DISTRIBUTED OPERATING SYSTEMS

1.

INTRODUCTION TO DISTRIBUTED SYSTEMS

The use of computers is in the process of undergoing a revolution. From 1945, when the modern computer era began, until about 1985, computers were large and expensive. Even minicomputers normally cost tens of thousands of dollars each. As a result, most organizations had only a handful of computers, and for lack of a way to connect them, these usually operated independently from one another.

Starting in the mid 1980s, however, two advances in technology began to change that situation. The first was the development of powerful microprocessors. Initially these were 8-bit machines, but soon 16, 32, and even 64-bit CPUs became common. Many of these had the computing power of a decent-sized mainframe (i.e., large) computer, but for a fraction of the price.

The second development was the invention of high-speed **local area networks** or **LANs**. These systems allowed dozens, or even hundreds, of machines to be connected in such a way that small amounts of information can be transferred between machines in a millisecond or so. Larger amounts of data can be moved between machines at rates of 10 million bits/sec and more.

The net result of these two technologies is that it is now not only feasible, but easy, to put together computing systems composed of large numbers of CPUs connected by a high-speed network. They are usually called **distributed systems**, in contrast to the previous **centralized systems** consisting of a single CPU, its memory, peripherals, and some terminals.

Distributed systems need radically different software than do centralized systems. In particular, the required operating systems for these distributed systems are only beginning to emerge. The first few steps have been taken, but there is still a long way to go. Nevertheless, enough is already known about these distributed operating systems that we can present the basic ideas.

1.1 GOALS

Just because it is possible to build distributed systems does not necessarily mean that it is a good idea. After all, with current technology it is possible to put 4 floppy disk drives on a personal computer. It is just that doing so would be pointless. In this section we will discuss the motivation and goals of typical distributed systems, and look at their advantages and disadvantages compared to traditional centralized systems.

1.1.1 Advantages of Distributed Systems over Centralized Ones

The real driving force behind the trend towards decentralization is economics. A quarter of a century ago, Herb Grosch stated what later came to be known as Grosch's law: the computing power of a CPU is proportional to the square of its price. By paying twice as much, you could get four times the performance. This observation fit the mainframe technology of its time quite well, and led most organizations to buy the largest single machine they could afford.

With microprocessor technology, Grosch's law no longer holds. For a few hundred dollars you can get a CPU chip that can execute more instructions per second than one of the largest 1980s mainframes. If you are willing to pay twice as much, you get the same CPU, only running at a slightly higher clock speed. As a result, the most cost effective solution is frequently to harness a large number of cheap CPUs together in a system. Thus reason number one for the trend towards distributed systems is that these systems

potentially have a much better price/performance ratio than a single large centralized system would.

A slight variation on this theme is the observation that a collection of microprocessors can not only give a better price/performance ratio than a single mainframe, but may yield an absolute performance that no mainframe can achieve at any price. For example, with current technology it is possible to build a system from 1000 modern CPU chips, each of which runs at 20 MIPS (Millions of Instructions Per Second) for a total performance of 20,000 MIPS. For a single processor (i.e., CPU) to achieve this, it would have to execute an instruction in 0.05 nsec (50 picosec). No existing machine even comes close to this, and both theoretical and engineering considerations make it unlikely for some time. Theoretically, Einstein's theory of relativity dictates that nothing can travel faster than light, which can only cover 1.5 cm in 50 picosec. Practically, a computer of that speed fully contained in a 1.5 cm cube would generate so much heat that it would melt. Thus whether the goal is normal performance at low cost or extremely high performance at greater cost, distributed systems have much to offer.

As an aside, some authors make a distinction between distributed systems, which are designed to allow many users to work together, and parallel systems, whose only goal is to achieve maximum speedup on a single problem, as our 20,000 MIPS machine might. We believe that this distinction is difficult to maintain because the design spectrum is really a continuum. We prefer to use the term "distributed system" in the broadest sense to denote any system in which multiple interconnected CPUs work together.

A next reason for building a distributed system is that some applications are inherently distributed. In a factory automation system controlling robots and machines all along an assembly line, it often makes sense to give each robot and machine its own computer to manage it. When these are connected, we have an industrial distributed system. Similarly, when all the branches of a bank are connected, we have a commercial distributed system.

Another potential advantage of a distributed system over a centralized one is higher reliability. By distributing the workload over many machines, a single chip failure will bring down at most one machine, leaving the rest intact. Ideally, if 5 percent of the machines are down at any moment, the system should be able to continue to work with a 5 percent loss in performance. For critical applications, such as control of nuclear reactors or aircraft, using a distributed system to achieve high reliability may be the dominant consideration.

Finally, incremental growth is also potentially a big plus. Often a company will buy a mainframe with the intention of doing all its work on it. If the company prospers and the workload grows, at a certain point the mainframe will no longer be adequate. The only solutions are to either replace the mainframe with a larger one (if it exists), or add a second mainframe. Both of these can wreak major havoc with the company's operations. In contrast, with a distributed system, it may be possible to simply add more processors to the system, thus allowing it to expand gradually as the need arises. These advantages are summarized in Table 1-l.

• Item	Description
Economics	Microprocessors offer a better price/performance than mainframes
Speed	A distributed system may have more total computing power than a mainframe
Inherent distribution	Some applications involve spatially separated machines
Reliability	If one machine crashes, the system as a whole can still survive
Incremental growth	Computing power can be added in small increments

Table 1-1. *Advantages of distributed systems over centralized ones.*

1.1.2 Advantages of Distributed Systems over Independent PCs

Given that microprocessors are a cost effective way to do business, why not just give everyone his own PC and let people work independently? For one thing, many users need to share data. For example, airline reservation clerks need access to the master data base of flights and existing reservations. Giving each clerk his own private copy of the entire data base would not work, since nobody would know which seats the other clerks had already sold. Shared data is absolutely essential to this and many other applications, so the machines must be interconnected. Interconnecting the machines leads to a distributed system. Data are not the only things that can be shared. Expensive peripherals, such as color laser printers, phototypesetters, and massive archival storage devices (e.g., optical juke boxes) are also candidates.

A third reason to connect a group of isolated computers into a distributed system is to achieve enhanced person-to-person communication. For many people, electronic mail has numerous attractions over paper mail, telephone, and FAX. It is much faster than paper mail, does not require both parties to be available at the same time as does the telephone, and unlike FAX, produces documents that can be edited, rearranged, stored in the computer, and manipulated with text processing programs.

Finally, a distributed system is potentially more flexible than giving each user an isolated personal computer. Although one model is to give each person a personal computer and connect them all with a LAN, this is not the only possibility. Another one is to have a mixture of personal and shared computers, perhaps of different sizes, and let jobs run on the most appropriate one, rather than always on the owner's machine. In this way, the workload can be spread over the computers more effectively, and the loss of a few machines may be compensated for by letting people run their jobs elsewhere. Table 1-2 summarizes these points.

Item	Description
Data sharing	Allow many users access to a common data base
Device sharing	Allow many users to share expensive peripherals like color printers
Communication	Make human-to-human communication easier, for example, by electronic mail
Flexibility	Spread the workload over the available machines in the most cost effective way

Table 1-2. *Advantages of distributed systems over isolated (personal) computers.*

1.1.3 Disadvantages of Distributed Systems

Although distributed systems have their strengths, they also have their weaknesses. We have already hinted at the worst problem: software. With the current state-of-the-art, we do not have much experience in designing, implementing, and using distributed software. What kinds of operating systems, programming languages, and applications are appropriate for these systems? How much should the users know about the distribution? How much should the system do and how much should the users do? The experts differ. As more research is done, this problem will diminish, but for the moment it should not be underestimated.

A second potential problem is due to the communication network. It can lose messages, which requires special software to handle, and it can become overloaded. When the network saturates, it must either be replaced or a second one must be added. In both cases, some portion of one or more buildings may have to be rewired at great expense, or network interface boards may have to be replaced (e.g., by fiber optics). Once the system comes to depend on the network, its loss or saturation can negate most of the advantages the distributed system was built to achieve.

Finally, the easy sharing of data, which we described above as an advantage, may turn out to be a two-edged sword. If people can conveniently access data all over the system, they may be equally able to conveniently access data that they have no business looking at. In other words, security is often a problem. For data that must be kept secret at all costs, it is often preferable to have a dedicated, isolated personal computer that has no network connections to any other machines, and is kept in a locked room with a secure safe in which all the floppy disks are stored. The disadvantages of distributed systems are summarized in Table 1-3.

Item	Description
Software	Little software exists at present for distributed systems
Networking	The network can saturate or cause other problems
Security	Easy access also applies to secret data

Table 1-3. *Disadvantages of distributed systems.*

Despite these potential problems, many people feel that the advantages outweigh the disadvantages, and it is expected that distributed systems will become increasingly important

in the coming years. In fact, it is likely that within a few years, most organizations will connect most of their computers into large distributed systems to provide better, cheaper, and more convenient service for the users.

1.2 HARDWARE CONCEPTS

Even though all distributed systems consist of multiple CPUs, there are several different ways the hardware can be organized, especially in terms of how they are interconnected and how they communicate. In this section we will take a brief look at distributed system hardware, in particular, how the machines are connected together. In the next section we will examine some of the software issues related to distributed systems.

Various classification schemes for multiple CPU computer systems have been proposed over the years, but none of them have really caught on and been widely adopted. Probably the most frequently cited taxonomy is Flynn's (l972), although it is fairly rudimentary. Flynn picked two characteristics that he considered essential: the number of instruction streams and the number of data streams. A computer with a single instruction stream and single data stream is called SISD. All traditional uniprocessor computers (i.e., those having only one CPU) fall in this category, from personal computers to large mainframes.

The next category is SIMD, single instruction stream, multiple data stream. This type refers to array processors with one instruction unit that fetches an instruction, and then commands many data units to carry it out in parallel, each with its own data. These machines are useful for computations that repeat the same calculation on many sets of data, for example, adding up all the elements of 64 independent vectors. Some supercomputers are SIMD.

The next category is MISD, multiple instruction stream, single data stream. No known computers fit this model. Finally comes MIMD, which essentially means a group of independent processors, each with its own program counter, program, and data. All distributed systems are MIMD, so this classification system is not tremendously useful for our purposes.

In Figure 1-1, we divide all MIMD computers into two groups: those that have shared memory, usually called **multiprocessors**, and those that do not, sometimes called **multicomputers**. The essential difference is this: In a multiprocessor, there is a single virtual address space that is shared by all CPUs. If any CPU writes, for example, the value 44 to word 1000, any other CPU subsequently reading from its word 1000 will get the value 44. All the machines share the same memory.

In contrast, in a multicomputer, every machine has its own private memory. If one CPU writes the value 44 to word 1000, when another CPU reads word 1000 it will get whatever value was there before. The write of 44 does not affect *its* memory at all. A common example of a multicomputer is a collection of personal computers connected by a network.

Each of these categories can be further divided based on the architecture of the interconnection network. In Figure 1-1 we describe these two categories as **bus** and **switched**. By bus we mean that there is a single network, backplane, bus, cable, or other medium that connects all the machines. Commercial cable television uses a scheme like this: the cable company runs a wire down the street, and all the subscribers have taps running from their televisions to it.

Switched systems do not have a single backbone like cable television. Instead there are individual wires from machine to machine, with many different wiring patterns in use.

Messages move along the wires, with an explicit switching decision made at each step to route the message along one of the outgoing wires. The world wide public telephone system is organized in this way.

Another dimension to our taxonomy is that in some systems the machines are **tightly coupled**, and in others they are **loosely coupled**. In a tightly-coupled system, the delay experienced when a message is sent from one computer to another is short, and the data rate is high, that is, the number of bits per second that can be transferred is large. In a loosely-coupled system, the opposite is true. The intermachine message delay is large, and the data rate is low. For example, two CPU chips on the same printed circuit board and connected by, wires etched onto the board are likely to be tightly coupled, whereas two computers connected by a 1200 bit/sec modem over the telephone system are certain to be loosely-coupled.

Tightly-coupled systems tend to be used more as parallel systems (working on a single problem) and loosely-coupled ones tend to be used as distributed systems (working on many unrelated problems), although this is not always true. One famous counterexample is a project in which hundreds of computers all over the world worked together trying to factor a huge number (about 100 digits). Each computer was assigned a different range of divisors to try, and they all worked on the problem in their spare time, reporting the results back by electronic mail when they finished.

On the whole, multiprocessors tend to be more tightly coupled than multicomputers, because they can exchange data at memory speeds, but some. fiber-optic based multicomputers can also work at memory speeds. Despite the vagueness of the terms "tightly-coupled" and "loosely-coupled", they are useful concepts.

In the following four sections, we will look at the four categories of Figure 1-1 in more detail, namely bus multiprocessors, switched multiprocessors, bus multicomputers, and switched multicomputers. Although these topics are not directly related to our main concern, distributed operating systems, they will shed some light on the subject because as we shall see, different categories of machines use different kinds of operating systems.

1.2.1 Bus-Based Multiprocessors

Bus-based multiprocessors consist of some number of CPUs all connected to a common bus, along with a memory module. A simple configuration is to have a high-speed backplane or motherboard, into which CPU and memory cards can be inserted. A typical bus has 32 address lines, 32 data lines, and perhaps 20 to 30 control lines, all of which operate in parallel. To read a word of memory, a CPU puts the address of the word it wants on the bus address lines, then puts a signal on the appropriate control lines to indicate that it wants to read. The memory responds by putting the value of the word on the data lines to allow the requesting CPU to read it in. Writes work in a similar way.

Since there is only one memory, if CPU *A* writes a word to memory and then CPU *B* reads that word back a microsecond later, *B* will get the value just written. A memory that has this property is said to be **coherent**. Coherence plays an important role in distributed operating systems in a variety of ways that we will study later.

The problem with this scheme is that with as few as 4 or 5 CPUs, the bus will usually be overloaded and performance will drop drastically. The solution is to add a high-speed **cache memory** between the CPU and the bus, as shown in Figure 1-2. The cache holds the most recently accessed words. All memory requests go through the cache. If the requested word is in the cache, the cache itself responds to the CPU, and no bus request is made. If the cache is large enough, the probability of success, called the **hit rate**, will be high, and the amount of bus traffic per CPU will drop dramatically, allowing many more CPUs in the

system. Cache sizes of 64K to 1M are common, which often gives a hit rate of 90 percent or more.

However, the introduction of caches also brings a serious problem with it. Suppose two CPUs, *A* and *B*, each read the same word into the respective caches. Then *A* overwrites the word. When *B* next reads that word, it gets its old value, not the value *A* just wrote. The memory is now incoherent, and the system hard to program.

Many researchers have studied this problem, and various solutions are known. Below we will sketch one of them. Suppose the cache memories are designed so that whenever a word is *written* to the cache, it also is written through to memory as well. Such a cache is, not surprisingly, called a **write-through cache**. In this design, cache hits for reads do not cause bus traffic, but cache misses for reads, and all writes, hits and misses, cause bus traffic.

In addition, all caches constantly monitor the bus. Whenever a cache sees a write occurring to a memory address present in its cache, it either removes that entry from its cache, or updates the cache entry with the new value. Such a cache is called a **snoopy cache** (or sometimes, snooping cache) because it is always snooping (eaves-dropping) on the bus. A design consisting of snoopy write-through caches is coherent and is invisible to the programmer. Nearly all bus-based multiprocessors use either this architecture or a closely-related one. Using it, it is possible to put about 32-64 CPUs on a single bus.

1.2.2 Switched Multiprocessors

To build a multiprocessor with more than 64 processors, a different method is needed to connect the CPUs with the memory. One possibility is to divide the memory up into modules, and connect them to the CPUs with a **crossbar switch**, as shown in Figure 1-3(a). Each CPU and each memory has a connection coming out of it, as shown. At every intersection is a tiny electronic **crosspoint switch** that can be opened and closed in hardware. When a CPU wants to access a particular memory, the crosspoint switch connecting them is momentarily closed, to allow the access to take place. The virtue of the crossbar switch is that many CPUs can be accessing memory at the same time, although if two CPUs try to access the same memory simultaneously, one of them will have to wait.

The downside of the crossbar switch is that with n CPUs and n memories, n^2 crosspoint switches are needed. For large n, this number can be prohibitive. As a result, people have looked for, and found, alternative switching networks that require fewer switches. The **omega network** of Figure 1-3(b) is one example. This network contains four 2x2 switches, each one having two inputs and two outputs. Each switch can route either input to either output. A careful look at the figure will show that with proper settings of the switches, every CPU can access every memory.

In the general case, with n CPUs and n memories, the omega network requires n switching stages, each containing $\log_2 n$ switches, for a total of $n\log_2 n$ switches. Although for large n this is considerably better than n^2, it is still substantial.

Furthermore, there is another problem: delay. For example, for $n = 1024$, there are 10 switching stages from the CPU to the memory, and another 10 for the requested word to come back again. Suppose the CPU is a modern RISC chip running at 50 MHz, that is, the instruction execution time is 20 nsec. If a memory request is to traverse a total of 20 switching stages (10 outbound and 10 back) in 20 nsec, the switching time must be 1 nsec. The complete multiprocessor will need 10,240 1-nsec switches. This is not going to be cheap.

People have attempted to reduce the cost by going to hierarchical systems. Some memory is associated with each CPU. Each CPU can access its own local memory quickly,

but accessing anybody else's memory is slower. This design gives rise to what is known as a **NUMA (NonUniform Memory Access)** machine. Although NUMA machines have better average access times than machines based on omega networks, they have the new complication that the placement of the programs and data becomes critical in order to make most access go to the local memory.

To summarize, bus-based multiprocessors, even with snoopy caches, are limited by the amount of bus capacity to about 64 CPUs at most. To go beyond that requires a switching network, such as a crossbar switch, an omega switching network, or something similar. Large crossbar switches are very expensive, and large omega networks are both expensive and slow. NUMA machines require complex algorithms for good software placement. The conclusion is clear: building a large, tightly-coupled, shared memory multiprocessor is difficult and expensive.

1.2.3 Bus-Based Multicomputers

On the other hand, building a multicomputer (i.e., no shared memory) is easy. Each CPU has a direct connection to its own local memory. The only problem left is how the CPUs communicate with each other. Clearly some interconnection scheme is needed here too, but since it is only for CPU-to-CPU communication, the volume of traffic will be several orders of magnitude lower than when the interconnection network is also used for CPU-to-memory traffic.

In Figure 1-4 we see a bus-based multicomputer. It looks topologically similar to the bus-based multiprocessor, but since there will be much less traffic over it, it need not be a high-speed backplane bus. In fact, it can be a much lower speed LAN (typically 10-100 Mbps, compared to 300 Mbps and up for a backplane bus). Thus Figure 1-4 is more often a collection of workstations on a LAN than a collection of CPU cards inserted into a fast bus (although the latter is definitely possible).

1.2.4 Switched Multicomputers

Our last category consists of switched multicomputers. Various interconnection networks have been proposed and built, but all have the property that each CPU has direct and exclusive access to its own, private memory. Figure 1-5 shows two popular topologies, a grid and a hypercube. Grids are easy to understand and lay out on printed circuit boards. They are best suited to problems that have an inherent two dimensional nature, such as graph theory or vision (e.g., robot eyes or analyzing photographs).

A **hypercube** is an *n*-dimensional cube. The hypercube of Figure 1-5(b) is 4-dimensional. It can be thought of as two ordinary cubes, each with 8 vertices and 12 edges. Each vertex is a CPU. Each edge is a connection between two CPUs. The corresponding vertices in each of the two cubes are connected.

To expand the hypercube to five dimensions, we would add another set of two interconnected cubes to the figure, and connect the corresponding edges in the two halves, and so on. For an n-dimensional hypercube, each CPU has n connections to other CPUs. Thus the complexity of the wiring increases only logarithmically with the size. Since only nearest neighbors are connected, many messages have to make several hops to reach their destination. However, the longest possible path also grows logarithmically with the size, in contrast to the grid, where it grows as the square root of the number of CPUs. Hypercubes with 1024 CPUs have been commercially available for several years, and hypercubes with as many as 16,384 CPUs are starting to become available.

1.3 SOFTWARE CONCEPTS

Although the hardware is important, the software is even more important. The image that a system presents to its users, and how they think about the system, is largely determined by the operating system software, not the hardware. In this section we will give an introduction to the various types of operating systems for the multiprocessors and multicomputers we have just studied, and discuss which kind of software goes with which kind of hardware.

It is more-or-less possible to distinguish two kinds of operating systems for multiple CPU systems: loosely coupled and tightly coupled. As we shall see, loosely and tightly-coupled software is roughly analogous to loosely and tightly-coupled hardware.

Loosely-coupled software allows machines and users of a distributed system to be fundamentally independent of one another, but still to interact to a limited degree where that is necessary. Consider a group of personal computers, each of which has its own CPU, its own memory, its own hard disk, and its own operating system, but which share some resources such as laser printers and data bases over a LAN. This system is loosely coupled, since the individual machines are clearly distinguishable, each with its own job to do. If the network should go down for some reason, the individual machines can still continue to run to a considerable degree, although some functionality may be lost (e.g., the ability to print files).

To show how difficult it is to make definitions in this area, now consider the same system as above, but without the network. To print a file, the user writes the file on a floppy disk, carries it to the machine with the printer, reads it in, and then prints it. Is this still a distributed system, only now even more loosely coupled? It's hard to say. From a fundamental point of view, there is not really any difference between communicating over a LAN and communicating by carrying floppy disks around. At most one can say that the delay and data rate are worse in the second example.

At the other extreme we might find a multiprocessor dedicated to running a single chess program in parallel. Each CPU is assigned a board to evaluate, and it spends its time examining that board and all the boards that can be generated from it. When the evaluation is finished, the CPU reports back the results, and is given a new board to work on. The software for this system, both the application program and the operating system required to support it, is clearly much more tightly coupled than in our previous example.

We have now seen four kinds of distributed hardware and two kinds of distributed software. In theory, there should be eight combinations of hardware and software. In fact, there are only four, because to the user, the interconnection technology is not visible. A multiprocessor is a multiprocessor, whether it uses a bus with snoopy caches or whether it uses an omega network. In the following sections we will look at some of the most common combinations of hardware and software.

1.3.1 Network Operating Systems

Let us start with loosely-coupled software on loosely-coupled hardware, since this is probably the most common combination at many organizations. A typical example is a network of engineering workstations connected by a LAN. In this model, each user has a workstation for his exclusive use. It may or may not have a hard disk. It definitely has its own operating system. All commands are normally run locally, right on the workstation.

However, it is sometimes possible for a user to remotely log into another workstation by using a command such as

```
rlogin machine
```

The effect of this command is to turn the user's own workstation into a remote terminal logged into the remote machine. Commands typed on the keyboard are sent to the remote machine, and output from the remote machine is displayed on the screen. To switch to a different remote machine, it is necessary to first log out, then to use the *rlogin* command to connect to another machine. At any instant, only one machine can be used, and the selection of the machine is entirely manual.

Networks of workstations often also have a remote copy command to copy files from one machine to another. For example, a command like

```
rcp machine1:file1 machine2:file2
```

might copy the file *file1* from *machine1* to *machine2* and give it the name *file2* there. Again here, the movement of files is explicit, and requires the user to be completely aware of where all files are located and where all commands are being executed.

While better than nothing, this form of communication is extremely primitive and has led system designers to search for more convenient forms of communication and information sharing. One approach is to provide a shared, global file system accessible from all the workstations. The file system is supported by one or more machines called **file servers**. The file servers accept requests from user programs running on the other (nonserver) machines, called **clients**, to read and write files. Each incoming request is examined and executed, and the reply is sent back, as illustrated in Figure 1-6.

File servers generally maintain hierarchical file systems, each with a root directory containing subdirectories and files. Workstations can import or mount these file systems, augmenting their local file systems with those located on the servers. For example, in Figure 1-7, two file servers are shown. One has a directory called *games*, while the other has a directory called *work*. These directories each contain several files. Both of the clients shown have mounted both of the servers, but they have mounted them in different places in their respective file systems. Client 1 has mounted them in its root directory, and can access them as */games* and */work*, respectively. Client 2, like client 1, has mounted *games* in its root directory, but regarding the reading of mail and news as a kind of game, has created a directory */games/work* and mounted work there. Consequently, it can access news using the path */games/work/news* rather than */work/news*.

While it does not matter where a client mounts a server in its directory hierarchy, it is important to notice that different clients can have a different view of the filesystem. The name of a file depends on where it is being accessed from, and how that machine has set up its file system. Because each of the workstations operates relatively independently from the rest, there is no guarantee that they all present the same directory hierarchy to their programs.

The operating system that is used in this kind of environment must manage the individual workstations, file servers and also take care of the communication between them. It is possible that the machines all run the same operating system, but this is not required. If the clients and servers run on different systems, then as a bare minimum they must agree on the format and meaning of all the messages that they may potentially exchange. In a situation like this, where each machine has a high degree of autonomy and there are few system-wide requirements, people usually speak of a **network operating system**.

One of the best-known network operating systems is Sun Microsystem's **Network File System**, universally known as **NFS**. It will be discussed in the chapter on distributed file systems.

1.3.2 True Distributed Systems

NFS is an example of loosely-coupled software on loosely-coupled hardware. Other than the shared file system, it is quite apparent to the users that the system consists of numerous computers, each one having a specific task. Each one can run its own operating system, and do whatever its owner wants. There is essentially no coordination at all, except for the rule that client-server traffic must obey the NFS protocols.

The next evolutionary step beyond this is tightly-coupled software on the same loosely-coupled (i.e., multicomputer) hardware. The goal of such a system is to create the illusion in the minds of the users that the entire network of computers is a single timesharing system, rather than a collection of distinct machines. This brings us to our definition of a distributed system:

A distributed system is one that runs on a collection of machines that do not have shared memory, yet looks to its users like a single computer.

Some authors refer to this property as the **single system image**. Others put it slightly differently, saying that a distributed system is one that runs on a collection of networked machines but acts like a **virtual uniprocessor**. No matter how it is expressed, the essential idea is that the users should not have to be aware of the existence of multiple CPUs in the system. No current system fulfills this requirement entirely yet, but a number of promising candidates are on the horizon. Some of these will be discussed later in this book.

What are some characteristics of a distributed system? To start with, there must be a single, global interprocess communication mechanism so that any process can talk to any other process. It will not do to have different mechanisms on different machines or different mechanisms for local communication and remote communication. There must also be a global protection scheme. Mixing access control lists, the UNIX protection bits, and capabilities will not give a single system image. Process management must also be the same everywhere. How processes are created, destroyed, started, and stopped must not vary from machine to machine. In short, the idea behind NFS, that any machine can do whatever it wants to as long as it obeys the NFS protocols when engaging in client-server communication, is not enough. Not only must there be a single set of system calls available on all machines, but these calls must be designed so that they make sense in a distributed environment.

As a logical consequence of having the same system call interface everywhere, it is normal that identical kernels run on all the CPUs in the system. Doing so makes it easier to coordinate activities that must be global. For example, when a process has to be started up, all the kernels have to cooperate in finding the best place to execute it. In addition, a global file system is needed, preferably one whose semantics are better defined than those of NFS.

Nevertheless, each kernel can have considerable control over its own local resources. For example, since there is no shared memory, it is logical to allow each kernel to manage its own memory. For example, if swapping or paging is used, the kernel on each CPU is the logical place to determine what to swap or page. There is no reason to centralize this authority. Similarly, if multiple processes are running on some CPU, it makes sense to do the scheduling right there too.

1.3.3 Multiprocessor Timesharing Systems

The last combination we wish to discuss is tightly-coupled software on tightly-coupled hardware. While various special-purpose machines exist in this category (such as dedicated data base machines), the most common general-purpose examples are multiprocessors such as those made by Sequent and Encore that are operated as a UNIX timesharing system, only with multiple CPUs instead of one CPU. To the outside world, a multiprocessor with 32 3 MIPS CPUs acts very much like a single 96 MIPS CPU. This is the single system image discussed above. Only implementing it on a multiprocessor makes life much easier, since the entire design can be centralized.

The key characteristic of this class of system is the existence of a single run queue: a list of all the processes in the system that are logically unblocked and ready to run. The run queue is a data structure kept in the shared memory. As an example, consider the system of Figure 1-8, which has three CPUs and five processes that are ready to run. All five processes are located in the shared memory, and three of them are currently executing: process *A* on CPU 1, process *B* on CPU 2, and process *C* on CPU 3. The other two processes, *D* and *E*, are also in memory, waiting their turn.

Now suppose that process *B* blocks waiting for I/O or its quantum runs out. Either way, CPU 2 must suspend it, and find another process to run. CPU 2 will normally begin executing operating system code (located in the shared memory). After having saved all of *B*'s registers, it will enter a critical region to run the scheduler to look for another process to run. It is essential that the scheduler be run as a critical region to prevent two CPUs from choosing the same process to run next. The necessary mutual exclusion can be achieved by using monitors, semaphores, or any of the other standard constructions.

Once CPU 2 has gained exclusive access to the run queue, it can remove the first entry, *D*, exit from the critical region, and begin executing *D*. Initially, execution will be slow, since CPU 2's cache is full of words belonging to that part of the shared memory containing process *B*, but after a little while, these will have been purged and the cache will be full of *D*'s code and data.

Because none of the CPUs have local memory and all programs are stored in the global shared memory, it does not matter on which CPU a process runs. If a long-running process is scheduled many times before it completes, on the average, it will spend about the same amount of time running on each CPU. The only factor that has any effect at all on CPU choice is the slight gain in performance when a process starts up on a CPU that is currently caching part of its address space. In other words, if all CPUs are idle, waiting for I/O, and one process becomes ready, it is slightly preferable to allocate it to the CPU it was last using, assuming that no other process has used that CPU since.

As an aside, if a process blocks for I/O on a multiprocessor, the operating system has the choice of suspending it or just letting it do busy waiting. If most I/O is completed in less time than it takes to do a process switch, then busy waiting is preferable. Some systems let the process keep its processor for a few milliseconds, in the hope that the I/O will complete soon, but if that does not occur before the timer runs out, a process switch is done.

An area in which this kind of multiprocessor differs appreciably from a network or distributed system is the organization of the file system. The operating system normally contains a traditional file system, including a single, unified block cache. When any process executes a system call, a trap is made to the operating system, which carries it out, using semaphores, monitors, or something equivalent to lock out other CPUs while critical sections are being executed or central tables are being accessed. In this way, when a WRITE system call is done, the central block cache is locked, the new data are entered into

the cache, and the lock released. Any subsequent READ call will see the new data, just as on a single processor system. On the whole, the file system is hardly different from a single processor file system. In fact, on some multiprocessors, one of the CPUs is dedicated to running the operating system; the other ones run user programs. This situation is undesirable, however, as the operating system machine is often a bottleneck.

It should be clear that the methods used on the multiprocessor to achieve the appearance of a virtual uniprocessor are not applicable to machines that do not have shared memory. Centralized run queues and block caches only work when all CPUs have access to them with very low delay. Although these data structures could be simulated on a network of machines, the communication costs make this approach prohibitively expensive.

Item	Network operating system	Distributed operating system	Multiprocessor operating system
Does it look like a virtual uniprocessor?	No	Yes	Yes
Do all have to run the same operating system?	No	Yes	Yes
How many copies of the operating system are there?	N	N	1
How is communication achieved?	Shared files	Messages	Shared memory
Are agreed upon network protocols required?	Yes	Yes	No
Is there a single run queue?	No	No	Yes
Does file sharing have well-defined semantics?	Usually no	Yes	Yes

Table 1-4. *Comparison of three different ways of organizing N CPUs.*

Table 1-4 shows some of the differences between the three kinds of systems we have examined above.

1.4 DESIGN ISSUES

In the previous sections we have looked at distributed systems and related topics from both the hardware and the software point of view. In the remainder of this chapter we will briefly look at some of the key design issues that people contemplating building a distributed operating system must deal with. We will come back to them in more detail later in the book.

1.4.1 Transparency

Probably the single most important issue is how to achieve the single system image. In other words, how do the system designers fool everyone into thinking that the collection of

machines is simply an old-fashioned single processor timesharing system? A system that realizes this goal is often said to be **transparent**.

Transparency can be achieved at two different levels. Easiest to do is to hide the distribution from the users. For example, when a UNIX user types *make* to recompile a large number of files in a directory, he need not be told that all the compilations are proceeding in parallel on different machines and are using a variety of file servers to do it. To him, the only thing that is unusual is that the performance of the system is half-way decent for a change. In terms of commands issued from the terminal and results displayed on the terminal, the distributed system can be made to look just like a single processor system.

At a lower level, it is also possible, but harder, to make the system look transparent to programs. In other words, the system call interface can be designed so that the existence of multiple processors is not visible. Pulling the wool over the programmer's eyes is harder than pulling the wool over the terminal user's eyes however. As we saw earlier, the semantics of file sharing are subtly different in NFS from UNIX. A programmer who is used to the UNIX semantics might well stumble over this problem. On the other hand, a distributed system could be designed to have well-defined, consistent semantics, even in the presence of file caching. It's just that these semantics might not be those of UNIX.

What does transparency really mean? It is one of those slippery concepts that sounds reasonable, but is hard to understand without looking at concrete examples. Let us look at a few to make the idea clearer. To start with, imagine a distributed system consisting of workstations each running some standard operating system. Normally system services (e.g., reading files) are obtained by issuing a system call that traps to the kernel. In such a system, remote files should be accessed the same way. A system in which remote files are accessed by explicitly setting up a network connection to a remote server and then sending messages to it is not transparent because remote services are then being accessed differently than local ones. The programmer can tell that multiple machines are involved, and this is not allowed.

The concept of transparency can be applied to several aspects of a distributed system, as shown in Table 1-5. **Location transparency** refers to the fact that in a true distributed system, users cannot tell where hardware and software resources such as CPUs, printers, files, and data bases are located. The name of the resource, must not secretly encode the location of the resource, so names like *machine1:prog.c* or */machine1/prog.c* are not acceptable.

Kind	Meaning
Location transparency	The users cannot tell where resources are located
Migration transparency	Resources can move at will without changing their names
Replication transparency	The users cannot tell how many copies exist
Concurrency transparency	Multiple users can share resources automatically
Parallelism transparency	Activities can happen in parallel without users knowing

Table 1-5. *Different kinds of transparency in a distributed system.*

Migration transparency means that resources must be free to move from one location to another without having their names change. In the example of Figure 1-7 we saw how server directories could be mounted in arbitrary places in the clients' directory hierarchy. Since a path like */work/news* does not reveal the location of the server, it is location transparent. However, now suppose that the folks running the servers decide that reading network news really falls in the category "games" rather than in the category "work". Accordingly, they move *news* from server 2 to server 1. The next time client 1 boots and mounts the servers in his customary way, he will notice that *work/news* no longer exists. Instead, there is a new entry */games/news*. Thus the mere fact that a file or directory has migrated from one server to another has forced it to acquire a new name because the system of remote mounts is not migration transparent.

If a distributed system has **replication transparency**, then the operating system is free to make additional copies of files and other resources on its own without the users noticing. Clearly in the previous example, automatic replication is impossible because the names and locations are so closely tied together. To see how replication transparency might be achievable, consider a collection of *n* servers logically connected to form a ring. Each server maintains the entire directory tree structure, but holds only a subset of the files themselves. To read a file, a client sends a message containing the full path name to any of the servers. That server checks to see if it has the file. If so, it returns the data requested. If not, it forwards the request to the next server in the ring, which then repeats the algorithm. In this system, the servers can decide by themselves to replicate any file on any or all servers, without the users having to know about this at all. Such a scheme is replication transparent because it allows the system to transparently make copies of heavily used files without the users even being aware this is happening.

Distributed systems usually have multiple, independent users. What should the system do when two or more users try to access the same resource at the same time? For example, what happens if two users try to update the same file at the same time? If the system is **concurrency transparent**, the users will not notice the existence of other users. One mechanism for achieving this form of transparency would be for the system to automatically lock a resource once someone had started to use it, unlocking it only when the access was finished. In this manner, all resources would only be accessed sequentially, never concurrently.

Finally we come to the hardest one, **parallelism transparency**. In principle, a distributed system is supposed to appear to the users as a traditional, uniprocessor timesharing system. What happens if a programmer knows that his distributed system has

1000 CPUs and he wants to use a substantial fraction of them for a chess program that evaluates boards in parallel? The theoretical answer is that together the compiler, runtime system, and operating system should be able to figure out how to take advantage of this potential parallelism without the programmer even knowing it. Unfortunately, the current state-of-the-art is nowhere near allowing this to happen. Programmers who actually want to use multiple CPUs for a single problem will have to program this explicitly, at least for the foreseeable future. Parallelism transparency can be regarded as the holy grail for distributed systems designers. When that has been achieved, the work will have been completed, and it will be time to move on to new fields.

1.4.2 **Flexibility**

The second key design issue is flexibility. It is important that the system be flexible because we are just beginning to learn about how to build distributed systems. It is likely that this process will incur many false starts and considerable backtracking. Design decisions that now seem reasonable may later prove to be wrong.: The best way to avoid problems is thus to keep one's options open.

Flexibility, along with transparency, is like parenthood and apple pie: who could possibly be against them? It is hard to imagine anyone arguing in favor of an inflexible system. However, things are not as simple as they seem. There are two schools of thought concerning the structure of distributed systems. One school maintains that each machine should run a traditional kernel that provides most services itself. The other maintains that the kernel should provide as little as possible, with the bulk of the operating system services available from user-level servers. These two models, known as the monolithic kernel and microkernel, respectively, are illustrated in Figure 1-9.

The monolithic kernel is basically today's centralized operating system augmented with networking facilities and the integration of remote services. Most system calls are made by trapping to the kernel, having the work performed there, and having the kernel return the desired result to the user process. With this approach, most machines have disks and manage their own local file systems. Many distributed systems that are extensions or imitations of UNIX use this approach because UNIX itself has a large, monolithic kernel.

If the monolithic kernel is the reigning champion, the microkernel is the up-and-coming challenger. Most distributed systems that have been designed from scratch use this method. The microkernel is more flexible because it does almost nothing. It basically provides just four minimal services:

- An interprocess communication mechanism.
- Some memory management.
- A limited amount of low-level process management and scheduling.
- Low-level input/output.

In particular, unlike the monolithic kernel, it does not provide the file system, directory system, full process management, or much system call handling. The services that the microkernel does provide are included because they are difficult or expensive to provide anywhere else. The goal is to keep it small.

All the other operating system services are generally implemented as user-level servers. To look up a name, read a file, or obtain some other service, the user sends a message to the appropriate server, which then does the work and returns the result. The

advantage of this method is that it is highly modular: there is a well-defined interface to each service (the set of messages the server understands), and every service is equally accessible to every client, independent of location. In addition, it is easy to implement, install, and debug new services, since adding or changing a service does not require stopping the system and booting a new kernel, as is the case with a monolithic kernel. It is precisely this ability to add, delete, and modify services that gives the microkernel its flexibility. Furthermore, users who are not satisfied with any of the official services are free to write their own.

As a simple example of this power, it is possible to have a distributed system with multiple file servers, one supporting MS-DOS file service and another supporting UNIX file service. Individual programs can use either or both, if they choose. In contrast, with a monolithic kernel, the file system is built into the kernel, and users have no choice but to use it.

The only potential advantage of the monolithic kernel is performance. Trapping to the kernel and doing everything there may well be faster than sending messages to remote servers. However, a detailed comparison of two distributed operating systems, one with a monolithic kernel (Sprite), and one with a microkernel (Amoeba), has shown that in practice this advantage is nonexistent. Other factors tend to dominate, and the small amount of time required to send a message and get a reply (typically about 1 msec) is usually negligible. As a consequence, it is likely that microkernel systems will gradually come to dominate the distributed systems scheme, and monolithic kernels will eventually vanish or evolve into microkernels.

1.4.3 **Reliability**

One of the original goals of building distributed systems was to make them more reliable than single processor systems. The idea is that if some machine goes down, some other machine takes over the job. In other words, theoretically the overall system reliability could be the Boolean OR of the component reliabilities. For example, with four file servers, each with a 0.95 chance of being up at any instant, the probability of all four being down simultaneously is $0.05^4 = 0.000006$, so the probability of at least one being available is 0.999994, far better than any individual server.

That is the theory. The practice is that current distributed systems sometimes count on a number of specific servers being up in order to function at all. As a result, some of them have an availability more closely related to the Boolean AND of the components, than the Boolean OR. In a widely-quoted remark, Leslie Lamport once defined a distributed system as: “One on which I cannot get any work done because some machine I have never heard of has crashed”. While this remark was (presumably) made somewhat tongue-in-cheek, there is clearly room for improvement here.

It is important to distinguish various aspects of reliability. **Availability**, as we have just seen, refers to the fraction of time that the system is usable. Lamport s system apparently did not score well in that regard. Availability can be enhanced by a design that does not require the simultaneous functioning of a substantial number of critical components. Another tool for improving availability is redundancy: key pieces of hardware and software should be replicated, so that if one of them fails the others will be able to take up the slack.

A highly reliable system must be highly available, but that is not enough. Data entrusted to the system must not be lost or garbled in any way, and if files are stored redundantly on multiple servers, all the copies must be kept consistent. In general, the more copies that are kept, the better the availability, but the greater the chance that they will be

inconsistent, especially if updates are frequent. The designers of all distributed systems must keep this dilemma in mind at all times.

Another aspect of overall reliability is **security**. Files and other resources must be protected from unauthorized usage. Although the same issue occurs in single-processor systems, in distributed systems it is more severe. In a single-processor system, the user logs in and is authenticated. From then on, the system knows who the user is and can check whether each attempted access is legal. In a distributed system, when a message comes in to a server asking for something, the server has no simple way of determining who it is from. No name or identification field in the message can be trusted, since the sender may be lying. At the very least, considerable care is required here.

Still another issue relating to reliability is **fault tolerance**. Suppose a server crashes and then quickly reboots. What happens? Does the server crash bring down users with it? If the server has tables containing important information about ongoing activities, recovery will be difficult at best. We saw earlier than NFS was designed to be stateless, just to avoid this problem.

In general, distributed systems can be designed to mask failures, that is, to hide them from the users. If a file service or other service is actually constructed from a group of closely cooperating servers, then it should be possible to construct it in such a way that users do not notice the loss of one or two servers, other than some performance degradation. Of course, the trick is to arrange this cooperation so that it does not add substantial overhead to the system in the normal case, when everything is functioning correctly.

1.4.4 Performance

Lurking in the background all the time is the issue of performance. Building a transparent, flexible, reliable distributed system will not win you any prizes if it is as slow as molasses. In particular, when running a particular application on a distributed system, it should not be appreciably worse than running that same application on a single processor. Unfortunately, achieving this is easier said than done.

Various performance metrics can be used. Response time is one, but so are throughput (number of jobs per hour), system utilization, and amount of network capacity consumed. Furthermore, the results of any benchmark are often highly dependent on the nature of the benchmark. A benchmark that involves a large umber of independent highly CPU-bound computations may give radically different results than a benchmark that consists of scanning a single large file for some pattern.

The performance problem is compounded by the fact that communication, which is essential in a distributed system (and absent in a single-processor system) is typically quite slow. Sending a message and getting a reply over a LAN takes about 1 msec. Most of this time is due to unavoidable protocol handling on both ends, rather than the time the bits spend on the wire. Thus to optimize performance, one often has to minimize the number of messages. The difficulty with this strategy is that the best way to gain performance is to have many activities running in parallel on different processors, but doing so requires sending many messages. (Another solution is to do all the work on one machine, but that is hardly appropriate in a distributed system.)

One possible way out is to pay considerable attention to the **grain size** of all computations. Starting up a small computation remotely, such as adding two integers, is rarely worth it, because the communication overhead dwarfs the extra CPU cycles gained. On the other hand, starting up a long compute-bound job remotely may be worth the trouble. In general, jobs that involve a large number of small computations, especially ones

that interact highly with one another, may cause trouble on a distributed system with relatively slow communication. Such jobs are said to exhibit **fine-grained parallelism**. On the other hand, jobs that involve large computations, low interaction rates, and little data, that is, **coarse-grained parallelism**, may be a better fit.

Fault tolerance also exacts its price. Good reliability is often best achieved by having several servers closely cooperating on a single request. For example, when a request comes in to a server, it could immediately send a copy of the message to one of its colleagues so that if it crashes before finishing, the colleague can take over. Naturally when it is done, it must inform the colleague that the work has been completed, which takes another message. Thus we have at least two extra messages, which in the normal case cost time and network capacity and produce no tangible gain.

1.4.5 Scalability

Most current distributed systems are designed to work with a few hundred CPUs. It is possible that future systems will be orders of magnitude larger, and solutions that work well for 200 machines will fail miserably for 200,000,000. Consider the following. The French PTT (Post, Telephone and Telegraph administration) is in the process of installing a terminal in every household and business in France. The terminal, known as a **minitel**, will allow on-line access to a data base containing all the telephone numbers in France, thus eliminating the need for printing and distributing expensive telephone books. It will also vastly reduce the need for information operators who do nothing but give out telephone numbers all day. It has been calculated that the system will pay for itself within a few years. If the system works in France, other countries will inevitably adopt similar systems.

Once all the terminals are in place, the possibility of also using them for electronic mail (especially in conjunction with printers-s) is clearly present. Since postal services lose a huge amount of money in every country in the world, and telephone services are enormously profitable, there are great incentives to having electronic mail replace paper mail.

Next comes interactive access to all kinds of data bases and services, from electronic banking to reserving places in planes, trains, hotels, theaters, and restaurants, to name just a few. Before long, we have a distributed system with tens of millions of users. The question is: “Will the methods we are currently developing scale to such large systems?”

Although little is known about such huge distributed systems, one guiding principle is clear: avoid centralized components, tables, and algorithms (see Table 1-6). Having a single mail server for 50 million users would not be a good idea. Even if it had enough CPU and storage capacity, the network capacity into and out of it would surely be a problem. Furthermore, the system would not tolerate faults well. A single power outage could bring the entire system down. Lastly, most mail is local. Having a message sent by a user in Marseille to another user two blocks away pass through a machine in Paris is obviously inefficient.

Concept	Example
Centralized components	A single mail server for all users
Centralized tables	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

Table 1-6. *Potential bottlenecks that designers should try to avoid in very large distributed systems.*

Centralized tables are almost as bad as centralized components. How should one keep track of the telephone numbers and addresses of 50 million people? Suppose each data record could be fit into 50 characters. A single 2.5 gigabyte disk would provide enough storage. But here again, having a single data base would undoubtedly saturate all the communication lines into and out of it. It would also be vulnerable to failures (a single speck of dust could cause a head crash and bring down the entire directory service). Furthermore, here too, valuable network capacity would be wasted shipping queries far away for processing.

Finally, centralized algorithms are also a bad idea. In a large distributed system, an enormous number of messages have to be routed over many lines. From a theoretical point of view, the optimal way to do this is collect complete information about the load on all machines and lines, and then run a graph theory algorithm to compute all the optimal routes. This information can then be spread around the system to improve the routing.

The trouble is that collecting and transporting all the input and output information would again be a bad idea for the reasons discussed above. In fact, any algorithm that operates by collecting information from all sites, sends it to a single machine for processing, and then distributes the results must be avoided. Only decentralized algorithms should be used. These have the following characteristics:

- No machine has complete information about the state of the system.
- Machines make decisions based only on locally available information.
- Failure of one machine does not ruin the algorithm.
- There is no implicit assumption that a global clock exists.

The first three follow from what we have said so far. The last is perhaps less obvious, but also important. Any algorithm that starts out with: “At precisely 12:00:00 all machines shall note the size of their output queue” will fail because it is impossible to get all the clocks exactly synchronized. Algorithms should take into account the lack of exact clock synchronization. The larger the system, the larger the uncertainty. On a single LAN, with considerable effort it may be possible to get all clocks synchronized down to a few milliseconds, but doing this nationally is tricky.

1.5 SUMMARY

Distributed systems have a number of potential selling points. They can offer good price/performance ratios, are a good fit to distributed applications, can be made highly

reliable, and can be gradually increased in size as the workload grows. They also have some disadvantages, such as having more complex software, potential communication bottlenecks, and weak security. Nevertheless, there is considerable interest worldwide in building and installing them.

Modern computer systems often have multiple CPUs. These can be organized as multiprocessors (with shared memory) or as multicomputers (without shared memory). Both types can be bus-based or switched. The former tend to be tightly coupled, while the latter tend to be loosely coupled.

The software for multiple CPU systems can be divided into three rough classes. Network operating systems allow users at independent workstations to communicate via a shared file system but otherwise leave each user as the master of his own workstation. Distributed operating systems turn the entire collection of hardware and software into a single integrated system, much like a traditional timesharing system. Shared-memory multiprocessors also offer a single system image, but do so by centralizing everything, so there really is only a single system. Shared-memory multiprocessors are not distributed systems.

Distributed systems have to be designed carefully, since there are many pitfalls for the unwary. A key issue is transparency – hiding all the distribution from the users and even from the application programs. Another issue is flexibility. Since the field is only now in its infancy, the design should be made with the idea of making future changes easy. In this respect, microkernels are superior to monolithic kernels. Other important issues are reliability, performance and scalability.

2.

PROCESSES AND PROCESSORS IN DTOS

Although processes are also an important concept in uniprocessor systems, in this chapter we will emphasize aspects of process management that are usually not studied in the context of classical operating systems. In particular, we will look at how the existence of multiple processors is dealt with.

In many distributed systems, it is possible to have multiple threads of control within a process. This ability provides some important advantages, but also introduces various problems. We will study these issues first. Then we come to the subject of how the processors and processes are organized, and see that several different models are possible. Finally we will look at processor allocation and scheduling in distributed systems.

2.1 THREADS

In most traditional operating systems, each process has an address space and a single thread of control. In fact, that is almost the definition of a process. Nevertheless, there are frequently situations in which it is desirable to have multiple threads of control that share a single address space, but run in quasi-parallel, as though they were in fact separate processes (except for the shared address space). In this section we will discuss these situations and how they can be best handled.

2.1.1 Introduction to Threads

Consider, for example, a file server that has to block occasionally waiting for the disk. If the server had multiple threads of control, a second thread could run while the first one was sleeping. The net result would be a higher throughput and better performance. It is not possible to achieve this goal by creating two independent server processes because they must share a common buffer cache, which requires them to be in the same address space. Thus a new mechanism is needed, one that is generally not found in single processor operating systems (although there is no reason why it could not be).

Figure 2-1. *(a) Three processes with one thread each. (b) One process with three threads.*

In Figure 2-1(a) we see a machine with three processes. Each process has its own program counter, its own stack, its own register set, and its own address space. The processes have nothing to do with each other, except that they may be able to communicate through the system's interprocess communication primitives, such as semaphores, monitors, or messages. In Figure 2-1(b) we see another machine, with one process. Only this process contains multiple threads of control, usually just called **threads**, or sometimes **lightweight processes**. In many respects, threads are like little mini-processes. Each thread runs strictly sequentially, and has its own program counter and stack to keep track of where it is. Threads share the CPU, just as processes do: first one thread runs, then another does (timesharing). Only on a multiprocessor do they actually run in parallel. Threads can create child threads and can block waiting for system calls to complete, just like regular processes. While one thread is blocked, another thread in the same process can run, in exactly the same way that when one process blocks, another process in the same machine can run. The analogy, thread is to process as process is to machine, holds in many ways.

Different threads in a process are not quite as independent as different processes, however. All threads have exactly the same address space, which means they also share the same global variables. Since every thread can access every virtual address, one thread can read, write, or even completely wipe out another thread's stack. There is no protection

between threads because (1) it is impossible, and (2) it should not be necessary. Unlike different processes, which may be from different users and which may be hostile to one another, a process is always owned by a single user, who has presumably created multiple threads so they can cooperate, not fight. In addition to sharing an address space, all the threads share the same set of open files, child processes, timers, and signals, etc. as shown in Table 2-1. Thus the organization of Figure 2-1(a) would be used when the three processes are essentially unrelated, whereas Figure 2-1(b) would be appropriate when the three threads are actually part of the same job and are actively and closely cooperating with each other.

Per thread items	Per process items
Program counter	Address space
Stack	Global variables
Register set	Open files
Child threads	Child processes
State	Timers
	Signals
	Semaphores
	Accounting information

Table 2-1. *Per thread and per process concepts.*

Like traditional processes (i.e., processes with only one thread), threads can be in any one of several states: running, blocked, ready, or terminated. A running thread currently has the CPU and is active. A blocked thread is waiting for another thread to unblock it (e.g., on a semaphore). A ready thread is scheduled to run, and will as soon as its turn comes up. Finally, a terminated thread is one that has exited, but which has not yet been collected by its parent (in UNIX terms, the parent thread has not yet done a WAIT).

2.1.2 Thread Usage

Threads were invented to allow parallelism to be combined with sequential execution and blocking system calls. Consider our file server example again. One possible organization is shown in Figure 2-2(a). Here one thread, the **dispatcher**, reads incoming requests for work from the system mailbox. After examining the request, it chooses an idle (i.e., blocked) **worker thread** and hands it the request, most likely by writing a pointer to the message into a special word associated with each thread. The dispatcher then wakes up the sleeping worker (e.g., by doing an UP on the semaphore on which it is sleeping).

When the worker wakes up, it checks to see if the request can be satisfied from the shared block cache, to which all threads have access. If not, it sends a message to the disk to get the needed block (assuming it is a READ), and goes to sleep awaiting completion of the disk operation. The scheduler will now be invoked, and another thread will be started, possibly the dispatcher, in order to acquire more work, or possibly another worker that is now ready to run.

Figure 2-2. *Three organizations of threads in a process. (a) Dispatcher/worker model. (b) Team model. (c) Pipeline model.*

Consider how the file server could be written in the absence of threads. One possibility is to have it operate as a single thread. The main loop of the file server gets a request, examines it and carries it out to completion before getting the next one. While

waiting for the disk, the server is idle and does not process any other requests. If the file server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the file server is waiting for the disk. The net result is that many fewer requests/sec can be processed. Thus threads gain considerable performance, but each thread is programmed sequentially, in the usual way.

So far we have seen two possible designs: a multithreaded file server and a single-threaded file server. Suppose that threads are not available, but the system designers find the performance loss due to single threading unacceptable. A third possibility is to run the server as a big finite-state machine. When a request comes in, the one and only thread examines it. If it can be satisfied from the cache, fine, but if not, send a message to the disk.

However, instead of blocking, it records the state of the current request in a table, and then goes and gets the next message. The next message may either be a request for new work, or a reply from the disk about a previous operation. If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table, and the reply processed. Since it is not permitted to send a message and block waiting for a reply here, RPC cannot be used. The primitives must be non-blocking calls to send and receive.

In this design, the "sequential process" model that we had in the first two cases is lost. The state of the computation must be explicitly saved and restored in the table for every message sent and received. In effect, we are simulating the threads and their stacks the hard way. The process is being operated as a finite-state machine that gets an event and then reacts to it, depending on what is in it.

It should now be clear what threads have to offer. They make it possible to retain the idea of sequential processes that make blocking system calls (e.g., RPC to talk to the disk) and still achieve parallelism. Blocking system calls make programming easier and parallelism improves performance. The single-threaded server retains the ease of blocking system calls, but gives up performance. The finite-state machine approach achieves high performance through parallelism, but uses nonblocking calls and thus is hard to program. These models are summarized in Table 2-2.

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-thread process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

Table 2-2. *Three ways to construct a server.*

The dispatcher structure of Figure 2-2(a) is not the only way to organize a multithreaded process. The **team** model of Figure 2-2(b) is also a possibility. Here all the threads are equals, and each one gets and processes its own requests. There is no dispatcher. Sometimes work comes in that a thread cannot handle, especially if each thread is specialized to handle a particular kind of work. In this case, a job queue can be maintained, with pending work kept in the job queue. With this organization, a thread should first check the job queue before looking in the system mailbox.

Threads can also be organized in the **pipeline** model of Figure 2-2(c). In this model, the first thread generates some data and passes them on to the next one for processing. The data continue from thread to thread, with processing going on at each step. Although this is

not appropriate for file servers, for other problems, such as the producer-consumer, it may be a good choice. Pipelining is widely used in many areas of computer systems, from the internal structure of RISC CPUs to UNIX command lines.

Threads are frequently also useful for clients. For example, if a client wants a file to be replicated on multiple servers, it can have one thread talk to each server. Another use for client threads is to handle signals, such as interrupts from the keyboard (DEL or BREAK). Instead of letting the signal interrupt the process, one thread is dedicated full time to waiting fur signals. Normally it is blocked, but when a signal comes in, it wakes up and processes the signal. Thus using threads can eliminate the need for user-level interrupts.

Another argument for threads has nothing to do with RPC or communication. Some applications are easier to program using parallel processes, the producer-consumer problem for example. Whether the producer and consumer actually run in parallel is secondary. They are programmed that way because it makes the software design simpler. Since they must share a common buffer, having them in separate processes will nut do. Threads fit the bill exactly here.

Finally, although we are not explicitly discussing the subject here, in a multiprocessor system, it is possible fur the threads in a single address space to actually run in parallel, on different CPUs. This is, in fact, one of the major ways sharing is done on such systems. On the other hand, a properly designed program that uses threads should work equally well on a single CPU that timeshares the threads or on a true multiprocessor, so the software issues are pretty much the same either way.

2.1.3 Design Issues for Threads Packages

A set of primitives (e.g., library calls) available to the user relating to threads is called a **threads package**. In this section we will consider some of the issues concerned with the architecture and functionality of threads packages. In the next one we will consider how threads packages can be implemented.

The first issue we will look at is thread management. Two alternatives are possible here, static threads and dynamic threads. With a static design, the choice of how many threads there will be is made when the program is written or when it is compiled. Each one is allocated a fixed stack. This approach is simple, but inflexible.

A more general approach is to allow threads to be created and destroyed on-the-fly during execution. The thread creation call usually specifies the thread's main program (as a pointer to a procedure) and a stack size, and may specify other parameters as well, for example, a scheduling priority. The call usually returns a thread identifier to be used in subsequent calls involving the thread. In this model, a process starts out with one (implicit) thread, but can create more threads as needed.

Threads can be terminated in one of two ways. A thread can exit of its own accord when it finishes its job, or it can be killed from outside. In this respect, threads are like processes. In many situations, such as the file servers of Figure 2-2, the threads are created immediately after the process starts up and are never killed.

Since threads share a common memory, they can, and usually do, use it for holding data that are shared among multiple threads, such as the buffers in a producer-consumer system. Access to shared data is usually programmed using critical regions, to prevent multiple threads from trying to access the same data at the same time. Critical regions are most easily implemented using semaphores, monitors, and similar constructions. One technique that is commonly used in threads packages is the **mutex**, which is a kind of watered-down semaphore. A mutex is always in one of two states, unlocked or locked. Two operations are defined on mutexes. The first one, LOCK, attempts to lock the mutex.

If the mutex is unlocked, the LOCK succeeds and the mutex becomes locked in a single atomic action. If two threads try to lock the same mutex at exactly the same instant, an event that is only possible on a multiprocessor, on which different threads run on different CPUs, one of them wins and the other loses. A thread that attempts to lock a mutex that is already locked, such as the loser above, is blocked.

The UNLOCK operation unlocks a mutex. If one or more threads are waiting on the mutex, exactly one of them is released. The rest continue to wait.

Another operation that is sometimes provided is TRYLOCK, which attempts to lock a mutex. If the mutex is unlocked, TRYLOCK returns a status code indicating success. If, however, the mutex is locked, TRYLOCK does not block the thread. Instead it returns a status code indicating failure.

Mutexes are like binary semaphores (i.e., semaphores that may only have the values 0 or l). They are not like counting semaphores. Limiting them in this way makes them easier to implement.

Another synchronization feature that is sometimes available in threads packages is the **condition variable**, which is similar to the condition variable used for synchronization in monitors. Each condition variable is normally associated with a mutex at the time it is created. The difference between mutexes and condition variables is that mutexes are used for short-term locking, mostly for guarding the entry to critical regions. Condition variables are used for long-term waiting until a resource becomes available.

The following situation occurs all the time. A thread locks a mutex to gain entry to a critical region. Once inside the critical region, it examines system tables and discovers that some resource it needs is busy. If it simply locks a second mutex (associated with the resource), the outer mutex will remain locked and the thread holding the resource will not be able to enter the critical region to free it. Deadlock results. Unlocking the outer mutex lets other threads into the critical region, causing chaos.

One solution is to use condition variables to acquire the resource, as shown in Example 2-1(a). Here, waiting on the condition variable is defined to automatically perform the wait and unlock the mutex atomically. Later, when the thread holding the resource frees it, as shown in Example 2-1(b), it calls *wakeup*, which is defined to either wakeup exactly one thread, or all the threads waiting on the specified condition variable. The use of WHILE instead of IF in Example 2-1(a) guards against the case that the thread is awakened but that someone else seizes the resource before the thread runs.

```
lock mutex;
      check data structures;
      while (resource busy)
            wait(condition variable);
      mark resource as busy;
unlock mutex;
```

(a)

```
lock mutex;
      mark resource as free;
unlock mutex;
wakeup(condition variable);
```

(b)

Example 2-1. *Use of mutexes and condition variables.*

The need for the ability to wake up all the threads, rather than just one, is demonstrated in the reader-writer problem. When a writer finishes, it may choose to wake up pending writers or pending readers. If it chooses readers, it should wake them all up, not just one. Providing thread primitives for waking up exactly one thread and for waking up all the threads provides the needed flexibility.

The code of a thread normally consists of multiple procedures, just like a process. These may have local variables, global variables, and procedure parameters. Local variables and parameters do not cause any trouble, but variables that are global to a thread but not global to the entire program can cause difficulty.

As an example, consider the *errno* variable maintained by UNIX. When a process or a thread) makes a system call that fails, the error code is put into *errno*. In Figure 2-3, thread 1 executes the system call ACCESS to find out if it has permission to access a certain file. The operating system returns the answer in the global variable *errno*. After control has returned to thread 1, but before it has a chance to read *errno*, the scheduler decides that thread 1 has had enough CPU time for the moment and decides to switch to thread 2. Thread 2 executes an OPEN call that fails, which causes *errno* to be overwritten and thread 1's access code to be lost forever. When thread 1 starts up later, it will read the wrong value and behave incorrectly.

Figure 2-3. *Conflicts between threads over the use of a global variable.*

Various solutions to this problem are possible. One is to prohibit global variables altogether. However worthy this ideal may be, it conflicts with much existing software, such as UNIX. Another is to assign each thread its own private global variables, as shown in Figure 2-4. In this way, each thread has its own private copy of *errno* and other global variables, so conflicts are avoided. In effect, this decision creates a new scoping level, variables visible to all the procedures of a thread, in addition to the existing scoping levels of variables visible only in a specific procedure and variables visible everywhere in the program.

Figure 2-4. *Threads can have private global variables.*

Accessing the private global variables is a bit tricky, however, since most programming languages have a way of expressing local variables and global variables, but not intermediate forms. It is possible to allocate a chunk of memory for the globals and pass it to each procedure in the thread, as an extra parameter. While hardly an elegant solution, it works.

Alternatively, new library procedures can be introduced to create, set, and read these thread-wide global variables. The first call might look like this:

```
create_global ("bufptr") ;
```

It allocates storage for a pointer called *bufptr* on the heap or in a special storage area reserved for the calling thread. No matter where the storage is allocated, only the calling thread has access to the global variable. If another thread creates a global variable with the same name, it gets a different storage location that does not conflict with the existing one.

Two calls are needed to access global variables: one for writing them and the other for reading them. For writing, something like

```
set_global ("bufptr", &buf);
```

will do. It stores the value of a pointer in the storage location previous created by the call to *create_global*. To read a global variable, the call might look like

```
bufptr = read_global ("bufptr") ;
```

This call returns the address stored in the global variable, so the data value can be accessed.

Our last design issue relating to threads is scheduling. Threads can be scheduled using various scheduling algorithms, including priority, round robin, and others. Threads packages

often provide calls to give the user the ability to specify the scheduling algorithm and set the priorities, if any.

2.1.4 An Example Threads Package

As an example threads package, in this section we will examine the Open Software Foundation's Distributed Computing Environment's threads package. Like most of OSF's software, this package is large and complicated. It has a total of 51 primitives (library procedures) relating to threads that user programs can call. Many of these are not strictly necessary, but are provided for convenience only. This approach is somewhat analogous to a four-function pocket calculator that not only has keys for +, -, x, and /, also has keys for +1, -1, x 2, x 10, x π, /2, and /10 on the grounds that these save the user time and effort. Due to the large number of calls, we will discuss only the most important ones (about half the total). Nevertheless, our treatment should give a reasonable impression of what the package can do.

Call	Description
Create	Create a new thread
Exit	Call by a thread when it is finished
Join	Like the WAIT system call in UNIX
Detach	Make it unnecessary for parent thread to wait when caller exits

Table 2-3. *Selected DCE thread calls for managing threads. All the calls in this and subsequent tables are actually prefixed by pthread_ (i.e., pthread_create, not create), which we have omitted to save space.*

For our discussion, it is convenient to group the calls into seven categories, each dealing with a different aspect of threads and their use. The first category, listed in Table 2-3, deals with thread management. These calls allow threads to be created, and for them to exit when done. A parent thread can wait for a child using join, which is similar to the WAIT system call in UNIX. If a parent has no interest in a child and does plan to wait for it, the parent can disown the child by calling *detach*.. In this case, when the child thread exits, its storage is reclaimed immediately, instead of having it wait for the parent to call *join*.

The DCE package allows the user to create, destroy, and manage templates for threads, mutexes, and condition variables. The templates can be set up to have appropriate initial values. When an object is created, one of the parameters to the create call is a pointer to a template. Thus, for example, a thread template can be created and given the attribute (property) that the stack size is 8K. Whenever a thread is created with that template as parameter, it will get an 8K stack. The point of having templates is to eliminate the need for specifying all the options as separate parameters. As the package evolves, the create calls can remain the same. Instead, new attributes can be added to the templates. Some of the template calls are listed in Table 2-4.

The *attr_create* and *attr_delete* calls create and delete thread templates, respectively. Other calls allow programs to read and write the template's attributes, such as the stack size and scheduling parameters to be used for threads created with the template. Similarly, calls are provided to create and delete templates for mutexes and condition variables. The need for the latter is not entirely obvious, since they have no attributes and no operations are

defined on them. Clearly the designers were hoping that someone would one day think of an attribute.

The third group deals with mutexes, which can be created and destroyed dynamically. Three operations are defined on mutexes, as shown in Table 2-5. The operations are for locking, unlocking mutexes, and for trying but accepting failure if locking cannot be done.

Call	Description
Attr_create	Create template for setting thread parameters
Attr_delete	Delete template for threads
Attr_setprio	Set the default scheduling priority in the template
Attr_getprio	Read the default scheduling priority from the template
Attr_setstacksize	Set the default stack size in the template
Attr_getstacksize	Read the default stack size from the template
Attr_mutexattr_create	Create template for mutex parameters
Attr_mutexattr_delete	Delete template for mutexes
Attr_mutexattr_setkind_np	Set the default mutex type in the template
Attr_mutexattr_getkind_np	Read the default mutex type in the template
Attr_condattr_create	Create template for condition variable parameters
Attr_condattr_delete	Delete template for condition variables

Table 2-4. *Selected template calls.*

Call	Description
Mutex_init	Create a mutex
Mutex_destroy	Delete a mutex
Mutex_lock	Try to lock a mutex; if it is already locked
Mutex_trylock	Try to lock a mutex; fail if it is already locked
Mutex_unlock	Unlock a mutex

Table 2-5. *Selected mutex calls.*

An obvious question to ask about mutexes is: “What happens if you unlock a mutex that is not currently locked?” Does the unlock get saved, or is it lost? Unfortunately, the answer here is not defined, meaning that the programmer cannot count on any particular behavior. This choice is especially unfortunate because it was precisely the problem of lost wakeups that led Dijkstra to invent semaphores. With semaphores, if one process (or thread) is trying to increase a semaphore and another is trying to decrease it, the precise order in which the two actions is carried out does not matter. As a result, race conditions do not occur and programming is relatively straightforward. With DCE mutexes, the order may or may not matter, depending on the implementation, which certainly does not make it any easier to write correct and portable programs.

Two kinds of mutexes are available: **fast mutexes** and **friendly mutexes**. They differ in the way they deal with nested locks. A fast mutex is analogous to a lock in a data base system. If a process tries to lock an unlocked record, it will succeed. However, if it tries to acquire the same lock a second time, it will block, waiting for the lock to be released, something that will never happen.

A friendly mutex allows a thread to lock a mutex that it has already locked. The idea is this. Suppose the main program of a thread locks a mutex, then calls a procedure that also locks the mutex. To avoid deadlock, the second lock is accepted. As long as the mutex is ultimately unlocked as many times as it is locked, the nesting can be arbitrarily deep. The package designers probably could not agree on whether a second lock by the same thread should block or not, so they threw in both versions.

Call	Description
Cond_init	Create a condition variable
Cond_destroy	Delete a condition variable
Cond_wait	Wait on a condition variable until a signal or broadcast arrives
Cond_signal	Wake up at most one thread waiting on the condition variable
Cond_broadcast	Wake up all the threads waiting on the condition variable

Table 2-6. *Selected condition variable calls.*

Next come the calls relating to condition variables, listed in Table 2-6. These, too, can be created and destroyed dynamically. Threads can sleep on condition variables pending the availability of some needed resource. Two wakeup operations are provided: signaling, which wakes up exactly one waiting thread, and broadcasting, which wakes them all up.

Table 2-7 lists the three calls for manipulating the per-thread global variables. These are variables that may be used by any procedure in the thread, but may not be used by procedures outside the thread. The concept of a per-thread global variable is not supported by any of the popular programming languages, so they have to be managed at run time. The first call allocates storage, the second one assigns a pointer to a global variable, and the third one allows the thread to read back a global variable value. Many computer scientists consider global variables to be in the same league as that all-time great pariah, the GOTO statement, so they would no doubt rejoice at the idea of making them cumbersome to use. (The author once tried to design a programming language with a

IKNOWTHISISASTUPIDTHINGTODOBUTNEVERTHELESSGOTO LABEL;

statement, but was forceably restrained from doing so by his colleagues.) On the other hand, it can be argued that having per-thread global variables use procedure calls instead of language scoping rules, like locals and globals, is an emergency measure introduced simply because most programming languages do not allow the concept to be expressed syntactically.

Call	Description
Keycreate	Create a global variable for this thread

Setspecific	Assign a pointer value to a per-thread global variable
Getspecific	Read a pointer value from a per-thread global variable

Table 2-7. *Selected per-thread global variable calls.*

The next group of calls (see Table 2-8) The cancel call tries to kill a thread, but sometimes killing a thread can have devastating effects, for example, if the thread has a mutex locked at the time. For this reason, threads can arrange for attempts to kill them to be enabled or disabled in various ways, very roughly analogous for the ability of UNIX processes to catch signals instead of being terminated by them.

Call	Description
Cancel	Try to kill another thread
Setcancel	Enable or disable ability of other threads to kill this thread

Table 2-8. *Selected calls relating to killing threads.*

Finally, our last group (see Table 2-9) The package allows the threads in a process to be scheduled according to FIFO, round robin, preemptive, nonpreemptive and other algorithms. By using these calls, the algorithm and priorities can be set. The system works best if threads do not elect to be scheduled with conflicting algorithms.

Call	Set the scheduling algorithm
Getscheduler	Read the current scheduling algorithm
Setprio	Set the scheduling priority
Getprio	Read the current scheduling priority

Table 2-9. *Selected scheduling calls.*

2.2 SYSTEM MODELS

Processes run on processors. In a traditional system, there is only one processor, so the question of how the processor should be used does not come up. In a distributed system, with multiple processors, it is a major design issue. The processors in a distributed system can be organized in several ways. There are two underlying models: the workstation model and the processor pool model.

2.2.1 The Workstation Model

The workstation model is straightforward: the system consists of workstations (high-end personal computers) scattered throughout a building or campus and connected by a high-speed LAN, as shown in Figure 2-5. Some of the workstations may be in offices, and thus implicitly dedicated to a single user, whereas others may be in public areas and have

several different users during the course of a day. In both cases, at any instant of time, a workstation either has a single user logged into it, and thus has an "owner", or it is idle.

Figure 2-5. *A network of personal workstations, each with a local file system.*

In some systems the workstations have local disks and in others they do not. The latter are universally called **diskless workstations**, but the former are variously known as **diskful workstations** or **disky workstations**. If the workstations are diskless, then the file system must be implemented by one or more file servers somewhere in the network. Requests to read and write files are sent to a file server, which performs the work and sends back the replies.

Diskless workstations are popular at universities and companies for several reasons, not the least of which is price. Having a large number of workstations equipped with small, slow disks is typically much more expensive than having one or two file servers equipped with huge, fast disks and accessed over the LAN.

A second reason why diskless workstations are popular is their ease of maintenance. When a new release of some program, say a compiler, comes out, the system administrators can easily install it on a small number of file servers in the machine room. Installing it on dozens or hundreds of machines all over a building or campus is another matter entirely. Backup and hardware maintenance is also simpler with one centrally located 5-gigabyte disk than with fifty 100-megabyte disks scattered over the building.

Another point against disks is that they have fans and make noise. Many people find this noise objectionable and do not want it in their office.

Finally, diskless workstations provide symmetry and flexibility. A user can walk up to any workstation in the system and log in. Since all his files are on the file server, one diskless workstation is as good as another. In contrast, when all the files are stored on local disks, using someone else's workstation means that you have easy access to his files, but getting to your own requires extra effort, and is certainly different from using your own workstation.

When the workstations have private disks, these disks can be used in one of at least four ways:

1. Paging and temporary files.
2. Paging, temporary files and system binaries.
3. Paging, temporary files and system binaries and file caching.
4. Complete local file system.

The first design is based on the observation that while it may be convenient to keep all user files on the central file servers (to simplify backup and maintenance, etc.), disks are also needed for paging and temporary files. In this model, the local disks are used only for paging anf files that are temporary, unshared, and can be discarded at the end of the login session. For example, most compilers consist of multiple passes, each of which creates a temporary file read by the next pass. When the file has been read onece, it is discarded. Local disks are ideal for storing such files.

The second model is a variant of the first one in which the local disks also hold the binary (executable) programs, such as the compilers, text editors and electronic mail handlers. When one of these programs is invoked, it is fetched from the local disk instead of from a file server, further reducing the network load. Since these programs rarely change, they can be installed on all the local disks and kept there for long periods of time. When a new release of some system program is available, it is essentially broadcast to all machines.

However, if that machine happens to be down when the program is sent, it will miss the program and continue to run the old version. Thus some administration is needed to keep track of who has which version of which program.

A third approach to using local disks is to use them as explicit caches (in addition to using them for paging, temporaries and binaries). In this mode of operation, users can download files from the file servers to their own disks, read and write them locally, and then upload the modified ones at the end of the login session. The goal of this architecture is to keep long-term storage centralized, but reduce network load by keeping files local while they are being used. A disadvantage is keeping the caches consistent. What happens if two users download the same file and then each modifies it in different ways? This problem is not easy to solve, and will be discussed later.

Fourth, each machine can have its own self-contained file system, with the possibility of mounting or otherwise accessing other machines' file systems. The idea here is that each machine is basically self-contained and that contact with the outside world is limited. This organization provides a uniform and guaranteed response time for the user and puts little load on the network. The disadvantage is that sharing is more difficult, and the resulting system is much closer to a network operating system than to a true transparent distributed operating system.

The one diskless and four diskful models we have discussed are summarized in Table 2-10. The progression from top to bottom in the table is the progression from complete dependence on the file servers to complete independence from them.

The advantages of the workstation model are manifold and clear. The model is certainly easy to understand. Users have a fixed amount of dedicated computing power, and thus guaranteed response time. Sophisticated graphics programs can be very fast, since they can have direct access to the screen. Each user has a large degree of autonomy, and can allocate his workstation's resources as he sees fit. Local disks add to this independence, and make it possible to continue working to a lesser or greater degree even in the face of file server crashes.

However, the model also has two problems. First, as processor chips continue to get cheaper, it will soon become economically feasable to give each user first 10 and later 100 CPUs. Having 100 workstations in your office makes it hard to see out the window. Second, much of the time users are not using their workstations, which are idle, while other users may need extra computing capacity and cannot get it.

The first problem can be addressed by making each workstation a personal multiprocessor. For example, each window on the screen can have a dedicated CPU to run its programs. Preliminary evidence from some early personal multiprocessors such as the DEC Firefly, suggest, however, that the mean number of CPUs utilized is rarely more than one, since users rarely have more than one active process at once.

Disk usage	Advantages	Disadvantages
(Diskless)	Low cost, aesy hardware and software maintenance, symmetry and flexibility	Heavy network usage; file servers may become bottlenecks
Paging, scratch files	Reduces network load over diskless case	Higher cost due to large number of disks needed

Paging, scratch files, binaries	Reduces network load even more	Higher cost; additional complexity of updating the binaries
Paging, scratch files, binaries, file caching	Still lower network load; reduces load on file servers as well	Higher cost; cache consistency problems
Full local file system	Hardly any network load; eliminates need for file servers	Loss of transparency

Table 2-10. *Disk usage on workstations.*

2.2.2 Using idle workstations

The second problem, idle workstations, has been the subject of considerable research, primiraly because many universities have a substantial number of personal workstations, some of which are idle (an idle workstation is the devil's playground?). Measurements show that even at peek periods in the middle of the day, often as many as 30 percent of the workstations are idle. A variety of schemes have been proposed for using idle or otherwise underutilized workstations.

The earliest attempt to allow idle workstations to be utilized was the *rsh* program that comes with Berkeley UNIX. This program is called by

```
rsh machine command
```

in which the first argument names a machine and the second names a command to run on it. What *rsh* does is run the specified program on the specified machine. Although widely used, this program has several serious flaws. First, the user must tell which machine to use, putting the full burden of keeping track of idle machines on the user. Second, the program executes in the environment of the remote machine, which is usually different from the local environment. Finally, if someone should log into an idle machine on which a remote process is running, the process continues to run and the newly logged in user has to either accept the lower performance or find another machine.

The research on idle workstations has centered on solving these problems. The key issues are:

1. How is an idle workstation found?
2. How can a remote process be run transparently?
3. What happens if the machine's owner comes back?

Let us consider these three issues, one at a time.

How is an idle workstation found? To start with, what is an idle workstation? At first glance, it might appear that a workstation with no one logged in at the console is an idle workstation, but with modern computer systems things are not always that simple. In many systems, even with no one logged in there may be dozens of processes running, such as clock daemons, mail daemons, nwes daemons, and all manner of other deamons. On the other hand, a user who logs in when arriving at his desk in the morning, but otherwise does not touch the computer for hours hardly puts any additional load on it. Different systems make different decisions as to what "idle" means, but typically if no one has touched the keyboard or mouse for several minutes and no user-initiated processes are running, the workstation can be said to be idle. Consequently, there may be substantial differences in

load between one idle workstation and another, due, for example, to the volume of mail coming into the first one but not the second.

The algorithms used to locate idle workstations can be divided into two categories: server driven and client driven. In the former, when a workstation goes idle, and thus becomes a potential compute server, it announces its availability. It can do this by entering its name, network address, and properties in a registry file (or database), for example. Later, when a user wants to execute a command on an idle workstation, he types something like:

```
remote command
```

and the *remote* program looks in the registry to find a suitable idle workstation. For reliability reasons, it is also possible to have multiple copies of the registry.

An alternative way for the newly idle workstation to announce the fact that it has become unemployed is to put a broadcast message onto the network. All other workstations then record this fact. In effect, each machine maintains its own private copy of the registry. The advantage of doing it this way is less overhead in finding an idle workstation and greater redundancy. The disadvantage is requiring all machines to do the work of maintaining the registry.

Whether there is one registry or many, there is a potencial danger of race conditions occurring. If two users simultaneously invoke the *remote* command, and both of them discover that the same machine is idle, they may both try to start up processes there at the same time. To detect and avoid this situation, the *remote* program can check with the idle workstation, which, if still free, removes itself from the registry and gives the go-ahead sign. At this point, the caller can send over its environment and start the remote process, as shown in Figure 2-6.

The other way to locate idle workstations is to use a client-driven approach. When *remote* is invoked, it broadcasts a request saying what program it wants to run, how much memory is needs, whether or not a floating point coprocessor chip is needed, and so on. These details are not needed if all the workstations are identical, but if the system is heterogeneous and not every program can run on every workstation, they are essential. When the replies come back, *remote* picks one and sets it up. One nice twist is to have "idle" workstations delay their responses slightly, with the delay being proportional to the current load. In this way, the reply from the least heavily loaded machine will come back first and be selected.

Finding a workstation is only the first step. Now the process has to be run there. Moving the code is easy. The trick is to set up the remote process so that it sees the same environment it would have locally, on the **home workstation**, and thus carries out the same computation it would have locally.

Figure 2-6. *A registry-based algorithm for finding and using idle workstations.*

To start with, it needs the same view of the file system, the same working directory, and the same environment variables (shell variables), if any. After these have been set up, the program can begin running. The trouble starts when the first system call, say a READ, is executed. What should the kernel do? The answer depends very much on the system architecture. If the system is diskless, with all the files located on file servers, the kernel can just send the request to the appropriate file server, the same way the home machine would have done had the process been running there. On the other hand, if the system has local disks, each with a complete file system, then the request has to be forwarded back to the home machine for execution.

Some system calls must be forwarded back to the home machine no matter what, even if all the machines are diskless. For example, reads from the keyboard and writes to the screen can never be carried out on the remote machine. However, other system calls must be done remotely under all conditions. For example, the UNIX system calls SBRK (adjust the size of the data segment), NICE (set CPU scheduling priority), and PROFIL (enable profiling of the program counter), cannot be executed on the home machine. In addition, all system calls that query the state of the machine have to be done on the machine that the process is actually running on. These include asking for the machine's name and network address, asking how much free memory it has, and so on.

System calls involving time are a problem because the clocks on different machines may not be synchronized. We saw how hard it is to achieve synchronization in the previous chapter. Using the time on the remote machine may cause programs that depend on time, like *make*, to give incorrect results. Forwarding all time-related calls back to the home machine, however, introduces delay, which also causes problems with time.

To further complicate matters, certain special cases of calls which normally might have to be forwarded back, such as creating and writing to a temporary file, can be done much more efficiently on the remote machine. In addition, mouse tracking and signal propagation have to be thought out carefully as well. Programs that directly write to hardware devices, such as the screen s frame buffer, diskette, or magnetic tape cannot be run remotely at all. All in all, making programs run on remote machines as though they were running on their home machines is possible, but it is a complex and tricky business.

The final question on our original list is what to do if the machine's owner comes back (i.e., somebody logs in or a previously inactive user touches the keyboard or mouse). The easiest thing is to do nothing, but this tends to defeat the idea of "personal" workstations. If other people can run programs on your workstation at the same time you are trying to use it, there goes your guaranteed response.

Another possibility is to kill off the intruding process. The simplest way is to do this abruptly and without warning. The disadvantage of this strategy is that all work will be lost and the file system may be left in a chaotic state. A better way is to give the process fair warning, by sending it a signal to allow it to detect impending doom, and shut down gracefully (write edit buffers to the disk, close files, and so on). If it has not exited within a few seconds, it is then terminated. Of course, the program must be written to expect and handle this signal, something most existing programs definitely are not.

In both cases, when the process is gone, it should leave the machine in the same state as it found it, to avoid disturbing the owner. Among other items, this requirement means that not only must the process go, but also all its children and their children. In addition, mailboxes, network connections, and other system-wide data structures must be deleted, and some provision must be made to ignore RPC replies and other messages that arrive for the process after it is gone. If there is a local disk, temporary files must be deleted, and if possible, any files that had to be removed from its cache restored.

2.2.3 The Processor Pool Model

Although using idle workstations adds a little computing power to the system, it does not address a more fundamental issue: "What happens when it is feasible to provide 10 or 100 times as many CPUs as there are active users?" One solution, as we saw, is to give everyone a personal multiprocessor. However this is a somewhat inefficient design.

An alternative approach is to construct a **processor pool**, a rack full of CPUs in the machine room, which can be dynamically allocated to users on demand. The processor pool approach is illustrated in Figure 2-7. Instead of giving users personal workstations, in this

model they are given high-performance graphics terminals, such as X terminals (although small workstations can also be used as terminals). This idea is based on the observation that what many users really want is a high-quality graphical interface and good performance. Conceptually, it is much closer to traditional timesharing than to the personal computer model, although it is built with modern technology (low-cost microprocessors).

Figure 2-7. *A system based on the processor pool model.*

The motivation for the processor pool idea comes from taking the diskless workstation idea a step further. If the file system can be centralized in a small number of file servers to gain economies of scale, it should be possible to do the same thing for compute servers. By putting all the CPUs in a big rack in the machine room, power supply and other packaging costs can be reduced, giving more computing power for a given amount of money. Furthermore, it permits the use of cheaper X terminals (or even ordinary ASCII terminals), and decouples the number of users from the number of workstations. The model also allows for easy incremental growth. If the computing load increases by 10 percent, just buy 10 percent more processors and put them in the pool.

In effect, we are converting all the computing power into "idle workstations" that can be accessed dynamically. Users can be assigned as many CPUs as they need for short periods, after which they are returned to the pool so other users can have them. There is no concept of ownership here: all the processors belong equally to everyone.

The biggest argument for centralizing the computing power in a processor pool comes from queueing theory. A queueing system is a situation in which users generate random requests for work from a server. When the server is busy, the users queue for service and are processed in turn. Common examples of queueing systems are bakeries, airport check-in counters, supermarket check-out counters, and numerous others. The bare basics are depicted in Figure 2-8.

Queueing systems are useful because it is possible to model them analytically. Let us call the total input rate λ requests per second, from all the users combined. Let us call the rate at which the server can process requests μ. For stable operation, we must have $\mu > \lambda$. If the server can handle 100 requests/sec, but the users continuously generate 110 requests/sec, the queue will grow without bound. (Small intervals in which the input rate exceeds the service rate are acceptable, provided that the mean input rate is lower than the service rate and there is enough buffer space.)

Figure 2-8. *A basic queueing system.*

It can be proven (Kleinrock, 1974) that the mean time between issuing a request and getting a complete response, T is related to λ and μ by the formula:

$$T = \frac{1}{\mu - \lambda}$$

As an example, consider a file server that can handle 50 requests/sec and which gets 40 requests per second. The mean response time will be 1/10 sec or 100 msec. Note that when λ goes to 0 (no load), the response time of the file server does not go to 0, but to 1/50 sec or 20 msec. The reason is obvious once it is pointed out. If the file server can process only 50 requests/sec, it must take 20 msec to process a single request, even in the absence of any competition, so the response time, which includes the processing time, can never go below 20 msec.

Suppose we have n personal multiprocessors, each with some number of CPUs, and each one forms a separate queueing system with request arrival rate λ and CPU processing rate μ. The mean response time, *T*, will be as given above. Now consider what happens if we scoop up all the CPUs and place them in a single processor pool. Instead of having n small queueing systems running in parallel, we now have one large one, with an input rate $n\lambda$ and a service rate $n\mu$. Let us call the mean response time of this combined system T_1. From the above formula we find

$$T_1 = \frac{1}{n\mu - n\lambda} = T / n$$

This surprising result says that by replacing *n* small resources by one big one that is *n* times more powerful, we can reduce the average response time *n*-fold.

This result is extremely general and applies to a large variety of systems. It is one of the main reasons that airlines prefer to fly a 300-seat 747 once every 5 hours to a 10-seat business jet every 10 minutes. The effect arises because dividing the processing power into small servers (e.g., personal workstations) each with one user, is a poor match to a workload of randomly arriving requests. Much of the time, a few servers are busy, even overloaded, but most are idle. It is this wasted time that is eliminated in the processor pool model, and the reason why it gives better overall performance. The concept of using idle workstations is a weak attempt at recapturing the wasted cycles, but it is complicated and has many problems, as we have seen.

In fact, this queueing theory result is one of the main arguments against having distributed systems at all. Given a choice between one centralized 1000-MIPS CPU and 100 private, dedicated, l0-MIPS CPUs, the mean response time of the former will be 100 times better, because no cycles are ever wasted. The machine only goes idle when no user has any work to do. This fact argues in favor of concentrating the computing power as much as possible.

However, mean response time is not everything. There are also arguments in favor of distributed computing. Cost is one of them. If a single 1000-MIPS CPU is vastly more expensive than 100 10-MIPS workstations, the price/performance ratio of the latter may be much better. It may not even be possible to build such a large machine at any price. Reliability and fault tolerance are factors as well.

Also, personal workstations have a uniform response, independent of what other people are doing (except when the network or file servers are jammed). For some users, a low variance in response time may be perceived as more important than the mean response time itself. Consider, for example, editing on a private workstation on which asking for the next page to be displayed always takes 500 msec. Now consider editing on a large, centralized, shared computer on which asking for the next page takes 5 msec 95 percent of the time and 5 sec one time in 20. Even though the mean here is twice as good as on the workstation, the users may consider the performance intolerable. On the other hand, to the user with a large *make* or a huge simulation to run, the big computer may win hands down.

So far we have tacitly assumed that a pool of *n* processors is effectively the same thing as a single processor that is n times as fast as a single processor. In reality, this assumption is only justified if all requests can be split up in such a way as to allow them to run on all the processors in parallel. If a job can be split into, say, only 5 parts, then the processor pool model only has an effective service time 5 times better than a single processor, not *n* times better.

Still, the processor pool model is a much cleaner way of getting extra computing power than looking around for idle workstations and sneaking over there while nobody is looking. By starting out with the assumption that no processor belongs to anyone, we get a design based on the concept of requesting machines from the pool, using them, and putting them back when done. There is also no need to forward anything back to a "home" machine because there are none. There is also no danger of the owner coming back, because there are no owners.

In the end, it all comes down to the nature of the workload. If all people are doing is simple editing and occasionally sending an electronic mail message or two, having a personal workstation is probably enough. If, on the other hand, the users are engaged in a large software development project, frequently running *make* on large directories, or are trying to invert massive sparse matrices, or do major simulations or run big artificial intelligence or VLSI routing programs, constantly hunting for substantial numbers of idle workstations will be no fun at all. In all these situations, the processor pool idea is fundamentally much simpler and more attractive.

2.2.4 A Hybrid Model

A possible compromise is to provide each user with a personal workstation, and to have a processor pool in addition. Although this solution is more expensive than either a pure workstation model or a pure processor pool model, it combines the advantages of both of the others.

Interactive work can be done on workstations, giving guaranteed response. Idle workstations, however, are not utilized, making for a simpler system design. They are just left unused. Instead, all noninteractive processes run on the processor pool, as does all heavy computing in general. This model provides fast interactive response, an efficient use of resources, and simple design.

2.3 PROCESSOR ALLOCATION

By definition, a distributed system consists of multiple processors. These may be organized as a collection of personal workstations, a public processor pool, or some hybrid form. In all cases, some algorithm is needed for deciding which process should be run on which machine. For the workstation model, the question is when to run a process locally and when to look for an idle workstation. For the processor pool model, a decision must be made for every new process. In this section we will study the algorithms used to determine which process is assigned to which processor. We will follow tradition and refer to this subject as "processor allocation" rather than "process allocation", although a good case can be made for the latter.

2.3.1 Allocation Models

Before looking at specific algorithms, or even at design principles, it is worthwhile saying something about the underlying model, assumptions, and goals of the work on processor allocation. Nearly all work in this area assumes that all the machines are identical, or at least code-compatible, differing at most by speed. An occasional paper assumes that the system consists of several disjoint processor pools, each of which is homogeneous. These assumptions are usually valid, and make the problem much simpler, but leave unanswered for the time being questions like whether a command to start up the text

formatter should be started up on a 486, SPARC, or MIPS CPU, assuming that binaries for all of them are available.

Almost all published models assume that the system is fully interconnected, that is, every processor can communicate with every other processor. We will assume this as well. This assumption does not mean that every machine has a wire to every other machine; just that transport connections can be established between every pair of machines. That messages may have to be routed hop by hop over a sequence of machines is only of interest to the lower layers. Some networks support broadcasting or multicasting, and some algorithms use these facilities.

New work is generated when a running process decides to fork or otherwise create a subprocess. In some cases the forking process is the command interpreter (shell) that is starting up a new job in response to a command from the user. In others, a user process itself creates one or more children, for example, in order to gain performance by having all the children run in parallel.

Processor allocation strategies can be divided into two broad classes. In the first one, which we shall call **nonmigratory**, when a process is created, a decision is made about where to put it. Once placed on a machine, the process stays there until it terminates. It may not move, no matter how badly overloaded its machine becomes and no matter how many other machines are idle. In contrast, with **migratory** allocation algorithms, a process can be moved even if it has already started execution. While migratory strategies allow better load balancing, they are substantially more complex, and have a major impact on system design.

Implicit in an algorithm that assigns processes to processors is that we are trying to optimize something. If this were not the case, we could just make the assignments at random or in numerical order. Precisely what it is that is being optimized, however, varies from one system to another. One possible goal is to maximize **CPU utilization**, that is, maximize the number of CPU cycles actually executed on behalf of user jobs per hour of real time. Maximizing CPU utilization is another way of saying that CPU idle time is to be avoided at all costs. When in doubt, make sure every CPU has something to do.

Another worthy objective is minimizing mean **response time**. Consider, for example, the two processors and two processes of Figure 2-9. Processor 1 runs at 10 MIPS; processor 2 runs at 100 MIPS, but has a waiting list of backlogged processes that will take 5 sec to finish off. Process *A* has 100 million instructions and process *B* has 300 million. The response times for each process on each processor (including the wait time) are shown in the figure. If we assign *A* to processor 1 and *B* to processor 2, the mean response time will be (10 + 8)/2 or 9 sec. If we assign them the other way around, the mean response time will be (30 + 6)/2 or 18 sec. Clearly the former is a better assignment in terms of minimizing mean response time.

Figure 2-9. *Response times of two processes on two processors.*

A variation of minimizing the response time is minimizing the **response ratio**. The response ratio is defined as the amount of time it takes to run a process on some machine, divided by how long it would take on some unloaded benchmark processor. For many users, response ratio is a more useful metric than response time since it takes into account the fact that big jobs are supposed to take longer than small ones. To see this point, which is better, a 1 sec job that takes 5 sec or a 1 min job that takes 70 sec? Using response time, the former is better, but using response ratio, the latter is much better because 5/1 >> 70/60.

2.3.2 Design Issues for Processor Allocation Algorithms

A large number of processor allocation algorithms have been proposed over the years. In this section we will look at some of the key choices involved in these algorithms and point out the various tradeoffs. The major decisions the designers must make can be summed up in five issues:

1. Deterministic versus heuristic algorithms.
2. Centralized versus distributed algorithms.
3. Optimal versus suboptimal algorithms.
4. Local versus global algorithms.
5. Sender-initiated versus receiver-initiated algorithms.

Other decisions also come up, but these are the main ones. Let us look at each of these in turn.

Deterministic algorithms are appropriate when everything about process behavior is known in advance. Imagine that you have a complete list of all processes, their computing requirements, their file requirements, their communication requirements, and so on. Armed with this information, it is possible to make a perfect assignment. In theory, one could try all possible assignments and take the best one.

In few, if any, systems, is total knowledge available in advance, but sometimes a reasonable approximation is obtainable. For example, in banking, insurance, or airline reservations, today's work is just like yesterday's. The airlines have a pretty good idea of how many people want to fly from New York to Chicago on a Monday morning in early Spring, so the nature of the workload can be accurately characterized, at least statistically, making it possible to consider deterministic allocation algorithms.

At the other extreme are systems where the load is completely unpredictable. Requests for work depend on who's doing what, and can change dramatically from hour to hour, or even from minute to minute. Processor allocation in such systems cannot be done in a deterministic, mathematical way, but of necessity uses ad hoc techniques called **heuristics**.

The second design issue is centralized versus distributed. This theme has occurred repeatedly throughout the book. Collecting all the information in one place allows a better decision to be made, but is less robust and can put a heavy load on the central machine. Decentralized algorithms are usually preferable, but some centralized algorithms have been proposed for lack of suitable decentralized alternatives.

The third issue is related to the first two: "Are we trying to find the best allocation, or merely an acceptable one?" Optimal solutions can be obtained in both centralized and decentralized systems, but are invariably more expensive than suboptimal ones. They involve collecting more information and processing it more thoroughly. In practice, most actual distributed systems settle for heuristic, distributed, suboptimal solutions due to the difficulty of obtaining optimal ones.

The fourth issue relates to what is often called **transfer policy**. When a process is about to be created, a decision has to be made whether or not it can be run on the machine where it is being generated. If that machine is too busy, the new process must be transferred somewhere else. The choice here is whether to base the transfer decision entirely on local information or not. One school of thought advocates a simple (local) algorithm: if the machine's load is below some threshold, keep the new process; otherwise try to get rid of it. Another school says that this heuristic is too crude. Better to collect (global) information

about the load elsewhere before deciding whether the local machine is too busy for another process or not. Each one has its points. Local algorithms are simple, but may be far from optimal, whereas global ones may only give a slightly better result at much higher cost.

The last issue in our list deals with **location policy**. Once the transfer policy has decided to get rid of a process, the location policy has to figure out where to send it. Clearly the location policy cannot be local. It needs information about the load elsewhere to make an intelligent decision. This information can be disseminated in two ways, however. In one method, the senders start the information exchange. In another, it is the receivers that take the initiative.

Figure 2-10. *(a) A sender looking for an idle machine. (b) A receiver looking for work to do.*

As a simple example, look at Figure 2-10(a). Here an overloaded machine sends out requests for help to other machines, in hopes of off-loading its new process on some other machine. The sender takes the initiative in locating more CPU cycles in this example. In contrast, in Figure 2-10(b), a machine that is idle or underloaded announces to other machines that it has little to do and is prepared to take on extra work. Its goal is to locate a machine that is willing to give it some work to do.

For both the sender-initiated and receiver-initiated cases, various algorithms have different strategies for whom to probe, how long to continue probing, and what to do with the results. Nevertheless, the difference between the two approaches should be clear by now.

2.4 SCHEDULING IN DISTRIBUTED SYSTEMS

There is not really a lot to say about scheduling in a distributed system. Normally, each processor does its own local scheduling (assuming it has multiple processes running on it), without regard to what the other processors are doing. Usually this approach works fine. However, when a group of related, heavily interacting processes are all running on different processors, independent scheduling is not always the most efficient way.

The basic difficulty can be illustrated by an example in which processes A and B run on one processor and processes *C* and *D* run on another. Each processor is timeshared in, say, 100 msec time slices, with *A* and *C* running in the even slices, and *B* and *D* running in the odd ones, as shown in Figure 2-11(a). Suppose that *A* sends many messages or makes many remote procedure calls to *D*. During time slice 0, A starts up and immediately calls *D*, which unfortunately is not running because it is now *C*'s turn. After 100 msec, process switching takes place, and *D* gets *A*'s message, carries out the work, and quickly replies. Because *B* is now running, it will be another 100 msec before *A* gets the reply and can proceed. The net result is one message exchange every 200 msec. What is needed is a way to ensure that processes that communicate frequently run simultaneously.

Figure 2-11. *(a) Two jobs running out of phase with each other. (b) Scheduling matrix for eight processors, each with six time slots. The Xs indicate allocated slots.*

Although it is difficult to dynamically determine the interprocess communication patterns, in many cases, a group of related processes will be started off together. For example, it is usually a good bet that the filters in a UNIX pipeline will communicate with each other more than they will with other, previously started processes. Let us assume that processes are created in groups, and that intragroup communication is much more prevalent than intergroup communication. Let us further assume that a sufficiently large number of

processors is available to handle the largest group, and that each processor is multiprogrammed with *N* process slots (*N*-way multiprogramming).

Ousterhout (1982) proposed several algorithms based on a concept he calls **co-scheduling**, which takes interprocess communication patterns into account while scheduling to ensure that all members of a group run at the same time. The first algorithm uses a conceptual matrix in which each column is the process table for one processor, as shown in Figure 2-11(b). Thus, column 4 consists of all the processes that run on processor 4. Row 3 is the collection of all processes that are in slot 3 of some processor, starting with the process in slot 3 of processor 0, then the process in slot 3 of processor 1, and so on. The gist of his idea is to have each processor use a round robin scheduling algorithm with all processors first running the process in slot 0 for a fixed period, then all processors running the process in slot 1 for a fixed period, etc. A broadcast message could be used to tell each processor when to do process switching, to keep the time slices synchronized.

By putting all the members of a process group in the same slot number, but on different processors, one has the advantage of *N*-fold parallelism, with a guarantee that all the processes will be run at the same time, to maximize communication throughput. Thus in Figure 2-11(b), four processes that must communicate should be put into slot 3, on processors l, 2, 3, and 4 for optimum performance. This scheduling technique can be combined with the hierarchical model of process management used in MICROS by having each department head maintain the matrix for its workers, assigning processes to slots in the matrix and broadcasting time signals.

Ousterhout also described several variations to this basic method to improve performance. One of these breaks the matrix into rows, and concatenates the rows to form one long row. With *k* processors, any *k* consecutive slots belong to different processors. To allocate a new process group to slots, one lays a window *k* slots wide over the long row such that the leftmost slot is empty but the slot just outside the left edge of the window is full. If sufficient empty slots are present in the window, the processes are assigned to the empty slots, otherwise the window is slid to the right and the algorithm repeated. Scheduling is done by starting the window at the left edge and moving rightward by about one window's worth per time slice, taking care not to split groups over windows. Ousterhout's paper discusses these and other methods in more detail and gives some performance results.

2.5 SUMMARY

Although threads are not an inherent feature of distributed operating systems, most of them have a threads package, so we have studied them in this chapter. A thread is a kind of lightweight process that shares an address space with one or more other threads. Each thread has its own program counter and stack, and is scheduled independently of the other threads. When a thread makes a blocking system call, the other threads in the same address space are not affected. Threads packages can be implemented in either user space or by the kernel, but there are problems to be solved either way. The use of lightweight threads has led to some interesting results in lightweight RPC as well.

Two models of organizing the processors are commonly used-the workstation model and the processor pool model. In the former, each user has his own workstation, sometimes with the ability to run processes on idle workstations. In the latter, the entire computing facility is a shared resource. Processors are then dynamically allocated to users as needed, and returned to the pool when the work is done. Hybrid models are also possible.

Given a collection of processors, some algorithm is needed for assigning processes to processors. Such algorithms can be deterministic or heuristic, centralized or distributed, optimal or suboptimal, local or global, and sender-initiated or receiver-initiated. Various examples were presented.

Finally, the chapter concluded with a discussion of co-scheduling.

3.

DISTRIBUTED FILE SYSTEMS

A key component of any distributed system is the file system. As in single processor systems, in distributed systems the job of the file system is to store programs and data, and make them available as needed. Many aspects of distributed file systems are similar to the conventional file systems we studied in Chap. 4, so we will not repeat that material here. Instead we will concentrate on those aspects of distributed file systems that are different from centralized ones.

To start with, in a distributed system, it is important to distinguish between the concepts of the file service and the file server. The **file service** is the specification of what the file system offers to its clients. It describes the primitives available, what parameters they take, and what actions they perform. To the clients, the file service defines precisely what service they can count on, but says nothing about how it is implemented. In effect, the file service specifies the file system's interface to the clients.

A **file server**, in contrast, is a process that runs on some machine and helps implement the file service. A system may have one file server or several, but in a properly designed distributed system, the clients should not be aware of how the file system is implemented. In particular, they should not know how many file servers there are and what the location or function of each one is. All they know is that when they call the procedures specified in the file service, the required work is performed somehow, and the required results are returned. In fact, the clients should not even know that the file service is distributed. Ideally, it should look the same as a normal single processor file system.

Since a file server is normally just a user process (or sometimes a kernel process) running on some machine, a system may contain multiple file servers, each offering a different file service. For example, a distributed system may have two servers that offer UNIX file service and MS-DOS file service, respectively, with each user process using the one appropriate for it. In that way, it is possible to have a terminal with multiple windows, with UNIX programs running in some windows and MS-DOS programs running in other windows, with no conflicts. Whether the servers offer specific file services, such as UNIX or MS-DOS, or more general file services is up to the system designers. The type and number of file services available may even change as the system evolves.

3.1 DISTRIBUTED FILE SYSTEM DESIGN

A distributed file system typically has two reasonably distinct components: the true file service and the directory service. The former is concerned with the operations on individual files, such as reading, writing, and appending, whereas the latter is concerned with creating and managing directories, adding and deleting files from directories, and so on. In this section we will discuss the true file service interface; in the next one we will discuss the directory service interface.

3.1.1 The File Service Interface

For any file service, whether for a single processor or for a distributed system, the most fundamental issue is: "What is a file?" In many systems, such as UNIX and MS-DOS, a file is an uninterpreted sequence of bytes. The meaning and structure of the information in the files is entirely up to the application programs; the operating system is not interested.

On mainframes however, many types of files exist, each with different properties. A file can be structured as a sequence of records, for example, with operating system calls to read or write a particular record. The record can usually be specified by either giving its

record number (i.e., position within the file), or the value of some field. In the latter case, the operating system either maintains the file as a B-tree or other suitable data structure, or uses hash tables to locate records quickly. Since most distributed systems are intended for UNIX or MS-DOS environments, most file servers support the notion of a file as a sequence of bytes, rather than as a sequence of keyed records.

A file can have **attributes**, which are pieces of information about the file but which are not part of the file itself. Typical attributes are the owner, size, creation date, and access permissions. The file service usually provides primitives to read and write some of the attributes. For example, it may be possible to change the access permissions but not the size (other than by appending data to the file). In a few advanced systems, it may be possible to create and manipulate user-defined attributes in addition to the standard ones.

Another important aspect of the file model is whether files can be modified after they have been created. Normally they can be, but in some distributed systems, the only file operations are CREATE and READ. Once a file has been created, it cannot be changed. Such a file is said to be **immutable**. Having files be immutable makes it much easier to support file caching and replication because it eliminates all the problems associated with having to update all copies of a file whenever it changes.

Protection in distributed systems uses essentially the same techniques as in single processor systems: capabilities and access control lists. With capabilities, each user has a kind of ticket, called a **capability**, for each object to which it has access. The capability specifies which kinds of accesses are permitted (e.g., reading is allowed but writing is not).

All **access control list** schemes associate with each file an implicit or explicit list of users who may access the file, and what kinds of access each one has. The UNIX scheme, with bits for controlling reading, writing, and executing each file separately for the owner, the owner's group, and everyone else is a simplified access control list.

File services can be split into two types, depending on whether they support an upload/download model or a remote access model. In the **upload/download model**, shown in Figure 3-1(a), the file service provides only two major operations: read file and write file. The former operation transfers an entire file from one of the file servers to the requesting client. The latter operation transfers an entire file the other way, from client to server. Thus the conceptual model is moving whole files in either direction. The files can be stored in memory or on a local disk, as needed.

The advantage of the upload/download model is its conceptual simplicity. Application programs fetch the files they need, then use them locally. Any modified files or newly created files are written back when the program finishes. No complicated file service interface has to be mastered to use this model. Furthermore, whole file transfer is highly efficient. The main disadvantage is that enough storage must be available on the client to store all the files required. Furthermore, if only a small fraction of a file is needed, moving the whole file is wasteful.

The other kind of file service is the **remote access model**, as illustrated in Figure 3-1(b). In this model, the file service provides a large number of operations for opening and closing files, reading and writing parts of files, moving around within files (LSEEK), examining and changing file attributes, and so on. Whereas in the upload/download model, the file service merely provides physical storage and transfer, here the file system runs on the servers, not on the clients. It has the advantage of not requiring much space on the clients, as well as eliminating the need to pull in entire files when only small pieces are needed.

3.1.2

The Directory Server Interface

The other part of the file service is the directory service, which provides operations for creating and deleting directories, naming and renaming files, and moving them from one directory to another. The nature of the directory service does not depend on whether individual files are transferred in their entirety or accessed remotely.

The directory service defines some alphabet and syntax for composing file (and directory) names. File names can typically be from 1 to some maximum number of letters, numbers, and certain special characters. Some systems divide file names into two parts, usually separated by a period, such as *prog.c* for a C program or *man.txt* for a text file. The second part of the name, called the **file extension**, identifies the file type. Other systems use an explicit attribute for this purpose, instead of tacking an extension onto the name.

All distributed systems allow directories to contain subdirectories, to make it possible for users to group related files together. Accordingly, operations are provided for creating and deleting directories as well as entering, removing, and looking up files in them. Normally, each subdirectory contains all the files for one project, such as a large program or document (e.g., a book). When the (sub)directory is listed, only the relevant files are shown; unrelated files are in other (sub)directories and do not clutter the listing. Subdirectories can contain their own subdirectories, and so on, leading to a tree of directories, often called a **hierarchical file system**. Figure 3-2(a) illustrates a tree with five directories.

In some systems, it is possible to create links or pointers to an arbitrary directory. These can be put in any directory, making it possible to build not only trees, but arbitrary directory graphs, which are more powerful. The distinction between trees and graphs is especially important in a distributed system.

The nature of the difficulty can be seen by looking at the directory graph of Figure 3-2(b). In this figure, directory D has a link to directory B. The problem occurs when the link from A to B is removed. In a tree-structured hierarchy, a link to a directory can only be removed when the directory pointed to is empty. In a graph, it is allowed to remove a link to a directory as long as at least one other link exists. By keeping a reference count, shown in the upper right-hand corner of each directory in Figure 3-2(b), it can be determined when the link being removed is the last one.

After the link from A to B is removed, B's reference count is reduced from 2 to 1, which on paper is fine. However, B is now unreachable from the root of the file system (A). The three directories, B, D, and E, and all their files are effectively orphans. This problem exists in centralized systems as well, but it is more serious in distributed ones. If everything is on one machine, it is possible, albeit somewhat expensive, to discover orphaned directories, because all the information is in one place. All file activity can be stopped and the graph traversed starting at the root, marking all reachable directories. At the end of this process, all unmarked directories are known to be unreachable. In a distributed system, multiple machines are involved and all activity cannot be stopped so getting a "snapshot" is difficult, if not impossible.

A key issue in the design of any distributed file system is whether or not all machines (and processes) should have exactly the same view of the directory hierarchy. As an example of what we mean by this remark, consider Figure 3-3. In Figure 3-3(a) we show two file servers, each holding three directories and some files. In Figure 3-3(b) we have a system in which all clients (and other machines) have the same view of the distributed file system. If the path */D/E/x* is valid on one machine, then it is valid on all of them.

In contrast, in Figure 3-3(c), different machines can have different views of the file system. To repeat the previous example, the path */D/E/x* might well be valid on client 1 but

not on client 2. In systems that manage multiple file servers by remote mounting, Figure 3-3(c) is the norm. It is flexible and straightforward to implement, but it has the disadvantage of not making the entire system behave like a single old-fashioned timesharing system. In a timesharing system, the file system looks the same to any process [i.e., the model of Figure 3-3(b)]. This property makes a system easier to program and understand.

A closely related question is whether or not there is a global root directory, which all machines recognize as the root. One way to have a global root directory is to have this root contain one entry for each server and nothing else. Under these circumstances, paths take the form */server/path*, which has its own disadvantages, but at least is the same everywhere in the system.

Naming Transparency

The principal problem with this form of naming is that it is not fully transparent. Two forms of transparency are relevant in this context, and are worth distinguishing. The first one, location transparency, means that the path name gives no hint as to where the file (or other object) is located. A path like */server1/dir1/dir2/x* tells everyone that *x* is located on server 1, but it does not tell where that server is located. The server is free to move anywhere it wants to in the network without the path name having to be changed. Thus this system has location transparency.

However, suppose file *x* is extremely large and space is tight on server 1. Furthermore, suppose that there is plenty of room on server 2. The system might well like to move *x* to server 2 automatically. Unfortunately, when the first component of all path names is the server, the system cannot move the file to the other server automatically, even if *dir1* and *dir2* exist on both servers. The problem is that moving the file automatically changes its path name from */server1/dir1/dir2/x* to */server2/dir1/dir2/x*. Programs that have the former string built into them will cease to work if the path changes. A system in which files can be moved without their names changing is said to have location independence. A distributed system that embeds machine or server names in path names clearly is not. location independent. One based on remote mounting is not either, since it is not possible to move a file from one file group (the unit of mounting) to another and still be able to use the old path name. Location independence is not easy to achieve, but it is a desirable property to have in a distributed system.

To summarize what we have said earlier, there are three common approaches to file and directory naming in a distributed system:

1. Machine + path naming, such as */machine/path* or *machine:path*.
2. Mounting remote file systems onto the local file hierarchy.
3. A single name space that looks the same on all machines.

The first two are easy to implement, especially as a way to connect up existing systems that were not designed for distributed use. The latter is difficult, and requires careful design, but it is needed if the goal of making the distributed system act like a single computer is to be achieved.

Two-Level Naming

Most distributed systems use some form of two-level naming. Files (and other objects) have symbolic names such as prog.c, for use by people, but they can also have internal, binary names for use by the system itself. What directories in fact really do, is provide a mapping between these two naming levels. It is convenient for people and programs to use symbolic (ASCII) names, but for use within the system itself, these names are too long and cumbersome. Thus when a user opens a file or otherwise references a symbolic name, the system immediately looks up the symbolic name in the appropriate directory to get the binary name that will be used to actually locate the file. Sometimes the binary names are visible to the users and sometimes they are not.

The nature of the binary name varies considerably from system to system. In a system consisting of multiple file servers each of which is self contained (i.e., does not hold any references to directories or files on other file servers), the binary name can just be a local i-node number, as in UNIX.

A more general naming scheme is to have the binary name indicate both a server and a specific file on that server. This approach allows a directory on one server to hold a file on a different server. An alternative way to do the same thing that is sometimes preferred is to use a symbolic link. A symbolic link is a directory entry that maps onto a (server, file name) string, which can be looked up on the server named to find the binary name. The symbolic link itself is just a path name.

Yet another idea is to use capabilities as the binary names. In this method, looking up an ASCII name yields a capability, which can take one of many forms. For example, it can contain a physical or logical machine number or network address of the appropriate server, as well as a number indicating which specific file is required. A physical address can be used to send a message to the server without further interpretation. A logical address can be located either by broadcasting or looking it up on a name server.

One last twist that is sometimes present in a distributed system but rarely in a centralized one is the possibility of looking up an ASCII name and getting not one, but several binary names (i-nodes, capabilities, or something else). These would typically represent the original file and all its backups. Armed with multiple binary names, it is then possible to try to locate one of the corresponding files, and if that one is unavailable for any reason, to try one of the others. This method provides a degree of fault tolerance through redundancy.

3.1.3 Semantics of File Sharing

When two or more users share the same file, it is necessary to define the semantics of reading and writing precisely to avoid problems. In single-processor systems that permit processes to share files, such as UNIX, the semantics normally state that when a READ operation follows a WRITE operation, the READ returns the value just written, as shown in Figure 3-4(a). Similarly, when two WRITEs happen in quick succession, followed by a READ, the value read is the value stored by the last write. In effect, the system enforces an absolute time ordering on all operations and always returns the most recent value. We will refer to this model as UNIX semantics. On a single-processor system (or even on a shared-memory multiprocessor) they are both easy to understand and straightforward to implement.

In a distributed system, UNIX semantics can be easily achieved as long as there is only one file server and clients do not cache files. All READs and WRITEs go directly to the file server, which processes them strictly sequentially. This approach gives UNIX semantics (except for the minor problem that network delays may cause a READ that

occurred a microsecond after a WRITE to arrive at the server first and thus get the old value).

In practice, however, the performance of a distributed system in which all file requests must go to a single server is frequently poor. This problem is often solved by allowing clients to maintain local copies of heavily used files in their private caches. Although we will discuss the details of file caching below, for the moment it is sufficient to point out that if a client locally modifies a cached file and shortly thereafter another client reads the file from the server, the second client will get an obsolete file, as illustrated in Figure 3-4(b).

One way out of this difficulty is to propagate all changes to cached files back to the server immediately. Although conceptually simple, this approach is inefficient. An alternative solution is to relax the semantics of file sharing. Instead of requiring a READ to see the effects of all previous WRITEs, one can have a new rule that says: "Changes to an open file are initially visible only to the process (or possibly machine) that modified the file. Only when the file is closed are the changes made visible to other processes (or machines)." The adoption of such a rule does not change what happens in Figure 3-4(b), but it does redefine the actual behavior (B getting the original value of the file) as being the correct one. When A closes the file, it sends a copy to the server, so that subsequent READs get the new value, as required. This rule is widely implemented and is known as session semantics.

Figure 3-4. *(a) On a single processor, when a READ follows a WRITE, the value returned by the READ is the value just written. (b) In a distributed system with caching, obsolete values may be returned.*

Using session semantics raises the question of what happens if two or more clients are simultaneously caching and modifying the same file. One solution is to say that as each file is closed in turn, its value is sent back to the server, so the final result depends on who closes last. A less pleasant, but slightly easier to implement, alternative is to say that the final result is one of the candidates, but leave the choice of which one unspecified.

One final difficulty with using caching and session semantics is that it violates another aspect of the UNIX semantics in addition to not having all READs return the value most recently written. In UNIX, associated with each open file is a pointer that indicates the current position in the file. READs take data starting at this position and WRITEs deposit data there. This pointer is shared between the process that opened the file and all its children. With session semantics, when the children run on different machines, this sharing cannot be achieved.

To see what the consequences of having to abandon shared file pointers are, consider a command like:

```
run >out
```

where run is a shell script that executes two programs, a and b, one after another. If both programs produce output, it is expected that the output produced by b will directly follow the output from a within out. The way this is achieved is that when b starts up, it inherits the file pointer from a, which is shared by the shell and both processes. In this way, the first byte that b writes directly follows the last byte written by a. With session semantics and no shared file pointers, a completely different mechanism is needed to make shell scripts and similar constructions that use shared file pointers work. Since no general-purpose solution to this problem is known, each system must deal with it in an ad hoc way.

A completely different approach to the semantics of file sharing in a distributed system is to make all files immutable. There is thus no way to open a file for writing. In effect, the only operations on files are CREATE and READ.

What is possible is to create an entirely new file and enter it into the directory system under the name of a previous existing file, which now becomes inaccessible (at least under that name). Thus although it becomes impossible to modify the file x, it remains possible to (atomically) replace x by a new file. In other words, although files cannot be updated, directories can be. Once we have decided that files cannot be changed at all, the problem of how to deal with two processes, one of which is writing on a file and the other of which is reading it, simply disappears.

What does remain is the problem of what happens when two processes try to replace the same file at the same time. As with session semantics, the best solution here seems to be to allow one of the new files to replace the old one, either the last one or nondeterministically.

A somewhat stickier problem is what to do if a file is replaced while another process is busy reading it. One solution is to somehow arrange for the reader to continue using the old file, even if it is no longer in any directory, analogous to the way UNIX allows a process that has a file open to continue using it, even after it has been deleted from all directories. Another solution is to detect that the file has changed and make subsequent attempts to read from it fail.

A fourth way to deal with shared files in a distributed system is to use atomic transactions, as we discussed in detail in Chap. 11. To summarize briefly, to access a file or a group of files, a process first executes some type of BEGIN TRANSACTION primitive to signal that what follows must be executed indivisibly. Then come system calls to read and write one or more files. When the work has been completed, an END TRANSACTION primitive is executed. The key property of this method is that the system guarantees that all the calls contained within the transaction will be carried out in order, without any interference from other, concurrent transactions. If two or more transactions start up at the same time, the system ensures that the final result is the same as if they were all run in some (undefined) sequential order.

The classical example of where transactions make programming much easier is in a banking system. Imagine that a certain bank account contains I00 dollars, and that two processes are each trying to add 50 dollars to it. In an unconstrained system, each process might simultaneously read the file containing the current balance (100), individually compute the new balance (150), and successively overwrite the file with this new value. The final result could either be 150 or 200, depending on the precise timing of the reading and writing. By grouping all the operations into a transaction, interleaving cannot occur and the final result will always be 200.

In Table 3-1 we summarize the four approaches we have discussed for dealing with shared files in a distributed system.

Method	Comment
UNIX semantics	Every operation on a file is instantly visible to all processes
Session semantics	No changes are visible to other processes until the file is closed
Immutable files	No updates are possible; simplifies sharing and replication
Transactions	All changes have the all-or-nothing property

Table 3-1. *Four ways of dealing with the shared files in a distributed system.*

3.2 DISTRIBUTED FILE SYSTEM IMPLEMENTATION

In the previous section, we have described various aspects of distributed file systems from the user s perspective, that is, how they appear to the user. In this section we will see how these systems are implemented. We will start out by presenting some experimental information about file usage. Then we will go on to look at system structure, the implementation of caching, replication in distributed systems, and concurrency control. Finally, we will conclude with a short discussion of some lessons that have been learned from experience.

3.2.1 File Usage

Before implementing any system, distributed or otherwise, it is useful to have a good idea of how it will be used, in order to make sure that the most commonly executed operations will be efficient. To this end, Satyanarayanan (1981) made a comprehensive study of file usage patterns. We will present his major results below.

However, first, a few words of warning about these and similar measurements are in order. Some of the measurements are static, meaning they represent a snapshot of the system at a certain instant. Static measurements are made by examining the disk to see what is on it. These measurements include the distribution of file sizes, the distribution of file types, and the amount of storage occupied by files of various types and sizes. Other measurements are dynamic, made by modifying the file system to record all operations to a log for subsequent analysis. These data yield information about the relative frequency of various operations, the number of files open at any moment, and the amount of sharing that takes place. By combining the static and dynamic measurements, even though they are fundamentally different, we can get a better picture of how the file system is used.

One problem that always occurs with measurements of any existing system is knowing how typical the observed user population is. Satyanarayanan's measurements were made at a university. Do they also apply to industrial research labs? To office automation projects? To banking systems? No one really knows for sure until these systems, too, are instrumented and measured.

Another problem inherent in making measurements is watching out for artifacts of the system being measured. As a simple example, when looking at the distribution of file names in an MS-DOS system, one could quickly conclude that file names are never more than 8 characters (plus an optional three-character extension). However, it would be a mistake to draw the conclusion that eight characters are therefore enough, since nobody ever uses more than eight characters. Since MS-DOS does not allow more than eight characters in a

file name, it is impossible to tell what users would do if they were not constrained to eight-character file names.

Finally, Satyanarayanan's measurements were made on more-or-less traditional UNIX systems. Whether or not they can be transferred or extrapolated to distributed systems is not really known.

This being said, the most important conclusions are listed in Table 3-2. From these observations, one can draw certain conclusions. To start with, most files are under 10K, which agrees with the results of Mullender and Tanenbaum (1984) made under different circumstances. This observation suggests that it may be feasible to transfer entire files rather than disk blocks between server and client. Since whole file transfer is typically simpler and more efficient, this idea is worth considering. Of course, some files are large, so provision has to be made for them too. Still, a good guideline is to optimize for the normal case and treat the abnormal case specially.

An interesting observation is that most files have short lifetimes. In other words, a common pattern is to create a file, read it (probably once) and then delete it. A typical usage might be a compiler that creates temporary files for transmitting information between its passes. The implication here is that it is probably a good idea to create the file on the client and keep it there until it is deleted. Doing so may eliminate a considerable amount of unnecessary client-server traffic.

The fact that few files are shared argues for client caching. As we have seen already, caching makes the semantics more complicated, but if files are rarely shared, it may well be best to do client caching and accept the consequences of session semantics in return for the better performance.

Finally, the clear existence of distinct file classes suggests that perhaps different mechanisms should be used to handle the different classes. System binaries need to be widespread but hardly ever change, so they should probably be widely replicated, even if this means that an occasional update is complex. Compiler and other temporary files are short, unshared, and disappear quickly, so they should be kept locally wherever possible. Electronic mailboxes are frequently updated but rarely shared, so replication is not likely to gain anything. Ordinary data files may be shared, so they may need still other handling.

Most files are small (less than 10 K)
Reading is much more common than writing
Reads and writes are sequential; random access is rare
Most files have a short lifetime
File sharing is unusual
The average process uses only a few files
Distinct file classes with different properties exist

Table 3-2. *Observed file system properties.*

3.2.2 System Structure

In this section we will look at some of the ways that file servers and directory servers are organized internally, with special attention to alternative approaches. Let us start with a very simple question: "Are clients and servers different?" Surprisingly enough, there is no agreement on this matter.

In some systems (e.g., NFS), there is no distinction between a client and a server. All machines run the same basic software, so any machine that wants to offer file service to the public at large is free to do so. Offering file service is just a matter of exporting the names of selected directories so other machines can access them.

In other systems, the file server and directory server are just user programs, so a system can be configured to run client and server software on the same machines or not, as it wishes. Finally, at the other extreme, are systems in which clients and servers are fundamentally different machines, either in terms of hardware or software. The servers may even run a different version of the operating system from the clients. While separation of function may seem a bit cleaner, there is no fundamental reason to prefer one approach over the others.

A second implementation issue on which systems differ is how the file and directory service is structured. One organization is to combine the two into a single server that handles all the directory and file calls itself. Another possibility, however, is to keep them separate. In the latter case, opening a file requires going to the directory server to map its symbolic name onto its binary name (e.g., machine + i-node) and then going to the file server with the binary name to actually read or write the file.

Arguing in favor of the split is that the two functions are really unrelated, so keeping them separate is more flexible. For example, one could implement an MS-DOS directory server and a UNIX directory server, both of which use the same file server for physical storage. Separation of function is also likely to produce simpler software. Weighing against this is that having two servers requires more communication.

Let us consider the case of separate directory and file servers for the moment. In the normal case, the client sends a symbolic name to the directory server, which then returns the binary name that the file server understands. However, it is possible for a directory hierarchy to be partitioned among multiple servers, as illustrated in Figure 3-5. Suppose, for example, we have a system in which the current directory, on server 1, contains an entry, a, for another directory on server 2. Similarly, this directory contains an entry, b, for a directory on server 3. This third directory contains an entry for a file c, along with its binary name.

To look up a/b/c, the client sends a message to server 1, which manages its current directory. The server finds a, but sees that the binary name refers to another server. It now has a choice. It can either tell the client which server holds b and have the client look up b/c there itself, as shown in Figure 3-5(a), or it can forward the remainder of the request to server 2 itself and not reply at all, as shown in Figure 3-5(b). The former scheme requires the client to be aware of which server holds which directory, and requires more messages. The latter method is more efficient, but cannot be handled using normal RPC since the process to which the client sends the message is not the one that sends the reply.

Looking up path names all the time, especially if multiple directory servers are involved, can be expensive. Some systems attempt to improve their performance by maintaining a cache of hints, that is, recently looked up names and the results of these lookups. When a file is opened, the cache is checked to see if the path name is there. If so, the directory-by-directory lookup is skipped and the binary address is taken from the cache.

For name caching to work, it is essential that when an obsolete binary name is inadvertently used, the client is somehow informed so it can fall back on the directory-by-

directory lookup to find the file and update the cache. Furthermore, to make hint caching worthwhile in the first place, the hints have to be right most of the time. When these conditions are fulfilled, caching hints can be a powerful technique that is applicable to many areas of distributed operating systems.

The final structural issue that we will consider here is whether or not file, directory, and other servers should maintain state information about clients. This issue is moderately controversial, with two competing schools of thought in existence.

One school thinks that servers should be **stateless**. In other words, when a client sends a request to a server, the server carries out the request, sends the reply, and then removes from its internal tables all information about the request. Between requests, no client-specific information is kept on the server. The other school of thought maintains that it is all right for servers to maintain state information about clients between requests. After all, centralized operating systems maintain state information about active processes, so why should this traditional behavior suddenly become unacceptable?

To better understand the difference, consider a file server that has commands to open, read, write, and close files. After a file has been opened, the server must maintain information about which client has which file open. Typically, when a file is opened, the client is given a file descriptor or other number which is used in subsequent calls to identify the file. When a request comes in, the server uses the file descriptor to determine which file is needed. The table mapping the file descriptors onto the files themselves is state information.

With a stateless server, each request must be self contained. It must contain the full file name and the offset within the file, in order to allow the server to do the work. This information increases message length.

Another way to look at state information is to consider what happens if a server crashes and all its tables are lost forever. When the server is rebooted, it no longer knows which clients have which files open. Subsequent attempts to read and write open files will then fail, and recovery, if possible at all, will be entirely up to the clients. As a consequence, stateless servers tend to be more fault tolerant than those that maintain state, which is one of the arguments in favor of the former.

The arguments both ways are summarized in Table 3-3. Stateless servers are inherently more fault tolerant, as we just mentioned. OPEN and CLOSE calls are not needed, which reduces the number of messages, especially for the common case in which the entire file is read in a single blow. No server space is wasted on tables. When tables are used, if too many clients have too many files open at once, the tables can fill up and new files cannot be opened. Finally, with a stateful server, if a client crashes when a file is open, the server is in a bind. If it does nothing, its tables will eventually fill up with junk. If it times out inactive open files, then a client that happens to wait too long between requests will be refused service, and correct programs will fail to function correctly. Statelessness eliminates these problems.

Advantages of stateless servers	Advantages of stateful servers
Fault tolerance	Shorter request messages
No OPEN/CLOSE calls needed	Better performance
No server space wasted on tables	Readahead possible
No limits on number of open files	Idempotency easier
No problems if a client crashes	File locking possible

Table 3-3. *A comparison of stateless and stateful servers.*

Stateful servers also have things going for them. Since READ and WRITE messages do not have to contain file names, they can be shorter, thus using less network bandwidth. Better performance is frequently possible since information about open files (in UNIX terms, the i-nodes) can be kept in main memory until the files are closed. Blocks can be read in advance to reduce delay, since most files are read sequentially. If a client ever times out and sends the same request twice, for example, APPEND, it is much easier to detect this with state (by having a sequence number in each message). Achieving idempotency in the face of unreliable communication with stateless operation takes more thought and effort. Finally, file locking is impossible to do in a truly stateless system, since the only effect setting a lock has is to enter state into the system. In stateless systems, file locking has to be done by a special lock server.

3.2.3 Caching

In a client-server system, each with main memory and a disk, there are four potential places to store files, or parts of files: the server's disk, the server's main memory, the client's disk (if available), or the client's main memory, as illustrated in Figure 3-6. These different storage locations all have different properties, as we shall see.

The most straightforward place to store all files is on the server's disk. There is generally plenty of space there and the files are then accessible to all clients. Furthermore, with only one copy of each file, no consistency problems arise.

The problem with using the server's disk is performance. Before a client can read a file, the file must first be transferred from the server's disk to the server's main memory, and then again over the network to the client's main memory. Both transfers take time.

A considerable performance gain can be achieved by **caching** (i.e., holding) the most recently used files in the server's main memory. A client reading a file that happens to be in the server's cache eliminates the disk transfer, although the network transfer still has to be done. Since main memory is invariably smaller than the disk, some algorithm is needed to determine which files or parts of files should be kept in the cache.

This algorithm has two problems to solve. First, what is the unit the cache manages? It can be either whole files or disk blocks. If entire files are cached, they can be stored contiguously on the disk (or at least in very large chunks), allowing high-speed transfers between memory and disk and generally good performance. Disk block caching, however, uses cache and disk space more efficiently.

Second, the algorithm must decide what to do when the cache fills up and something must be evicted. Any of the standard caching algorithms can be used here, but because

cache references are so infrequent compared to memory references, an exact implementation of LRU using linked lists is generally feasible. When something has to be evicted, the oldest one is chosen. If an up-to-date copy exists on disk, the cache copy is just discarded. Otherwise, the disk is first updated.

Having a cache in the server's main memory is easy to do and totally transparent to the clients. Since the server can keep its memory and disk copies synchronized, from the clients' point-of-view, there is only one copy of each file, so no consistency problems arise.

Although server caching eliminates a disk transfer on each access, it still has a network access. The only way to get rid of the network access is to do caching on the client side, which is where all the problems come in. The tradeoff between using the client's main memory or its disk is one of space versus performance. The disk holds more, but is slower. When faced with a choice between having a cache in the server's main memory versus the client's disk, the former is usually somewhat faster, and it is always much simpler. Of course, if large amounts of data are being used, a client disk cache may be better. In any event, most systems that do client caching do it in the client's main memory, so we will concentrate on that.

If the designers decide to put the cache in the client's main memory, three options are open as to precisely where to put it. The simplest is to cache files directly inside each user process' own address space, as shown in Figure 3-7(b). Typically, the cache is managed by the system call library. As files are opened, closed, read, and written, the library simply keeps the most heavily used ones around, so that when a file is reused, it may already be available. When the process exits, all modified files are written back to the server. Although this scheme has an extremely low overhead, it is only effective if individual processes open and close files repeatedly. A data base manager process might fit this description, but in the usual program development environment, most processes only read each file once, so caching within the library wins nothing.

The second place to put the cache is in the kernel, as shown in Figure 3-7(c). The disadvantage here is that a kernel call is needed in all cases, even on a cache hit, but the fact that the cache survives the process more than compensates. For example, suppose a two-pass compiler runs as two processes. Pass one writes an intermediate file read by pass two. In Figure 3-7(c), after the pass one process terminates; the intermediate file will probably be in the cache, so no server calls will have to be made when the pass two process reads it in.

The third place for the cache is in a separate user-level cache manager process, as shown in Figure 3-7(d). The advantage of a user-level cache manager is that it keeps the (micro)kernel free of file system code, is easier to program because it is completely isolated, and is more flexible.

On the other hand, when the kernel manages the cache, it can dynamically decide how much memory to reserve for programs and how much for the cache. With user-level cache manager running on a machine with virtual memory, it is conceivable that the kernel could decide to page out some or all of the cache to a disk, so that a so-called "cache hit" requires one or more pages to be brought in. Needless to say, this defeats the idea of client caching completely. However, if it is possible for the cache manager to allocate and lock in memory some number of pages, this ironic situation can be avoided.

When evaluating whether caching is worth the trouble at all, it is important to note that in Figure 3-7(a), it takes exactly one RPC to make a file request, no matter what. In both Figure 3-7(c) and Figure 3-7(d) it takes either one or two, depending on whether the request can be satisfied out of the cache or not. Thus the mean number of RPCs is always greater when caching is used. In a situation in which RPCs are fast and network transfers are slow (fast CPUs, slow networks) caching can give a big gain in performance. If,

however, network transfers are very fast (e.g., with high-speed fiber optic networks), the network transfer time will matter less, so the extra RPCs may eat up a substantial fraction of the gain. Thus the performance gain provided by caching depends to some extent on the CPU and network technology available, and of course, on the applications.

Cache Consistency

As usual in computer science, you never get something for nothing. Client caching introduces inconsistency into the system. If two clients simultaneously read the same file and then both modify it, several problems occur. For one, when a third process reads the file from the server, it will get the original version, not one of the two new ones. This problem can be defined away by adopting session semantics (officially stating that the effects of modifying a file are not supposed to be visible globally until the file is closed). In other words, this "incorrect" behavior is simply declared to be the "correct" behavior. Of course, if the user expects UNIX semantics, the trick does not work.

Another problem, unfortunately, that cannot be defined away at all, is that when the two files are written back to the server, the one written last will overwrite the other one. The moral of the story is that client caching has to be thought out fairly carefully. Below we will discuss some of the problems and proposed solutions.

One way to solve the consistency problem is to use the **write through** algorithm. When a cache entry (file or block) is modified, the new value is kept in the cache, but is also immediately sent to the server. As a consequence, when another process reads the file, it gets the most recent value.

However, the following problem arises. Suppose a client process on machine A reads a file, f. The client terminates, but the machine keeps f in its cache. Later, a client on machine B reads the same file, modifies it, and writes it through to the server. Finally, a new client process is started up on machine A. The first thing it does is open and read f, which is taken from the cache. Unfortunately, the value there is now obsolete.

A possible way out is to require the cache manager to check with the server before providing any client with a file from the cache. This check could be done by comparing the time of last modification of the cached version with the server's version. If they are the same, the cache is up-to-date. If not, the current version must be fetched from the server. Instead of using dates, version numbers or checksums can also be used. Although going to the server to verify dates, version numbers, or checksums takes an RPC, the amount of data exchanged is small. Still, it takes some time.

Another trouble with the write through algorithm is that although it helps on reads, the network traffic for writes is the same as if there were no caching at all. Many system designers find this unacceptable, and cheat: instead of going to the server the instant the write is done, the client just makes a note that a file has been updated. Once every 30 seconds or so, all the file updates are gathered together and sent to the server all at once. A single bulk write is usually more efficient than many small ones.

Besides, many programs create scratch files, write them, read them back, and then delete them, all in quick succession. In the event that this entire sequence happens before it is time to send all modified files back to the server, the now-deleted file does not have to be written back at all. Not having to use the file server at all for temporary files can be a major performance gain.

Of course, delaying the writes muddies the semantics, because when another process reads the file, what it gets depends on the timing. Thus postponing the writes is a tradeoff

between better performance and cleaner semantics (which translates into easier programming).

The next step in this direction is to adopt session semantics and only write a file back to the server after it has been closed. This algorithm is called write-on-close. Better yet, wait 30 seconds after the close to see if the file is going to be deleted. As we saw earlier, going this route means that if two cached files are written back in succession, the second one overwrites the first one. The only solution to this problem is to note that it is not nearly as bad as it first appears. In a single CPU system, it is possible for two processes to open and read a file, modify it within their respective address spaces, and then write it back. Consequently, write-on-close with session semantics is not that much worse than what can happen on a single CPU system.

A completely different approach to consistency is to use a centralized control algorithm. When a file is opened, the machine opening it sends a message to the file server to announce this fact. The file server keeps track of who has which file open, and whether it is open for reading, writing, or both. If a file is open for reading, there is no problem with letting other processes open it for reading, but opening it for writing must be avoided. Similarly, if some process has a file open for writing, all other accesses must be prevented. When a file is closed, this event must be reported, so the server can update its tables telling which client has which file open. The modified file can also be shipped back to the server at this point.

When a client tries to open a file and the file is already open elsewhere in the system, the new request can either be denied or queued. Alternatively, the server can send an **unsolicited message** to all clients having the file open, telling them to remove that file from their caches and disable caching just for that one file. In this way, multiple readers and writers can run simultaneously, with the results being no better and no worse than would be achieved on a single CPU system.

Although sending unsolicited messages is clearly possible, it is inelegant, since it reverses the client and server roles. Normally, servers do not spontaneously send messages to clients or initiate RPCs with them. If the clients are multithreaded, one thread can be permanently allocated to waiting for server requests, but if they are not, the unsolicited message must cause an interrupt.

Even with these precautions, one must be careful. In particular, if a machine opens, caches, and then closes a file, upon opening it again the cache manager must still check to see if the cache is valid. After all, some other process might have subsequently opened, modified, and closed the file. Many variations of this centralized control algorithm are possible, with differing semantics. For example, servers can keep track of cached files, rather than open files. All these methods have a single point of failure and none of them scale well to large systems.

The four cache management algorithms discussed above are summarized in Table 3-4. To summarize the subject of caching as a whole, server caching is easy to do and almost always worth the trouble, independent of whether client caching is present or not. Server caching has no effect on the file system semantics seen by the clients. Client caching, in contrast, offers better performance at the price of increased complexity and possibly fuzzier semantics. Whether it is worth doing or not depends on how the designers feel about performance, complexity, and ease of programming.

Method	Comments

Write through	Works, but does not affect write traffic
Delayed write	Better performance but possibly ambiguous semantics
Write on close	Matches session semantics
Centralized control	UNIX semantics, but not robust and scales poorly

Table 3-4. *Four algorithms for managing a client file cache.*

Earlier in this chapter, when we were discussing the semantics of distributed file systems, we pointed out that one of the design options is immutable files. One of the great attractions of an immutable file is the ability to cache it on machine A without having to worry about the possibility that machine B will change it. Changes are not permitted. Of course, a new file may have been created and bound to the same symbolic name as the cached file, but this can be checked for whenever a cached file is reopened. This model has the same RPC overhead discussed above, but the semantics are less fuzzy.

3.2.4 Replication

Distributed file systems often provide file replication as a service to their clients. In other words, multiple copies of selected files are maintained, with each copy on a separate file server. The reasons for offering such a service vary, but among the major reasons are:

1. To increase reliability by having independent backups of each file. If one server goes down, or is even lost permanently, no data are lost. For many applications, this property is extremely desirable.
2. To allow file access to occur even if one file server is down. The motto here is: "The show must go on." A server crash should not bring the entire system down until the server can be rebooted.
3. To split the workload over multiple servers. As the system grows in size, having all the files on one server can become a performance bottleneck. By having files replicated on two or more servers, the least heavily loaded one can be used.

The first two relate to improving reliability and availability; the third concerns performance. All are important.

A key issue relating to replication is transparency (as usual). To what extent are the users aware that some files are replicated? Do they play any role in the replication process, or is it handled entirely automatically? At one extreme, the users are fully aware of the replication process, and can even control it. At the other, the system does everything behind their backs. In the latter case, we say the system is **replication transparent**.

Figure 3-8 shows three ways replication can be done. The first way, shown in Figure 3-8(a), is for the programmer to control the entire process. When a process makes a file, it does so on one specific server. Then, it can make additional copies on other servers, if desired. If the directory server permits multiple copies of a file, the network addresses of all copies can then be associated with the file name, as shown at the bottom of Figure 3-8(a), so that when the name is looked up, all copies will be found. When the file is subsequently opened, the copies can be tried sequentially in some order, until an available one is found.

To make the concept of explicit replication more familiar, consider how it can be done in a system based on remote mounting in UNIX. Suppose that a programmer's home directory is */machine1/usr/ast*. After creating a file, for example the file, */machine1/usr/ast/xyz*, the programmer, process, or library can use the *cp* command (or equivalent) to make copies in */machine2/usr/ast/xyz* and */machine3/usr/ast/xyz*. Programs can be written to accept strings like */usr/ast/xyz* as arguments, and successively try to open the copies until one succeeds. While this scheme can be made to work, it is a lot of trouble. For this reason, a distributed system should do better.

In Figure 3-8(b) we see an alternative approach, lazy replication. Here, only one copy of each file is created, on some server. Later, the server itself makes replicas on other servers automatically, without the programmer's knowledge. The system must be smart enough to be able to retrieve any of these copies if need be. When making copies in the background like this, it is important to pay attention to the possibility that the file might change before the copies can be made.

Our final method is to use group communication, as shown in Figure 3-8(c). In this scheme, all WRITE system calls are simultaneously transmitted to all the servers at once, so extra copies are made at the same time the original is made. There are two principal differences between lazy replication and using a group. First, with lazy replication, one server is addressed, rather than a group. Second, lazy replication happens in the background, when the server has some free time, whereas when group communication is used, all copies are made at the same time.

Update Protocols

Above we looked at the problem of how replicated files can be created. Now let us see how existing ones can be modified. Just sending an update message to each copy in sequence is not a good idea because if the process doing the update crashes part way through, some copies will be changed and others not. As a result, some future reads may get the old value and others may get the new value, hardly a desirable situation. We will look at two well-known algorithms that solve this problem.

The first one is called **primary copy replication**. When it is used, one server is designated as the primary. All the others are secondaries. When a replicated file is to be updated, the change is sent to the primary server, which makes the change locally, and then sends commands to the secondaries, ordering them to change too. Reads can be done from any copy, primary or secondary.

To guard against the situation that the primary crashes before it has had a chance to instruct all the secondaries, the update should be written to stable storage prior to changing the primary copy. In this way, when a server reboots after a crash, a check can be made to see if any updates were in progress at the time of the crash. If so, they can still be carried out. Sooner or later, all the secondaries will be updated.

Although the method is straightforward, it has the disadvantage that if the primary is down, no updates can be performed. To get around this asymmetry, Gifford (1979) proposed a more robust method, known as voting. The basic idea is to require clients to request and acquire the permission of multiple servers before either reading or writing a replicated file.

As a simple example of how the algorithm works, suppose that a file is replicated on N servers. We could make a rule stating that to update a file, a client must first contact at least half the servers plus 1 (a majority) and get them to agree to do the update. Once they have agreed, the file is changed and a new version number is associated with the new file.

The version number is used to identify the version of the file, and is the same for all the newly updated files.

To read a replicated file, a client must also contact at least half the servers plus 1 and ask them to send the version numbers associated with the file. If all the version numbers agree, this must be the most recent version because an attempt to update only the remaining servers would fail because there are not enough of them.

For example, if there are five servers, and a client determines that three of them have version 8, it is impossible that the other two have version 9. After all, any successful update from version 8 to version 9 requires getting three servers to agree to it, not just two.

Gifford's scheme is actually somewhat more general than this. In it, to read a file of which N replicas exist, a client needs to assemble a **read quorum**, an arbitrary collection of any N_r servers, or more. Similarly, to modify a file, a write quorum of at least N_w servers is required. The values of N_r and N_w are subject to the constraint that $N_r + N_w > N$. Only after the appropriate number of servers has agreed to participate can a file be read or written.

To see how this algorithm works, consider Figure 3-9(a), which has $N_r = 3$ and $N_w = 10$. Imagine that the most recent write quorum consisted of the 10 servers C through L. All of these get the new version and the new version number. Any subsequent read quorum of three servers will have to contain at least one member of this set. When the client looks at the version numbers, it will know which is most recent and take that one.

In Figure 3-9(b) and Figure 3-9(c) we see two more examples. The latter is especially interesting because it sets N_r to 1, making it possible to read a replicated file by finding any copy and using it. The price paid, however, is that write updates need to acquire all copies.

An interesting variation on voting is **voting with ghosts** (Van Renesse and Tanenbaum, 1988). In most applications, reads are much more common than writes, so N_r is typically a small number and N_w is nearly N. This choice means that if a few servers are down, it may be impossible to obtain a write quorum at all.

Voting with ghosts solves this problem by creating a dummy server, with no storage, for each real server that is down. A ghost is not permitted in a read quorum (it does not have any files, after all), but it may join a write quorum, in which case it just throws away the file written to it. A write only succeeds if at least one server is real.

When a failed server is rebooted, it must obtain a read quorum to locate the most recent version, which it then copies to itself before starting normal operation. The algorithm works because it has the same property as the basic voting scheme, namely, N_r and N_w are chosen so that acquiring a read quorum and a write quorum at the same time is impossible. The only difference here is that dead machines are allowed in a write quorum, subject to the condition that when they come back up they immediately obtain the current version before going into service.

3.3

A CASE STUDY : NFS (NETWORK FILE SYSTEM)

NFS was originally designed and implemented by Sun Microsystems for use on its UNIX-based workstations. Other manufacturers now support it as well, for both UNIX and other operating systems (including MS-DOS). NFS supports heterogeneous systems, for example, MS-DOS clients making use of UNIX servers. It is not even required that all the machines use the same hardware. It is common to find MS-DOS clients running on Intel 386 CPUs getting service from UNIX file servers running on Motorola 68030 or Sun SPARC CPUs.

Three aspects of NFS are of interest: the architecture, the protocol, and the implementation. Let us look at these in turn.

3.3.1 NFS Architecture

The basic idea behind NFS is to allow an arbitrary collection of clients and servers to share a common file system. In most cases, all the clients and servers are on the same LAN, but this is not required. It is possible to run NFS over a wide-area network. For simplicity we will speak of clients and servers as though they were on distinct machines, but in fact, NFS allows every machine to be both a client and a server at the same time.

Each NFS server exports one or more of its directories for access by remote clients. When a directory is made available, so are all of its subdirectories, so in fact, entire directory trees are normally exported. The list of directories a server exports is maintained in the */etc/exports* file, so these directories can be exported automatically when the server is booted.

Clients access exported directories by mounting them. When a client mounts a (remote) directory, it becomes part of its directory hierarchy, as shown in Figure 1-7. Many Sun workstations are diskless. If it so desires, a diskless client can mount a remote file system on its root directory, resulting in a file system that is supported entirely on a remote server. Those workstations that do have local disks can mount remote directories anywhere they wish on top of their local directory hierarchy, resulting in a file system that is partly local and partly remote. To programs running on the client machine, there is (almost) no difference between a file located on a remote file server and a file located on the local disk.

Thus the basic architectural characteristic of NFS is that servers export directories and clients remotely mount them. If two or more clients mount the same directory at the same time, they can communicate by sharing files in their common directories. A program on one client can create a file, and a program on a different one can read the file. Once the mounts have been done, nothing special has to be done to achieve sharing. The shared files are just there in the directory hierarchy of multiple machines, and can be read and written the usual way. This simplicity is one of the great attractions of NFS.

3.3.2 NFS Protocols

Since one of the goals of NFS is to support a heterogeneous system, with clients and servers possibly running different operating systems on different hardware, it is essential that the interface between the clients and servers be well defined. Only then is it possible for anyone to be able to write a new client implementation and expect it to work correctly with existing servers, and vice versa.

NFS accomplishes this goal by defining two client-server protocols. A **protocol** is a set of requests sent by clients to servers, along with the corresponding replies sent by the servers back to the clients. (Protocols are an important topic in distributed systems; we will come back to them later in more detail.) As long as a server recognizes and can handle all the requests in the protocols, it need not know anything at all about its clients. Similarly, clients can treat servers as "black boxes" that accept and process a specific set of requests. How they do it is their own business.

The first NFS protocol handles mounting. A client can send a path name to a server and request permission to mount that directory somewhere in its directory hierarchy. The place where it is to be mounted is not contained in the message as the server does not care where it is to be mounted. If the path name is legal and the directory specified has been exported, the server returns a **file handle** to the client. The file handle contains fields uniquely identifying the file system type, the disk, the i-node number of the directory, and security information. Subsequent calls to read and write files in the mounted directory use the file handle.

Many clients are configured to mount certain remote directories without manual intervention. Typically, these clients contain a file called */etc/rc*, which is a shell script containing the remote mount commands. This shell script is executed automatically when the client is booted.

Alternatively, Sun's version of UNIX also supports **automounting**. This feature allows a set of remote directories to be associated with a local directory. None of these remote directories are mounted (or their servers even contacted) when the client is booted. Instead, the first time a remote file is opened, the operating system sends a message to each of the servers. The first one to reply wins, and its directory is mounted.

Automounting has two principal advantages over static mounting via the */etc/rc* file. First, if one of the NFS servers named in */etc/rc* happens to be down, it is impossible to bring the client up, at least not without some difficulty, delay, and quite a few error messages. If the user does not even need that server at the moment, all that work is wasted. Second, by allowing the client to try a set of servers in parallel, a degree of fault tolerance can be achieved (because only one of them need to be up), and the performance can be improved (by choosing the first one to reply – presumably the least heavily loaded).

On the other hand, it is tacitly assumed that all the file systems specified as alternatives for the automount are identical. Since NFS provides no support for file or directory replication, it is up to the user to arrange for all the file systems to be the same. Consequently, automounting is most often used for read-only file systems containing system binaries and other files that rarely change.

The second NFS protocol is for directory and file access. Clients can send messages to servers to manipulate directories and to read and write files. In addition, they can also access file attributes, such as file mode, size, and time of last modification. Most UNIX system calls are supported by NFS, with the perhaps surprising exception of OPEN and CLOSE.

The omission of OPEN and CLOSE is not an accident. It is fully intentional. It is not necessary to open a file before reading it, nor to close it when done. Instead, to read a file, a client sends the server a message containing the file name, with a request to look it up and return a file handle, which is a structure that identifies the file. Unlike an OPEN call, this LOOKUP operation does not copy any information into internal system tables. The READ call contains the file handle of the file to read, the offset in the file to begin reading, and the number of bytes desired. Each such message is self contained. The advantage of this scheme is that the server does not have to remember anything about open connections in between

calls to it. Thus if a server crashes and then recovers, no information about open files is lost, because there is none. A server like this that does not maintain state information about open files is said to be **stateless**.

In contrast, in UNIX System V, the **Remote File System**, **RFS**, requires a file to be opened before it can be read or written. The server then makes a table entry keeping track of the fact that the file is open, and where the reader currently is, so each request need not carry an offset. The disadvantage of this scheme is that if a server crashes and then quickly reboots, all open connections are lost, and client programs fail. NFS does not have this property.

Unfortunately, the NFS method makes it difficult to achieve the exact UNIX file semantics. For example, in UNIX a file can be opened and locked so that other processes cannot access it. When the file is closed, the locks are released. In a stateless server such as NFS, locks cannot be associated with open files, because the server does not know which files are open. NFS therefore needs a separate, additional mechanism to handle locking.

NFS uses the UNIX protection mechanism, with the *rwx* bits for the owner, group, and others. Originally, each request message simply contained the user and group ids of the caller, which the NFS server used to validate the access. In effect, it trusted the clients not to cheat. Several years' experience abundantly demonstrated that such an assumption was –how shall we put it? – naive. Currently, public key cryptography can be used to establish a secure key for validating the client and server on each request and reply. When this option is enabled, a malicious client cannot impersonate another client because it does not know that client's secret key. As an aside, cryptography is used only to authenticate the parties. The data themselves are never encrypted.

All the keys used for the authentication, as well as other information are maintained by the **NIS** (**Network Information Service**). The NIS was formerly known as the yellow pages. Its function is to store (key, value) pairs. When a key is provided, it returns the corresponding value. Not only does it handle encryption keys, but it also stores the mapping of user names to (encrypted) passwords, as well as the mapping of machine names to network addresses, and other items.

The network information servers are replicated using a master/slave arrangement. To read their data, a process can use either the master or any of the copies (slaves). However, all changes must be made only to the master, which then propagates them to the slaves. There is a short interval after an update in which the data base is inconsistent.

3.3.3 NFS Implementation

Although the implementation of the client and server code is independent of the NFS protocols, it is interesting to take a quick peek at Sun's implementation. It consists of three layers, as shown in Figure 3-10. The top layer is the system call layer. This handles the calls like OPEN, READ, and CLOSE. After parsing the call and checking the parameters, it invokes the second layer, the virtual file system (VFS) layer.

The task of the VFS layer is to maintain a table with one entry for each open file, analogous to the table of i-nodes for open files in UNIX. In ordinary UNIX, an i-node is uniquely indicated by a (device, i-node number) pair. Instead, the VFS layer has an entry, called a **v-node** (**virtual i-node**), for every open file. V-nodes are used to tell whether the file is local or remote. For remote files, enough information is provided to be able to access them.

To see how v-nodes are used, let us trace a sequence of MOUNT, OPEN, and READ system calls. To mount a remote file system, the system administrator calls the *mount* program specifying the remote directory, the local directory on which it is to be mounted, and other information. The mount program parses the name of the remote directory to be mounted and discovers the name of the machine on which the remote directory is located. It then contacts that machine asking for a file handle for the remote directory. If the directory exists and is available for remote mounting, the server returns a file handle for the directory. Finally, it makes a MOUNT system call passing the handle to the kernel.

The kernel then constructs a v-node for the remote directory and asks the NFS client code in Figure 3-10 to create an **r-node** (**remote i-node**) in its internal tables to hold the file handle. The v-node points to the r-node. Each v-node in the VFS layer will ultimately contain either a pointer to an r-node in the NFS client code, or a pointer to an i-node in the local operating system (see Figure 3-10). Thus from the v-node it is possible to see if a file or directory is local or remote, and if it is remote, to find its file handle.

When a remote file is opened, at some point during the parsing of the path name, the kernel hits the directory on which the remote file system is mounted. It sees that this directory is remote and in the directory's v-node finds the pointer to the r-node. It then asks the NFS client code to open the file. The NFS client code looks up the remaining portion of the path name on the remote server associated with the mounted directory and gets back a file handle for it. It makes an r-node for the remote file in its tables and reports back to the VFS layer, which puts in its tables a v-node for the file that points to the r-node. Again here we see that every open file or directory has a v-node that points to either an r-node or an i-node.

The caller is given a file descriptor for the remote file. This file descriptor is mapped onto the v-node by tables in the VFS layer. Note that no table entries are made on the server side. Although the server is prepared to provide file handles upon request, it does not keep track of which files happen to have file handles outstanding and which do not. When a file handle is sent to it for file access, it checks the handle, and if it is valid, uses it. Validation can include verifying an authentication key contained in the RPC headers, if security is enabled.

When the file descriptor is used in a subsequent system call, for example, READ, the VFS layer locates the corresponding v-node, and from that determines whether it is local or remote and also which i-node or r-node describes it.

For efficiency reasons, transfers between client and server are done in large chunks, normally 8192 bytes, even if fewer bytes are requested. After the client's VFS layer has gotten the 8K chunk it needs, it automatically issues a request for the next chunk, so it will have it should it be needed shortly. This feature is known as **read ahead**, and improves performance considerably.

For writes an analogous policy is followed. If a WRITE system call supplies fewer than 8192 bytes of data, the data is just accumulated locally. Only when the entire 8K chunk is full is it sent to the server. However, when a file is closed, all of its data are sent to the server immediately.

Another technique used to improve performance is caching, as in ordinary UNIX. Servers cache data to avoid disk accesses, but this is invisible to the clients. Clients maintain two caches, one for file attributes (i-nodes) and one for file data. When either an i-node or a file block is needed, a check is first made to see if it can be satisfied out of the client's cache. If so, network traffic can be avoided.

While client caching helps performance enormously, it also introduces some nasty problems. Suppose two clients are both caching the same file block, and one of them

modifies it. When the other one reads the block, it gets the old (stale) value. The cache is not coherent. We saw the same problem with multiprocessors earlier. However, there it was solved by having the caches snoop on the bus to detect all writes and invalidate or update cache entries accordingly. With a file cache that is not possible, because a write to a file that results in a cache hit on one client does not generate any network traffic. Even if it did, snooping on the network is nearly impossible with current hardware.

Given the potential severity of this problem, the NFS implementation does several things to mitigate it. For one, associated with each cache block is a timer. When the timer expires, the entry is discarded. Normally the timer is 3 sec for data blocks and 30 sec for directory blocks. Doing this reduces the risk somewhat. In addition, whenever a cached file is opened, a message is sent to the server to find out when the file was last modified. If the last modification occurred after the local copy was cached, the cache copy is discarded and the new copy fetched from the server. Finally, once every 30 sec a cache timer expires, and all the dirty (i.e., modified) blocks in the cache are sent to the server.

Still, NFS has been widely criticized for not implementing the proper UNIX semantics. A write to a file on one client may or may not be seen when another client reads the file, depending on the timing. Furthermore, when a file is created, it may not be visible to the outside world for as much as 30 sec. Other similar problems exist as well.

From this example we see that although NFS provides a shared file system, because the resulting system is kind of a patched up UNIX, the semantics of file access are not entirely well defined, and running a set of cooperating programs again may give different results, depending on the timing. Furthermore, the only issue NFS deals with is the file system. Other issues, such as process execution, are not addressed at all. Nevertheless, NFS is popular and widely used.

3.4 SUMMARY

The heart of any distributed system is the distributed file system. The design of such a file system begins with the interface: “What is the model of a file and what functionality is provided?” As a rule, the nature of a file is no different for the distributed case than for the single-processor case. As usual, an important part of theinterface is file naming and the directory system. Naming quickly brings up the issue of transparency. To what extent is the name of a file related to its location? Can the system move a single file on its own without the file name being affected? Different systems have different answers to these questions.

File sharing in a distributed system is a complex but important topic. Various semantic models have been proposed, ineluding UNIX semantics, session semantics, immutable files, and transaction semantics. Each has its own strengths and weaknesses. UNIX semantics is intuitive and familiar to most programmers (even non-UNIX programmers), but it is expensive to implement. Session semantics is less deterministic, but more efficient. Immutable files are unfamiliar to most people, andmake updating files difficult. Transactions are frequently overkill.

Implementing a distributed file system involves making many decisions. These include whether the system should be stateless or stateful, if and how caching should be done, and how file replication can be managed. Each of these has far-ranging consequences for the designers and the users. As an example, we looked at AFS and saw that it works by caching whole files on client disks, uploading them when they are closed.

Finally, we briefly examined some issues that are likely to become more important in the design of future distributed file systems, ineluding changes in hardware technology, scalability, wide-area systems, mobile users, and fault tolerance.

4.

MEMORY MANAGEMENT IN DTOS

4.1 VIRTUAL MEMORY

Virtual memory is the abstraction of single-level storage that is implemented, transparently, by a combination of primary memory, such as RAM chips, and backing storage, that is a high speed persistent storage mechanism such as a disk. Virtual memory is of considerable interest as an aspect of the design of distributed operating systems. First, a virtual memory implementation may need to use a backing store at a separate computer from the one that contains the primary memory. Secondly, it is possible to share data which is simultaneously mapped into the address spaces of processes residing at different computers in the form of distributed shared memory.

Much of the implementation of virtual memory in a distributed system is common to that found in a conventional operating system. The main difference is that the backing store interface is to a server, instead of a local disk. For a full description of virtual memory and its implementation in conventional, uniprocessor operating systems, the reader is referred to Tanenbaum [1992]. This section outlines those general features of the virtual memory concept and implementation that are applicable to distributed systems.

A central aim of virtual memory systems is to be able to execute large programs. and combinations of programs, whose entire code and data are too large to be stored in main memory at any one time. In virtual memory systems, part of main memory is used as a cache of the contents of backing storage. By storing only those sections of code and data currently being accessed by processes, it is possible (a) to run programs whose associated code and data exceeds the capacity of main memory, (b) to increase the level of multiprogramming by increasing the number of processes whose working code and data can be stored in main memory simultaneously, and (c) to remove the concerns of physical memory limitations from programmers.

This idea was then generalized so that any open file (and not just sections of executable files) could be made to correspond with an address space region. A process reads or writes the data in mapped files by reading or writing the corresponding virtual memory locations. In a typical high-level language, the open file appears as an array of bytes. The underlying kernel is responsible for fetching data from secondary storage as the array is accessed, and for writing modified data to permanent store. MULTICS was the original operating system to include mapped files. A more recent example of an operating system which suppons them is SunOS.

The most common implementation of virtual memory is called *demand paging*. Each page is fetched into primary memory upon demand: that is, when a process attempts to read or write data in a page which is not currently resident, it is fetched from backing store.

A virtual memory system is required to make decisions in two areas. First, its *frame allocation policy* is an algorithm for deciding how much main memory should be allocated to each running process. Secondly, *a page replacement policy is* used when a page must be fetched from secondary storage and there is no room in the main memory cache. A page is chosen to be replaced by the page to be brought in. The vinual memory system applies its policies at two points in the system's operation: (a) when a process attempts to reference a non-resident page, causing a page fault to be raised and handled by the kernel, and (b) periodically, upon measurement of page fault rates and/or each process's page reference patterns.

4.1.1 External pagers

In a distributed system, the computer running a process that incurs a page fault is not necessarily the same computer that manages the corresponding page data. For example, the first computer could be diskless. Even where a local disk is used for some paging, the pages of mapped files could be managed by a remote file server. The natural development for virtual memory implementation in distributed systems is for page data to be stored by a server, and not directly by the kernel using a local secondary storage device. These user-level servers are variously called **external pagers** *or external mappers* or memory *managers* (Figure 4-1).

Recall that a vinual address space is organized in regions. To consider a general model, we shall assume that any region in an address space is mapped to pan or all of some underlying **memory object**. A memory object is a contiguous, addressable resource such as a file or a set of pages managed by an external pager. If the region is an execution stack, for example, then the underlying memory object is one which persists only as long as the process executes. In the case of a region into which a file has been mapped, the underlying memory object is the file itself. By using an external server for paging, instead of keeping this functionality in the kernel, customized paging schemes can be implemented. External pagers also provide an approach to the implementation of distributed shared memory.

To map a memory object into a region, the process sends a request to the external pager that manages the memory object (see Figure 4-1). After the mapping, messages pass between the kernel and the external server to deal with paging. The kernel fetches initial data values from the external pager, and pages data to it, just as a conventional virtual memory operating system kernel pages in initial data values and pages out data to disk.

Figure 4-1. *An external pager manages mapped memory objeets.*

The kernel retains responsibility for handling page fault exceptions generated by local processes. It is responsible for main (physical) memory management, and therefore for implementing a frame allocation policy. The kernel is normally left to implement its own page replacement policy. The information necessary for applying the page replacement policy, such as bits set by the memory management unit when pages are referenced or modified, is local to the kernel.

The roles of the external pager are:

1. to receive and deal appropriately with data that have been purged by a kernel from its cache of pages, as part of the kernel's page replacement policy;
2. to supply page data as required by a kernel to complete its page fault handling; and
3. to impose consistency constraints determined by the underlying memory object abstraction, given that several kernels might attempt to cache modifiable pages of the object simultaneously.

Of these, (1) and (2) are straightforward extensions of the conventional virtual memory subsystem that stores each page as a disk block and records its disk address, or which looks up the address of pages and fetches them from disk, respectively. The difference in the distributed case is that the kernel requires not a disk address but a capability for the memory object. It uses this to transmit or request page data in messages to the corresponding external pager. The kernel is unaware of the whereabouts of the extemal pager. Data are copied in messages, so there is no presumption of the pager being local.

The external pager's role number (3), the maintenance of consistency, is relevant to mapped file access and distributed shared memory.

In summary, virtual memory is a framework for accessing any collection of memory objects that can be mapped to individual regions. The common requirement for any memory object abstraction is that it consists of contiguously addressable data items. which may be read and modified.

4.2 DISTRIBUTED SHARED MEMORY

This section describes distributed shared memory (DSM), an abstraction used for sharing data between processes in computers that do not share physical memory. The motivation for DSM is that it allows a shared memory programming model to be employed, which has some advantages over message-based models. For example, programmers do not have to marshal data items in DSM.

A central problem in implementing DSM is how to achieve good performance that is retained as systems scale to large numbers of computers. Accesses to DSM involve potential underlying network communication. Processes competing for the same or neighbouring data items may cause large amounts of communication to occur. The amount of communication is strongly related to the consistency model of a DSM - the model that determines which, of possibly many, written values will be returned when a process reads from a DSM location.

The section discusses DSM design issues such as the eonsistency model and implementation issues such as whether copies of the same data item are invalidated or updated when one copy is written. It goes on to discuss invalidation protocols in more detail. Finally, it describes release consistency - a relatively weak consistency model which is adequate for many purposes and relatively cheap to implement.

4.2.1 Introduction

Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory. Processes access DSM by reads and updates to what appears to be ordinary memory within their address space. However, an underlying run-time system ensures transparently that processes executing at different computers observe the updates made by one another. It is as though the processes access a single shared memory, but in fact the physical memory is distributed (see Figure 4-2).

The main point of DSM is that it spares the programmer the concerns of message passing when writing applications that might otherwise have to use it. DSM is primarily a tool for parallel applications or for any distributed application or group of applications in which individual shared data items can be accessed directly. DSM is in general less appropriate in client-server systems, where clients normally view server-held resources as abstract data and access them by request (for reasons of modularity and protection). However, servers can provide DSM that is shared between clients. For example, memory-mapped files that are shared and for which some degree of consistency is maintained are a form of DSM. (Mapped files were introduced with the MULTICS operating system.)

Message passing cannot be avoided altogether, of course, in a distributed system: in the absence of physically shared memory, the DSM run-time support has to send updates in messages between computers. DSM systems manage replicated data: each computer has a local copy of recently accessed data items stored in DSM, for speed of access. The problems of implementing DSM are in this respect related to those discussed in Chapter 11, as well as those of caching shared iles discussed in Chapter 7.

DSM has been an active area of research since the early 1980s. One of the first notable examples of a DSM implementation was the Apollo Domain file system [Leach et al.1983], in which processes hosted by different workstations share files by mapping them simultaneously into their address spaces. This example shows that distributed shared memory can be persistent. That is, it may outlast the execution of any process or group of processes that accesses it, and be shared by different groups of processes over time.

Figure 4-2. *The distributed shared memory abstraction.*

The significance of DSM has grown alongside the development of shared memory multiprocessors. Much research has gone into investigating algorithms suitable for parallel computation on these multiprocessors, and into the use of hardware caching for maximizing the number of processors that can be sustained by them. However, the practical limit stands at about 10 or 20 processors, where these are connected to memory modules over a common bus; after this the benefit of adding extra processors tails off rapidly.

Distributed memory multiprocessors and networked distributed systems, on the other hand, scale to far greater numbers of processors or computers. Emerging shared memory multiprocessor designs employ memory modules that are interconnected by high-speed networks, although processors see a single address space. A central question that has been pursued by the DSM and multiprocessor research communities is whether the investment in knowledge of shared memory algorithms and the associated software can be directly transferred to a more scalable distributed memory architecture.

Message passing versus DSM: As a communication mechanism, DSM is comparable to message passing rather than to request-reply-based communication, since its application to parallel processing, in particular, entails the use of asynchronous communication. The DSM and message passing approaches to programming can be contrasted as follows:

Programming model: Under the message passing model, variables have to be marshalled from one process, transmitted and unmarshalled into other variables at the receiving process. By contrast, with shared memory the processes involved share variables directly, so no marshalling is necessary - even of pointers to shared variables – and of course no separate communication operations are necessary. Most implementations allow variables stored in DSM to be named and accessed similarly to ordinary unshared variables. In favour of message passing, on the other hand, is that it allows processes to communicate while being protected from one another by having private address spaces, whereas processes sharing DSM can, for example, cause one another to fail by erroneously altering data. Furthermore, when message passing is used between heterogeneous computers, marshalling takes care of differences in data representation; but how can memory be shared between computers with, for example, different integer representations?

Synchronization between processes is achieved in the message model through message passing primitives themselves, using techniques such as the lock server implementation discussed in Chapter 10. In the case of DSM, synchronization is via normal constructs for shared memory programming such as locks and semaphores (although these require different implementations in the distributed memory environment). Chapter 6 discussed briefly such synchronization objects in the context of programming with threads.

Finally, since DSM can be made persistent, processes communicating via DSM may execute with non-overlapping lifetimes. A process can leave information in an agreed

location for the other to examine when it runs. By contrast, processes communicating via message passing must execute at the same time.

Efficiency: Experiments show that certain parallel programs developed for DSM can be made to perform about as well as functionally equivalent programs written for message passing platforms on the same hardware [Carter et al.1991] - at least in the case of relatively small numbers of computers (ten or so). However, this result cannot be generalized. The performance of a program based on DSM depends upon many factors, as we shall discuss below - particularly the pattem of data sharing (such as whether an item is updated by several processes).

There is a difference in the visibility of costs associated with the two types of programming. In message passing, all remote data accesses are explicit, and therefore the programmer is always aware of whether a particular operation is inprocess or involves the expense of communication. Using DSM, however, any parrticular read or update may or may not involve communication by the underlying run-time support. Whether it does or not depends upon such factors as whether the data have been accessed before, and the sharing pattern between processes at different computers.

There is no definitive answer as to whether DSM is preferable to message passing for any particular application. DSM is a promising tool whose ultimate status depends upon the efficiency with which it can be implemented.

Main approaches to DSM: There are three main approaches to the implementation of distributed shared memory, which respectively involve the use of hardware, virtual memory or library support. Note that they are not necessarily mutually exclusive:

Hardware-based: Some multiprocessor architectures rely on specialized hardware to handle load and store instructions applied to addresses in DSM, and to communicate with remote memory modules as necessary in order to implement them. These hierarchical designs, in which clusters of processors and memory modules are connected via a high-speed network, are aimed at orders of magnitude more processors than the limit of 10 or so that can be accommodated over a common bus: the prototype Dash multiprocessor has 64 nodes.

Page-based: All implement DSM as a region of virtual memory occupying the same address range in the address space of every participating process. In each case the kernel maintains the consistency of data within DSM regions as part of page fault handling.

Library-based: Some languages or language extensions such as Orca and Linda support forms of DSM. In this type of implementation sharing is not implemented through the virtual memory system but by communication between instances of the language run-time. Processes make library calls inserted by a compiler when they access data items in DSM. The libraries access local data items and communicate as necessary to maintain consistency. Processes may also execute threads to handle incoming messages requesting operations related to DSM.

The requirement for specialized hardware makes the first approach currently suitable only for multiprocessors; this chapter concentrates on the use of software to implement DSM on standard computers. (Even with hardware support, high-level software techniques may be used to minimize the amount of communication between components of a DSM implementation.) Of the other two, the page-based approach is receiving the most attention.

This is primarily because it imposes no particular structure on the DSM, which appears as a sequence of bytes; and it avoids the user-level overheads associated with libraries. Modern microkeroels such as Mach and Chorus provide virtual memory facilities which support DSM (and other memory abstractions). The advent of microprocessors such as the MIPS R4000 and DEC Alpha with their 64-bit address spaces widens the scope for this form of DSM by relaxing constraints on address space management. In object-based systems such as Clouds, Choices and COOL, page-based DSM allows threads to execute in parallel in objects that are replicated at multiple computers.

```
 #include "world.h"
struct shared {
        int a, b;
};

Program Writer:

main()
{
        struct shared *p;
        methersetup();                          /* Initialize the Mether run-time */
        p = (struct shared *)METHERBASE;        /* overlay structure on METHER segment */
        p->a = p->b = 0;                        /* initialize fields to zero */
        while(TRUE){                            /* continuously update structurefields */
                p ->a = p ->a + 1;
                p ->b = p ->b - 1;
        }
}

Program Reader:

main()
(
        struct shared *p;
        methersetup();
        p = (struct shared *)METHERBASE;
        while(TRUE){                            /* read the fields once every second */
                printf( "a = %d, b = %d\n", p ->a, p ->b);
                sleep(1);
        }
}
```

Example 4-1. *Mether system program*

The example in Example 4-1 is of two C programs, *Reader* and *Writer,* which communicate via the page-based DSM provided by the Mether system [Minnich and Farber 1989]. *Writer* updates two fields in a structure overlaid upon the beginning ofthe Mether DSM segment (beginning at address *METHERBASE),* and *Reader* periodically prints out the values it reads from these fields.

The two programs contain no special operations; they are compiled into machine instructions that access a common range of virtual memory addresses (starting at *METHERBASE).* Mether runs over conventional Sun workstation and network hardware.

4.2.2 Design and implementation issues

This section discusses design and implementation options concerning the main featuces that characterize a DSM system. These are: the structure of data held in DSM; the

synchronization model used to access DSM consistently at the application level: the DSM consistency model, which governs the consistency of data values accessed from different computers; the update options for communicating written values between computers; the granularity of sharing in a DSM implementation; and the problem of thrashing.

Structure

What is the application programmer's view of the contents of DSM? Three main approaches have been implemented, which view DSM as being composed respectively of bytes, shared objects or immutable data items.

> *Byte-oriented:* This is the view illustrated above by the Mether system. It is also the view of, for example, the Ivy system. It allows applications (and language implementations) to impose whatever data structures they want on the shared memory. Byte-oriented DSM is accessed as ordinary virtual memory.
>
> *Shared objects:* An advantage of viewing shared memory as a collection of objects is that synchronization can be applied at the level of the object operation. Orca views DSM as a collection of shared objects, and automatically serializes operations upon any given object.
>
> *Immutable data:* Both Agora and Linda view DSM as a collection of immutable data items that all processes can read. Processes have to replace data items in the DSM in lieu of modifying them. For example, Linda uses the tuple model. This comprises the notion of tuple space – a collection of immutable tuples with typed data items in their fields – and a set of primitives for accessing tuple space that can be added to any base language. Processes may place tuples in tuple space, and read or extract them from tuple space. Processes select tuples by specifying the number of their fields and their values or types.

The byte-oriented view allows the data types of any language to be superimposed relatively straightforwardly (as long as the DSM is shared between computers of homogeneous architecture - Exercise 17.2 explores the problem of heterogeneous computer architectures).

Synchronization model

Many applications apply constraints concerning the values stored in shared memory. This is as true of applications based on DSM as it is of applications written for shared memory multiprocessors (or indeed for any concurrent programs that share data, such as operating system kernels and multi-threaded servers). For example, if a and b are two variables stored in DSM, then a constraint might be that a = b always. If two or more processes execute the following code:

```
a := a + 1;
b := b + 1;
```

then an inconsistency may arise. Suppose a and b are initially zero, and that Process 1 gets as far as setting a to 1. Before it can increment b, Process 2 sets a to 2 and b to 1. The constraint has been broken. The solution is to make this code fragment into a critical section: to synchronize processes to ensure that only one may execute it at a time.

In order to use DSM, then, a distributed synchronization service needs to be provided, which includes familiar constructs such as locks and semaphores. Even when DSM is structured as a set of objects, the implementors of the objects have to be concerned with synchronization. Synchronization constructs are implemented using message passing.

Special machine instructions such as test-and-set, which are used for synchronization in shared memory multiprocessors, are in principle applicable to page-based DSM, but their operation in the distributed case is very inefficient. The most recent DSM implementations take advantage of application synchronization to reduce the amount of update transmission. The DSM then includes synchronization as an integrated component.

Consistency model

DSM implementations employ caching for enhanced performance. Each process has a local replica manager, which holds replicas of some DSM data items. In most implementations, data items are read using local values for efficiency, but updates have to be propagated to the other replica managers. In a pagebased DSM implementation, the kernels are replica managers: they cache DSM pages, which they map into the address spaces of local processes sharing DSM. In the Orca implementation, the processes themselves act as replica managers (when they execute library code). If DSM segments are persistent, then one or more storage servers (for example, file servers) will also act as replica managers.

In addition to caching, a DSM implementation could in principle buffer write accesses, and thus amortize communication costs by spreading them over multiple writes. Given that these techniques are available, the question arises of whether DSM memory itself is consistent when they are used. Note that this aspect of consistency, called *memory consistency*, is different from that discussed under the heading of application synchronization above.

Cheriton [1985] describes how forms of DSM can be envisaged for which some degree of inconsistency is acceptable. For example, DSM might be used to store the loads of computers on a network, in order that clients can select the least-loaded computers for running applications. Since such information is by its nature liable to become inaccurate on relatively small time scales, it would be a waste of effort to keep it consistent at all times.

Most applications do, however, have stronger consistency requirements. Care must be taken to give programmers a model that conforms to reasonable expectations of the way memory should behave. Before describing memory consistency requirements in more detail, it is helpful first to look at an example.

Process 1	Process 2
br := b;	*a := a+1;*
ar := a;	*b := b+1;*
if(ar >= br) then	
print ("OK");	

Example 4-2. *Two processes accessing shared variables.*

Consider an application in which two processes access two variables, *a* and *b* (Example 4-2), which are initialized to zero. Process 2 increments *a* and *b*, in that order. Process 1 reads the values of *b* and a into auxiliary variables *br* and *ar*, in that order. Note that there is no application-level synchronization. Intuitively, Process 1 should expect to see one of the following combinations of values, depending upon the points at which the read operations applied to *a* and *b* (implied in the statements *br := b* and *ar := a*) occur with respect to process 2's execution: *ar = 0, br = 0; ar =1, br = 0; ar =1, br = 1*. In other words, the condition *ar >= br* should always be satisfied, and Process 1 should print 'OK'. However, a DSM implementation might deliver the updates to *a* and *b* out of order to the replica manager for Process 1, in which case the combination *ar = 0, br =1* could occur.

The reader's immediate reaction to the example just given is probably that the DSM implementation, which reverses the order of two updates, is incorrect. If Process 1 and Process 2 execute together at a single-processor computer, we would assume that the memory subsystem was malfunctioning. However, it may be a correct implementation, in the distributed case, of a consistency model that is weaker than what many of us would intuitively expect, but which nonetheless can be useful and is relatively efficient.

The literature can only be described as inconsistent on the matter of the terms used to describe types of memory consistency model. However, Mosberger [1993] delineates a range of models that have been devised for shared-memory multiprocessors and software DSM systems. The main consistency models that can be practically realized in DSM implementations are sequential consistency and models that are based on weak consistency.

Sequential consistency: The central question to be asked in order to characterize a particular memory consistency model is this: when a read access is made to a memory location, which write accesses to the location are candidates whose values could be supplied to the read? At the weakest extreme, the answer is: any write that was issued before the read. This model would be obtained if replica managers could delay propagating updates to their peers indefinitely. It is too weak to be useful.

At the strongest extreme, all written values are instantaneously available to all processes: a read returns the most recent write at the time that the read takes place. Unfortunately, this definition is problematic in two respects. First, neither writes nor reads take place at a single point in time, so the meaning of 'most recent' is not always clear. Each type of access has a well-defined point of issue, but they complete at some later time (for example, after message passing has taken place). Secondly, clocks cannot be synchronized sufficiently accurately in a distributed system to resolve events at this level of granularity. So the relation 'before' (in absolute time) is not a useful one for these events. A model called atomic consistency has been formulated to remove these ambiguities, but it remains impractical for a DSM system; it is described by Mosberger.

Fortunately, it is possible to define useful and practical memory consistency models logically, so that they can be reasoned about and compared with what programmers intuitively expect from memories. The strongest and most important such model is **sequential consistency**. Sequential consistency is similar to one-copy serializability – but it applies to memory systems at the granularity of individual memory operations, rather than transactions. Under it, the result of any execution is equivalent to one in which the following conditions obtain:

- all the reads and writes issued by the individual processes concerned are satisfied in program order - as the programmers of those processes surely expect, and
- the memory operations belonging to different processes occur in some serial order.

The condition can be restated more concretely as follows: no process may read values that conflict with its program order or that of any other participating process. In an actual execution, memory operations may be overlapped and some updates may be ordered differently at different processes, as long as the definition's constraints are not thereby broken. Note that memory operations upon the entire DSM have to be serializable to satisfy the conditions of sequential consistency - and not just the operations on each individual location.

The combination *ar = 0, br =1* in the above example could not occur under sequential consistency, because Process 1 would be reading values that conflict with Process 2's

program order. An example interleaving of the processes' memory accesses is shown in Figure 4-3.

Figure 4-3. *An example serialization under sequential consistency.*

Sequentially consistent DSM could be implemented by using a single server to hold all the shared data, and by making all processes perform reads or writes by sending requests to the server, which globally orders them. This architecture is of course too inefficient for a DSM implementation, and practical means of achieving sequential consistency are described below.

Weak consistency: Dubois et al. [1988] developed the weak consistency model in an attempt to avoid the costs of sequential consistency on multiprocessors. This model exploits knowledge of synchronization operations in order to relax memory consistency, while appearing to the programmer to implement sequential consistency (at least, under certain conditions that are beyond the scope of this book). For example, if the programmer uses a lock to implement a critical section, then a DSM system can assume that no other process may access the data items accessed under mutual exclusion within it. It is therefore redundant for the DSM system to propagate updates to these items until the process leaves the critical section. While items are left with 'inconsistent' values some of the time, they are not accessed at those points; the execution appears to be sequentially consistent. Adve and Hill [1990] describe a generalization of this notion called weak ordering: '(A DSM system) is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all software that obeys the synchronization model'. Release consistency is a form of weak ordering.

Figure 4-4. *DSM using write-update.*

Update options

Two main implementation choices have been devised for propagating updates made by one process to the others: write-update and write-invalidate. These are applicable to a variety of DSM consistency models, including sequential consistency. In outline, the options are as follows:

Write-update: The updates made by a process are made locally and multicast to all other replica managers possessing a copy of the data item, which immediately modify the data read by local processes (Figure 4-4). Processes read the local copies of data items, without the need for communication. In addition to allowing multiple readers, several processes may write the same data item at the same time; this is known *as multiple-reader-multiple-writer sharing.*

The memory consistency model that is implemented with write-update depends on several factors, mainly the multicast ordering property. Sequential consistency can be achieved by using multicasts that are totally ordered, which do not return until the update message has been delivered locally. All processes then agree on the order of updates. The set of reads that take place between any two consecutive updates is well-defined, and their ordering is immaterial to sequential consistency.

Reads are cheap in the write-update option. However, ordered multicast protocols are relatively expensive to implement in software. Orca uses write-update, and employs the Amoeba multicast protocol, which uses hardware support for multicast. Munin supports writeupdate as an option. A write-update protocol is used with specialized hardware support in the PLUS multiprocessor architecture.

Write-invalidate: This is commonly implemented in the form of multiple-readersingle-writer sharing. At any time, a data item may either be accessed in read-only mode by one or more processes, or it may be read and written by a single process. An item that is currently accessed in read-only mode can be copied indefinitely to other processes. When a process attempts to write to it, a multicast message is first sent to all other copies to invalidate them, and this is acknowledged before the write can take place; the other processes are thereby prevented from reading stale data (that is, data that are not up-to-date). Any processes attempting to access the data item are blocked if a writer exists. Eventually, control is transferred from the writing process and other accesses may take place once the update has been sent. The effect is to process all accesses to the item on a first-come-first-served basis. By the proof given by Lamport [1979], this scheme achieves sequential consistency. Invalidations may be delayed under release consistency.

Under the invalidation scheme, updates are only propagated when data are read, and several updates can take place before communication is necessary. Against this must be placed the cost of invalidating read-only copies before a write can occur. In the multiple-reader-single-writer scheme described, this is potentially expensive. But if the read/write ratio is sufficiently high, then the parallelism obtained by allowing multiple simultaneous readers offsets this cost. Where the read/write ratio is relatively small, a single-reader-single-writer scheme can be more appropriate: that is, one in which at most one process may be granted read-only access at a time.

Granularity

An issue that is related to the structure of DSM is the granularity of sharing. Conceptually, all processes share the entire contents of a DSM. As programs sharing DSM execute, however, only certain parts of the data are actually shared, and then only for certain times during the execution. It would clearly be very wasteful for the DSM implementation always to transmit the entire contents of DSM as processes access and update it. What should be the unit of sharing in a DSM implementation? That is, when a process has written to DSM, which data does the DSM run-time send in order to provide consistent values elsewhere?

We focus here on page-based implementations, although the granularity issue does arise in library-based implementations. In a page-based DSM, the hardware supports alterations to an address space efficiently in units of pages – essentially by the placement of a new page frame pointer in the page table. Page sizes can typically range up to 8 kilobytes, so this is an appreciable amount of data that must be transmitted over a network to keep remote copies consistent when an update occurs. By default, the price of the whole page transfer must be paid whether the entire page has been updated, or just one byte of it.

Using a smaller page size - say 512 bytes or 1 kilobyte - does not necessarily lead to an improvement in overall performance. First, in cases where processes do update large amounts of contiguous data, it is better to send one large page rather than several smaller pages in separate updates, because of the fixed software overheads per network packet. Secondly, using a small page as the unit of distribution leads to a large number of units that must be separately administered by the DSM implementation.

Figure 4-5. *Data items laid out over pages.*

To complicate matters further, processes tend to contend more for pages when the page size is large, because the likelihood that the data they access will lie within the same page increases with the page size. Consider, for example, two processes, one of which

accesses only data item *A* and the other accesses only data item *B*, which lie within the same page (Figure 4-5). For the sake of concreteness, let us assume that one process reads *A* and the other updates *B*. There is no contention at the application level. However, the entire page must be transmitted between the processes, since the DSM run-time does not by default know which locations in the page have been altered. This phenomenon is known as *false sharing:* two or more processes share parts of a page, but only one in fact accesses each part. In write-invalidate protocols, false sharing can lead to unnecessary invalidations. In write-update protocols, when several writers falsely share data items they may cause them to be overwritten with older versions.

In practice, the choice of the unit of sharing has to be made based on the physical page sizes available, although a unit of several contiguous pages may be taken if the page size is small. The layout of data with respect to page boundaries is an important factor in determining the number of page transfers made when a program executes.

Thrashing

A potential problem with write-invalidate protocols is thrashing. Thrashing is said to occur where the DSM run-time spends an inordinate amount of time invalidating and transferring shared data, compared to the time spent by application processes doing useful work. It occurs when several processes compete for the same data item, or for falsely shared data items. If, for example, one process repeatedly reads a data item that another is regularly updating, then this item will be constantly transferred from the writer and invalidated at the reader. This is an example of a sharing pattern for which write-invalidate is inappropriate, and write-update would be better.

4.3 SUMMARY

This chapter has described and motivated the concept of distributed shared memory as an abstraction of shared memory that is an alternative to message-based communication in a distributed system. DSM is primarily intended for parallel processing and data sharing. It has been shown to perform as well as message passing for cerrtain parallel applications, but it is difficult to implement efficiently and its performance varies with applications.

The chapter has concentrated on software implementations of DSM − particularly those using the virtual memory subsystem − but it has been implemented with hardware supporrt.

The main design and implementation issues are the DSM structure, the means by which applications synchronize, the memory consistency model, the use of write-update or write-invalidation protocols, the granularity of sharing and thrashing.

The DSM is structured either as a series of bytes, a collection of language-specific data structures, a collection of shared objects, or a collection of immutable data items.

Applications using DSM require synchronization in order to meet applicationspecific consistency constraints. They use objects such as locks for this purpose, implemented using message passing for efficiency.

The most common and strictest type of memory consistency implemented in DSM systems is sequential consistency. An emerging weaker level of consistency is release consistency, which requires integration of the synchronization mechanism with the DSM mechanism.

Write-update protocols are those in which updates are propagated to all copies as data items are updated. These are implemented usually in hardware, although software implementations using totally ordered multicast exist. Write-invalidation protocols prevent

stale data from being read by invalidating copies as data items are updated. These are more suited to page-based DSM, for which write-update may be an expensive option.

The granularity of DSM affects the likelihood of contention between processes that falsely share data items because they are contained in the same unit of sharing (for example, page). It also affects the cost per byte of transferring updates between computers.

Thrashing may occur when write-invalidation is used. This is the repeated transfer of data between competing processes, at the expense of application progress. This may be reduced by application-level synchronization, by allowing computers to retain a page for a minimum time, or by labelling data items so that both read and write access are always granted together.