io_exit(rct, irb, SUCCESS_CODE);
    break;
}
SRVR_RISC_ADDR(ptr, (char *), iorb->lk_adr, irb);
if (iorb->lk_ct2 & l_b)
    ptr++;
PUT4(iorb->lk_seek, ftbl->current_wseek);
do
    if (ftbl->status & Lk_MOD)
    {
        memcpy(&tempbuff, ptr, iorb->lk_rng);
        tempbuff[iorb->lk_rng] = '\n';
        tempbuff[iorb->lk_rng + 1] = 0;
        bytes_tran = fputs(tempbuff, ftbl->fd.stream);
        if (bytes_tran >= 0)
            bytes_tran = strlen(tempbuff) - 1;
    }
    else
        bytes_tran = write(ftbl->fd.fdes, ptr, iorb->lk_rng);
while ((bytes_tran < 0)

```

70

```

        && ((errno == EINTR)
            && (!(fbl->status & (Lk_BREAK | Lk_SIG))));
iorb->lk_rsr = iorb->lk_rng - bytes_tran;
if (bytes_tran != iorb->lk_rng)
{
    if (bytes_tran < 0)
    {
        if (fbl->status & Lk_SIG)
        {
            fbl->status &= ~Lk_SIG;
        }
    }
    /*
    * temp change for p1686; file system does not kill task on LINK_LOST.
    * if/when this is fixed, uncomment next two lines & delete UNAVL io_exit
    */
    /*
    iorb->lk_fsfail = LINK_LOST;*/
    /*
    io_exit(rct, irb, SUCCESS_CODE);*/
    io_exit(rct, irb, UNAVL_CODE);
    break;
}
if (fbl->status & Lk_BREAK)
{
    fbl->status &= ~Lk_BREAK;
    iorb->lk_fsfail = END_OF_FILE;
    iorb->lk_st = l_prg;
    io_exit(rct, irb, SUCCESS_CODE);
    break;
}
syslog(LOG_ERR, "E950: LINK write failed: %m");
}
else
    syslog(LOG_ERR, "E951: LINK write failed");
io_exit(rct, irb, FAILURE_CODE);
break;
}
fbl->current_wseek += bytes_tran;
if (iorb->lk_control & l_curnc)
    fbl->current_wseek = fbl->current_rseek;
io_exit(rct, irb, SUCCESS_CODE);
break;
case DISC_FUNCTION:
/*
* if we created the pipe then delete it on a physical disconnect..
* ignore signals - we are closing here
*/
fbl->status &= ~(Lk_SIG | Lk_BREAK);
if (iorb->lk_dvs & l_hu)
{
    if (!(fbl->status & Lk_CONN))
    {
        io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
        break;          /* already logically disconnected */
    }
    if (fbl->status & Lk_OPEN)
    {

```

71

```

do
    if (ftbl->status & Lk_MOD)
        status = fclose(ftbl->fd.stream);
    else
        status = close(ftbl->fd.fdes);
    while ((status < 0) && (errno == EINTR));
    if (status == -1)
    {
        iorb->lk_iofail = FAILURE_CODE;
        syslog(LOG_ERR, "E952: LINK close failed: %m");
    }
    else
    {
        iorb->lk_iofail = SUCCESS_CODE;
        ftbl->fd.fdes == -1;
        ftbl->status &= ~Lk_CONN;
    }
    ftbl->status &= ~Lk_OPEN;
}
else
{
    iorb->lk_iofail = SUCCESS_CODE;
    ftbl->fd.fdes == -1;
    ftbl->status &= ~Lk_CONN;
}
io_exit(rct, irb, iorb->lk_iofail);
break;
}
/* logical disconnect */
if (ftbl->status & Lk_CONN)
{
    ftbl->fd.fdes == -1;
    ftbl->status &= ~Lk_CONN;
}
}
/*
* don't unlink any pipe not created by us
*/
if (ftbl->status & Lk_CRPIPE)
    unlink(ftbl->pathname);
ftbl->status &= ~(P_CONN | Lk_CRPIPE);
iorb->lk_ct2 += FUNC_DELTA;
if (rtn_ftbl(irb) == 0)
    disc_detected = TRUE;
break;
default:
    io_exit(rct, irb, ILLEGAL_FUNCTION_CODE);
    break;
}
/* switch */
}
/* for */
/* sleep forever, the dual decor server will kill us in a moment */
for (;;)
{
    sleep(1000);
}
}
/* lfs_pipe */

```

72

APPENDIX BLN COMMAND EXAMPLES**Example A:**

Using Link Files

```
LK >>sid>clm_x "x:/usr/hvx/active/clm/clm_x" -T -FILE  
ed >>sid>clm_x
```

In this example, the user uses the LK command to link the emulated system file >>sid>clm_x to the host file /usr/hvx/active/clm/clm_x. The user then edits >>sid>clm_x using the emulated system editor.

73

HOST

EMULATED SYSTEM

KEY 12345678-ABCDEFGH

MEMPOOL I AB 360KB

DISK WIN00 PATH=/hvxvols/bootdisk

NTD_MODE ATD=IORB

NTD NTD00 DEV=/dev/console CONSOLE

NTD NTD01 DEV=/dev/tty1

NTD NTD02 DEV=/dev/tty2 ATD=NONE

NTD NTD03 DEV=/dev/tty3 ATD=CB

LK>>sid>clm_x "x:/usr/hvx/
active/clm/clm_x" -T/usr/hvx/active/clm/clm_x**Emulated System user creates a link to a host file.**KEY 12345678-ABCDEFGH

MEMPOOL I AB 360KB

DISK WIN00 PATH=/hvxvols/bootdisk

NTD_MODE ATD=IORB

NTD NTD00 DEV=/dev/console CONSOLE

NTD NTD01 DEV=/dev/tty1

NTD NTD02 DEV=/dev/tty2 ATD=NONE

NTD NTD03 DEV=/dev/tty3 ATD=CB

/usr/hvx/active/clm/clm_xKEY 12345678-ABCDEFGH

MEMPOOL I AB 360KB

DISK WIN00 PATH=/hvxvols/bootdisk

NTD_MODE ATD=IORB

NTD NTD00 DEV=/dev/console/CONSOLE

NTD NTD01 DEV=/dev/tty1

NTD NTD02 DEV=/dev/tty2 ATD=NONE

NTD NTD03 DEV=/dev/tty3 ATD=CB

>>sid>clm_x

Emulated user can now access the host file from the emulated system.

74

Example B:
Using Linked Named Pipes

A pipe is a FIFO (first-in, first-out) stream that is used to synchronize and pass information between cooperating tasks. In the system, host processes (tasks) can cooperate with emulated system tasks.

Pipes can be used to send data from a host application to an emulated system application and vice versa. For example, a pipe can be used to send a query from an emulated system application to a database residing on the host system. The database could return the reply using a second pipe. This is illustrated below.

When a pipe is created, the file name in the host pathname specified is not an actual file. It is the name of a pipe, that is, a named pipe. Pipes can be permanent or transient. If a link is created to a named pipe that already exists in the host file system, it is a permanent pipe. The pipe can be opened and closed repeatedly as needed. If a link is created to a named pipe that does not exist, the pipe is created when it is opened. When it is closed, the pipe is deleted.

To create a named pipe in the host file system, the `mknod` command is used whose syntax is: `mknod node__name p`.

An instance of a pipe can transmit data in only one direction. Once a pipe is closed, it can be opened again and transmit in the opposite direction from its previous instance. For example, application A transmits data to application B over the pipe named PIPE01. The transmission is finished and the pipe is closed. Application B could then open PIPE01 and transmit to application A.

Host and emulator system applications must open pipes in the same order. All initial references to pipes in the emulated system must occur in the same order as the pipes are opened in the host file system. This matching process is needed to establish protocols. Once the protocols are established, the pipes can be used in whatever sequence needed, as long as the protocols are maintained.

75

HOST EMULATED SYSTEM

LK PIPE1 "X:/usr/lee/pipe1" -B -PIPE

Host Application	pipe1	PIPE1	ES Application
---------------------	-------	-------	-------------------

Applications Using One Pipe

Data can be transmitted in only one direction for each instance of the open pipe.

Different instances of the pipe can transmit in different directions.

LK pipe1 "X:/usr/lee/pipe1" -B -PIPE

LK pipe2 "x:/usr/lee/pipe2" -B -PIPE

Host Application	pipe1	pipe1	data	ES Application
	data			
	pipe2	pipe2		

Applications Using Two Pipes

Applications can send data back and forth.

Linked Named Pipes

76

Example C:
Using Linked Named Pipes

In this example, a COBOL application running in the emulated system can accept SQL statements. When it receives a query, it writes the query to a named pipe that passes the query to a host application that is waiting for queries. The host application queries the database and receives a response. The response is written to a named pipe that passes the information to the COBOL application, which has been waiting in a read state.

To implement this example, pipe files must be created on the host side and they are linked on the emulated system side. On the host side, two named pipes are created with the mknod command as follows:

```
mknod /usr/dir/sqlout p
mknod /usr/dir/sqlin p.
```

On the emulated system side, links are created to the two named pipes with the LK command, as follows:

```
LK sqlout "x:/usr/dir/sqlout" -T -PIPE
LK sqlin "x:/usr/dir/sqlin" -T -PIPE.
```

The two cooperating applications must open the pipes in the same order. If the host application opens sqlout first, the emulated system application must open sqlout first.

		SQL Query	
		(sqlout)	Emulated System
Data	AIX		Application
base	Application	(sqlin)	
		SQL Response	

77

It will be apparent to those skilled in the art that many changes may be made to the preferred embodiment of the present invention.

While in accordance with the provisions and statutes there has been illustrated and described the best form of the invention, certain changes may be made without departing from the spirit of the invention as set forth in the appended claims and that in some cases, certain features of the invention may be used to advantage without a corresponding use of other features.

CLAIMS

73

1. A host system having a memory organized into shared and host memory areas and a hardware platform including a plurality of input/output devices operatively connected for executing host system instructions and for emulating the execution of emulated system instructions by an emulator running as an application process on said host system, said plurality of input/output devices including input/output devices for storing files contained within emulation system and host file systems, said emulator including a number of emulated system executive service components operating in said shared memory area comprising a command handler unit and file management component operatively coupled to said command handler unit for managing files of said emulated system file system and an interpreter, an emulator monitor call unit (EMCU) and server facilities operating in said host memory area, said host system further including operating system facilities for providing a number of services for host programs, said operating system facilities being coupled to said plurality of input/output devices and to said EMCU, said host system further including a link file facility for linking together files of said emulated file system and said host file systems, said link file facility comprising:

link file command handler means included within said command handler unit, said command handler means responsive to each link (LK) command specifying creating a link to a host file generating a standard file monitor call to said file management component accompanied by information coded in a predetermined manner for specifying characteristics of a host file for which said link is being created for writing as a directory link entry in said file of said emulated file system;

link file means included in said file management component in response to each standard file monitor call from said link file command handler means pertaining to accessing a file within said emulated file system associated with said link to said host file initializing predetermined data structures associated with said emulated system file operation in a predetermined manner for executing special link operation and issuing an input/output request;

a user table (USTBL) mechanism located in said host memory area, said USTBL mechanism having a number of locations for storing unique user description entries generated by said EMCU for emulated system users whose identities were previously validated during login; and,

said server facilities including first server handler means including user security validation means, said security validation means in response to input/output

79

request generated by said file management component in response to an emulated system user request specifying access to the linked host file, performing a validation operation on said emulated system user by accessing said user table mechanism entry associated with said user requesting access to ascertain that access is being requested by a trusted user and would not compromise host system security and a second server means operatively coupled to said first server handler means, said second server means receiving said input/output request for executing a link file operation only when said validation operation confirms that the request was issued by a trusted user.

2. The host system of claim 1 wherein said second server includes access control means for performing an access control check for establishing that said emulated system user has the required access permissions to execute the link file operation.

3. The host system of claim 2 wherein the link file can be a file or a named pipe.

4. The host system of claim 1 wherein each unique user description entry includes emulated system descriptive information associated with the user by said executive service components which is not modifiable by the user.

5. The host system of claim 4 wherein said emulated system user descriptive information used to perform said validation operation includes a first portion containing user-id information and second and third portions containing addresses of predetermined data structures used by the executive service components to uniquely identify the user.

6. The host system of claim 5 wherein said first server handler means performs said validation operation by comparing the portions of said emulated system user descriptive information previously stored in said user table mechanism as an entry for the user with corresponding portions of emulated system user descriptive information obtained from said emulated memory area obtained by said first server handler which is associated with a current user issuing said file request.

7. The host system of claim 6 wherein said first server handler means upon determining that there is no identical comparison indicating that the current user is untrustworthy, signals the occurrence of a user security violation and does not initiate a series of operations for enabling execution of the link file operation.

8. The host system of claim 6 wherein said first server handler means upon determining that there is an identical comparison identifying no change in user personality confirming that the current user is trustworthy, initiates a series of operations for enabling execution of the link file operation.

9. The host system of claim 3 wherein said second server means performs said validation of access permissions by said access control means at a time when file access is to take place which corresponds to an open command in the case of a file and on a first read or write request in the case of a named pipe.

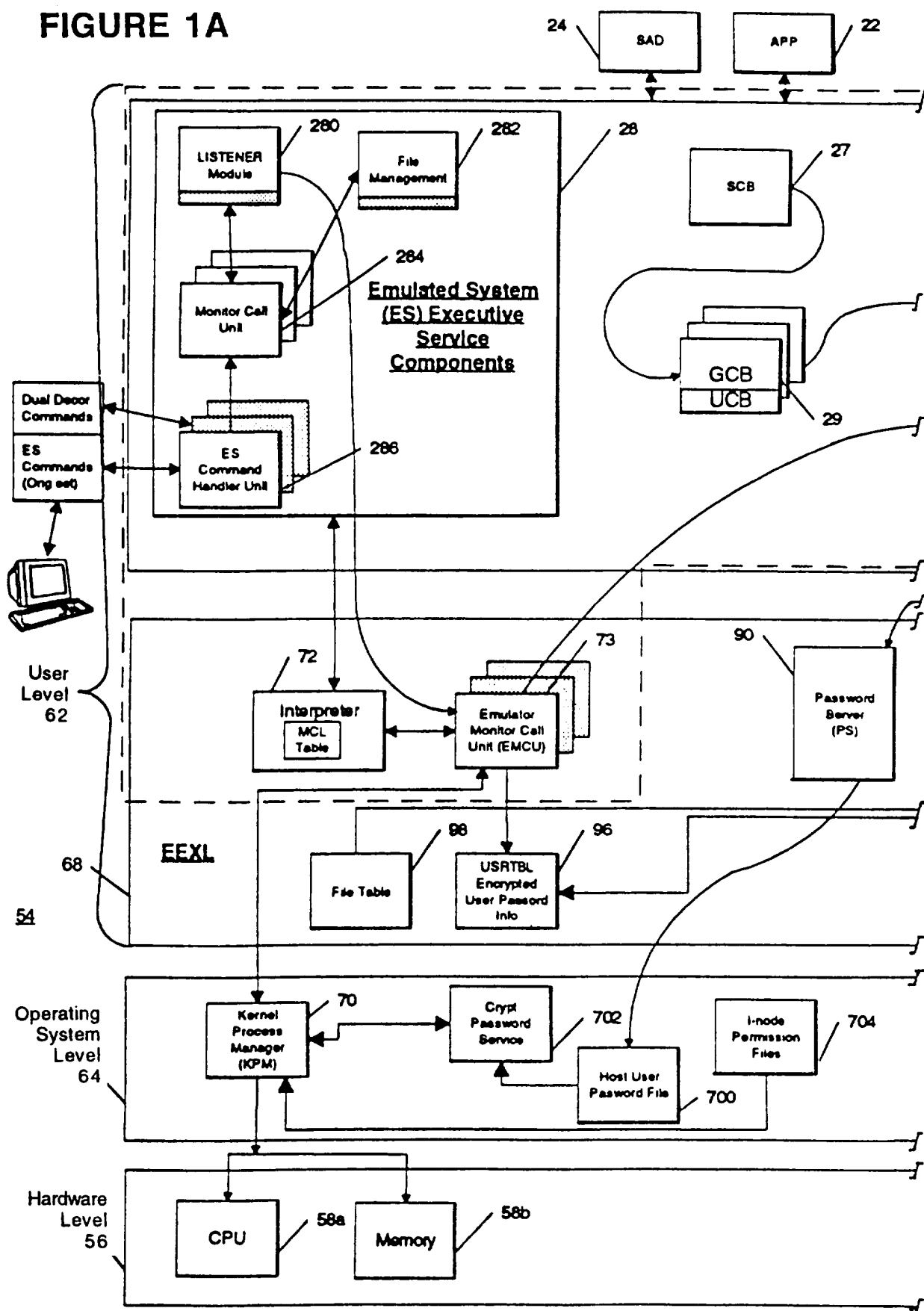
10. The host system of claim 2 wherein said operating system includes Inode permission files defining access permissions for each file of said host system, said access control means generating a number of system calls to said operating system for validating user access on the basis of both basic and extended permissions defined by said Inode permission files.

11. The host system of claim 1 wherein said LK command is an extension of an existing command in a set of commands utilized by said emulated system.

12. The host system of claim 11 wherein said LK command has a predetermined format having a number of different portions, a first portion being coded to specify a link command code for invoking said command handler means, a second portion being coded for designating an emulated system name of the link, a third portion being coded to include a prefix designating a host path name for the link and a fourth portion containing information for defining certain characteristics of the host file being linked.

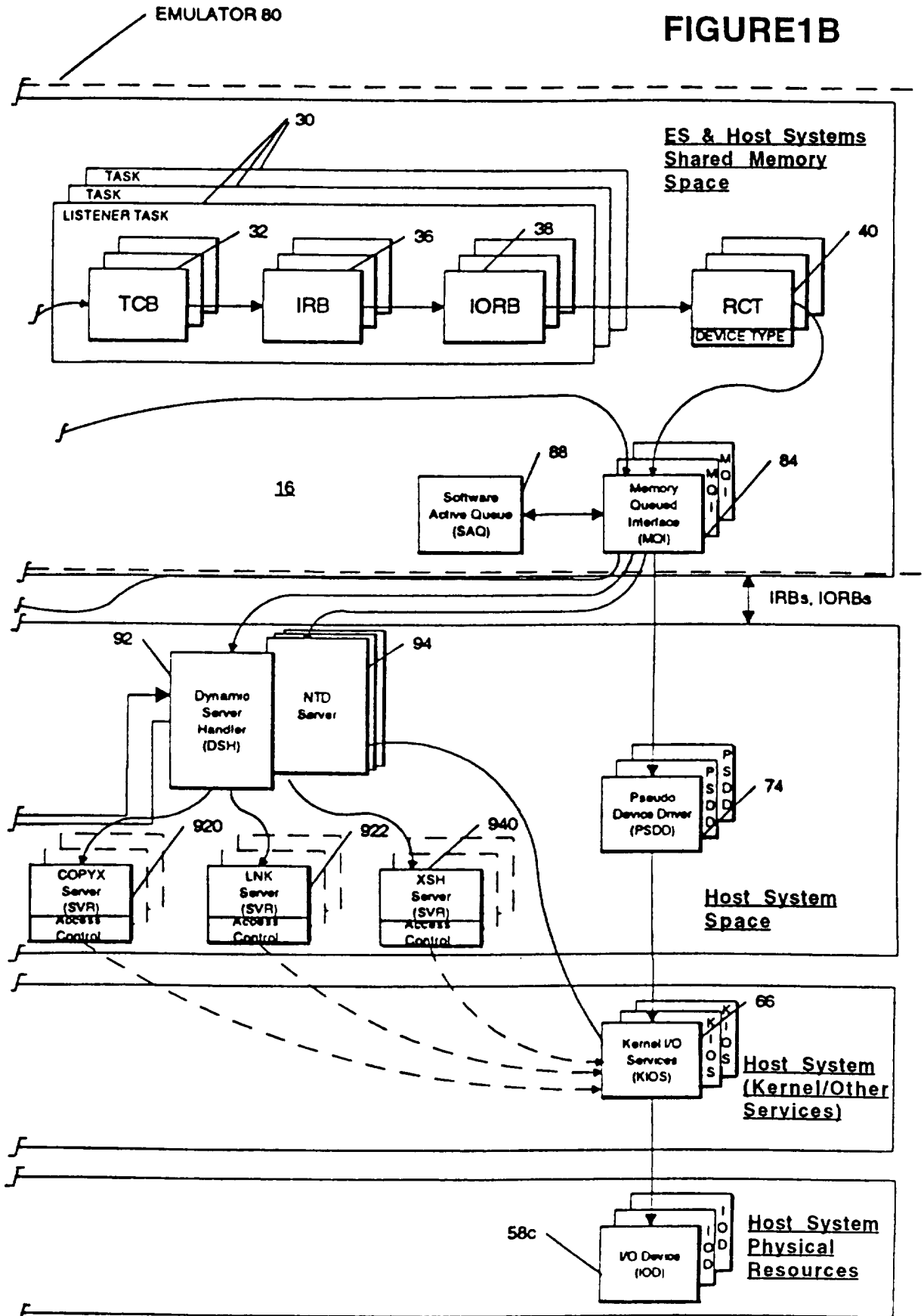
1/9

FIGURE 1A



2/9

FIGURE 1B



3/9

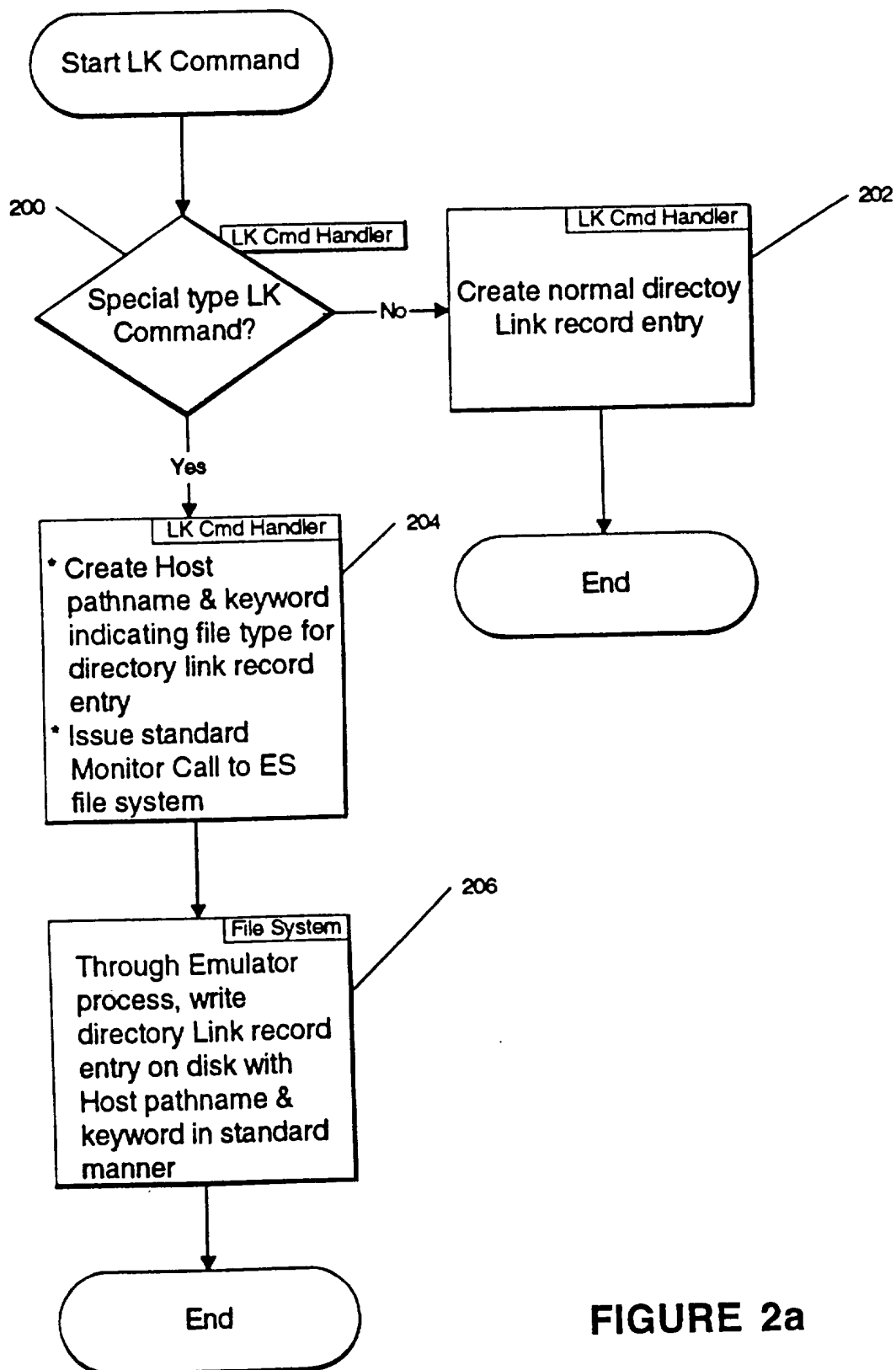


FIGURE 2a

4/9

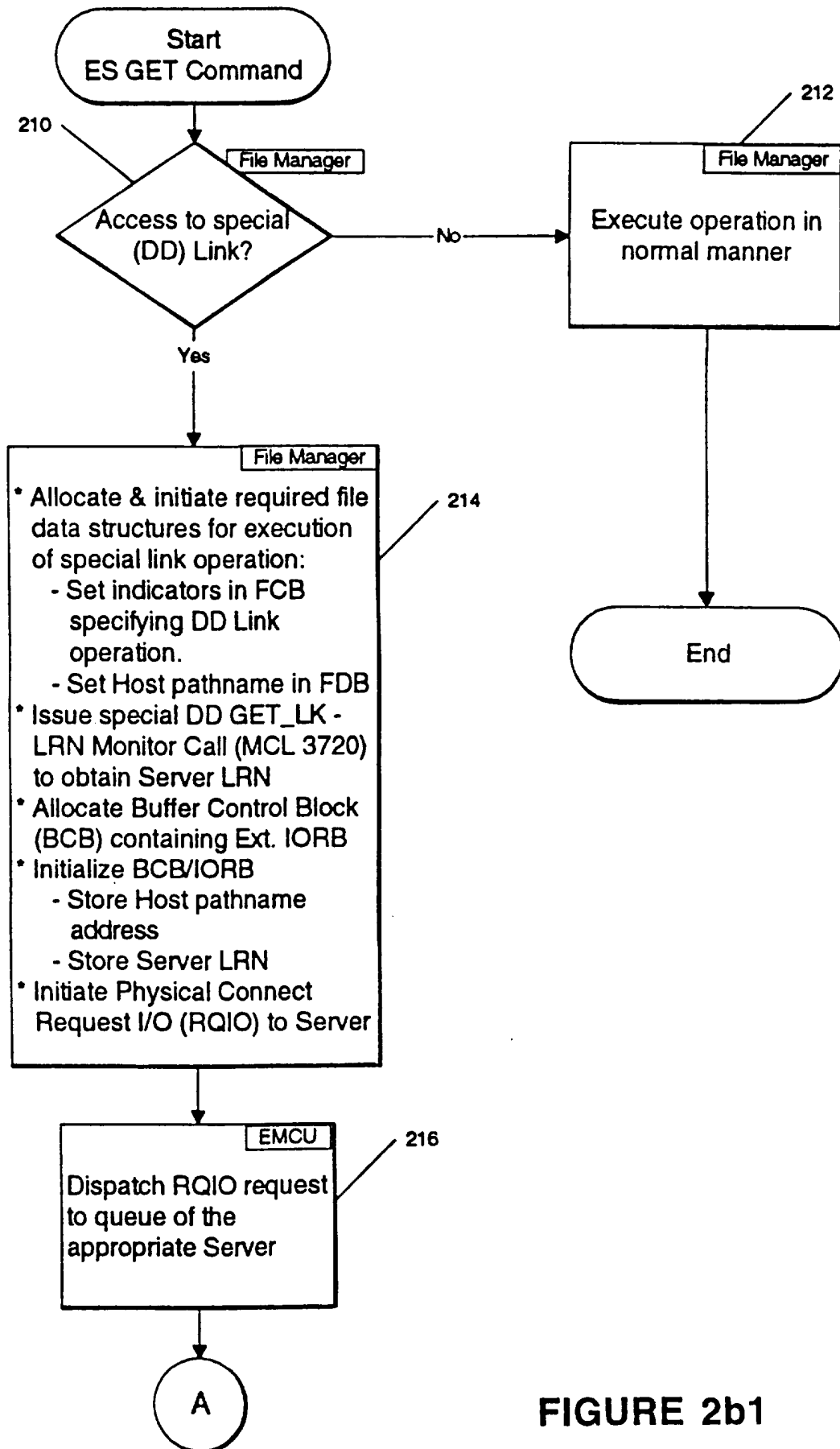


FIGURE 2b1

5/9

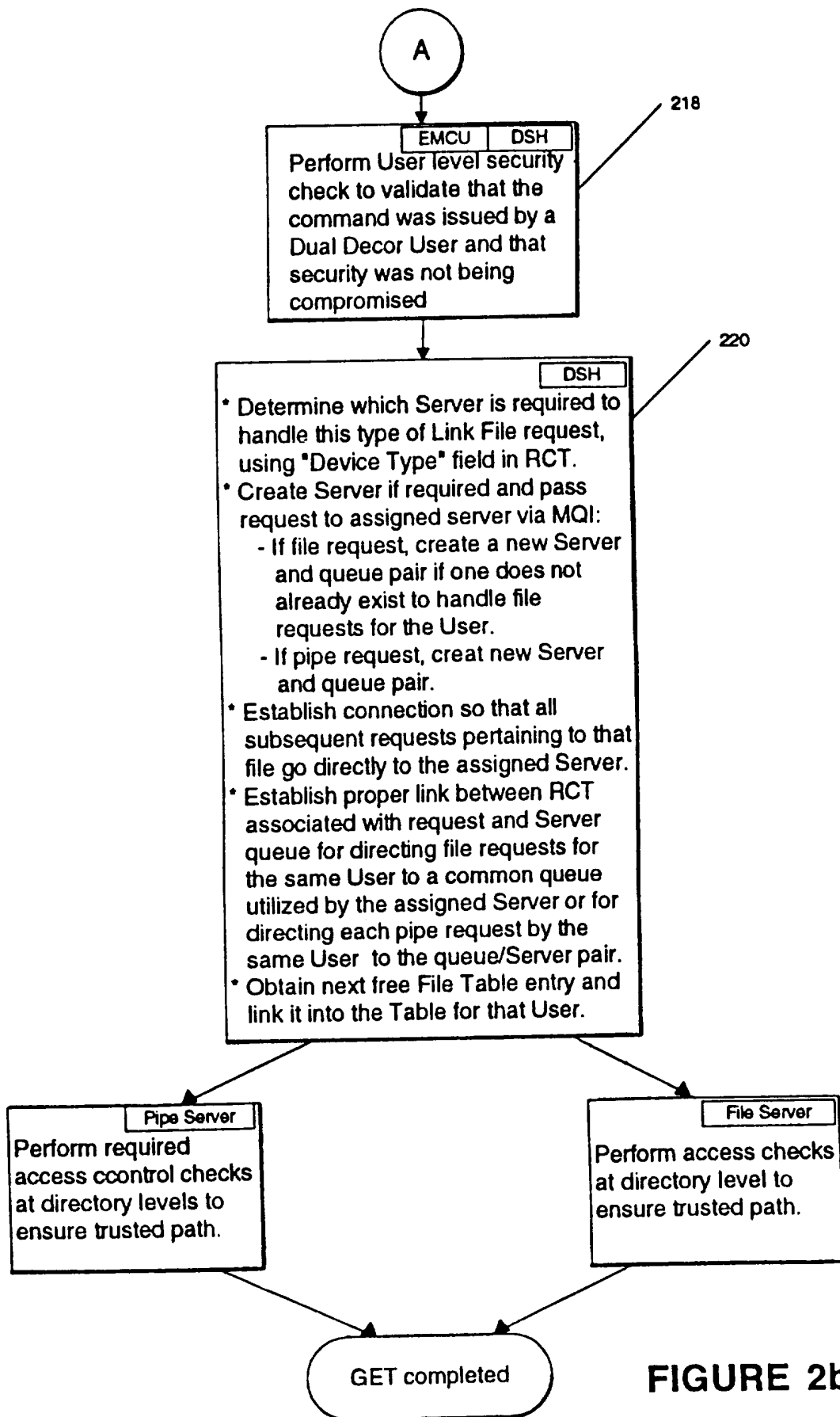


FIGURE 2b2

6/9

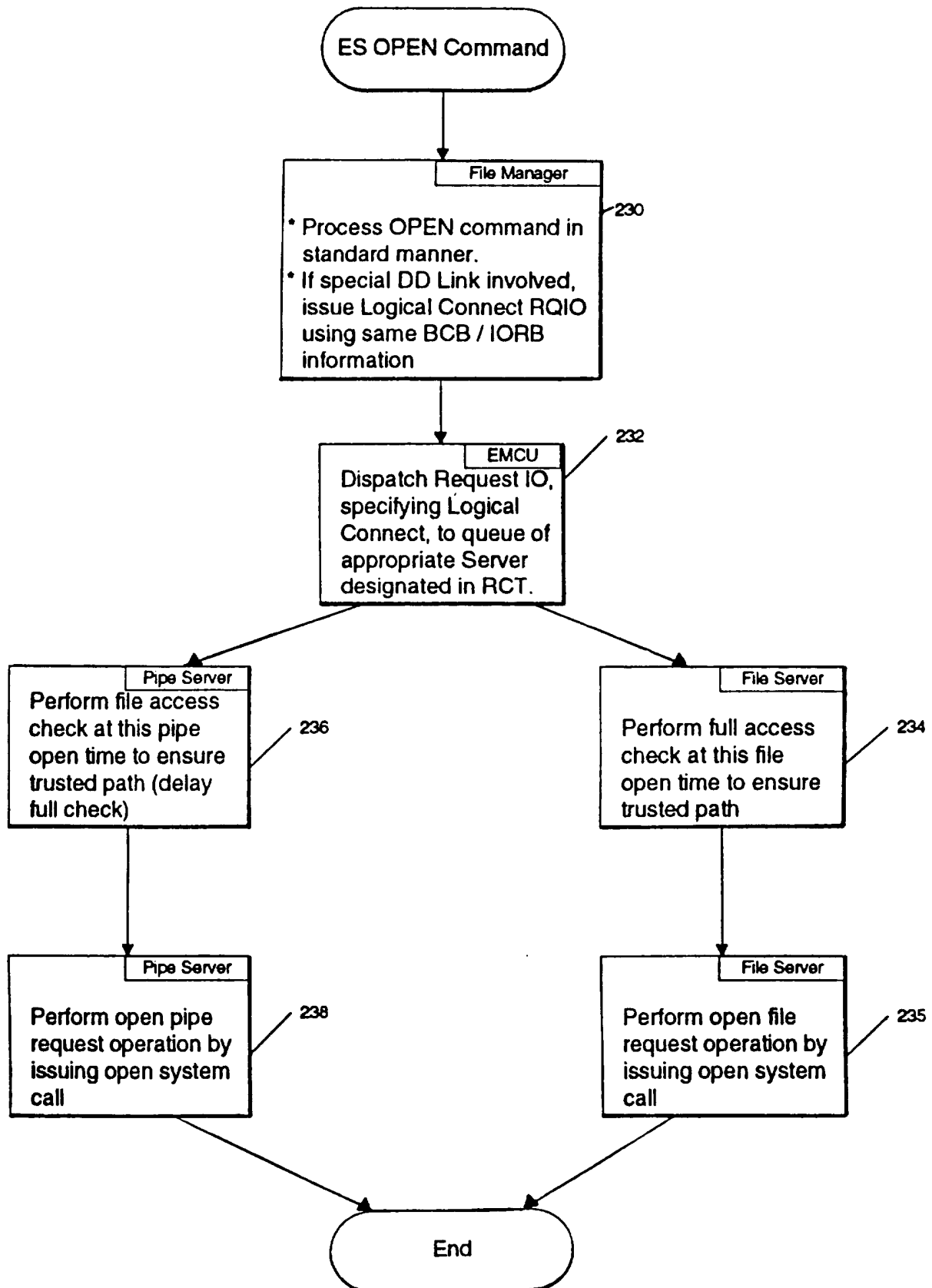


FIGURE 2c

7/9

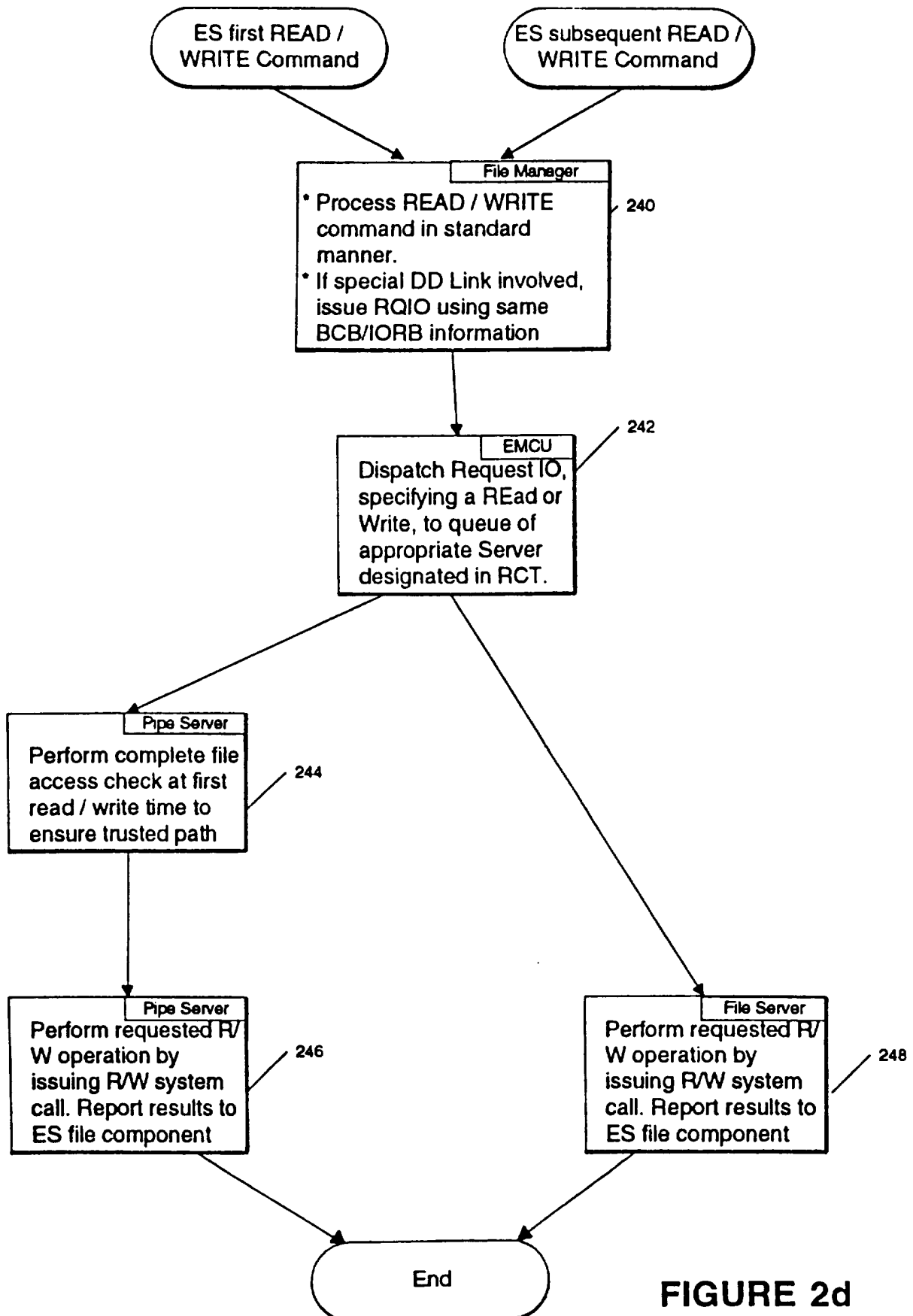


FIGURE 2d

8/9

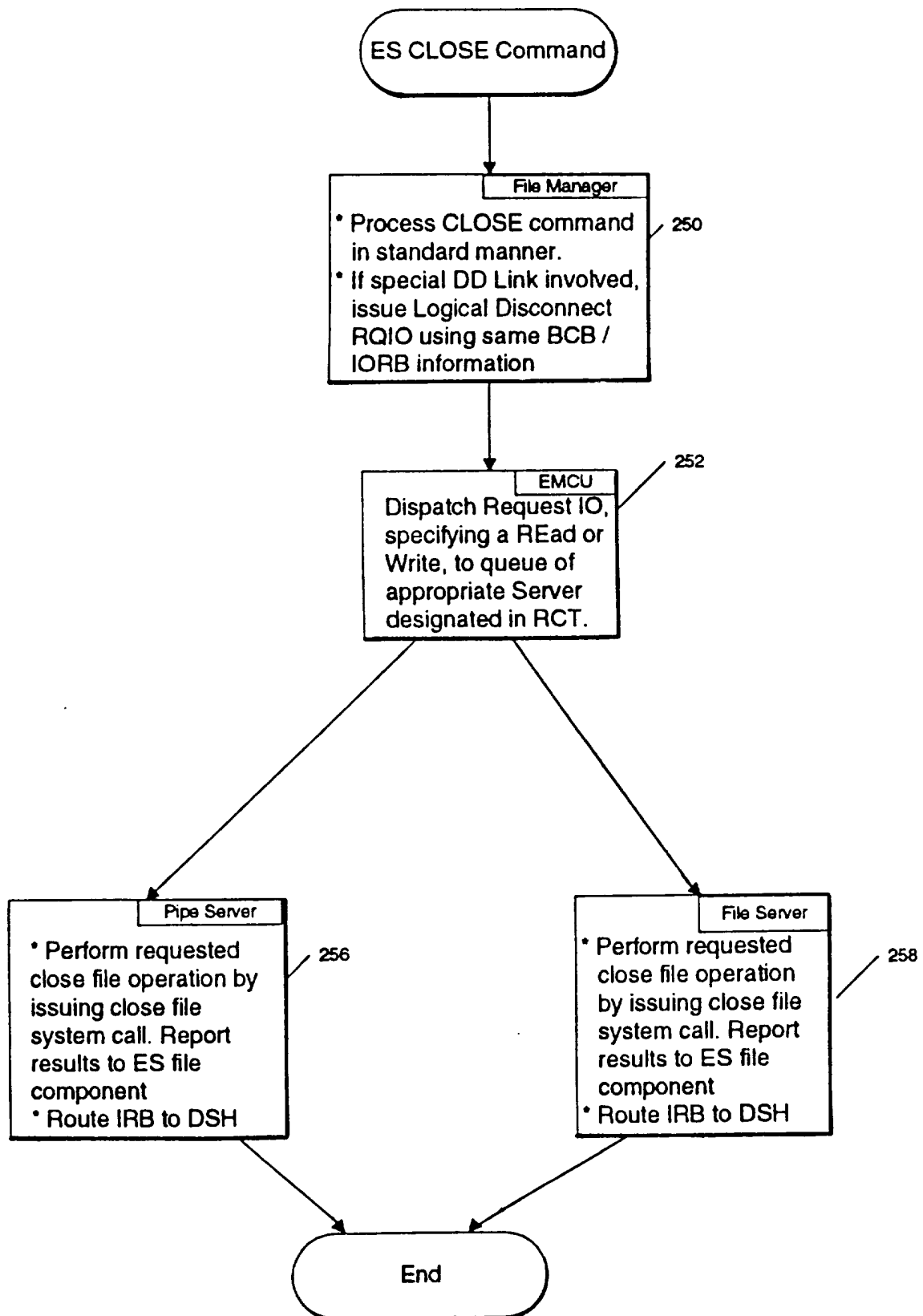


FIGURE 2e

9/9

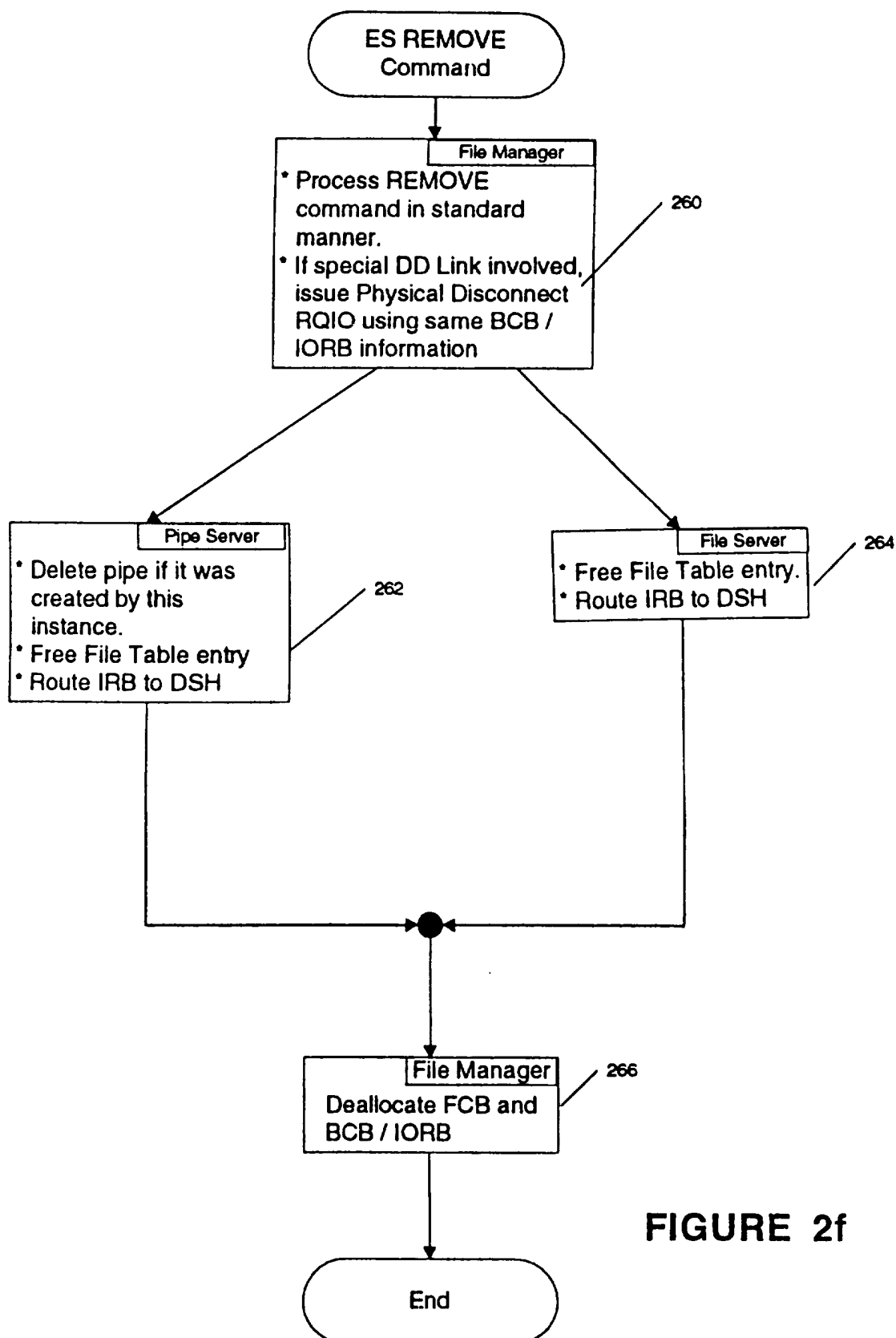


FIGURE 2f