XOS: An Operating System for the X-TREE Architecture

Barton Miller
David Presotto

University of California
Computer Science Division
Berkeley, CA 94720

Abstract
This paper describes the fundamentals of the X-TREE Operating System (XOS), a system developed to investigate the effects of the X-TREE architecture on operating system design. It outlines the goals and constraints of the project and describes the major features and modules of XOS. Two concepts are of special interest: The first is demand paging across the network of nodes and the second is separation of the global object space and the directory structure used to reference it. Weaknesses in the model are discussed along with directions for future research.

1. INTRODUCTION

X-TREE is an architecture for the design and construction of distributed, multiprocessor computer systems. Its major intent is to provide a model for building powerful, low-cost systems comprised of many identical microprocessor chips (known in previous papers as monolithic microprocessors) [Desp78, Ditz80].

It was recognized early in the X-TREE project that it would not be sufficient to consider only paper operating systems designs when attempting to evaluate the feasibility of the X-TREE architecture. Therefore a project was undertaken from April until December in 1979 to write an actual system that could run under X-TREE. At the time, the hardware was largely hypothetical, which gave us both an advantage and a problem. On the one hand we could have great influence on the eventual design. On the other, we had no hardware to run on. To solve this problem, we constructed a simulator running under UNIX on a VAX 11/780. The simulator of the hardware was both ad hoc and slow, but it gave us an opportunity to try out our ideas without a real X-TREE.

The X-TREE Operating System (XOS) is a general purpose operating system for X-TREE. Our major goals were to discover the aspects of the architecture that influence the design of a distributed operating system and to suggest possible features for the X-TREE that would aid an operating system.

1.1. The X-TREE Architecture

While the details of the architecture are rather complicated [Pres80], only a few are relevant to our discussion and need be presented here. They are:

Topology: Figure 1.1 depicts a possible X-TREE system. It is important to note that all devices are connected to leaf nodes. This was a major influence in the way the operating system was eventually divided.

This study was sponsored in part by the Joint Services Electronics Program, Contract F44620-76-C-0100 and by the National Science Foundation under Grant No. MCS 7807291.

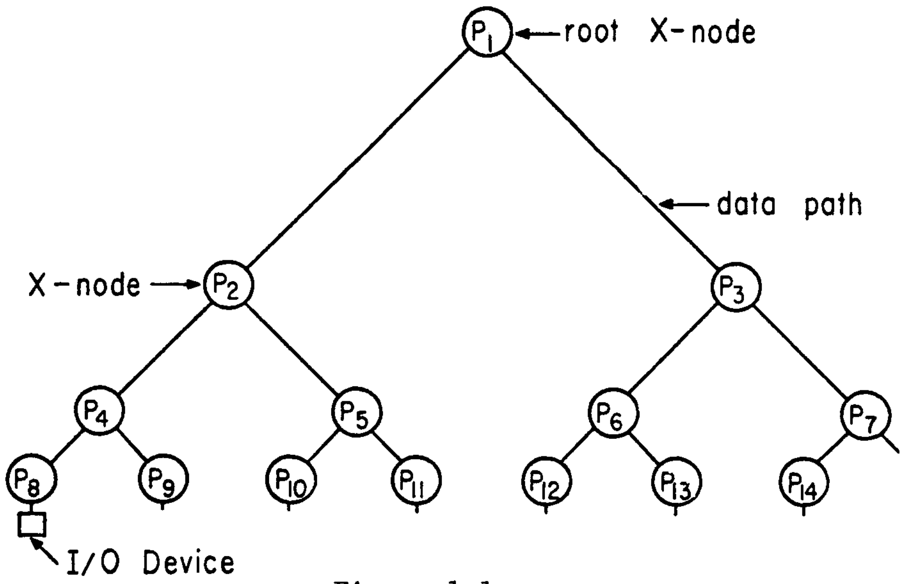

Figure 1.1

Communications Hardware: X-TREE has a sophisticated communications system implemented in hardware. Points significant to the operating system are:

1) Multiple streams of messages between nodes are routed and maintained completely by hardware.
2) The only data needed by the hardware to route the information is the node address of the destination. Node addresses are of variable length to permit an expandable address space. Refer to the node numbers in Figure 1.1.

CPU flexibility: An X-TREE CPU is microprogrammed. Therefore we had the ability to choose data structures or addressing mechanisms that were appropriate.

Node memory size: Each node has on the order of 64K bytes of private memory. Part of the memory contains the local kernel code, and part of it is managed as a cache for the data and capabilities being referenced by processes on that node. This placed a restriction on the maximum kernel size of approximately 32K bytes.

1.2. Specific Goals

As we have said above, our major goal was an operating system design for a general purpose system to investigate aspects of the X-TREE architecture. Additional goals of the design were:

Resource sharing: Since this was to be a distributed, general purpose operating system it had to allow processes anywhere in the tree to share information.

Effective use of the connection tree: The main reason for many of the architectural decisions in X-TREE was to reduce traffic in the tree (especially the decision on topology). Therefore one of our major goals was to minimize the traffic due to the operating system.

Process migration: As a step toward reducing traffic caused by user processes, we added the requirement that processes be able to migrate across the system to nodes which would minimize traffic in the tree.

The rest of this paper describes the XOS system and how it attempts to meet these criteria.

2. SYSTEM STRUCTURE

XOS is made up of five major modules (see figure 2.1):

1) the Microcoded Kernel (MK)
2) the Capability Manager (CM)
3) the Object Manager (OM)
4) the Directory System (DS)
5) the Command Interpreter (CI)

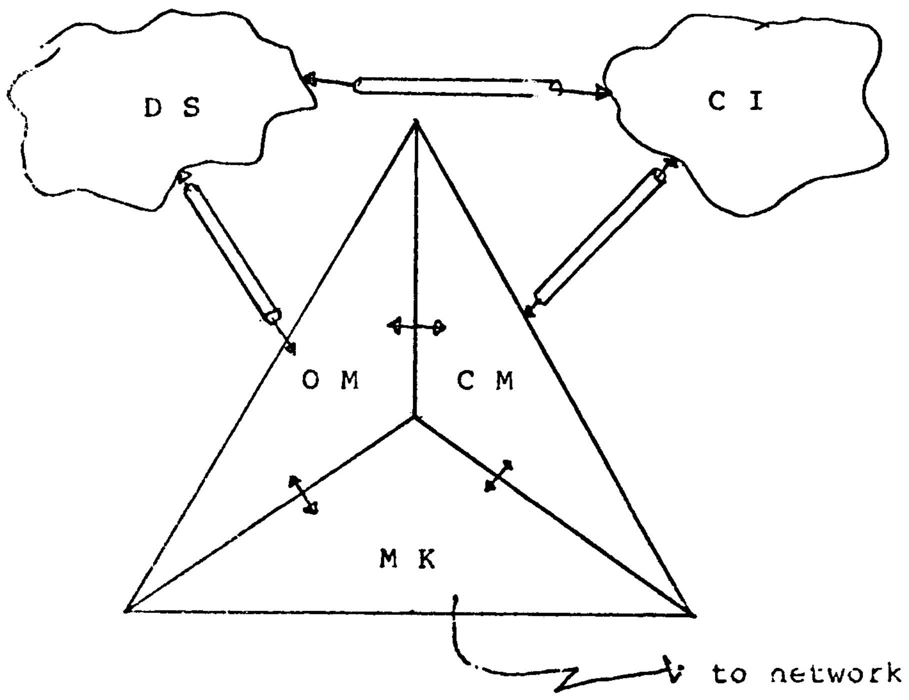

Figure 2.1

We chose to implement these particular modules in order to write a complete system that would cover the full spectrum of activity from user login to process execution. We had little desire to offer this to a user community since the actual hardware was not yet available.

The lowest level of the system is implemented in the microcoded kernel (MK). The MK serves as the direct interface to the processor and provides the rest of the system with an abstract machine which offers memory management, communications, and scheduling.

Above the MK are two modules that work in close cooperation. The Object Manager (OM) is called upon to handle all page faults, resolving them by retrieving the requested page. The Capability Manager (CM) checks access to objects and performs all bit level operations on capabilities. The OM and CM often send messages to one another since each performs a function the other needs, the OM fetching pages of C-lists for the CM and the CM resolving access rights for the OM. Both the CM and OM exist in the kernel address space since they need access into the representation of objects and capabilities.

The only user processes implemented are the Directory System (DS) and Command Interpreter (CI). These have no privileged commands and are therefore full-fledged user programs. The DS is a library of user routines used to provide an access structure to the object space. The CI is a terminal interface process used to test the system.

The remaining sections will describe these structures (except for the CI) and the concepts important to them.

3. ADDRESSING

X-TREE is an object-based system, with capabilities used for addressing [Fabr74]. While capabilities in X-TREE are similar to capabilities in previous systems [Wulf74, Need77], various extensions and changes have been made to work within the X-TREE architecture.

3.1. Objects

The basic addressable unit of X-TREE is the *object*. Data, programs, processes, directories, files, and ports are all types of objects, and the actions of sending a message, starting a process, accessing a file, or storing data all involve operations on an object.

The address space of X-TREE is a virtual object space. All objects (except ports and processes) reside at the leaf nodes of the tree, and all processes in the tree have equal access. This is similar to the virtual segment space in Multics [Bens72], in that Multics segments form a uniform address space, potentially available to all processes. In Multics, a segment may be in main memory, on a swapping device, or on secondary storage. In the X-TREE system, an object may reside on secondary storage (at the leaf nodes of the tree), or in a processor's (internal node) local memory. Processes residing in any node of the tree can address any object in the system, independent of their logical or physical locations. XOS maintains its segments in a uniform object space, allowing organizational groupings such as directories to be provided by systems running above the operating system kernel (i.e. the Directory System).

3.2. Capabilities

All addressing of objects in X-TREE is performed via *capabilities* [Fabr74]. A capability is the unforgeable key that is required to access an object. It may reside only in objects. Capabilities serve three main functions: addressing, data abstraction, and protection. A capability consists of a triple:

<address> <access rights> <object type>

Except for the address field, the capability in XOS is almost identical to those in HYDRA [Cohe75]. Therefore we will focus only on the address portion and refer the reader to the HYDRA papers for a complete description of capabilities.

3.3. Addresses

The address field of a capability defines the unique address of an object. The address is in two parts:

<global node address> <local node address>

The *global node address* is the address of the particular node of the tree on which the object resides, and the *local node address* identifies the object within the node. One of the basic goals of X-TREE is to avoid the limited addressing range of many previous architectures. Object and node addresses are both variable length; there are no intrinsic limits on either the size of the tree, or the number of objects local to a single node. In principle, the address space is indefinitely expandable.

A global address, which identifies a node within the tree, is encoded in the standard X-TREE notation that is used by the message routing hardware [Sequ78]. Since all standard objects reside in the leaves of the tree, their global addresses always name leaf nodes. For ports and processes, the global address names the node in which the process or port was created.

It is important to note that the global address is the address of a *physical* node within the tree. This means that a given object must reside on a particular node (e.g., a particular disk drive or set of disk drives). The case where a section of the tree fails and data must be moved (e.g., mounted on a different leaf node's disk drive) cannot be handled by the current structure.

3.4. Local address translation

Each node has on the order of 64K bytes of local memory. Part of the memory contains kernel code, and part of it is managed as a cache for the data and capabilities being referenced by the processes on that node.

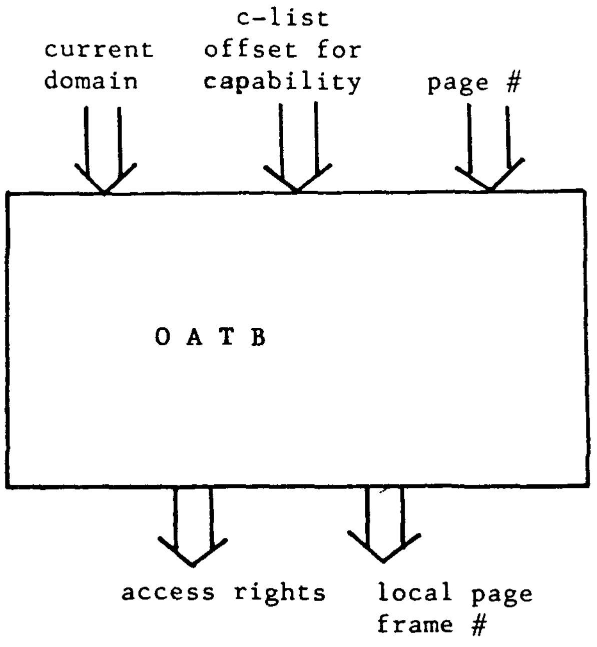

Figure 3.1

Each node must keep a list of pages currently residing in the local memory, and the objects to which they belong. Since each capability address is variable length, and on each data or capability reference the page cache must be searched, hardware support is provided to increase efficiency. Each capability reference is actually an index operation (offset) into the c-list for the current domain (see PWO description in section 4). There is an associative memory (called the Object Address Translation Buffer, or OATB [McCr80]) that translates the current domain, c-list offset for the capability, and page number (from the offset portion of the address) to the access rights and local page frame number. This is illustrated in figure 3.1.

3.5. Pre-fetching and forwarding

Objects residing at the leaf nodes are stored on mass storage devices such as disk. It is the responsibility of the Object Manager (OM) to access pages from disk and transfer them to the appropriate node. When a request for a page of an object originates from a node in the tree, the request is sent to an agent for the OM residing in that node. The page request is then forwarded to the proper leaf node where the OM accesses the page and sends it to the requesting node.

The OM attempts to optimize disk accesses by pre-fetching pages. Different pre-fetching strategies are used depending on the type of access. For example, access to an object containing executable code would typically use a working set organization [Denn80], while access through a sequential file would always attempt to have the next page available. The proper strategy is selected by examining an history of accesses.

For synchronization and shared access when an access request is made (at the leaf node) to a shared page whose latest version is not currently resident at the leaf node, the request is forwarded to the node that actually has the page. The pages is then sent to the requesting node. Each time the latest version of the page moves to another node, a message is sent to the leaf node that is the home for that page, notifying it of the new location. If a request is forwarded to a node that no longer has the page, the request is forwarded on to the node that does have the page. After a certain number of forwardings, the request is returned to the leaf node.

It is important to note that the OM that is resident on each node is not the same. Non-leaf nodes contain only an agent OM which sends its requests to the OM's on the leaf nodes containing the secondary storage devices. This greatly decreases the size of the kernel in non-leaf nodes.

3.6. Structure of addressing an object

Revocation of access to an object once a capability has been issued has been a problem in previous capability-based systems. A refinement of this problem is the ability to change (reduce or increase) the access rights to an object after the capability for that object has been distributed. One solution to this problem is to use a resource manager. Each reference to some type of object is through this resource manager. It is up to the manager to decide whether a given access request should be honored. Using this organization, complex criteria may be used. A drawback of this approach is that for simple accesses, the resource manager may be much too slow. A different solution to the revocation problem was proposed by Redell [Rede74]. This method was applied in XOS as follows.

X-TREE capabilities come in two forms: direct and indirect. Each object has, as part of its structure, an *indirect* field. This field consists of a special capability associated with the object. A direct capability ignores the indirect field, and behaves as described previously. An indirect capability addresses the object through the capability stored in the indirect field. It is possible then, to invalidate the indirect field of an object and thereby revoking access to anyone with an indirect capability for that object. The access rights of the capability in the indirect field may be modified causing the access rights of anyone with an indirect capability to be changed. If the indirect field is substituted with a capability for another object, this acts as a renaming function.

4. PROCESSES

4.1. Processes and Messages

XOS uses the paradigm of processes communicating via messages and message streams. A process exists on only one processor at a time although it may migrate from one processor to another during its existence. The process is described in its entirety by the Process Work Object, defined below. The same types of communications mechanisms are used both by processes for communications and by the disk system for paging traffic. As mentioned above, all data, instruction, and device accesses are treated as object accesses and therefore use the same paging mechanism.

4.2. Process Work Object (PWO)

The PWO contains all information for defining an object including stack pointers and registers. A single capability (pointing to the PWO) describes a process. There are two reasons for this compact representation. One is to provide an object understood by the processor that can be used to provide fast context switches (similar to the VAX's System Control Block [DEC78]). The other reason is to facilitate the migration of processes. The act of swapping the PWO to disk and then back up to another node (or directly to the other node) is sufficient to migrate the process to another processor.

Like other objects, the PWO contains both a data part and a c-list. This is necessary since the program counter (PC), the current frame, and execution stack all are made up of a capability section and a data section. For the PC, the capability points to the current program object and the data part is the offset into the object of the current instruction.

Figure 4.1 depicts the PWO. The beginning of the data part contains:

- 8 32-bit general purpose registers
- the offset to the top of the data part of the execution stack
- the offset to the top of the capability part of the execution stack
- the offset to the beginning of the data part of the current routine frame
- the offset to the beginning of the capability part of the current routine frame
- the offset into the current program object of the current instruction

The first capability in the c-list is that of the current program object. These locations are actually shadows of physical registers resident in the processor. Whenever a PWO is written out, these shadows are filled in by the processor microcode with the values of the corresponding registers.

Starting a process is performed by loading the process pointer (PP) with the capability for its PWO. The microcode then loads the registers from the shadows. The actual loading of the PP is performed by the dispatching hardware, which is described in the scheduling section.

4.3. Inter-Process Communication (IPC)

Communication in XOS is similar to DEMOS [Bask76]. It is message-based, unidirectional, and capability accessed. Remote and local communication appear the same to processes.

PROCESS WORK OBJECT

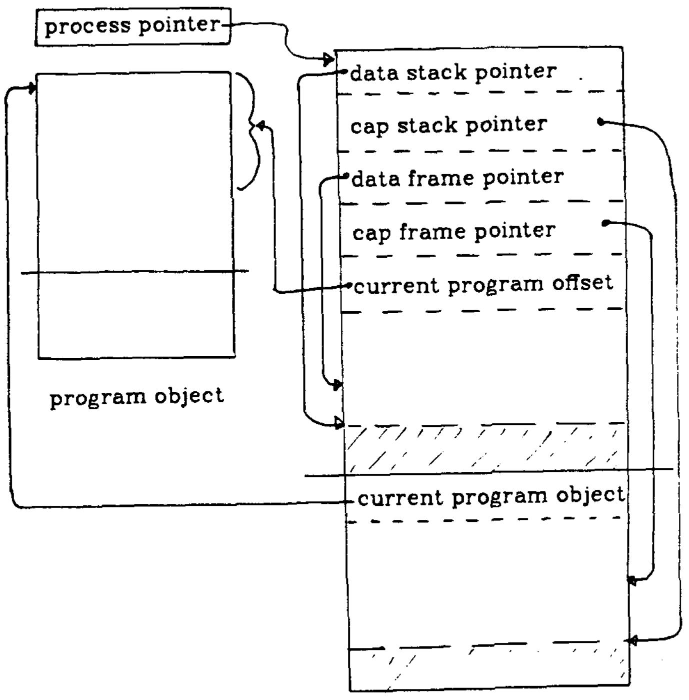

Figure 4.1

All messages are sent to a special object called a port. Ports are owned by processes. When a process wishes to receive messages it creates a port object and passes send capabilities for that port to other processes. It can do this either by saving the capabilities in commonly accessible objects or by handing them off to a switchboard process with whom every process can communicate. If two-way communications are to be established, the other process can send back a port capability with which to talk. The actual port object exists in the global space and contains the node location of the process that created the port.

Messages contain both data and capabilities. This allows the transfer of capabilities between processes. Messages can be sent either in *datagram* or *virtual circuit* mode. The receiver has no idea in which mode the messages are being sent. *Virtual circuits* provide sequenced data streams with routing performed only once for the whole stream. A process may set up sequenced communications by sandwiching the message stream between two special system calls which open (set-up) and close (tear-down) a *virtual circuit.*

Remote process to process communication corresponds directly to physical circuits. *Virtual channels* comprise a uni-directional message stream traveling over a physical circuit, with a return stream in another physical circuit providing acknowledgments for the messages. Software is used to guarantee sequentiality and to make circuit time-outs invisible to the processes. When a circuit times out, a special tear-down message is sent by the communications software to free the circuit. *Datagrams* are messages enveloped by set-up and tear-down messages. Again a return path is provided for acknowledges. Time-outs are not a factor since a tear-down follows the message. Protocols in the *virtual circuit* case are negative acknowledge. In the case of *datagrams* every message must be acknowledged.

Since the number of possible physical connections is finite, communications may become saturated despite the availability of time-out tear-downs. A Statistical Time-division Multiplex (STDM) channel is reserved in each node to allow message flow in the case where the channels become saturated. By convention, only datagrams are sent over this channel.

4.4. Process Scheduling

XOS process scheduling is very primitive, dealing only with a priority microcoded scheduler. All processes are allowed to run until either they complete, they block awaiting resources, or a higher priority process preempts. This simple scheme is felt to be sufficient since the expected number of processes per node is small – on the order of three or four.

The kernel microcode implements instructions for linking and unlinking processes on 8 priority levels. The processes run round robin at each level, each running until it blocks. The highest priority process always runs. The priority queues are linked lists, each element containing a pointer to a PWO resident in the node's memory.

4.5. Process Migration

One of XOS's objectives was to undertake investigations of process migration from node to node in the system. Process migration could be used in attempting to dynamically move processes closer to its data in order to reduce message traffic. To facilitate process migration, the representation of a process (its state) was designed to be easily transported from one processor to another. A process can be completely represented by its PWO. Since the PWO is an object, it can be moved between processors, thereby moving the process to another node. The problem is more complex when port objects are involved. When a process moves, all processes communicating with it are now pointing to the old node. However, in XOS, all such pointers are treated as hints. If a message send arrives at the wrong node, a special negative acknowledge is sent to the sender. The sending node's kernel then retrieves the actual location of the port by reading the port object itself, a copy of which is kept on disk. After getting the new hint it tries the send again. This process continues until the message catches up with the roving process.

5. DIRECTORY SYSTEM

The XOS Directory System [Mill79] is included in the XOS design for a number of reasons. First, it is an example of a user-level (non-kernel) protected subsystem. This provided an application to test the low-level parts of the operating system. Second, the Directory System supplied a name mapping mechanism. And last, the Directory System provided a data organization facility.

5.1. Directory System as a non-unique, user subsystem.

The DS is an example of a user-level, protected subsystem. Operations on objects of type *directory* are control by the DS. It should be noted that the DS is a "non-file system". A directory is merely a list of capabilities for some objects, and any object whose capability is contained in a directory is termed a *file*. The directory system only provides access control to the capabilities for files; not to the files themselves. There is no concept of opening or closing a file. This could be built as another subsystem which makes use of the DS.

5.2. Directory structures as a forest

A capability for a directory object might be contained in some other directory-type object. This forms a *subdirectory* structure. Access to subdirectories and files in subdirectories is as in MULTICS or UNIX [Salt74, Ritc78]. It is possible to build directory structures that form arbitrary directed graphs, but this is not desirable for maintenance purposes, and so is prevented. All directory structures are restricted to trees.

For each DS function, the user may supply a "starting directory", which forms the root of the tree for that particular operation. This allows the user to have a number of totally independent directory structures, or a forest of structures.

5.3. Directories as a collection of objects.

A major role of the DS is that of providing a name mapping mechanism. Associated with each capability (file) in a directory is a symbolic name for that object. The naming function, incorporated with a syst^m directory (one to which each user has access), provides a mechanism for allowing objects to be available throughout the system. Accessing such facilities as editors and compilers, or establishing initial connections for communications is done through the directory system. In this role, it similar to the "switch board" in DEMOS [Bask77].

6. CONCLUSION

We have presented an operating system for the X-TREE architecture. XOS is a capability-based system because the communications structure easily facilitated capability addressing. A number of structures, among them the OATB and PWO, have been proposed to assist an architecture running a capability-based operating system.

The uniform, global address space simplified the structure of the operating system over multiple processors. Each processor had a consistent and equivalent view of the address space. The ability to influence the CPU design (microcode) allowed an efficient implementation of many of the O.S. primitives.

Communications channels were used for both message and paging traffic. This allowed common usage of a single mechanism, thus simplifying the overall structure.

It was also demonstrated that the directory structure is easily separated from the object name space allowing the possibility of several separate access structures to be implemented on the same system.

The X-TREE structure facilitated the rapid construction of the O.S., but the system proved larger and more complex than may be appropriate for the VLSI implementation (24,000 lines of high-level language code). The system was constructed by 6 people in eight months, at which time the basic system, a command interpreter (shell) and directory system were working.

A number of problems still exist in the system. The physical dependency of object names makes it difficult to move objects to different locations in the tree. This can prove to be awkward when expanding the tree or when parts of the tree fail. A number of solutions have been proposed during our initial implementation and may be tried out in the future. The migration of processes has been made possible and even easy. However no attempt has been made to formulate an algorithm for determining to where to migrate them. This is also a topic for future research.

7. Acknowledgements

This report reflects the work of a number of people besides the authors. In particular, we would like to acknowledge John Lisowski for his work on the Capability Manager, Edie Merritt for her work on the Object Manager, Gordon Spencer for his work on the Command Interpreter, and Tim McCreery for his work on the Kernel.

We would like to thank Al Despain for his guidance during the course of the project, and Michael Powell for his help during the preparation of this report.

8. REFERENCES

[Bask76] F. Baskett, J.H. Howard, & J.T. Montague, "Task Communication in DEMOS," *Proc. Sixth Symposium on Operating Systems Principles*, appearing as *Operating Systems Review* **11**, 5 (November 1977), pp. 23-31.

[Bens72] A. Bensoussan, C.T. Clingen, & R.C Daley, "The MULTICS Virtual Memory: Concepts and Design", *Comm ACM*, **15**, 5 (May 1972), 308-318.

[Cohe75] E. Cohen, D. Jefferson, "Protection in the HYDRA operating system", *Proc. of the Fifth Symposium on Operating Systems Principles*, November, 1975, 141-160.

[DEC78] *VAX 11/780 Hardware Manual*, Digital Equipment Corp., 1978.

[Denn80] P.J. Denning, "Working Sets Past and Present", *IEEE Transactions on Software Engineering* **SE-6**, 1 (Jan. 1980), pp. 64-84.

[Desp80] A.M. Despain, "The Architecture of Multiple Microcomputer Systems", *Proceedings of the National Communications Forum of the National Electronics Consortium*, 1980.

[Ditz80] D.R. Ditzel, "Investigation of a High Level Language Oriented Computer for X-TREE", *Masters Thesis*, University of California, Berkeley, 1980.

[Fabr74] R.S. Fabry, "Capability-Based Addressing," *Comm ACM*, **17**, 7 (July 1974), 403-412.

[McCr80] T. McCreery, "The Development and Simulation of the XOS Kernel", *Masters Thesis*, University of California, Berkeley, 1980.

[Mill79] B.P. Miller, "X-TREE Operating System: The Directory System and the Role of User Processes", *Masters Thesis*, University of California, Berkeley, 1979.

[Need77] R.M. Needham, A.D. Birreel, "The Cambridge CAP Computer and its Protection System", *Proc. Sixth Symposium on Operating Systems Principles*, West Lafayette, November, 1977.

[Patt79] D.A. Patterson, E.S. Fehr, & C.H. Sequin, "Design Considerations for the VLSI Processor of X-TREE", *Proc. of the Sixth Annual Symposium on Computer Architecture*, April 1979.

[Pres79] D.L. Presotto, "X-TREE Operating System: Simulation, Disk Consistency, and Communication", *Masters Thesis*, University of California, Berkeley, 1979.

[Pres80] D.L Presotto, "X-TREE: The Node, Network, and Operating System Architectures", *Proceedings of the National Communications Forum of the National Electronics Consortium*, 1980.

[Rede74] D.D. Redell, "Naming and protection in extensible operating systems", *Ph.D. Thesis*, University of California, Berkeley, 1974.

[Ritc78] Ritchie, D.M., and Thompson, K., "The UNIX Time-Sharing System," *The Bell System Technical Journal* **57**, 6, Part 2, pp. 1905-1929.

[Salt74] J.H. Saltzer, "Protection and Control of Information Sharing in MULTICS", *Comm ACM*, **17**, 7 (July 1974), 388-402.

[Sequ78] C.H. Sequin, A.M. Despain, & D.A. Patterson, "Communication in X-TREE, a Modular Multiprocessor System", *Proc. of the ACM 1978 Conference*, December 1978.