Approaches to Distributed UNIX Systems

Jonathan M. Smith

CUCS-241-86

Jonathan M. Smith
Columbia University
New York, NY 10027
2 April 1986

This research has been supported by the Defense Advanced Research Projects Agency under contract N0039-84-C-0165.

Table of Contents

List of Figures

Abstract

This paper examines several approaches to developing a distributed version of the UNIX[1] operating system. Relevant UNIX concepts are introduced and brief overviews of a number of distributed UNIX implementations are provided.

The major issues discussed are concurrency and file system architecture:

1. The multiprocessor designs examined have the common problem of implementing critical sections in the kernel; several approaches are studied.
2. The file system architectures studied reflect differing views as to how the UNIX name space should be implemented in a distributed environment, and what the names will refer to; the issue of transparency is central to both, and is examined in some detail.

[1]UNIX is a trademark of AT&T Bell Laboratories.

1. Introduction

1.1 Definitions

A *distributed UNIX system* is a *distributed system* which runs the UNIX operating system. The UNIX system may consist of several independent cooperating UNIX systems, or the operating system may be distributed over several processing units.

We define a *distributed system* to consist of two or more communicating autonomous processing units[2]. They are autonomous in the sense that a processing unit can perform processing tasks independently of other processing units. For example, the channel and central processing unit (CPU) which comprise a mainframe are not considered a distributed system under this definition; the restriction is made on the basis of the degree of autonomy enjoyed by each processing unit.

In addition, the processing units must be able to communicate in some fashion. Examples of communication methods are shared memory[3], local networks, local area networks, or long-haul networks.

We adopt (with some distinctions due to Deitel [22]) the categorization of distributed systems introduced by Stone in [96]. The differences between these system organizations are usually a function of the ratio of communication bandwidth to memory bandwidth; this decides the organization of the UNIX system which will utilize the hardware. Bandwidth can be defined as the rate at which data can be transferred from point to point; thus, memory bandwidth would be given by the rate at which memory to memory copying can take place. These categorizations are *tightly-coupled*, *closely-coupled*, *loosely-coupled*, *connected*, and *miscellaneous*. Each is discussed in more detail below.

In *tightly-coupled* distributed systems, memory is a shared resource. Since bus timing is often an issue in such systems, the systems are spread over a small area, dictated by signal propagation delay[4]. The shared memory is addressable by all processors; the bus or switch logic which supports this is typically rather complex, in order that certain instructions used for concurrency control work in an atomic (indivisible) fashion. While Direct Memory Access (DMA) devices[5] work in this fashion, we have not included them in our definition. *Multiprocessors* are tightly-coupled distributed systems consisting of a multiplicity of homogeneous CPUs. [7] states, "Multiprocessors are computers that contain two or more processors capable of independently executing instructions and gaining access to programs and data held in a common memory". The only private memory is typically a cache local to each processing unit.

---

[2]A processing unit is another name for a *processor*, which "carries out computational work by retrieving instructions from memory and performing operations on data that were also retrieved from memory." [7]

[3]Examples are shared bus, crossbar-switch matrix, and multiport storage (see [22]).

[4]Electronic signals are subject to the formula Distance = Rate * Time, and they travel at a fixed rate, so that the time taken to travel from point to point is directly proportional to the distance between those points. Note that due to the propagation delay (i.e. the time it takes a signal to go from A to B) if A requires B to acknowledge the data sent by A, then unless each transfer involves a large block of data, the performance will be poor for large distance between A and B.

[5]Direct Memory Access means that an Input/Output Device attached to a bus can carry out data transfers to memory also attached to that bus without requiring intervention by the Central Processing Unit.

Sometimes processors are equals, e.g. IBM 3084 [94], and other times a master-slave relationship is enforced, e.g. AT&T 3B20A [4], DEC VAX-11/782 [23, 50]. These latter designs are often referred to as *attached processor* configurations. Further discussion of these designs and terminology is given later in the paper. A distinction in the case where there are many microprocessors combined as a multiprocessor is made by Bell in [7, 8, 9]; he refers to these systems as *Multis*. Some multiprocessors have many more processors than others [26, 60, 93, 95] but these machines are experimental and hence not in general use; at present massive parallelism seems difficult to exploit. All multiprocessor systems have the characteristic that memory bandwidth is equal to communications bandwidth.

Closely-coupled distributed systems are distinguished from tightly-coupled systems by the amount of memory which is shared. Only a portion of the addressable memory is common to the connected processors. Closely-coupled distributed systems share with tightly-coupled systems the property that the memory bandwidth and communications bandwidth are equal, but have some possible advantages and disadvantages. There are several possible advantages. First, there should be less bus contention, thus speeding the average memory access time. This is because the private memory references do not have to contend with other processors. Second, the private memory will make caching more effective, since global cache invalidations affect only shared memory. Third, the systems architects can take advantage of the fact that fewer accesses to data structures have to be synchronized. Finally, a distinction between memory for communication and memory for private data storage should aid structuring of applications. The major disadvantage is the complexity inherent in the addressing logic for such systems, both in hardware and software.

Loosely-coupled distributed systems consist of several independent, interconnected computers, each containing a processor. There is no shared memory. The connection is achieved by some sort of data communications mechanism, e.g. data-switch, channel-to-channel links, and *local-area* networks. A variation on this scheme requires the aggregate to resemble a single computer system. These are referred to as *netcomputers* [32, 33] or *multicomputers* [7]. A *local-area* network is one which is high speed (on the order of 1-10 million bits per second or more), and distributed over a relatively small geographic area, say on the order of 10 square miles. Examples abound, such as Datakit [19, 20, 34], ETHERNET [69], and the Network Systems Corporation HYPERChannel [35]. Here, communication bandwidth is less than memory bandwidth, but sufficient for providing reasonable response time for distributed operations. We restrict ourselves to *local-area* networks in this discussion.

Connected distributed systems are systems in which the slow speed of the communications channel effectively prevents them from being used as general distributed computing environments. While long-haul networks are inappropriate for connecting a loosely-coupled system on the basis of response-time considerations[6], they are entirely adequate for many communications tasks which are not as sensitive to round-trip delay. The local nodes are characterized by more autonomy than previously mentioned designs, although the loosely-coupled distributed systems mentioned previously may display quite a bit of heterogeneity. These connections are more often used for transferring files and electronic mail applications than sharing of computational workload. The heterogeneity of systems coupled with the

[6]Note, however, that the bandwidth of the communications subsystem is not the only determining factor of the performance of a distributed application: it only provides an upper bound on information flow. See [54] for an interesting study.

communications channel properties make most distributed systems applications inappropriate for this environment. There are some exceptions, such as systems involving distributed databases [11, 87] Examples of connected distributed systems are ARPAnet [99], CSNet [14], and USENet. The communications bandwidth is *much* less than the memory bandwidth of individual CPUs. (There are exceptions to the bandwidth criteria, e.g. ARPAnet satellite channels, but this is a case where the exception defines the rule, as satellite communications is characterized by long round-trip delays.)

Miscellaneous systems are systems which have attempted to communicate via some means such as shared disk drives; it is hard to characterize these as communications media. These systems (e.g. VAXCluster[7]) are however useful if a large pool of disk storage must be accessible to several processors; this is often the most effective way to share large amounts of data.

Another, more detailed taxonomy can be found in [2], and an interesting discussion of multiprocessors based on VLSI in [91].

1.2 Motivation

There are many motivations for distributed systems, and the study of them. Among the more important [76] are:

1. **increased reliability.** This potential increase in robustness is due to several reasons. First, there need be no *single* point of failure, although designs which rely on central service administration or control points are subject to this. Second, a system which balances load across processors effectively can offer graceful performance degradation under increased system workload. (Note that this is not always true. Since balancing algorithms may require significant CPU and communications resources, it may actually make the degradation more rapid.) Dynamic work placement algorithms can gracefully handle component failure conditions [5]. Third, MTBF can increase as the product of the MTBFs of individual components in a properly designed system. Fourth, the effects of local disasters can be lessened by geographic distribution. Fifth, in distributed systems where data are replicated, an extra measure of disaster protection is given.

2. **increased performance.** Distributed systems designers are attracted by the potential of distributed systems for increased performance. This potential exists in several areas.

   - Monoprocessor architectures have crept ever nearer to an upper bound on instructions per second (IPS)[8]. Thus, designers look to parallel processing as a means to achieve a greater *aggregate* performance, in IPS or other metrics. Supercomputers often disguise a somewhat distributed architecture, e.g. pipelining, vector processing, and the CDC 6600's Peripheral Processing Units (PPUs) [96].

   - Access to large resource pools (e.g. large databases) is often desirable. Local access reduces communications delay. Replicating some or all of the data in such a database by keeping a copy of the database at each local site provides potentially good performance if the database isn't changed often; otherwise the communications overhead of updates to maintain consistency may quickly exceed that of reading data.

---

[7]VAXCluster is a trademark of Digital Equipment Corporation.

[8]As mentioned above, signal propagation delay is a function of distance. Thus, an upper bound can be calculated for the speed of a simple logic circuit by using the speed of light in a vacuum as a propagation velocity, and molecular diameters for the sizes of electronic junction dimensions.

Multiple copies of a large database are wasteful if copies exist at each site, as it requires expensive local disk resources. Distributed architectures balance tradeoffs between cost, reliability, geographic, and other constraints in order to provide acceptable performance.

- Application-specific processors typically outperform general purpose processing units by several orders of magnitude on their assigned tasks, and yet may need general purpose facilities in order to provide a useful operating environment. For example, channels are ideal for handling asynchronous communication because they can field interrupts effectively, but they cannot easily provide sophisticated operating system services.
- A problem can be too large for a monoprocessor. Such a problem must be solved by cooperating communicating modules.

3. **lower cost/performance ratio.** Economic justifications are always strong ones, and several of these are very compelling.
   - Components may be very cheap, e.g. microprocessors (which reap the economic benefits of mass production) can be connected with busses to form multiprocessors [6, 7, 30, 44], with local area networks to form netcomputers [32, 33, 80, 81], or in several other fashions [91]. In contrast, several components of supercomputers [96] used to achieve architectural goals can be very expensive, e.g. exotic high-speed cache technology, fast main memory, and lookaside buffers.
   - If performance requirements exceed a certain point, cost may be less of an issue. This is also true of reliability. Examples are the Safeguard ABM system [16, 40, 73], and the AT&T 3B20D [103], used for phone switching applications.
   - Networking hardware and software may already be in existence; creating a distributed system may only require adding an interface layer of software, thus achieving many benefits without major capital investment.
   - Resource-sharing can amortize the cost of an expensive, necessary, and possibly underutilized resource (e.g. a tape drive or line printer) over the price of several processing units.
4. **linear speed-up.** There is the attractive promise of linear speed-up, which means that there exists some coefficient which when multiplied by the number of processors will give a performance figure. Hence, the availability of an arbitrary number of processors could lead to an arbitrary performance gain. While this sort of speed up is the theoretical maximum when the coefficient is one, it is easily seen that even fractional coefficients of reasonable magnitude are desirable. A better way of looking at linearity may be in terms of the cost of the processing units, as the tradeoff between communications overhead and quantity discounts may be sufficient to make the coefficient attractive.
5. **incremental expansion.** Distributed systems offer another potential advantage in terms of cost. An organization or enterprise may be expanding over some period of time and may want their computing resources to expand along with them at the same pace [57]. In a well-designed distributed system small pieces can be added to the system, e.g. more microprocessors or disks, as the needs of the organization change, *without* the huge transition costs of changing mainframes, which may involve major software and hardware upgrades, or even a change of vendor.
6. **structuring advantages.** Paul in [76] notes that there are several structuring advantages that can be realized by a distributed system. For example, the system by its nature may be geographically distributed, for example Automatic Teller Machines and defense warning system sensors. The warning system sensors are an example of an application where the

> signal strengths force *in loco* processing, although the virtue of multiple geographically distributed sites for the sake of robustness is important. Another structuring advantage is realized by applications where each parallel process is independently located in an autonomous processor.

For the purposes of this paper, we will not restrict ourselves to the study of distributed systems of any particular architecture but will rather cover only systems that run an operating system based on UNIX [84, 85], and run the distributed UNIX system directly on the configuration. What we are interested in is the changes which have been made in UNIX in order to handle a distributed environment.

Note that this definition excludes the AT&T 3B20D[9].

First, we will describe some mechanisms in UNIX[10] of particular relevance to implementing distributed UNIX systems. These mechanisms are used for concurrency control, implementing co-routine logic in the kernel, demountable volumes, and path name interpretation. We will then look at some multiprocessor systems. These systems are characterized by the desire for increased performance through use of multiple independent processors. The discussion centers on changes in the kernel logic which are necessary to support multiple processors. Reliability as a function of the multiplicity of processors seems to be a secondary concern in most current designs. Finally, we will examine several distributed UNIX file systems. The file systems are interesting because they permit sharing of economically expensive system resources such as disk storage between geographically distributed UNIX machines[11]. The different approaches are examined in light of an issue which is central to distributed file system implementations, namely that of transparency.

---

[9]The AT&T 3B20D [103] is a fault-tolerant architecture consisting of two equivalent CPUs; in the event of a CPU failure the other (idle) CPU takes over the processing [41]. However, the fault tolerant aspects of this system are not handled by the UNIX Operating System. UNIX runs as a supervisor process under DMERT [39, 49], the system which actually deals with failed processors. UNIX is oblivious to the failure and recovery procedure.

[10]The mechanisms we will examine have undergone very little change in the various common UNIX variants such as 4.2BSD, System III, Version 7, System V, Xenix, etc. Hence, the description at the level given is valid for all of these variants on the version described in the Ritchie & Thompson article [84].

[11]Another potential advantage is that a distributed file system (DFS) can remove architectural limitations on the amount of disk storage supported. For example, a certain manufacturer's I/O subsystem may support a maximum number of disk controllers, and the controllers support a maximum number of disk drives, thus giving an upper bound on the disk storage provided by the system. However, when several systems are combined, the aggregate disk storage available and accessable can exceed this bound. This is particularly relevant where huge databases are necessary to support applications; these can and do range beyond the multi-gigabyte range.

2. Monoprocessor UNIX

This section describes techniques used in UNIX systems to achieve various resource management goals. The purpose of the section is provide sufficient background in the details of UNIX operation for later discussion. The background material is by no means complete; points are covered solely for reasons of differentiating mechanisms used in distributed versions of UNIX.

2.1 Mutual exclusion

Critical sections are sections of code which must be executed indivisibly; that is, without interruption and therefore without intervening change to the data structures that are operated upon by the code in the critical section. They are used to implement certain actions which must be performed atomically in order that a multiprocessing kernel can both perform changes and guarantee consistency (i.e. ensure that certain constraints are met). UNIX uses a technique for implementing mutual exclusion which does not port well to multiprocessors. The technique used is to raise the processor priority level before entering a critical section. This ensures that the executing code will not be preempted by any interrupts at or below the selected level. For example, when data structures which would suffer from destructive concurrent access are referenced in a disk driver, the priority level is raised to a point at which no other disk interrupts can affect the processor, while higher priority interrupts, e.g. hardware clock interrupts, remain enabled. (For a more detailed example, see the discussion of a section of the teletype driver code in [4].)

2.2 Implementing Coroutines

The sleep() and wakeup() calls are used throughout the UNIX kernel to voluntarily relinquish control of the processor and to alert other processes waiting on an event[12] that the event has happened. The flow of control is illustrated in the following figure: An *event* is the address of some integer[13]; processes which want to block awaiting some later event call **wakeup()** with an address as an argument. **Wakeup()** then examines the process table entries to determine which processes are blocked awaiting completion of that event (by comparing the address passed with one stored with the process), and marks all such processes as "runnable".[14] This ensures that there will be a process to execute when the current process executes **sleep()**, which takes an address (the one later used by **wakeup()**), stores it with the process, and then yields the processor.

This mechanism would be used, for example, in a disk driver. A user process has issued a **read()** system call, and the system does not have the data available. The system allocates a system buffer to the process, initiates a call to the disk device to fetch the desired block of data, calls **wakeup()** using some address (it has not yet relinquished the processor) and then calls **sleep()** using the address of the first word of its system buffer as an argument. When the disk driver reads the block of data and has transferred it into the system buffer, the process (and maybe others) is awakened; the process is known by the address

---

[12]These processes had previously called sleep() with an event as argument

[13]An example of such an integer is the first word of an *i-node* table entry, or the first word in a buffer used for block I/O.

[14]Note that all the processes which have been waiting on the event are made runnable simultaneously. This means that the order in which they are serviced will be determined by the scheduling algorithm, rather than how long they have been waiting.

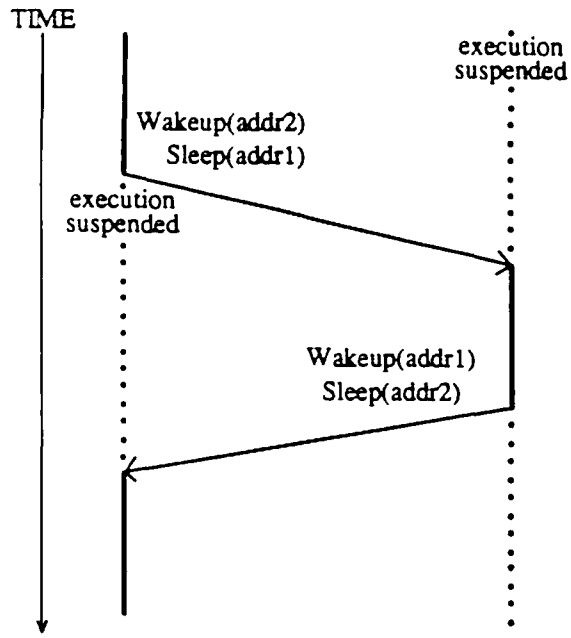

Figure 2-1: Coroutines: Sleep() and Wakeup()

of its buffer. It can then complete the **read()**, returning data to the user process.

Care must be taken to ensure that there is always at least one runnable process in the system, and that the users of **sleep()** and **wakeup()** calls preserve this condition, e.g. by always calling **wakeup()** prior to the sleep() call. Note also that many processes waiting on an event may be awakened several times; this is a fair policy, but under certain conditions can be wasteful, as a great deal of context switching may result.

2.3 Path Name Interpretation

A file in UNIX is identified uniquely by its *path-name*, where the path name consists of a path along the branches of a hierarchical file tree. All non-leaf nodes are directories. The path-name is the external handle of the file: it is the means by which users address the file. The internal handle of the file is the *i-node*; a file descriptor returned by **open()** provides access to a pointer to the *i-node*. There is an *i-node* associated with each path component. The *i-node* is the file control block used by the kernel to access the file contents; it contains such obviously useful information as the addresses of the disk blocks which comprise the file. The point of having such a 2-level naming scheme is increased system performance: the overhead of performing a search through the name tree is very high, and thus is performed once, when the file is opened.

The interpretation is performed by a routine in the kernel called **namei()** [62], which takes a character

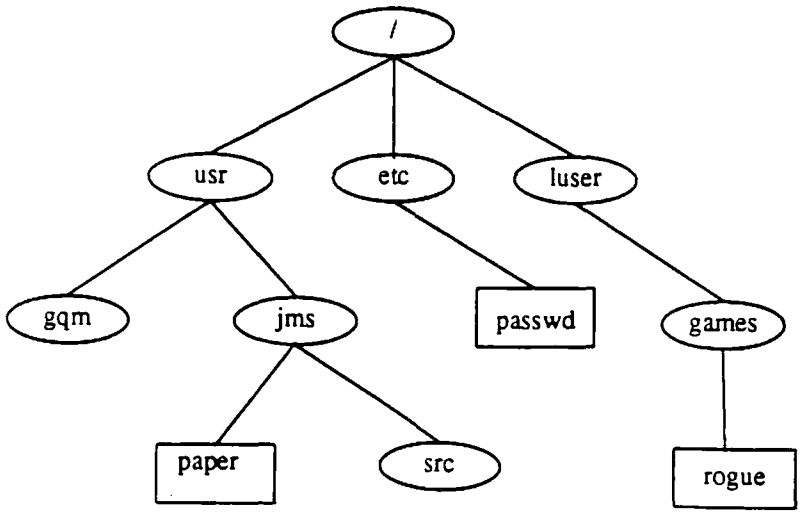

Figure 2-2: UNIX File Tree Illustration

string as an argument and returns a pointer to an in-core[15] *i-node* table[16] entry. The interpretation process starts by reading a directory (we'll call it the "interpretation directory") via information contained in the *i-node*, either the the root directory if the path-name is preceded by a "/" or the current working directory of the process (contained in an *i-node* table pointer in the process's user area). Directories consist of lists of <name,i-number> ordered pairs. The directory contents are scanned. When a name matching the current portion of the path-name is found, the *i-number* portion of the pair is used to fetch the *i-node* from the *i-list* of the file system it resides on, and place a copy in the in-core table. Note that "/"'s separate path-name components. If this is the last component of the path name, then **namei()** returns a pointer to the in-core *i-node* table entry it has just gotten. Otherwise, the process continues with the newly fetched *i-node* being used as the interpretation directory, which will now be scanned for the next component in the path name.

2.4 System Calls

UNIX implements system calls as operating system "traps": the system call is implemented as a processor instruction which is privileged and hence causes a fault when executed from user (unprivileged) execution mode [61, 62]. The processor traps the fault[17] and passes control to a routine handling traps via a processor interrupt vector. If the trap is of the appropriate type, the trap number is used to index a table of kernel entry points. The table consists of pointers to kernel functions which implement the appropriate

---

[15]While the technology of core memory is a bit dated, the name has come to indicate main memory in general usage; in the case of *i-nodes* the designation is used to distinguish it from the on-disk *i-nodes* in the *i-list*

[16]A table of active *i-nodes* is maintained in order to avoid referencing the disk for every action which would affect the information in the *i-node*

[17]It is easy to set up sets of privileged and non-privileged instructions; if the privilege level at which the processor is currently executing is available in hardware, instruction interpretation microcode can check the privilege level (available in, e.g. the VAX Processor Status Word [23, 24, 50] and invoke the microcode sequence which generates a processor fault.

system call; when the trap occurs, the flow of control passes to the selected function. The called routine then accesses required arguments through data preserved in the user area by the calling process.

2.5 Demountable Volumes

It is important to note that crossing file system boundaries involves special logic; a file is fully qualified in a monoprocessor UNIX system as <file system,*i-number*>: *i-numbers* are only unique within the context of the file system. Thus namei() must be aware of the current file system which is used to find an *i-node* using only its *i-number*. This extra state information is maintained in the *mount table*; there is one entry for each mounted file system. Since a file system can be mounted at any point in the directory hierarchy, the detection of boundaries has to be incorporated in the path name interpretation process.

The volumes ("file systems") have the following structure:

1. *super block* - This serves as the control block for the file system; it contains such data as the last mount time, the size of the file system, the size of data blocks, and the size of the *i-list*.
2. *i-list* - This is the area where *i-nodes* are kept; an *i-number* is a direct index into this list.[18]
3. *data blocks* - These are the contents of the files referenced by the *i-nodes* in a file system's *i-list*.

This structure is illustrated in the following figure:

[18]This is no longer true in 4.2BSD UNIX [68]. The logic is the same, but the *i-list* is distributed across the disk to improve seek times.

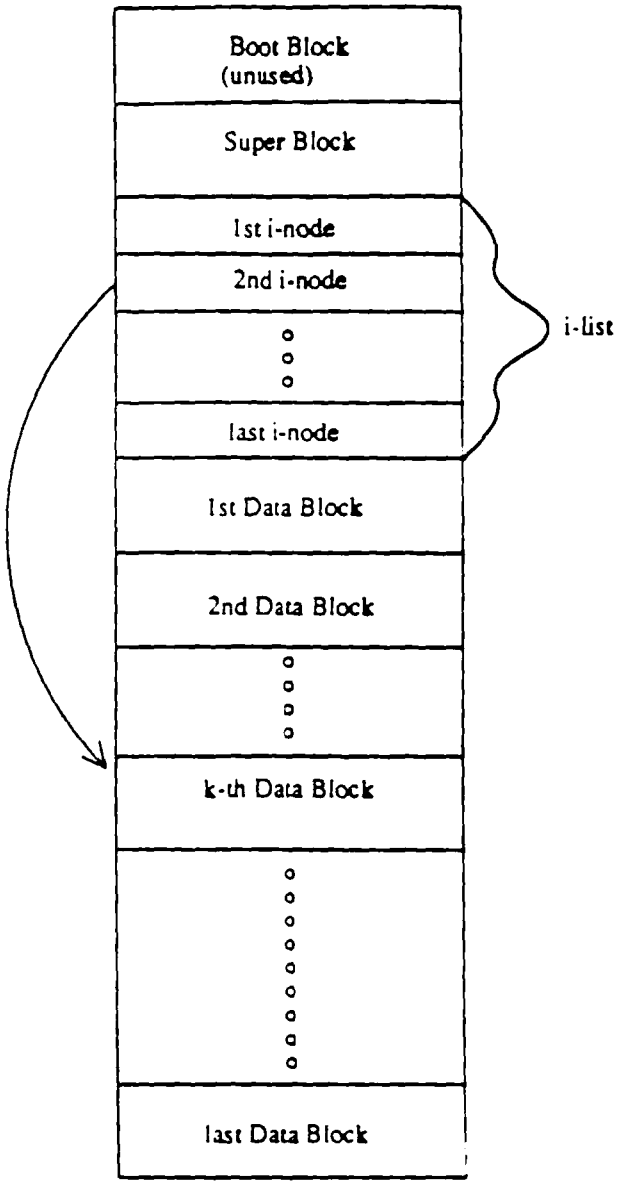

Figure 2-3: UNIX File System Organization

3. Multiprocessor UNIX Systems

This section will be confined to a discussion of systems consisting of autonomous processors with *shared* primary and secondary storage media. The processors communicate using a bus (thus implementing shared memory), and have access to all peripherals (which may or may not be on the bus), so that disks or other secondary storage media are a shared resource.

As a result of the layering of the UNIX architecture's file access mechanisms, they avoid or solve at lower layers many of the problems facing distributed file system designers which are discussed in later sections. For example, file access synchronization logic is important here, because a kernel can no longer assume that it is the sole interface to the local set of disk drives, since no disk drives are "local" to a processor. These problems are however subsumed by the concurrency control problems, and thus by their solutions.

3.1 True Multiprocessor Designs

"True" multiprocessor configurations allow each autonomous processing unit to perform I/O. They are distinguished from the Attached Processor designs mentioned above (and discussed below) which do not show this symmetry of "true" multiprocessors with respect to I/O.

While many designs for the underlying hardware exist, there are some common problems which must be addressed in the design of a UNIX kernel for a multiprocessor environment. These are [48]:

- mutual exclusion
- interrupt distribution
- interprocessor communication
- process dispatching (where to put runnable processes)

The most important of these is the problem of mutual exclusion [22, 64, 67], as achieved with critical sections defined by interrupt masks in monoprocessor UNIX. Once primitives for supporting mutual exclusion are in place, interprocessor signalling, etc. can be handled. Interrupt distribution is a hardware concern, and while useful for load-balancing, is handled by the hardware and not the UNIX software.

While the technique discussed above is effective where the only source of preemption is interrupts, it does not effectively handle the problem of resource locking in a multiprocessor environment; autonomous processors disabling interrupts for themselves should not be able to affect the other processors. Some sort of effective object locking mechanism, one which can lock out other processors, must exist. While other operating systems have tried monitors [15, 43, 52]; semaphores [25] seem more appropriate in the UNIX kernel enviroment [4, 46, 101]. This is a function of the existing UNIX design. Monitors consist of a collection of shared critical data and all of the procedures that reference the data. For example, UNIX device drivers act as monitors for user programs wishing to access system hardware. Since monitors are a more sophisticated structuring concept than semaphores, it would require significant modification to incorporate monitors into the present kernel design (which relies heavily on data structures referenced throughout the kernel, such as the *user area* associated with each process [61, 62]), and it is not clear that synchronizing on entities of this granularity could occur without performance penalties. For example, the *user area* is referenced on every system call as a register save area, an argument passing block, and

perhaps as a data structure to be modified. Consequently, only one processor could be executing code which performs access to this data structure at a time. Another factor is that the simplicity of semaphores makes them amenable to hardware-level support; the monitor abstraction is provided by software. In addition, monitors may require language-level support [15, 22].

There are of course new problems to which semaphores give rise. There are several forms of deadlock possible when counting semaphores are used to allocate resources. The most typical solution and the cheapest (in terms of software complexity, and hence performance) is to order resources such that processes can not accumulate sets of resources in such a way that deadlock is possible. (For more detail on deadlock prevention see [22].)

The semantics of semaphores are rather different than those of the **sleep()** and **wakeup()** calls. In [4], the implementation hides semaphore operations under calls to **sleep()** and **wakeup()** in device drivers so that the driver code can remain unchanged across both multi- and monoprocessors. Other sections of the kernel are explicitly recoded to use the new facility, as the semantic change affects them drastically.

Since the implementation of semaphores (counting or binary) requires an atomic *P* operation, typically some form of hardware support is provided. If nothing else, an atomic test-and-set on shared memory is sufficient. In the following system descriptions, hardware support for semaphores is described where appropriate.

3.1.1 Sequent Balance 8000

The Sequent Balance 8000 [6] is a multiprocessor system which uses from 2 to 12 National Semiconductor Series 32000 microprocessors as the processing units. The processors each have a System Link and Interrupt Controller (SLIC). The SLIC is used to implement the shared memory, and a per-processor cache is provided to attempt to decrease bus contention. Semaphores require atomic (non-interruptable) instructions in order to implement the semantics of the *P* and *V* operations; memory is typically the place where this atomicity property is guaranteed. These services are provided by the SLICs.

SLICs communicate via a private bus, separate from the main memory bus; the private bus is used to, for example, arbitrate among the SLICs as to which will service an interrupt. The SLIC performs the following functions:

1. *interrupt distribution:* The SLIC communicates with the processor with which it is associated; this communication serves to remove much of the traffic which would otherwise pass across the main interprocessor bus, e.g. that necessary to support locking. Each OS kernel running on a processor loads its current priority[19] into a register in the SLIC. When an interrupt arrives (all SLICs receive the interrupt), the SLICs use an arbitration mechanism. The servicing processor is selected based on the current priority displayed to the SLIC by the processor. Idle processors have the lowest priority; the scheme attempts to distribute interrupts in such a way that the load is effectively balanced.

[19]The priority at which the operating system is running is used to derive this number. High prority activities might be those such as servicing page faults, while low priority tasks are exemplified by computing the next million decimal places of pi. These priorities are associated with each activity in the system by the OS designer.

2. *locking:* Each SLIC provides a number of binary semaphores and commands to manipulate them. These are displayed to the associated processor. The semaphores (called gates) are effectively global; the SLICs determine between themselves whether something is locked or unlocked; the separate bus ensures that this process does not consume memory bus bandwidth. Consequently, atomic operations on these semaphores are seen by all processors, and useful for effective mutual exclusion.

3. *interprocessor signalling:* The SLICs are programmable in such a way that a processor can use its SLIC to send another processor an interrupt. This can be used to, for example, to notify a processor that an I/O which it had scheduled had completed.

4. *monoprocessor device driver support:* Since the SLICS are programmable, they can be instructed to send all device interrupts to a given processor. Other system code (the driver) can be bound to that same processor. As mentioned in [4], semaphores can be used to emulate the sleep()/wakeup() mechanism, at least in driver code. This combination of features allows device drivers written for monoprocessor environments to serve effectively in the Balance 8000 as well. Some experiences in writing drivers for this architecture are described in [38].

The cache design is interesting as well, and from the performance measurements given in [6] and also discussed in [48], seems to be effective. The essential idea is that the cache watches the bus for changes in the state of variables it currently has copies of. This is effective because all writes are to main memory (i.e., cache write through) and hence can be detected on the system bus.

The configuration is fully symmetric; any code can execute on any processor in the system. Because of the interrupt scheme, all processors can handle I/O and system calls, unlike the attached processors described elsewhere in this paper.

3.1.2 Encore Multimax

The Encore Computer Corporation's Multimax architecture is a symmetric multiprocessor architecture supporting from 2 to 20 processors [105]. An overview of the architecture and the concepts underlying this particular computer structure are provided in [7, 8]; fairly detailed descriptions of various components can be found in [27, 28, 29].

The Multimax is designed to achieve high-performance processing and high I/O data rates. Up to 20 processors can be configured into the system, attached to the Nanobus, which supports 100 megabyte/second data transfer rates. Considerable I/O processing power is derived from the use of a separate network communications support processor [29].

The Multimax is composed of processors and memory, connected by the high-speed bus. Processors are National Semiconductor NS32032 microprocessors, connected two to a Dual Processor Card (DPC). The processors share bus interface logic and a 32K cache. The DPC also contains an NS32082 floating point unit and hardware to support demand-paged virtual memory. The Multimax can support from 1-8 Shared Memory Cards (SMCs), each of which contains 4 Megabytes of memory. The memory can be interleaved in order to alleviate the mismatch between processor and memory access speeds.

Concurrency control is supported via processor-provided atomic instructions on memory cards; these are interlocked bit-setting instructions, performed on-board, that are used to support memory locks. The

locks are used by the UMAX 4.2 System [27] in order to provide higher-level concurrency control mechanisms, such as spin locks, exclusion semaphores, and read/write locks. Nanobus-monitoring electronics keep the DPC-local caches consistent. Since the locks are implemented at the memory level, the number of available locks is much larger than on a system such as the Sequent Balance 8000, described above, which supports the locking operations at the level of the SLIC bus-control hardware, in bounded numbers. These locks, in turn, are used to support critical sections in the UMAX 4.2 kernel, and to lock access to data structures such as the in-core *i-node* table.

The Multimax architecture, like the Balance 8000's, supports automatic (e.g. hardware) distribution of interrupts between processors; while this feature is cited in [27, 28], it is not described further.

The Multimax off-loads much time-consuming character I/O[20] to peripheral processors; this significantly reduces processor overhead devoted to servicing per-character and per-packet interrupts, and performing data transfers. In addition, several UNIX-specific tasks, such as character echoing and line-editing, are supported by these communications processors.

The most significant feature of the architecture is the extremely fast bus. The bus speed is used to achieve what the Sequent architecture achieves with sophisticated bus-control logic in the SLIC. While the Encore processor does have some cache-control logic, it provides its locking primitives, etc. at the main-memory level. The Encore machine can be characterized as relying on the ratio of bus speed to processor memory referencing behavior in order to provide its locking services; this is the "fast bus" approach. Sequent, on the other hand, relies on a "smart bus" seperate from the processor memory bus, to provide locking facilities. As various technologies improve, it will be interesting to see whether the "fast bus" approach or the "smart bus" approach keeps better pace with processor performance.

3.1.3 Alliant FX/Series

The Alliant Computer FX/Series is a multiprocessor system [47, 92, 100] composed of up to 8 Computational Elements (CEs), which are custom microprogrammed, pipelined processors which implement the Motorola 68020 MPU instruction set, as well as floating point, vector, and special interprocessor communication oriented instruction sets. In addition, up to 12 68012-based Interactive Processors (IPs) are supported. The IPs service all I/O interrupts and communicate with peripherals via a Multibus interface. They, in effect, implement the operating system, while the CEs are reserved for computational tasks (i.e. user applications) only.

The CEs have cache memory; the cache is shared by all CEs. While the IPs also have cached access to the main memory data bus, a given cache supports a maximum of three IPs. As in the Sequent and Encore systems, the cache monitors the bus in order to detect memory write operations which will invalidate cache contents.

The architecture relies on a distributed memory bus (DMB) to achieve shared memory; the cache subsystems are connected to this bus. The bus can support data transfers at 188 Megabyte per second

[20] As opposed to block I/O; see [101]

rates, assuming sequential access (the speed drops considerably otherwise, e.g. for random byte access) of the memory locations.

The architecture as related to UNIX bears further examination, as it has a very unusual implementation of critical sections. The Alliant architecture runs a 4.2BSD based UNIX system called Concentrix. [100] makes a distinction between two classes of computing resource in the Alliant System: individual IPs and the CE-Complex[21]. Concentrix process scheduling policy relies on these classes as the basis for coarse scheduling decisions; each IP is scheduled independently while the CEs in the complex are scheduled *as a unit.* The object file (image) used to create a process determines which type of computing resource will be allocated to the process. Processes which utilize vector-processing and concurrency can only be scheduled on the CE-Complex; other images can be scheduled, and thus executed, anywhere.

A further distinction is made between different *streams*[22] of execution; there are *system-streams* and *interrupt-streams*. System-streams comprise the normal execution of the machine, with control passing back and forth between user-mode and kernel-mode activities. Interrupt-streams are code-sequences executed entirely in kernel mode, and are initiated by some event external to the processing unit. Synchronization between system-streams is enforced by UNIX's non-preemptive scheduling discipline; system-streams protect themselves from inadvertent modification of critical data structures by raising the processor priority level in the uniprocessor case. As we have discussed elsewhere in this paper, that method is not effective in multiprocessor systems.

Concentrix uses the priority-level method in order to handle the system-stream/interrupt-stream interaction within the context of a single processor. Multiprocessor locking primitives handle cross-processor conflicts of the interrupt-stream/interrupt-stream and interrupt-stream/system stream varieties. The globally accessible locks are apparently (it is not clearly specified in [100]) implemented at the memory level to provide a **test-and-set** type of atomic operation; locks are then implemented by "spin-waiting"[23] Care must be taken to minimize lock holding time in order for such a scheme to be effective. Locks are tracked and organized in a stack structure in order to provide information to the kernel for freeing locks upon changing process state and provide structuring necessary for deadlock prevention.

---

[21]The Alliant concurrency instruction set allows CEs to work together as a computational-complex (CE-Complex) on a single application. It is used to, for example, alter looping constructs so that they can be executed in parallel. A for loop iterating with variable *i* ranging over 1..8 could be implemented by executing the body of the loop on 8 CEs, each with *i* set to a different value. Since such facilities are provided to the process and not necessary for the kernel's execution, we will not study them further.

[22]As in single-instruction-stream/single-data-stream [31]

[23]*Spin-waiting* refers to the process of continuously trying to gain access to a memory location; the test-and-set atomic operation returns 0 if a location is "clear", 1 if "set"; in any case, it sets it to 1 in the same machine cycle. Thus, locking is implemented as

```
lock:
    while( test_and_set( location ) )
        ;
```

the process releases the lock by setting the locked location to 0. This is also referred to as "busy-waiting".

The processors communicate by interrupting each other; interrupts can be sent to processors or groups of processors.

While fully symmetric, the Alliant processor supports I/O devices on a per IP basis. That is, each set of devices is attached to a particular Multibus, which in turn is connected to a particular IP. The accessibility of the I/O devices is provided via a global mechanism for mapping Multibus addresses, such that each Multibus device has a unique address. However, there does not appear to be a dynamic interrupt distribution scheme as we have seen previously. This has two disadvantages. Interrupt load-leveling must be performed by moving boards from Multibus to Multibus, and failure of an IP makes attached devices inaccessible. However, there is compensation: by attaching only one device to an IP, real-time processing for that device can be delivered.

3.2 Attached Processor Designs

Attached Processors (APs), while still multiprocessors, are characterized by their asymmetry with respect to input/output (I/O). That is, only one processor (the "master") is in control of I/O operations (which may in fact be carried out by dedicated I/O processors); the other processor(s) ("slaves") can perform all other operations. This control mechanism may be enforced in several ways. In hardware, it might be enforced by only attaching I/O devices to a designated master processor. In software, the goal might be accomplished by operating system scheduling logic assigning a process performing I/O to the designated master.

An AP configuration is especially attractive where computationally intensive work is being done, since much of the computation can be off-loaded to the slave processor(s). I/O intensive workloads are less desirable, as there is only one processor doing the transfers, and each of the slave processors must both wait and cause the master to perform a context switch in order to perform their operations. However, the interface to the I/O subsystem is restricted to the "master", and so much of the systems programming is made easier, e.g. writing disk drivers.

A major disadvantage of AP configurations is that master failure implies system failure; thus there is a single point of failure.

3.2.1 3B20A

The AT&T 3B20S [3] is a 32-bit superminicomputer, capable of executing about 1 MIPS. The architecture consists of a CPU, some main memory, and a number of programmable I/O Processors (IOPs). Peripheral devices are attached to the IOPs; they are capable of removing much of the burden of processing I/O traffic from the CPU. They carry on transfers by means of DMA so that the central processor need not perform byte-by-byte transfers. Intelligent devices such as terminal controllers can be connected to the IOPs as well. Device drivers are simplified in that the interface to the IOP comprises the major portion of many drivers rather than device-dependent code.

The AT&T 3B20A is essentially a 3B20S with an additional CPU. Each of the CPUs can execute kernel software; for example, the scheduler code allows any CPU to pick the highest priority process off

the runnable queue[24]. The master CPU performs all I/O operations; this is accomplished by setting a flag in the requesting process's process table entry indicating it can only run on the master and generating a high priority interrupt to alert the master processor.

Note that a new process state indicating that a process is executing on a processor had to be added to the processor state descriptor to distingush running processes from runnable processes. This distinction is irrelevant in a single-processor system, but with more than one processor operational, processes which are not "runnable" can not be assumed to be blocked for I/O; if a process running on one CPU is left on the runnable queue with no indication of its status, it might be run by both CPUs simultaneously.

3.2.2 Purdue VAX

The VAX-11/780 [23, 24, 50] is a 32-bit superminicomputer produced by the Digital Equipment Corporation. The architecture consists of a central processing unit, a console subsystem, a main memory, and the I/O subsystems. The I/O subsystems consist of several types of Bus Adapters (e.g. UNIBUS, MASSBUS) which connect onto the processor backplane, known as the Synchronous Backplane Interconnect (SBI). Intelligent peripherals, such as disk controllers or terminal controllers, are connected to one of the Bus Adapter subsystems; these devices can perform DMA by means of the adapter subsystems, which, like main memory, are linked by the SBI.

The Purdue VAX [36] configuration is essentially two VAXes with shared memory. The sharing is achieved by attaching the second processor to the SBI, in the manner of memory or a high-speed bus interface unit. This gives the second machine access to memory, etc. in a completely natural way. Other details of the configuration, along with details for constructing the system, are available in [36].

The Purdue VAX possesses a much greater master/slave asymmetry than the AT&T 3B20A implementation of UNIX; only the master can execute system calls. The kernel can detect if the system-call generated interrupt came from the slave. If so, the generating process is blocked and rescheduled for the master by setting an appropriate flag in the executing process's process table entry.

This implementation can take advantage of both processors if the processing load is computationally intensive and does not make a lot of system calls. Otherwise the master processor will be the source of queueing delays in a busy system. As noted in [4], about 40 or 50 percent of a UNIX system's time is typically spent in kernel (privileged) mode, and since system calls are handled by the master, no kernel code is executed by the slave. In a two-processor configuration, this might be mitigated by a workload which always has a process which does not need kernel services in the ready queue. This would ensure that the slave is kept busy, thus keeping throughput high, perhaps at the expense of response time.

---

[24]The runnable queue is a list of processes which could execute if there were a processor available. It is a shared data structure; both processors can examine and alter it.

3.2.3 Conclusions on APs

The 3B20A approach generalizes more effectively to configurations of three and more machines, and it provides a finer grain of parallelism by offering only I/O asymmetry as opposed to asymmetry with respect to execution of all system functions. However, given the nature of UNIX workloads (highly I/O intensive, in most environments) the idea of channeling all of the I/O (or worse, all of the system calls, as in the case of the Purdue VAX) through one processor is potentially a severe performance bottleneck[25]. A two processor configuration may be able to take advantage of the asymmetry in the system by distinct scheduling policies for each CPU. For example, the auxiliary CPU could give CPU-bound jobs higher priority and longer time slices, thus improving throughput, while the master CPU would favor I/O bound jobs and provide a shorter time slice, thus improving response time. This specializing of the CPUs process selection criteria based on their capabilities may possibly yield greater system utilization than the present UNIX scheduling algorithm.

In addition, unless the "master" is dynamically chosen from the set of available processing units, the system has a central point of failure, thus gaining none of the potential fault-tolerance of a multiprocessor design. However, the lessons learned in altering the UNIX kernel to run on an AP configuration [4] have proven to be directly applicable in true multiprocessor systems supporting up to 12 processors [46].

3.3 Analysis of Multiprocessor UNIX Systems

We have examined several multiprocessor designs, of both symmetric and asymmetric varieties. These systems are chosen as representative of available systems, rather than provide an exhaustive enumeration of available hardware and features. This section generalizes some of the observations made concerning multiprocessor UNIX systems.

Attached processor systems with one master and one slave provide a significant speedup for CPU intensive workloads, but adding further slaves becomes fruitless rather quickly, even assuming a much smaller percentage of time devoted to executing operating system functions than cited in [4]. Significant speedup is noted when the slave(s) can execute OS code, but the single processor devoted to I/O still becomes a bottleneck.

Symmetric multiprocessors offer much greater potential for configurations consisting of many processors; this stems from the ability to distribute all system functions and I/O processing across the processors. Multiple processors handling interrupts prevents particular processors or groups of processors from being overburdened with interrupt service[26].

The solution taken to implementing critical sections has been to provide hardware-supported locks seen across the set of processors in a multiprocessor; the traditional UNIX method of ignoring interrupts during the course of a critical section is untenable in a multiprocessor configuration. These locks are used

---

[25]Although [4] reports a 70 percent increase in throughput for the 3B20AP, the per-processor gain can be expected to decrease with the number of processors attached.

[26]Servicing interrupts involves a context switch, where the state (*context*) of the currently executing process is saved, and replaced with the interrupt handler. After servicing the interrupt, the process state must be restored.

to prevent concurrent access to essential data structures by kernel code executing in different processing units; they can be implemented in several ways, for example at the memory hardware level, as in the Encore multiprocessor, or as a higher-level structure, as in the Sequent machine. The locks are used to implement traditional operating system control primitives such as semaphores. Machines requiring even higher levels of parallelism, e.g. the NYU Ultracomputer [26, 37] and the IBM RP3 [72, 77], may require different techniques to support UNIX systems, as the sequentiality enforced by lock access[27] is unacceptable.

An interesting observation is that techniques invented for use in high-performance uniprocessor mainframes seem to be re-emerging in the UNIX multiprocessor realm. For example, the Alliant machine and the Encore machine both off-load much of the I/O processing activity to out-board processors. These units are conceptually similar to the *channels* used by, for example, the IBM 370 Architecture [45].

Also interesting is the apparent lack of interest in building explicitly fault-tolerant architectures [5] which run UNIX. While the AT&T 3B20D mentioned in Section 1.2 is fault-tolerant, it does not run UNIX in native mode[28]. UNIX may require significant changes to support fault-tolerant operation; otherwise a potential advantage of multiprocessor UNIX systems will be wasted.

Since one of the major reasons for implementing multiprocessors is for achieving greater performance, performance is an important issue, both on an absolute basis and when related to the cost of the system. Several of the systems discussed have shown significant cost/performance gains over uniprocessor designs, but it remains to be seen how far this can be carried. The performance gains seem to be very dependent on the offered workload [6]; only further experience in the field will demonstrate if the gains are available to a sufficiently wide range of applications .

---

[27]Processes waiting at a lock must block; the number of such processes increases with the number of processes, and thus can severely degrade parallel activity.

[28]*Native mode* means that the operating system has direct access to all of the hardware. In the 3B20D case, UNIX is merely a privileged process running under DMERT, as mentioned in Section 1.2.

4. Distributed UNIX File Systems

We first provide some motivations for using a distributed UNIX file system. Next, several sections are devoted to issues which must be dealt with by such systems. The final few sections are devoted to case studies, and an analysis of the studied systems.

4.1 Motivation

There are several reasons why one might want to distribute files across several processing units. Much of the research in the past has been on distributed database [11, 53, 97, 104] systems, such as airline reservation systems. Some motivations for distributed file systems are as follows:

1. The database or file structure is too large for one processor. For example, the database of U.S. taxpayers and their returns may be so large that one processor cannot physically support all the disk drives necessary to store the database.

2. The database or file structure is by nature distributed across several sites. For example, it makes sense for bank branches to keep relevant data locally for purposes of quick access and lower communications costs. Users may want to share files, and yet primarily utilize machines physically local to them, for example desk-top workstations.

3. There are many reliability concerns. For example, disaster protection may be crucial; a large enterprise should be able to carry on its activities in spite of local disasters; earthquakes in Boston shouldn't disable the San Francisco office (or vice-versa!). Replication (multiple copies) is often a technique used to enhance *system* robustness. A short synopsis of the plusses and minuses of replication is given later.

4. Cost issues may dictate that one (or a few) copies of a file be accessible to many nodes, in order to cut down on the utilization of often expensive secondary storage. This must be weighed against single point of failure issues and the penalty of decreased data access rates from contention and communications increases.

5. The usual locality arguments apply. That is, the data should be near where it is to be manipulated. Thus, local copies often exist even if the data is not explicitly distributed. For example, when a data file is edited, file contents are often stored in in-core arrays in order to save disk accesses. Likewise, data files, sources, and executables may be cached on local disk in order to save communications overhead.

6. There are several absolute performance issues which may have to be addressed. The major one is that a single system may not be able to support the required transaction rate for an application. The transaction rate of a file system is typically limited by access rates for the mass storage device on which the DB resides. Therefore, carrying on activity in parallel can bring about an otherwise unachievable increase in the aggregate transaction rate.

There are several major issues which must be addressed in the design of a distributed file system. These issues affect both the user interface to the file system and the underlying implementation of the interface, and are discussed in the following sections, beginning with naming.

4.2 Naming

The interesting issue of naming arises at the user interface level. In most modern operating systems, a file is known by its name, rather than some obscure parameters such as its disk locations or social security number. In essence, the issue here is how do users reference distributed files (files on remote locations) as opposed to local files? Should they be able to see any difference? Can common utility programs

reference files in a transparent[29] way? (For a lively, if somewhat critical discussion of naming, see [78].)

Transparency is an important point when dealing with the issue of naming. If an entirely distinct convention for naming remote files is used, utility programs, such as directory listing programs, must be aware of the difference, so that directory listings of remotely located file directories can be done in a transparent way. If the details of accessing remote files do not include their being named in any specific way, then the operating system can handle the details of referencing the remote data, thus providing utility programs some more uniform way of doing their job.

The major decision in designing a distributed UNIX file system, as opposed to designing a file system without such stringent interface requirements [98] is what the system will offer in terms of transparency. *Transparency* is the idea that the user of a facility should be oblivious to certain aspects of the use of the facility. For example, if all remote files must be accessed with different system calls, e.g. *remote_open(location, file_name)*, then access of remote files is not transparent. Thus, decisions as to what UNIX file system syntax and semantics are to be preserved are central to any distributed UNIX file system architecture [88]. Reliability issues, though distinct, become a function of transparency as a result of the semantic transparency aspect of file system design.

4.2.1 File Access Transparency

If no new system calls are necessary to access remote files, then the system has *access transparency*. This means that remote resources, although they may be named differently, are accessed in the same way as local files. While the file may have a name of the form {remote-machine}/usr/file[30] one can still use open() to obtain a file descriptor, and this file descriptor can be used as if the file were local. The read() and write() primitives should also behave the same way. The idea here is that programs should not have to be rewritten to be aware of the location of the files they are accessing:

/bin/cat {remote-machine}/usr/file

should dump the contents of that file on my terminal just as if the user had typed the command

/bin/cat /usr/file

on {remote-machine}. The *syntax* of the UNIX file system primitives is preserved, although the path-name syntax or semantics may be altered.

4.2.2 Syntactic Transparency

While access may be transparent to the system calls, it may not be transparent at the file naming level. System names may be prepended using some character indicating that the remote system name field is separated from the file name field. If this is inconsistent with the normal rules for generating file names, ambiguities may result in naming files. A naming scheme which is *syntactically transparent* is one which uses the UNIX file naming convention of "/"-delimited names of nodes in a tree structure.

---

[29] A definition of transparency is given below.

[30] For example, consider machine zack and file /etc/passwd. If all remote machines are named in a directory /n, the file would be named /n/zack/etc/passwd; if machines are named by prepending {name}:, the file would be named zack:/etc/passwd, and so on.

Note that this implies that the remote system names must somehow be accessible using the existing UNIX directory mechanisms. Note also that syntactic transparency does not imply access transparency: something which is nameable unambiguously using the UNIX file name syntax may not be accessible without new system calls or library routines.

4.2.3 File Location Transparency

This frees the user from remembering at which host a file is located. "Placement of files become the responsibility of the file system, permitting it to exploit placement strategies that improve efficiency" [102]. Two of these strategies are *file migration* and *file replication. File migration* moves files towards the point where they are read or written, thus reducing network access and reducing response time. *File replication* makes copies of the data contents of the file, in order that read access rates (on the average) and fault tolerance (as a result of redundancy) are increased.

A variation on the idea of location transparency is one which attempts to hide the fact that the architecture is distributed. We call this variation the "One Big Machine" idea[31]. In this sort of system, an attempt is made to present the user with a view of the distributed system as a single large rooted tree; the fact that the system is distributed across several processors is hidden from the user. Such a system ideally provides *no* indication of a given file's location; thus it is *location-transparent.* No use of the strategies mentioned in the previous subsection is implied, only that the monoprocessor UNIX file system tree structure is not semantically altered.

4.3 Implementation

At the design level, one has to deal with issues such as whether or not to replicate data. This has a crucial effect on both the performance of the system, and algorithm design while implementing the system. The notion of consistency among the multiple copies must be preserved across them; if a given entity has alternate communication paths, it should receive the same answer anywhere it asks. This is essentially a distributed complication of the readers and writers problem; the difficulty is preserving the multiple readers / single writer paradigm [22] in the context of a distributed data referencing environment. Also at issue is what balance is to be struck between the attractiveness of replication in terms of increased robustness and increased read rate (more copies means less contention and more likelihood of local reference) versus the increased complexity of the update (writing) protocol and the system-wide resource utilization (of disk resources).

There are arguments for relaxing these consistency constraints in certain situations. While on the one hand, selling an airline seat twice may potentially create an irritating situation, it is not as crucial as maintaining account consistency in a database of checking accounts accessible through several automatic teller machines at once. Consequently, the cost of consistency may be balanced against the potential loss of not maintaining it. Too, it is possible to relax constraints "locally", as long as consistency is maintained [10].

---

[31]The *LOCUS* [80, 81] terminology is "Network Transparency". The idea there is that you are oblivious to the fact that the file system structure is implemented by a set of processors communicating over a network.

Implementation problems such as reliability and dealing with failures (e.g. "backing out") are major, and are often dealt with in inelegant but simple fashion, such as transaction logs. If the file system is to support, e.g. a database system, and it is to be of general use, it may not be able to take an application's characteristics into account. For example, if there are two applications, one which can afford to be wrong and the other which cannot, the system must be designed to handle the more demanding of the two applications. In general, it is assumed that system software cannot deliver a wrong answer.

Communication is typically structured as being between clients and servers.

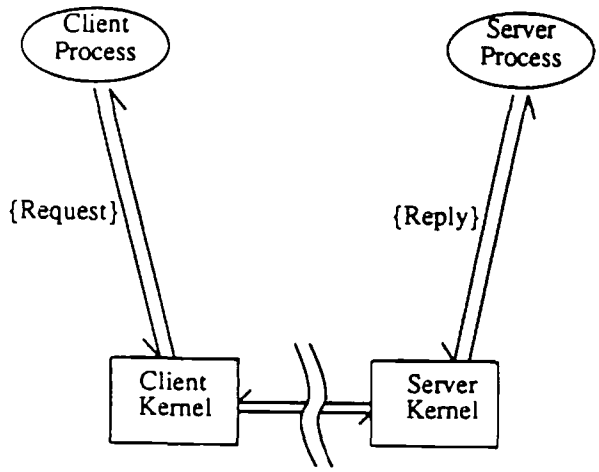

Figure 4-1: Client/Server Interaction

These can be normal UNIX processes, or some kernel implemented mechanism in order to prevent context switching. Structuring the communication is as an issue; the choices are typically between a virtual circuit, analogous to a phone call, and a datagram system, analogous to sending a letter through the postal system. Also at issue is what sort of state information to preserve at the server. State information is the enemy of robustness; if a connection is broken, the state information is lost. Thus *stateless server* designs exist as attempts to improve system robustness. However, the advantage of statelessness must be balanced against the potential difficulty of preserving UNIX file system semantics in such an implementation. For example, remote tape management becomes a serious problem unless a file descriptor is held on the remote system. This is because the tape drive must both maintain the position of the tape reel, and provide some indication that it is "in use". It does this by means of state information indexed by the file descriptor, which in turn is state information that a server process must maintain.

Updating data becomes a much more complex issue in distributed file systems. If multiple copies are kept, as in the case with buffered block I/O[32], a philosophy must be established as to which version of the data is correct. An enforcement mechanism for this policy must be put into place throughout the

[32]Block I/O on UNIX is *buffered*, that is, it is read in from the disk a block at a time, and then dispensed to the reading process as required. Occasionally, when sequential data access is noted, another block is fetched in anticipation of its being used shortly. Unfortunately, as with other buffering schemes, the data can become outdated rather easily.

distributed system, in order to guarantee correctness of the data for future access.

4.3.1 Inhomogeneity

An issue in the lowest reaches of the implementation is the subject of inhomogeneity. This obviously is not of concern to homogeneous environments (environments consisting solely of like processing units), but a general (e.g. "open system") distributed design should not assume homogeneity.

Data inhomogeneity presents many problems. Some are not particularly hard to solve, particularly those objects which have very simple meaning, such as a number. For example, at the very lowest level of data representations are things such as bit and byte orderings and notions of data units such as words. Numbers may have a different representation on two architectures, e.g. DEC PDP-11 and Motorola 68000. These data inhomogeneity issues are typically resolvable through translation conventions; a reasonable translation convention can preserve a good deal of transparency for network users, although it of course adds considerable complexity to the system software. A very effective technique has been chosen by SUN for its Network File System [89], where a standard External Data Representation (XDR) [65] is defined and used to pass data of different types between nodes. Other interpretation issues are rather more difficult to resolve correctly.

The data inhomogeneity usually presents more of a problem once transparency is required for access to data structures displaying a higher level of abstraction. Two readily understood examples are file control blocks and instantiated processes, both dealt with by an operating system at a higher level of abstraction than raw data. For example, a file control block on one architecture may contain the file's complete path name in a hierarchical file system, while another may only preserve the last component of the path. System calls which access the components of the files must either distinguish between the two types of files and act accordingly, (perhaps returning an error if the full path name is asked for; not very transparent) or perhaps using some complex technique for delivering the full path name component from the remote system.

4.3.2 Data Interpretation

While UNIX files are nominally ordered arrays of characters, a natural use of the system dictates that some decisions must be made about how remote data is interpreted in certain situations. Executable ("a.out") files provide an illustrative example. Among the different types of data stored in files are process images, executable program text which can be loaded into main memory and executed.

The important question is what to do with a request to execute one of these modules which comes from a remote (and perhaps incompatible) system. This process instantiation problem is interesting, because it involves two important points. First, what should be able to be passed across a network in a sensible fashion? A process instantiation consists of loading executable program text and initialized data segments into memory, and then passing control to the new process. While an external data representation technique may carry the data across a network unaltered, it doesn't guarantee that the meaning of the data as an executable program is preserved. Hence, even if the executable image can be loaded, it can't execute on another arcitecture. Programs use machine features such as machine registers as well as pointers to data areas used by the process. Process context, by its very nature, seems not to be readily

portable. Machine registers are not compatible between machines. Different machines have different numbers of registers; some machines such as the Burroughs B7700 [75] provide none at all at the architectural level. (Of course, many of these questions are moot if the remote system can easily provide an emulation of the local architecture)

Second, the process instatiation problem raises the question of what should be addressable across the network. That is, if one processor has memory addresses that refer to its local memory locations, what should these addresses mean to a remote site, if anything? For example, the address of an object on a VAX-11/780 connected to other machines via the ARPAnet is not expected to have any validity to other members of the ARPA community. On the other hand, Apollo's Domain Architecture [55, 56, 71]presents a One-Level Store approach [74], where the addresses have meaning everywhere, since all nodes are part of the same "address space".

The simplest solution, to fail when the meaning of an object is not covered by an explicit definition or standard, is probably the most reasonable. This saves designers from sacrificing general performance in order to handle unusual and unlikely special cases.

4.4 Case Studies

We have provided motivations for distributed file systems, and a discussion of some issues which must be addressed in the design of such systems. Descriptions of several design approaches for a distributed UNIX file system are given in the following sections.

4.4.1 8th Edition UNIX Network File System

A Network File System[33] has been implemented in the 8th Edition UNIX System[34] [51, 79, 83, 86, 109]. The concept of a file system type is utilized in order to incorporate remote file access into the UNIX name space. The file system type is used in order to detect logical boundaries between remote and local access, as crossing a mount point indicates a change in the physical device context in a normal UNIX system. A connection is established between a client machine **A** and a server machine **B**, where **A** accesses the file and **B** provides access to the file. The connection established is a virtual circuit between the machines. The connection is considered asymmetric in the sense that another connection must be established if processes on **B** wish to access files on **A**. **A** sees files on **B** via a variation on the mount() system call; **B**'s root directory appears to be a mounted file system located at a mount point. When path name interpretation is done, the fact that an *i-node* is located remotely is detected at the mount point, and the connection is utilized to gain the necessary information about the file. (The information at the mount point contains sufficient information to detect which virtual circuit to use.)

By convention, remote root directories are mounted in the directory /n. Since the remote access is natural and occurs in the path name interpretation process, one can type the command line:

---

[33]Another system, Remote File Sharing (RFS) [42] has been demonstrated by AT&T Information Systems. While based on the 8th Edition NFS, it apparently has enhanced features for system administration.

[34]Consistent with AT&T Bell Labs tradition, the UNIX system is named for the edition of the UNIX Programmer's manual which describes it.

"cat /n/*/etc/passwd | grep '^dmr:' | wc -l"

in order to find out how many systems contain a password file entry for user **dmr**. The shell [13] expands the "*" in the normal fashion; it matches all entries in the /n directory, which happen to be machine names.

All UNIX File System Semantics are preserved by the 8th Edition UNIX NFS; this is eased by the state information (e.g. file table entries, reference counts, etc.) maintained by the server. For example, unlike the SUN system described below, the 8th Edition UNIX NFS allows transparent access to remote devices. The client machines do not cache any state information; this is simple and ensures that there is a consistent view of the file. It also provides concurrency control.

Since the system is constructed using the stream I/O described in [86], it is independent of any particular network implementation. All nodes must, however, run UNIX.

4.4.2 SUN Network File System

The SUN Network File System (NFS) extends the 4.2BSD [18, 58, 59, 68] kernel. The SUN NFS has several design goals [89], with the overall intent to provide easy sharing of filesystem resources in a network of heterogeneous machines. These goals are: machine and operating system independence (e.g. NFS Servers can supply files to many varieties of client), transparent access, crash recovery, UNIX semantics maintained on client, and reasonable performance.

Several of the goals are achieved by means of a protocol [90] which uses an external data representation [65] (XDR) technique[35] to remove many of the problems of data representation present in a heterogeneous environment. The external data representation technique achieves the goal of machine and operating system independence. The NFS protocol is based on an underlying remote procedure call mechanism [66].

The crash recovery goal is achieved by means of the NFS protocol. The protocol specifies that all information necessary to implement a client's request on a server will be carried with the request. Hence, servers need not maintain any state information which may be lost in a server crash, and since the client maintains the information it can retry the request elsewhere. Retries are harmless, as the NFS protocol guarantees idempotence of NFS requests.

The kernel was extended to deal with the remote objects. The "filesystem interface" was abstracted into two parts, the Virtual File System (VFS) interface, which defines the operations which can be performed on a filesystem object, and the vnode interface which defines the operations which can take place on a file object within that filesystem. (These abstract the normal file system and *inode* objects.) The namei() routine was extensively modified, as much of the activity taking place in this routine had to respect the new vnode semantics. In addition, some code where global state was used as a

[35]The idea of an external data representation is that agreed upon data will take on an agreed upon form when in the representation, e.g. a certain coding of floating-point numbers will be used. The machine dependent portion of the system will then be limited to routines which convert to and from the XDR.

communication device had to be modified. Many of the vnode interface procedures map one-to-one with NFS protocol procedures. Remote file systems are mounted volumes, and the path name interpretation process done in namei() (which now returns vnodes) spurs the remote reference through the vnode interface. Note that unlike the 8th Edition UNIX Network File System, there is no virtual circuit maintained once the file system is mounted.

Currently, root filesystems are not shareable. Filesystems are named by a convention which indicates their location, as in the 8th Edition UNIX NFS.

The SUN NFS performance is about 70% of the 4.2 file system performance on a workload composed of commonly used UNIX commands. Several graphs of the performance on these tasks are available in [89].

The SUN NFS does not replicate files nor does it provide any locking mechanisms.

4.4.3 Newcastle Connection

The Newcastle Connection (TNC) idea [12, 17] was developed at the Computing Laboratory of the University of Newcastle-upon-Tyne. Unlike the other Distributed UNIX File Systems described here, TNC is implemented completely at the application library level rather than in the kernel. Path names containing the string "/.."[36] are intercepted and processed via a remote procedure call mechanism which continues processing the request, e.g. opening a file on zack would require referencing the file as "/../zack/etc/passwd". Thus, a remote machine's filesystem is treated as a subdirectory of "/..".

The remote procedure call mechanism utilizes whatever underlying network services are available on the local node. TNC designers expect the system to be able to support an infinite system; that is, a UNIX United system supporting an infinite number of nodes. They achieve this by means of the notion of "name neighbours" to break up the potentially infinite naming tree into manageable components. A "name neighbour" is a UNIX system addressable in one hop, i.e. in the name-space provided by the underlying network. Such systems comprise a "name neighbour space". System calls destined for such systems are sent to and executed by a server process at the destination; this server can be be in existence or created dynamically. The underlying transport software is responsible for delivery, fragmentation of large data objects, etc. The server is responsible for execution of the system call; it is encoded in the packet header; access permissions are checked on the server's system.

If the path name refers to a system which is not in the "name neighbour space", a *relay process* is instantiated which transparently passes the datagram along to a server process or another relay. The *relay process* on a remote system serves as a logical gateway connecting two "name neighbour spaces"; the transitive closure of the connection graph [1] is the name tree of UNIX United.

The path-naming convention of UNIX United is perhaps the most elegant of the distributed file systems

---

[36]While in normal usage, ".." indicates the parent directory of a directory, ".." in the root directory is taken to be, by convention, the root directory itself. In TNC, the meaning is taken literally; the parent directory of the root directory is used as a syntactic mechanism to lift the pathname out of the realm of the local machine.

examined; the syntactic "/.." is in fact well-related to the semantics under TNC. The implementation is quite portable; underlying network software need only supply abstract **send** and **receive** operations in order to implement TNC. This is a considerable advantage over systems such as LOCUS which require extensive kernel modification, or other systems which implement distributed file system transport and name interpretation in the kernel. However, the performance penalty of several intermediate layers of software may be unacceptable; user-level software is subject to many delays and is removed from the device-interrupt level of processing by several layers of software, each requiring synchronization.

Since TNC intercepts the system calls via library routines[37], programs must be recompiled once in order to use the system. Since all access to file system data is made by system calls, the implementation is network transparent; in principle, all utilities should work as before. It is somewhat unclear how such operations as **cd /../zack/usr** would be implemented, as the current working directory has no local file descriptor object, but it is possible.

4.4.4 IBIS

IBIS [102] is a file system developed in a research effort at Purdue University [21]. IBIS is built on top of the 4.2BSD Interprocess Communication (IPC) Facilities, with which it performs all remote operations. The UNIX file access primitives have had the pathname syntax extended so that a pathname is now of the form [<**hostname**>:<pathname>], i.e. it prepends a machine name to the path name, so that the password file on machine **zack** would be accessed as **zack:/etc/passwd.** What happens during pathname interpretation is that the system calls (e.g. open()) detect the prepended machine name, and pass the remaining string to the named remote system for interpretation; it does this via communication with a server process[38] on the remote machine. The modified pathname syntax can, to some degree, be hidden by the use of symbolic links. For example,

ln -s zack:/ /n/zack

should allow one to create a directory structure usable to access sets of systems in a manner similar to the 8th Edition UNIX Network File System discussed elsewhere in this paper.

IBIS provides file migration and file replication. File replication and migration are both done on demand, by default. File replication serves to increase the read rate; a copy of a file is made on the local system when a file is read. This copy is typically marked for deletion when the file is changed; this merely requires sending a status change notice rather than the new file contents. File migration refers to the movement of the "primary" location of a file; this location is essentially the most recent location which has written the file contents. It invalidates other copies of the file which are in existence; they will update themselves the next time something is read from them. The advantage of migration is that it speeds write access to files; only the local copy is written to, and remote copies are simply invalidated.

---

[37]That is, the system entry point reference open() is replaced by a procedure call to the library routine open(), which interfaces with the Remote Procedure Call facility.

[38]Each *client process* (the process requesting file access is a client) has a dedicated server; if more than one file is open, the server multiplexes access to that set of files. The connection is set up by a server creation process on the remote hist; this is done over a secure privileged port. If the client process *fork()*s, a corresponding server process is created in order to preserve the semantics of *fork()*

The idea is that frequently read files (e.g. those near the root in the directory tree) will be replicated nearly everywhere, so that nearly all accesses will be local. Frequently written files will not incur the overhead of moving their data contents; rather, they follow the most recent writer.

4.4.5 LOCUS

LOCUS [63, 80, 81, 82, 106, 107, 108] is the result of a continuing research effort at the University of California at Los Angeles. The LOCUS system is available commercially, and development continues, both at UCLA and at the corporation. As LOCUS was one of the first distributed UNIX systems, and was driven more by research interests than perceived market needs, it is somewhat distinct from the other distributed file systems.

While the essence of the LOCUS system is the distributed UNIX file system, the way in which it is implemented reflects an attempt to achieve the "One Big Machine" point of view mentioned above. A LOCUS "system" may consist of an aggregate of several processors, but the user interface is always a single rooted file system. LOCUS extends the UNIX name space to the entire netcomputer[39], such that there is a "logical" root to the name space to which the local nodes (on individual processors) are in some sense subservient.

Like IBIS, LOCUS replicates files. In LOCUS, the *i-node*s are replicated. A logical *file group* is defined within which the *i-node* is guaranteed to be unique. There can be several physical locations for the file group, each of which contain *some* of the *i-nodes* associated with that file group. For example, if a file group /luser is found at 10 sites, the *i-node* associated with /luser/games/rogue may be found at 1 or more of these sites. A file group is logically equivalent to a file system. Thus, if there is a data file /usr/jms/data with *i-node number* 17, there may be several copies of that *i-node*. When the file is to be read, the name is mapped to an *i-number*, that is used to find the file in the file group[40] with which the file is associated.

Since all the network communication takes place beneath the level of the name space, the name space can be entirely ignorant of the file's location. This enables applications to be used with LOCUS in a very transparent manner; they have no idea where the files they are accessing are physically located, and both file system syntax and file system semantics are preserved.

The replication also increases performance and robustness, since the likelihood that a file is local is increased with the number of copies in the netcomputer, and the likelihood that all copies of a file will be destroyed is likewise decreased. The replication is not dynamic at the operating system level: the number of copies and replication sites (storage sites in LOCUS terminology) are set either by default (the values are associated with the current process state) or by system calls added to the UNIX repertoire.

---

[39]While this term seems to originate with [32], it applies to LOCUS

[40]A file group is the LOCUS name for a collection of inodes and data blocks which would normally comprise a file system; an *i-node* always refers to the same file data, which may or may not be instantiated within a given instantiation of the file group on a processor.

LOCUS provides sophisticated algorithms for rejoining partitioned[41] sets of processors; such algorithms (for restoring the global file system to a sane state) are beyond the scope of this paper. The mechanism for achieving this, a *version vector* scheme, is however interesting as it is used to ensure synchronization and consistency among the several storage sites which can be associated with a given i-node number in some file group.

The LOCUS system also provides more support for database style *transactions* [53, 70] than the standard UNIX filesystem. This however, requires no alteration in the semantics of the filesystem primitives, as the interfaces necessary to support transactions are separated from the file manipulation interfaces. (There is extensive literature devoted to distributed database systems [11, 87, 97, 104].)

While the network is entirely transparent to the users in LOCUS, this may not necessarily be desirable. The element of control is somewhat lacking. For example, it may be desirable to update a file on a subset of the processors underneath the LOCUS file system, for example, a file which indicates that the machine type is a VAX-11/750[42] or machines geographically located in New York City. While this is very natural in a design where the machine name is part of the file name (either by convention or by requirement) it is not quite as easy with LOCUS.

4.5 Analysis of Distributed UNIX File System Designs

In this section, we analyze the distributed UNIX file systems discussed in the case studies. The analysis is done with respect to the issues that were presented above.

4.5.1 Naming

Since the remainder of the user interface to files is more or less defined by the UNIX system calls and their semantics, the major interface issue is the naming distinctions made between local and remote files.

It seems preferable with current technology to have some method for specifying the the location of files in systems; this control is removed in LOCUS, and it is unclear how much control is available to IBIS users; the similarity to the 4.2 BSD interface indicates that quite a lot of information about the location of files can be specified via the name. The motivation for selecting some subset of systems is obvious; it is similar to a multicast facility in LANs.

The prepending of system names to file names seems somewhat unnatural in a UNIX file system, but UNIX users are familiar with the concept due to previous methods for file transfer such as uucp, a utility which transfers files using telephone connections. Uucp syntax for the password file on system zack would be:

zack!/etc/passwd

---

[41] A set of processors is *partitioned* if it is operational, communicating with other processors in the partioned set, and it is disconnected (by, e.g. network failure) from a larger set of processors of which it is normally a part.

[42] E.g. "/bin/vax750". This file shouldn't be updated on, e.g. a PDP-11/70.

In addition, since such prepending is not a natural component of the path name syntax, storing system names in directories is somewhat more difficult; thus utilities may require modification to act in a reasonable manner, for example the expansion of the shell "*" filename wildcard.

A possible danger with the "{name}:" sort of prefix is interference with local activities. For example, assume that a user named jms has a file "/luser/jms/zack:/etc/passwd", and that his current working directory is "/luser/jms". "zack:" is a valid file or directory name under monoprocessor UNIX systems. However, given that a remote system "zack" exists, user jms's request to access "zack:/etc/passwd" can become ambiguous by virtue of the new syntax. These dangers are avoided by using the normal directory structure and path name field separator syntax to access remote files. While the ":" can be hidden using 4.2BSD symbolic links, it is still difficult to avoid the ambiguity.

The most syntactically reasonable interface seems to be that of the Newcastle Connection; the use of "/.." is quite clever and very natural. In fact, it seems to be somewhat more reasonable than the 8th Edition UNIX NFS, at least as a naming convention.

4.5.2 Implementation

Path name interpretation can be done at several points in the process of obtaining a file descriptor. While it can be handled by libraries, this makes the interpretation of such ubiquitious examples as the shell wildcard designator "*" somewhat more difficult. (The present algorithm is to read a directory, looking for matches. This implies that names are available in directory structures.) The decision is one influenced by performance and portability more than any other factors. Performance can be potentially increased by a kernel-level implementation *if* the percentage of remote references is high enough, and if the overhead of implementing the remote file names outside the kernel is high enough. If there are not a lot of remote references, then the additional code for dealing with remote references may enlarge and slow down the kernel needlessly.

When remote path name interpretation is handled by the kernel, data communications *must* be handled by the kernel as well, rather than by user-implemented networking or RPC code[43]. Thus, the package becomes less portable from UNIX system to UNIX system; the distributed file system becomes tied to the communications subsystem by virtue of software engineering decisions. Of course, placing the file system logic in the kernel (as well as removing layers of data communications software) will improve a system's performance. Experience has shown, however, that systems which are engineered for simplicity and portability outlast those designed to meet goals centered around performance. UNIX itself is the obvious example.

One other important issue is that of the underlying semantics of a file: Does a single file's representation change in the distributed file system? In the case of IBIS and LOCUS, file replication, where multiple copies of a file are kept, is implemented. IBIS also implements dynamic file migration; that is, the location of the file may change during the operation of the system. If the system is to make the

[43]Although Dennis Ritchie's *streams* system may provide a means for user level network interface code to be accessed in the 8th Edition UNIX NFS.

decision about where a file's contents are located, there must be a freedom to do this implied by the naming convention, or the naming convention must somehow be subverted[44]. There must also be logic for determining which version of a replicated file is correct after some system modifies the data contained in a file, and if operation after partition is enabled, after the partitions are rejoined.

4.5.3 Miscellaneous Issues

While the data interpretation problem is an issue, it is mainly dealt with by either assuming homogeneity or guaranteeing nothing where it does not exist. LOCUS addresses the issue of instantiating processes; the single-machine point of view would be difficult to preserve across hardware types without such a mechanism.

LOCUS also provides support for other data base style operations, such as transactions. This may or may not be a wise design decision; it certainly seems somewhat inconsistent to provide anything other than primitives necessary for user processes which wish to use such services. It's inconsistent in the sense that it adds overhead, and gives the kernel a point of view about file objects other than the "linear array of characters" view which has proven useful.

[44]For example, the file control block might have a location distinct from the data blocks, in which case the naming convention might only insure that the file control block is located at the referenced location.

5. Conclusions

UNIX is a popular working environment in the academic and industrial community, and, as it is simple, well-understood, and easily modifiable, it has often been the tool of choice in OS experiments. Many of the systems we examined had their source in industrial and academic research environments, where UNIX is the standard O.S. As distributed systems become more mature, many users want to take the environment they are comfortable with along with them into the distributed environment. Consequently, many distributed UNIX systems have appeared.

UNIX provides a familiar environment for developers as well. Adjusting it to the needs of multiprocessors and distributed file systems is considerably easier than developing a new system from scratch. Most of what is done in UNIX is done right, and most of the adjustments made are isolated to specific kernel segments. Making these isolated adjustments is often sufficient to support the remainder of UNIX, thus saving massive development effort and enabling the new system to use a large pool of existing high quality software.

Multiprocessor UNIX systems are the approach of choice for high performance computation which can take place without being widely geographically distributed. They can take advantage of the potential for fault-tolerance provided by multiple CPUs, but this seems under-exploited. It's unclear whether this is a function of UNIX or of the hardware. UNIX may need significant changes to be fault-tolerant, but the changes could possibly be isolated to a small segment of the kernel, as other changes have been. Another possibility is that hardware designs are subject to catastrophic failure, for example bus failure, which are not easily prevented without special purpose hardware such as redundant bus logic.

Distributed file systems are utilized when the processing nodes require more independence, either because they can't share memory because they're too far apart, or because they do not desire to, for structuring reasons. This is because, in the ideal case, they provide distributed access to the most significant shared resource under UNIX, data files. The designs we have examined in this paper work, but it is too soon to tell what the user acceptance of various naming conventions and performance tradeoffs will be. Distributed UNIX file systems provide somewhat more system robustness than multiprocessor systems, in the sense that the independence of the processors creates a more fault-tolerant system. Very few of the systems have addressed problems such as partitioning; in many cases the system response is somewhat fragile. For example, if, when using the SUN NFS, a remote system whose files are mounted locally undergoes a failure, processes which access the apparently accessible files will be blocked in a kernel wait state. This wait state is at a high enough priority that keyboard interrupts are ignored. This is a function of SUN's stateless server design, but the interface doesn't seem to provide reasonable handling, at least from a user perspective. Another issue, which hasn't been addressed in our discussion, is security. SUN has discussed the problem somewhat in the literature, but at this point, it appears to be a design issue which has been postponed.

5.1 Summary

Essentially, UNIX has had two types of growing pains, those of rethinking old assumptions, and those of adding necessary features. For example, the priority-raising scheme for implementing critical sections was based on an assumption about the underlying hardware configuration which is not valid in a

multiprocessor kernel. Hence the logic of certain sections of the kernel had to be changed to use a different methodology, that of semaphores.

The second type of growing pain is that often encountered with a successful software system which must have features added. Questions of compatibility with the existing version of the system and associated utilities are often crucial to the new systems useability and success. These have certainly influenced the designers seeking transparency in an effort to preserve UNIX file system semantics.

6. Acknowledgements

I'd like to thank Bell Communications Research and Bell Laboratories for providing an atmosphere in which it was easy to learn about UNIX. Gerald Maguire and Steven Feiner provided immense amounts of constructive criticism. Gerald Leitner provided the impetus for the paper and was instrumental in its completion. Any faults which remain are the author's alone.

7. References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, 1974.

[2] George A. Anderson and E. Douglas Jensen.
Computer Interconnection Structures: Taxonomy, Characteristics, and Examples.
ACM Computing Surveys Volume 7(Number 4):pp. 197-213, December, 1975.

[3] AT&T Technologies.
AT&T 3B20S Computer - Model 2 System Description.
AT&T Technologies, 1984.

[4] M.J. Bach and S.J. Buroff.
Multiprocessor UNIX Systems.
AT&T Bell Laboratories Technical Journal Volume 63(Number 8, Part 2):pp. 1733-1751, October, 1984.

[5] J.F. Bartlett.
A NonStop Operating System.
Technical Report, Tandem Computers, Inc., 1977.

[6] Bob Beck and Bob Kasten.
VLSI Assist in Building a Multiprocessor UNIX System.
In *USENIX Proceedings*, pages 255-275. June, 1985.

[7] C. Gordon Bell.
Multis: A New Class of Multiprocessor Computers.
Science Volume 228:pp. 462-467, April, 1985.

[8] C. Gordon Bell, Steve Emmerich, Ivor Durham, Daniel P. Siewiorek, and Andrew Wilson.
Computer Structures are Changing: Will UNIX Change with Them?
In *USENIX Summer '85 Conference Proceedings*, pages 1-4. USENIX Association, June, 1985.

[9] Rob Warnock.
Interview with Gordon Bell.
UNIX Review :pp. 62-68, 1986.

[10] E.J. Berglund and D.R. Cheriton.
Amaze: A Multiplayer Computer Game.
IEEE Software Volume 2(Number 3):pp. 30-39, May, 1985.

[11] Bernstein, P.A., et al.
Concurrency Control in a System for Distributed Databases (SDD-1).
ACM Transactions on Database Systems Volume 5(Number 1):pp. 18-51, March, 1980.

[12] J.P. Black.
The Newcastle Connection: Current Status and Future Plans.
In *USENIX Proceedings*, pages 377-382. July, 1983.

[13] S.R. Bourne.
The UNIX Shell.
Bell System Technical Journal Volume 57(Number 6, Part 2):pp. 1971-1990, July-August 1978, 1978.

[14] L. Breeden and M. O'Brien.
CSNET: A Computer Science Research Network.
In *USENIX Proceedings*, pages 371-376. July, 1983.

[15] Per Brinch-Hansen.
The Programming Language Concurrent Pascal.
IEEE Transactions on Software Engineering :pp. 199-207, June, 1975.

[16] N.H. Brown, M.P. Fabisch, and C.J. Rifenberg.
SAFEGUARD Data-Processing System: Introduction and Overview.
Bell System Technical Journal :pp. S9-S25, 1975.
Safeguard Supplement.

[17] D.R. Brownbridge, L.F. Marshall, and B. Randell.
The Newcastle connection or UNIXes of the world unite!
Software Practice and Experience Volume 12:pp. 213-237, 1982.

[18] *Unix Programmer's Manual (4.2BSD)*
University of California, Berkeley, 1983.

[19] Chesson, G.L.
Datakit Software Architecture.
In *Proceedings I.C.C.*, pages 20.2.1-20.2.5. 1979.

[20] G.L. Chesson and A.G. Fraser.
Datakit Network Architecture.
In *Proceedings Compcon*, pages 59-61. February, 1980.

[21] Douglas P. Comer.
Transparent Integrated Local and Distributed Environment (TILDE) Project Overview.
Technical Report Number CSD-TR-466, Purdue University, 1984.

[22] H.M. Deitel.
An Introduction to Operating Systems.
Addison-Wesley, 1984.

[23] Digital Equipment Corporation.
VAX Hardware Handbook.
Digital Equipment Corporation, 1982.

[24] Digital Equipment Corporation.
VAX Architecture Handbook.
Digital Equipment Corporation, 1982.

[25] E.W. Dijkstra.
Solution of a Problem in Concurrent Programming Control.
Communications of the ACM Volume 8(Number 9):pp. 569-578, September, 1965.

[26] Jan Edler, Allan Gottlieb, and Jim Lipkis.
Considerations for Massively Parallel Unix Systems on the NYU Ultracomputer and the IBM RP3.
In *1986 Winter USENIX Technical Conference*, pages 193-210. USENIX Association, January, 1986.

[27] Encore.
UMAX 4.2 Operating System.
Sales Literature, Encore Computer Corporation, 1985.

[28] Encore.
Multimax Multiprocessor System.
Sales Literature, Encore Computer Corporation, 1985.

[29] Encore.
Annex Network Communications Computers.
Sales Literature, Encore Computer Corporation, 1985.

[30] P. Ewens, D.R. Blythe, M. Funkenhauser, and R.C. Holt.
Tunis : A Distributed Multiprocessor Operating System.
In *USENIX Proceedings*, pages 247-254. June, 1985.

[31] Flynn, M.J.
Some Computer Organizations and Their Effectiveness.
IEEE Transactions on Computers Volume 25, September, 1972.

[32] A.J. Frank, L.D. Wittie and A.J. Bernstein.
Group Communication on Netcomputers.
In *Proceedings, Fourth International Conference on Distributed Computing Systems*, pages 326-335. 1984.

[33] A.J. Frank, L.D. Wittie, and A.J. Bernstein.
Multicast Communication on Network Computers.
IEEE Software Volume 2(Number 3):pp. 49-61, May, 1985.

[34] A.G. Fraser.
Datakit - A Modular Network for Synchronous and Asynchronous Traffic.
In *Proceedings I.C.C.*, pages 20.1.1-20.1.3. 1979.

[35] T.E. Fritz, J.E. Hefner, and T.M. Raleigh.
A Network of Computers Running the UNIX System.
AT&T Bell Laboratories Technical Journal Volume 63(Number 8, Part 2):pp. 1877-1896, October, 1984.

[36] G.H. Gobel and M.H. Marsh.
A Dual Processor VAX 11/780.
Technical Report Number TR-EE 81-31, Purdue University, September, 1981.

[37] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir.
The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer.
IEEE Transactions on Computers :pp. 175-189, February, 1983.

[38] Ed Gould.
Device Drivers in a Multiprocessor Environment.
In *USENIX Proceedings*, pages 357-360. June, 1985.

[39] M.E. Grzelakowski, J.H. Campbell, and M.R. Dubman.
DMERT Operating System.
Bell System Technical Journal Volume 62(Number 1):pp. 303-323, January, 1983.

[40] J.P. Haggerty.
SAFEGUARD Data-Processing System: Central Logic and Control Operating System.
Bell System Technical Journal :pp. S89-S99, 1975.
Safeguard Supplement.

[41] R.C. Hansen, R.W. Peterson, and N.O. Whittington.
Fault Detection and Recovery.
Bell System Technical Journal Volume 62(Number 1):pp. 349-366, January, 1983.

[42] M.J. Hatch, M. Katz, and J. Rees.
AT&T's RFS and SUN's NFS: A Comparison of Heterogeneous Distributed File Systems.
UNIX/World :pp. 38-52, December, 1985.

[43] C.A.R. Hoare.
Monitors: An operating system structuring concept.
Communications of the ACM Volume 17(Number 10):pp. 549-557, October, 1974.

[44] R.C. Holt, M.P. Mendel, S.G. Perelgut.
TUNIS: A Portable, UNIX Compatible Kernel Written in Concurrent Euclid.
In *USENIX Proceedings*, pages 61-74. July, 1983.

[45] *IBM System/370 Principles of Operation*
IBM Corporation, 1970.
Form number GA22-7000.

[46] Jack Inman.
Implementing Loosely Coupled Functions on Tightly Coupled Engines.
In *USENIX Proceedings*, pages 277-298. June, 1985.

[47] Herb Jacobs.
A User Tunable Multiprocessor Scheduler.
In *1986 Winter USENIX Technical Conference*, pages 183-191. USENIX Association, January, 1986.

[48] Ian Johnstone.
Strength in Numbers.
UNIX Review :pp. 53-60, 1986.

[49] J.R. Kane, R.E. Anderson, and P.S. McCabe.
Overview, Architecture, and Performance of DMERT.
Bell System Technical Journal Volume 62(Number 1):pp. 291-302, January, 1983.

[50] Lawrence J. Kenah and Simon F. Bate.
VAX/VMS Internals and Data Structures.
Digital Press, 1984.

[51] T.J. Killian.
Processes as Files.
In *USENIX Proceedings*, pages 203-207. June, 1984.

[52] B.W. Lampson and D.D. Redell.
Experiences with Processes and Monitors in Mesa.
Communications of the ACM Volume 23(Number 2):pp. 105-116, February, 1980.

[53] B. Lampson.
Atomic Transactions.
Lecture Notes in Computer Science. Distributed Systems: An Advanced Course.
Springer-Verlag, 1981.

[54] K.A. Lantz, W.I. Nowicki, and M.M. Theimer.
An Emprical Study of Distributed Application Performance.
IEEE transactions on Software Engineering Volume SE-11(Number 10):pp. 1162-1174, October, 1985.

[55] Leach, P.J., Stumpf, B.L., Hamilton, J.A., and Levine, P.H.
UID's as internal names in a distributed file system.
Proceedings of the 1st Symposium on Principles of Distributed Computing :pp. 34-41, 1982.

[56] Leach, P.J., Levine, P.H., Douros, B.P., Hamilton, J.A., Nelson, D.L., and Stumpf, B.L.
The architecture of an integrated local network.
IEEE Journal Selected Areas of Communication :pp. 842-856, 1983.

[57] Paul J. Leach.
The Whys and Wherefores of Distributed Resource Sharing.
UNIX Review :pp. 20-27,90, May, 1985.

[58] S.J. Leffler, R.S. Fabry, W.N. Joy.
A 4.2BSD Interprocess Communication Primer.
Technical Report, University of California, Berkeley, 1983.

[59] S.J. Leffler, W.N. Joy, R.S. Fabry.
4.2BSD Networking Implementation Notes.
Technical Report, University of California, Berkeley, July, 1983.

[60] Mark D. Lerner, Gerald Q. Maguire Jr., and Salvatore J. Stolfo.
An Overview of the DADO Parallel Computer.
In *Proceedings*. National Computer Conference, 1985.

[61] J. Lions.
A Commentary on the UNIX Operating System.
Bell Laboratories, 1977.

[62] J. Lions.
UNIX Operating System Source Code, Level Six.
Bell Laboratories, 1977.

[63] G.J. Popek.
The Locus Distributed System Architecture.
Technical Report, Locus Computing Corporation, Santa Monica, California, 1983.

[64] H. Lorin and H.M. Deitel.
The Systems Programming Series: Operating Systems.
Addison-Wesley, 1981.

[65] B. Lyon.
Sun External Data Representation Specification.
Technical Report, Sun Microsystems, Inc., 1984.

[66] B. Lyon.
Sun Remote Procedure Call Specification.
Technical Report, Sun Microsystems, Inc., 1984.

[67] S. Madnick and J. Donovan.
Operating Systems.
MacGraw-Hill, 1974.

[68] M.K. McKusick, S.J. Leffler, R.S. Fabry, W.N. Joy.
A Fast File System for UNIX.
Technical Report, University of California, Berkeley, 1983.

[69] R.M.Metcalfe and D.R.Boggs.
ETHERNET: Distributed Packet Switching for Local Computer Networks.
Communications of the ACM :pp. 395-404, July, 1976.

[70] Mueller, Erik T., Moore, Johanna D., and Popek, Gerald J.
A Nested Transaction Mechanism for LOCUS.
In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 71-89.
Association for Computing Machinery, October, 1983.

[71] D.L. Nelson and P.J. Leach.
The Architecture and Applications of the Apollo Domain.
IEEE Computer Graphics :pp. 58-66, April, 1984.

[72] V.A. Norton and G.F. Pfister.
A Methodology for Predicting Multiprocessor Performance.
In *Proceedings*, pages 772-781. International Conference on Parallel Processing, 1985.

[73] J.W. Olson.
SAFEGUARD Data-Processing System: Architecture of the Central Logic and Control.
Bell System Technical Journal :pp. S41-S61, 1975.
Safeguard Supplement.

[74] Elliott I. Organick.
The Multics System.
Massachusetts Institute of Technology Press, 1972.

[75] Elliott I. Organick.
Computer System Organization: The B5700/B6700 Series.
Academic Press, 1973.

[76] M. Paul and H.J. Siegert (editors).
Lecture Notes in Computer Science. Number 190: *Distributed Systems: Methods and Tools for Specification, An Advanced Course* .
Springer-Verlag, 1985.

[77] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton, and J. Weiss.
The IBM Research Parallel Processor Prototype (RP3).
In *Proceedings*, pages 764-771. International Conference on Parallel Processing, 1985.

[78] R. Pike and P.J. Weinberger.
The Hideous Name.
In *USENIX Proceedings*, pages 563-568. June, 1985.

[79] R. Pike and D. Presotto.
Face the Nation.
In *USENIX Proceedings*, pages pp. 81-86. June, 1985.

[80] G.J. Popek, et al.
LOCUS: A Network Transparent, High Reliability Distributed System.
ACM SIGOPS Operating Systems Review Volume 15:pp. 169-177, December, 1981.

[81] Popek, G.J., Walker, B., English, R., Kline, C., and Thiel, G.
The LOCUS distributed operating system.
ACM SIGOPS Operating Systems Review Volume 17:pp. 49-70, October, 1983.

[82] David Butterfield and Gerald Popek.
Network Tasking in the LOCUS Distributed UNIX System.
In *USENIX Proceedings*, pages 62-71. June, 1984.

[83] D. Presotto and D.M. Ritchie.
Interprocess Communication in the 8th Edition Unix System.
In *USENIX Proceedings*, pages 309-316. June, 1985.

[84] D.M. Ritchie and K.L. Thompson.
The *UNIX* Time-Sharing System.
Bell System Technical Journal Volume 57(Number 6):pp. 1905-1930, July-August, 1978.

[85] D.M. Ritchie.
UNIX Time-Sharing System: A Retrospective.
Bell System Technical Journal Volume 57(Number 6):pp. 1947-1969, July-August, 1978.

[86] D.M. Ritchie.
A Stream Input-Output System.
AT&T Bell Laboratories Technical Journal Volume 63(Number 8, Part 2):pp. 1897-1910, October, 1984.

[87] Rothnie, J.B., et al.
Introduction to a System for Distributed Databases (SDD-1).
ACM Transactions on Database Systems Volume 5(Number 1):pp. 1-15, March, 1980.

[88] G.R. Sager and R.B. Lyon.
Distributed File System Strategies.
UNIX Review :pp. 28-33,94, May, 1985.

[89] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, R. Lyon.
The Design and Implementation of the Sun Network File System.
In *USENIX Proceedings*, pages 119-130. June, 1985.

[90] R. Sandberg.
Sun Network File System Protocol Specification.
Technical Report, Sun Microsystems, Inc., 1985.

[91] Charles L. Seitz.
Concurrent VLSI Architectures.
IEEE Transactions on Computers Volume C-33(Number 12):pp. 1247-1265, December, 1984.

[92] Omri Serlin.
Alliant Unveils Parallel Multiprocessor.
UNIX/World :pp. 17-27, December, 1985.

[93] Shaw, D.E., Stolfo, S.J., Ibrahim, H., Hillyer, B., Wiederhold, G., and Andrews, J.A.
The NON-VON Database Machine: A Brief Overview.
Database Engineering Volume 4(Number 2), December, 1981.

[94] D.P. Siewiorek, C.G. Bell, and A. Newell.
Computer Structures: Principles and Examples.
McGraw-Hill, 1982.

[95] Stolfo, S.J., Miranker, D.P., and Shaw, D.E.
Architecture and Applications of DADO: A Large Scale Parallel Computer for Artificial Intelligence.
In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence.* IJCAI, 1983.

[96] H.S. Stone.
Introduction to Computer Architecture.
SRA, 1980.

[97] M. Stonebraker.
Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES.
IEEE Transactions on Software Engineering Volume 5(Number 3):pp. 188-194, 1979.

[98] H. Sturgis, J. Mitchell, and J. Israel.
Issues in the Design and Use of a Distributed File System.
ACM SIGOPS Operating Systems Review Volume 14, 1980.

[99] A.S. Tanenbaum.
Computer Networks.
Prentice-Hall, 1981.

[100] Jack Test.
Concentrix -- A Unix for the Alliant Multiprocessor.
In *1986 Winter USENIX Technical Conference*, pages 172-182. USENIX Association, January, 1986.

[101] K.L. Thompson.
UNIX Implementation.
Bell System Technical Journal Volume 57(Number 6):pp. 1931-1946, July-August, 1978.

[102] W.F. Tichy and Z. Ruan.
Towards a Distributed File System.
In *USENIX Proceedings*, pages 87-97. 1984.

[103] W.N. Toy and L.E. Gallaher.
Overview and Architecture of the 3B20D Processor.
Bell System Technical Journal Volume 62(Number 1):pp. 181-190, January, 1983.

[104] I.L. Traiger, J. Gray, C.A. Galtieri, and B.G. Lindsay.
Transactions and Consistency in Distributed Database Systems.
ACM Transactions on Database Systems Volume 7(Number 3):pp. 323-342, September, 1982.

[105] Staff.
Trends: For the Record.
UNIX/World :p. 10, December, 1985.

[106] G.J. Popek and B.J. Walker.
Issues of Network Transparency and file replication in the Distributed Filesystem Component of Locus.
Technical Report CSD830905, University Of California, Los Angeles Computer Science Department, 1983.

[107] Bruce J. Walker and Gerald J. Popek.
The LOCUS Distributed File System.
1983.

[108] Bruce J. Walker.
Network Transparency and its Limits in a Distributed Operating System.
Technical Report CSD840228, University Of California, Los Angeles Computer Science Department, 1984.

[109] P.J. Weinberger.
The Version 8 Network File System.
In *USENIX Proceedings*, pages pp. 86. June, 1984.