E.G. Coffman et al., Eds., *Handbooks in OR & MS, Vol. 3*

Chapter 3

Operating Systems – The State of the Art

Andrew S. Tanenbaum

Department of Mathematics and Computer Science, Vrije Universiteit De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

An operating system is a program that controls the resources of a computer and provides the users with an interface that is more attractive than the bare machine. The first operating systems date from the early 1950s. This chapter traces the evolution of operating systems from their inception to the late 1980s and describes the various types. It also discusses one particularly significant system, UNIX®, in some detail. Finally, it concludes with a discussion of current work in network and distributed operating systems.

1. What is an operating system?

Most computer users have had some experience with an operating system, but it is difficult to pin down precisely what an operating system is. Part of the problem is that operating systems perform two basically unrelated functions, and depending on who is doing the talking, you hear mostly about one function or the other. Let us now look at both.

1.1. The operating system as an extended machine

The actual hardware of a modern computer, especially the I/O (Input/Output) hardware, is extremely complex. Consider for a moment the floppy disk on an IBM PC, which is a very unsophisticated device. To program it at the hardware level, the user must be aware of the 16 commands that it accepts. Each of these commands needs parameters. The READ command, for example, requires 13 parameters packed into 9 bytes, which must be passed to the disk at the proper rate, not too fast and not too slow. When the READ is completed, the disk returns 23 different status and error fields packed into 7 bytes.

Without going into the *real* details, it should be clear that the average programmer probably does not want to get too intimately involved with the programming of floppy disks (or any other disks). Instead, what the programmer wants is a simple, high-level abstraction to deal with. In the case of disks, a typical abstraction would be that the disk contains a collection of named files. Each file can be opened, read or written, and then finally closed. Details such as whether or not recording should use modified frequency modulation and what the current state of the motor is should not appear in the abstraction presented to the user.

The program that hides the truth about the hardware from the programmer and presents a nice, simple view of names files that can be read and written is, of course, the operating system. Just as the operating system shields the programmer from the disk hardware and presents a simple file-oriented interface, it also conceals a lot of unpleasant business concerning interrupts, timers, memory management, and other low-level features. In each case, the abstraction presented to the user of the operating system is simpler and easier to use than the underlying hardware.

In this view, the function of the operating system is to present the user with the equivalent of an *extended machine* or *virtual machine* that is easier to program than the underlying hardware.

1.2. The operating system as a resource manager

The concept of the operating system as primarily providing its users with a convenient interface is a top-down view. An alternative, bottom-up, view holds that the operating system is there to manage all the pieces of a complex system. Modern computers consist of processors, memories, timers, disks, terminals, magnetic tape drives, network interfaces, laser printers, and a wide variety of other devices. In this alternative view, the job of the operating system is to provide for an orderly and controlled allocation of the processors, memories, and I/O devices among the various programs competing for them.

Imagine what would happen if three programs running on some computer all tried to print their output simultaneously on the same printer. The first few lines of printout might be from program 1, the next few from program 2, then some from program 3, and so forth. The result would be chaos. The operating system can bring order to the potential chaos by buffering all the output destined for the printer on the disk. When one program is finished, the operating system can then copy its output from the disk file where it has been stored to the printer, while at the same time the order program can continue generating more output, oblivious to the fact that the output is not really going to the printer (yet).

When a computer has multiple users, the need for managing and protecting the memory, I/O devices, and other resources is even more apparent. This need arises because it is frequently necessary for users to share expensive resources such as tape drives and phototypesetters. Economic issues aside, it is also often necessary for users who are working together to share information. In short, this view of the operating system holds that its primary task is to keep track of who is using which resource, to grant resource requests, to account for usage, and to mediate conflicting requests from different programs and users.

2. A brief history of operating systems

Operating systems have been evolving through the years. Since operating systems have historically been closely tied to the architecture of the computers on which they run, we will look at successive generations of computers to see what their operating systems were like.

The first true digital computer was designed by the English mathematician Charles Babbage (1792–1871). Although Babbage spent most of his life and fortune trying to built his *analytical engine*, he never got it working properly because it was a purely mechanical design, and the technology of his day could not produce the wheels, gears, cogs and other mechanical parts to the high precision that he needed. Needless to say, the analytical engine did not have an operating system.

2.1. The first generation (1945–1955): Vacuum tubes and plugboards

After Babbage's unsuccessful efforts, little progress was made in constructing digital computers until World War II. Around the mid-1940s, Howard Aiken at Harvard, John von Neumann at the Institute for Advanced Study in Princeton, J. Presper Eckert and William Mauchley at the University of Pennsylvania, and Konrad Zuse in Germany, among others, all succeeded in building calculating engine using vacuum tubes. These machines were enormous, filling up entire rooms with tens of thousands of vacuum tubes, but were much slower than even the cheapest home computer available today. Since a failure in any one of the thousands of tubes could bring the machine to a grinding halt, these machines were not very reliable. They were exclusively used for numerical calculations, were programmed in (binary) machine language, and had no operating systems.

2.2. The second generation (1955–1965): Transistors and batch systems

The introduction of the transistor in the mid-1950s changed the picture radically. Computers became reliable enough that they could be manufactured and sold to paying customers with the expectation that they would continue to function long enough to get some useful work done. The normal way of programming these machines was for the programmer to first write the program on paper (in FORTRAN or assembly language), then punch it on cards. He would then bring the card deck down to the input room and hand it to one of the operators.

When the computer finished whatever job it was currently running, an operator would take one of the card decks that had been brought from the input room and read it in. If the FORTRAN compiler was needed, the operator would have to get it from a file cabinet and read it in. Much computer time was wasted while operators were walking around the machine room.

Given the high cost of the equipment, it is not surprising that people quickly looked for ways to reduce the wasted time. The solution generally adopted was the *batch system*. The idea behind it was to collect a tray full of jobs in the input room, and then read them onto a magnetic tape using a small, inexpensive computer, such as the IBM 1401.

After about an hour of collecting a batch of jobs, the tape was rewound and brought into the machine room, where it was mounted on a tape drive. The operator then loaded a special program (the ancestor of today's operating system), which read the first job from tape and ran it. The output was written onto a second tape, instead of being printed. After each job finished, the operating system automatically read the next job from the tape and began running it. When the whole batch was done, the operator removed the input and output tapes, replaced the input tape with the next batch, and took the output tape offline for printing.

2.3. The third generation (1965–1980): ICs and multiprogramming

Third generation computers introduced a major advance in their operating systems: *multiprogramming*. On the second generation machines, when the current job paused to wait for a tape or other I/O operation to complete, the CPU simply sat idle until the I/O finished. With heavily CPU-bound scientific calculations, I/O is infrequent, so this wasted time is not significant. With commercial data processing the I/O wait time can often be 80 or 90 percent of the total time, so something had to be done about it.

The solution that evolved was to partition memory into several pieces, with a different job in each partition. While one job was waiting for I/O to complete, another job could be using the CPU. If enough jobs could be held in main memory at once, the CPU could be kept busy nearly 100 percent of the time. Having multiple jobs in memory at once requires special hardware to protect each job against snooping and mischief by the other ones, so third generation systems were equipped with this hardware.

Another major feature present in third-generation operating systems was the ability to read jobs from cards onto the disk as soon as they were brought to the computer room. Then, whenever a running job finished, the operating system could load a new job from the disk into the now-empty part of memory and run it. This technique is called *spooling* (from Simultaneous Peripheral Operation On Line) and was also used for output. With spooling, the 1401s were no longer needed, and much carrying of tapes disappeared.

Although third-generation operating systems were well-suited for big scientific calculations and massive commercial data processing runs, they were still basically batch systems. Many programmers pined for the first generation days when they had the machine all to themselves for a few hours, so they could debug their programs quickly. With third generation systems, the time between submitting a job and getting back the output was often several hours, so a single misplaced comma could cause a compilation to fail, and the programmer to waste half a day.

This desire for quick response time paved the way for *time-sharing*, a variant of multiprogramming, in which each user has an on-line terminal. In a time-sharing system, if 20 users are logged in and 17 of them are thinking or talking or drinking coffee, the CPU can be allocated in turn to the three jobs that want service. Since people debugging programs usually issue short commands (e.g., compile a five-page program) rather than long ones (e.g., sort a million-record tape), the computer can provide fast, interactive service to a number of users and perhaps also work on big batch jobs in the background when the CPU is otherwise idle. The first serious time-sharing system (CTSS) was developed at MIT on a specially modified IBM 7094 [Corbato, Merwin-Dagget & Daley, 1962].

After the success of the CTSS system, MIT, Bell Labs, and General Electric (then a major computer manufacturer) decided to embark on the development of a 'computer utility', a machine that would support hundreds of simultaneous time-sharing users. Their model was the electricity distribution system – when you need electric power, you just stick a plug in the wall, and within reason, as much power as you need will be there. The designers of this system, known as *MULTICS* (MULTiplexed Information and Computing Service), envisioned one huge machine providing computing power for everyone in Boston. The idea that machines as powerful as their GE-645 would be sold as personal computers for a few thousand dollars only 20 years later was pure science fiction at the time.

To make a long story short, MULTICS introduced many seminal ideas into the computer literature, but building it was a lot harder than anyone had expected. Bell Labs dropped out of the project, and General Electric quit the computer business altogether. Eventually, MULTICS ran well enough to be used in a production environment at MIT and a few dozen sites elsewhere, but the concept of a computer utility fizzled out. Still, MULTICS had an enormous influence on subsequent systems. It is described by Corbato & Vyssotsky [1965], Daley & Dennis [1968], Organick [1972] and Saltzer [1974].

Another major development during the third generation was the phenomenal growth of minicomputers, starting with the DEC PDP-1 in 1961. The PDP-1 had only 4K of 18-bit words, but at 120 000 dollars per machine (less than 5 percent of the price of a 7094), they sold like hotcakes. For certain kinds of non-numerical work, it was almost as fast as the 7094, and gave birth to a whole new industry. It was quickly followed by a series of other PDPs (unlike IBM's family, all incompatible) culminating in the PDP-11.

One of the computer scientists at Bell Labs who had worked on the MULTICS project, Ken Thompson, subsequently found a small PDP-7 that no one was using and set out to write a stripped-down, one-user version of MULTICS. Brian Kernighan somewhat jokingly dubbed this system UNICS (UNiplexed Information and Computing Service), but the spelling was later changed to UNIX. It was later moved to a small PDP-11/20, where it worked well enough to convince Bell Labs' management to invest in a larger PDP-11/45 to continue the work.

Another Bell Labs computer scientist, Dennis Ritchie, then teamed up with

Thompson to rewrite the system in a high-level language called C, designed and implemented by Ritchie. Bell Labs licensed UNIX to universities almost for free, and within a few years hundreds of them were using it. It soon spread to the Interdata 7/32, VAX, Motorola 68000, and many other computers. UNIX has been moved ('ported') to more CPU types than any other operating system in history, and its use is still rapidly increasing.

2.4. The fourth generation (1980–1990): Personal computers

With the development of LSI (Large Scale Integration) circuits, chips containing thousands of transistors on a square centimeter of silicon, the age of the personal computer dawned. In terms of architecture, personal computers were not that different from minicomputers of the PDP-11 class, but in terms of price they certainly were different. Where the minicomputer made it possible for a department in a company or university to have its own computer, the microprocessor chip made it possible for a single individual to have his or her own personal computer.

The widespread availability of computing power, especially highly interactive computing power usually with excellent graphics, led to the growth of a major industry producing software for personal computers. Much of this software was *user-friendly*, meaning that it was intended for users who did not know anything about computers, and furthermore had absolutely no intention whatsoever of learning. This was certainly a major change from OS/360, whose job control language, JCL, was so arcane that entire books have been written about it [e.g., Cadow, 1970].

Two operating systems have dominated the personal computer scene: MS-DOS, written by Microsoft, Inc. for the IBM PC and other machines using the Intel 8088 CPU and its successors, and UNIX, which is dominant on the larger personal computers using the Motorola 68000 CPU family. It is perhaps ironic that the direct descendant of MULTICS, designed for a gigantic computer utility, has become so popular on personal computers, but mostly it shows how well thought out the basic ideas in MULTICS and UNIX were. Although the initial version of MS-DOS was relatively primitive, subsequent versions have included more and more features from UNIX, which is not entirely surprising given that Microsoft was a major UNIX supplier, using the trade name of XENIX®.

An interesting development that began taking place during the mid-1980s is the growth of networks of personal computers running *network operating systems* and *distributed operating systems*. In a network operating system, the users are aware of the existence of multiple computers, and can log in to remote machines and copy files from one machine to another. Each machine runs its own local operating system and has its own user (or users).

A distributed operating system, in contrast, is one that appears to its users as a traditional uniprocessor system, even though it is actually composed of multiple processors. In a true distributed system, users should not be aware of where their programs are being run or where their files are located; that should all be handled automatically and efficiently by the operating system.

Network operating systems are not fundamentally different from single-processor operating systems. They obviously need a network interface controller and some low-level software to drive it, as well as programs to achieve remote login and remote file access, but these additions do not change the essential structure of the operating system.

True distributed operating systems require more than just adding a little code to a uniprocessor operating system, because distributed and centralized systems differ in critical ways. Distributed systems, for example, often allow programs to run on several processors at the same time, thus requiring more complex processor scheduling algorithms in order to optimize the amount of parallelism achieved.

Communication delays within the network often mean that these (and other) algorithms must run with incomplete, outdated, or even incorrect information. This situation is radically different from a single-processor system in which the operating system has complete information about the system state.

Fault-tolerance is another area in which distributed systems are different. It is common for a distributed system to be designed with the expectation that it will continue running, even if part of the hardware is currently broken. Needless to say, such an additional design requirement has enormous implications for the operating system.

3. An example operating system – UNIX

In this section we will discuss one operating system, as an example of the current state-of-the-art. The system we have chosen is the UNIX operating system [Ritchie & Thompson, 1974; Kernighan & Mashey, 1979]. The UNIX system was developed by Ken Thompson and Dennis Ritchie at AT & T Bell Laboratories in the 1970s and has since been developed further and has spread to a tremendous number of computers ranging from personal computers (e.g., IBM PC) to supercomputers (Cray-2) as mentioned above. It is the de facto standard on scientific workstations and in nearly all university computer science departments, and is making inroads in the microcomputer world.

3.1. The UNIX user interface

To the user at a terminal, the two primary aspects of UNIX are the file system and the shell. We will now look at each of these in turn. Information in UNIX is stored in *files*. Each file consists of a sequence of bytes, from 0 up to the last byte. Files can be of any length of the size of the device. The operating system does not distinguish between different kinds of files such as Pascal programs, binary programs, English text, and so on. As far as it is concerned, a file is just a sequence of bytes.

Much of the power of UNIX comes from this model of a file. Users do not have to think in terms of cylinders and heads or other physical device characteristics. There are no physical or logical records, and no blocking factors. No distinction is made between sequential files, random access files, or files with various access methods. There are no file types. Finally, there are no buffers or other file-related data structures in user programs. All of these points may seem obvious, but UNIX is almost the only operating system around that does not burden the user with any of these items.

Files can be grouped into *directories*. Each directory can hold as many or as few files as needed, and may also contain subdirectories, which themselves may also contain files and subdirectories. Thus the file system as a whole takes the form of a tree, with the *root directory* at the top. By convention, the root directory contains subdirectories

bin	– for binary (executable) programs
lib	– for libraries
dev	– for special files (see below)
tmp	– for temporary (scratch) files
usr	– for user directories

Within the *usr* directory, there is one subdirectory for each user, known as the user's *home directory*. Users often have subdirectories for various projects in their home directory. File names can be specified by giving their *absolute path name*. For example, the path */usr/jim/research/report.4* references a file *report.4* in a directory *research* which is itself in a directory *jim*, which is located in the top-level directory *usr*. The initial / indicates that the path is absolute (i.e., starting in the root directory).

It is possible for a user to issue a command to designate a given directory as the *working directory*. Path names that do not begin with a / are relative to the working directory. Thus if */user/jim/research* is the working directory, the path *report.4* is equivalent to */user/jim/research/report.4*. In practice, it is common to go to a working directory at the start of a login session, and stay there for a substantial period, so most path names can be short relative names, rather than long absolute names.

UNIX allows both files and directories to be protected. A user can set the protection for a file or directory so that only the owner can read and/or write it, so that members of the owner's group (e.g., project team or department, depending on the installation) can read and/or write it, or so that everyone on the system has access. For example, it is easy to the protect a file so that the owner can both read and write (i.e., modify) it, members of the owner's group can read it but not write it, and no one else has any access at all.

In addition to the regular files described above, UNIX also supports *special files*. These files can be used to access I/O devices. For example, there may be a file */dev/disk* that represents the disk, and another file */dev/tty* that represents the terminal. Assuming the protection system allows it, these files can be used to real and write raw disk blocks and the terminal, respectively. Allowing I/O devices to be accessed the same way as files makes it easy to implement real-time and other applications that need direct access to I/O devices.

The other key feature of the user interface is the command interpreter, called the *shell*. Command lines typed by the user on the terminal are analyzed and executed by the shell. Some commands give output, and others do their work quietly. For example, the command

```
date
```

might result in

```
Fri Sep 4 17:16:09 EDT 1988
```

On the other hand, the command

```
cp file1 file2
```

just copies *file1* to *file2* (i.e., it creates a new file called *file2* and gives it the same contents as *file1*).

Some commands require multiple file names, such as *cp* above. The shell permits certain kinds of abbreviations to reduce the amount of typing needed. For example, it is customary to have files that contain C programs have the suffix *.c* as in *prog1.c* and *prog2.c*. To get a listing of all the C programs in the current directory, one can type

```
ls *.c
```

The * matches all strings, and thus lists all files ending in *.c*. What actually happens here is that the shell reads the working directory before calling the *ls* program, finds all the file names ending in *.c* and expands the **.c* in the command line. The command that actually gets executed might be something like

```
ls prog1.c prog2.c
```

The * is not the only 'magic' character. The ? character matches any single character, and [b-h] matches any character between 'b' and 'h' inclusive. Other abbreviations of this type are available, and are commonly used.

The standard output of most programs is written to the terminal. However, in some cases, the user would like to put it on a file, to be saved for later use. This can easily be done by *redirecting standard output*. The command

```
ls >files
```

runs the command *ls*, and puts the output, the list of all files in the working directory, in the file *files* where it can be examined later, printed, edited, and so on. Similarly, commands that normally take their input from the terminal can also have that redirected. For example, the editor, *ed*, normally takes its commands from the terminal, but the command

```
ed <script
```

causes it to read from the file *script* instead. This feature, known as *redirecting standard input* might be useful when a series of files has to be edited using the same editor commands.

The ability to redirect standard input and standard output can be utilized to connect programs together. Consider, for example, the program *prep* that reads a document and outputs each word on a separate line, eliminating all punctuation. To get an alphabetical list of all the words in the document, one could type:

```
prep <document >temp
sort <temp >output
rm temp
```

which first runs *prep*, putting the output on a temporary file, then runs *sort* to sort the words and write the output on a file *output*. Finally, the temporary file is removed. Alternatively we could type:

```
prep <document | sort >output
```

The '|' symbol, called a *pipe*, arranges for the output of the first program to be fed into the second one without using a temporary file. If we now wanted to remove duplicates from the list, we could use the program *uniq*:

```
prep <document | sort | uniq >output
```

Programs that take their input from standard input and write their output on standard output are called *filters*. By combining filters with pipes, it is often possible to get a job done without having to write a program. Numerous filters that can be used as building blocks are part of UNIX.

Normally, when a command is given from the terminal, the user has to wait until it completes before proceeding. However, it is also possible to start a job up in the background by typing an ampersand (&) after the command. For example,

```
pc prog.p &
```

starts up the Pascal compiler as a background job. The user can continue issuing normal commands from the terminal. The only effect of the background job will be some loss of performance due to the CPU cycles used by the compiler.

Another interesting feature of the shell is its ability to take commands from a file instead of a terminal. In fact, the shell is actually a small programming language. For example, consider a file, *words*, containing one line:

```
prep <$1 | sort | uniq >output
```

When the user types:

```
words doc3
```

the shell substitutes *doc3* for the first (and in this case, only) formal parameter, $1, and carries out the command:

```
prep <doc3 | sort | uniq >output
```

Files containing shell programs are called *shell scripts*. A more complex example is the shell script

```
for i in $*
do
  echo Results of $i
  prep <$i | sort | uniq
done
```

in which $* is replaced by all the parameters and the variable *i* iterates through them, one per iteration. The command *echo* just outputs its parameters to standard output. Thus if the above shell script were called *multiwords*, the command

```
multiwords doc1 doc2 doc3
```

would be equivalent to typing

```
echo Results of doc1
prep <doc1 | sort | uniq
echo Results of doc2
prep <doc2 | sort | uniq
echo Results of doc3
prep <doc3 | sort | uniq
```

The output of *multiwords* can, of course, also be redirected or used as input to a pipe. The shell also has variables, *while* loops and many other features that make it extremely powerful and easy to use. It is frequently possible to put together a little shell script to get a job done instead of spending a lot of time writing a program.

3.2. The UNIX *implementation*

Let us now turn from the user interface to take a quick look at how UNIX is implemented internally. Associated with each file is a small data structure called an *i-node*. The i-node tells who owns the file, how big it is, who may access it, and gives a list of the disk blocks that contain the data. All the i-nodes for all the files on a disk are located in consecutive blocks at the start of the disk. They are numbered from 1 up to some maximum. Given an i-node number, the system can find all the information about the file, both the administrative information and the data themselves.

Another key data structure is the *directory*. A directory is just a file containing (name, i-node number)-pairs. The system keeps track of the working directory by remembering its i-node number. When a user wants to read a file 'data', for example, the system locates the working directory using its i-node number, finds the blocks, and begins reading the entries until it finds one would name is 'data'. At that point it has the i-node number and can read in the i-node, to see if the user is permitted to access the file, where the blocks are, and so on. This scheme makes it possible for two or more users working together to share a file: they just make sure that each one has a (name, i-node number)-entry in one of their directories. Although there may be multiple directory entries for a file, there is always only one copy of the file itself.

The second major concept in the UNIX implementation is that of a *process*. A process is a program in execution. When the user types in a command, the shell (which is itself a process), creates a new process, and loads the program to be executed in the process. Normally, the shell waits until this new process finishes before reading the next command line from the terminal. However, if the previous command contained the ampersand symbol, the shell does not wait for termination. It reads and handles the next command immediately. This is how background jobs are implemented.

When a command line with multiple pipes is typed, the shell creates a separate process for each program to be executed. It then arranges for the standard output of the first one to refer to a kind of dummy file, and the standard input of the second one to refer to the same dummy file, and so on. These dummy files are handled internally in a more efficient way than ordinary scratch files.

The shell is not the only process that creates other processes. Any process that wants to can create new processes. In fact, when the system is brought up, a program called *init* is started. This program looks at a configuration file to see how many terminals the system has, and creates one process for each terminal. This process initially runs the *login* program, which asks for a name and password, and if successful, starts up a shell for that terminal. Thus all the processes in the whole system form a single tree, with the *init* process at the root.

3.3. Virtual memory

Many operating systems, including most versions of UNIX that run on minicomputers and mainframes have a feature called *virtual memory*, that allows users to execute programs larger than the amount of memory the computer actually has. The basic idea behind virtual memory is that each program has a certain amount of address space that it can reference. On a computer with 32-bit addresses, this address space would normally be 32-bits wide, allowing for over 4 billion addresses. Programs can use this address space for program text or data.

Of course a problem exists if the physical memory is much smaller than the allowed address space. The problem is typically solved by a technique called *paging*. Both the address space and the physical memory are broken up into fixed-size chunks called *pages*. Page sizes are always powers of two, typically 1K, 2K, 4K, or 8K bytes.

The computer's hardware maintains a mapping between address space pages and memory pages, but the mapping is not 1-to-1. For example, the first 4 pages of the address space might be mapped onto physical memory pages 7, 19, 2, and 5, respectively. With 1K pages, all memory references to addresses between 0K and 1K would automatically be mapped onto addresses between 7K and 8K. Similarly, references to addresses between 1K and 2K would be mapped onto 19K to 20K. Some address space pages are not mapped at all onto physical memory (because there are not enough physical memory pages). When an address on one of these pages is used (e.g., a jump is made to program text on it, or an attempt is made to read data on it), the hardware causes a trap to the operating system. The operating system then removes some other page from memory, brings in the needed page, changes the hardware page map, and restarts the instruction that failed.

A considerable amount of research has gone into discovering efficient algorithms for managing virtual memory systems, especially finding algorithms for choosing the page to remove [Denning, 1970]. An obvious (but unimplementable) strategy is to evict the page that will be referenced farthest in the future of all the pages currently in memory. Many algorithms have been devised that attempt to approximate this strategy. For example, one popular algorithm evicts the page least recently used, on the grounds that if it has not been used in a long time, it probably will not be used for a long time in the future, if ever again. Another one chooses the page that was loaded into memory before any of the other pages, on the grounds that it has been around long enough and it is time to give some other page a change. Many more complex algorithms have also been discussed in the literature. See any of the books cited in Section 6 for a fuller discussion of virtual memory.

4. Network operating systems

We have now finished our discussion of the UNIX system. As technology has progressed and computers have gotten cheaper, many organizations have become interested in connecting multiple computers together using local-area or wide-area networks. There are several reasons why this trend has occurred. In the first place, the price/performance ratio of smaller machines is much better than larger ones. Thus hooking up a number of smaller computers into one large system often given a more cost effective system. In the second place, a system with multiple processors is more redundant. If one of them fails, the others can continue. In the third place, many applications are inherently distributed and fit well with the model of multiple processors.

We can distinguish two kinds of operating systems for these multiple computer systems: *network operating systems* and *distributed operating systems*. In the former case, the systems are weakly coupled. Each machine retains its own identity and users. Each user logs into one specific machine, and to run programs or access files on another one requires explicitly accessing the other machine. In the latter case, all the computing resources act like a big pool. Users are generally not aware of where their computations or files are located. Decisions about where to put what are made by the operating system. In this section we will discuss network operating systems, quite a few of which are currently in operation. In the next section we will look at research into distributed operating systems.

The key feature that distinguishes a network operating system from a distributed operating system is the transparency – to what extent are the users aware that multiple machines are involved. If the users are clearly aware of the existence of multiple machines, and have to do remote login, file transfer, etc. by themselves, it is a network operating system. If these things all happen automatically, it is a distributed operating system. The visibility occurs in three primary areas: the file system, protection, and execution location. We will now look at each of these issues in turn.

4.1. File system

When connecting two or more distinct systems together, the first issue that must be faced is how to merge the file systems. Three approaches have been tried. The first approach is not to merge them at all. Going this route means that a program on machine *A* cannot access files on machine *B* by making system calls. Instead, the user must run a special file transfer program that copies the needed remote files to the local machine, where they can then be accessed normally. Sometimes remote printing and mail is also handled this way. One of the best-known examples of networks that primarily support file transfer and mail via special programs, and not system call access to remote files is the UNIX 'uucp' program, and its network, UUCPNET.

The next step upward in the direction of a distributed file system is to have *adjoining file systems*. In this approach, programs on one machine can open files on another machine by providing a path name telling where the file is located. For example, one could say

```
open("/machine1/pathname",READ_ONLY);
open("machine1!pathname",READ_ONLY); or
open("/../machine1/pathname",READ_ONLY)
```

The latter naming scheme is used in the Newcastle Connection [Brownbridge, Marshall & Randell, 1992] and Netix [Wambecq, 1983] and is derived from the creation of a virtual 'superdirectory' above the root directories of all the connected machines. Thus '/..' means start at the local root directory and go upwards one level (to the superdirectory), and then down to the root directory of *machine*. To access file *x* on machine *C*, one might say

```
open("/../C/x", READ_ONLY)
```

In the Newcastle system, the naming tree is actually more general, since 'machine1' may really be any directory, so one can attach a machine as a leaf anywhere in the hierarchy, not just at the top.

The third approach is the way it is done in distributed operating systems, namely to have a single global file system visible from all machines. When this method is used, there is one 'bin' directory for binary programs, one password file, and so on. When a program wants to read the password file it does something like

```
open("/etc/passwd",READ_ONLY)
```

without reference to where the file is. It is up to the operating system to locate the file and arrange for transport of data as it is needed. LOCUS is an example of a system using this approach [Popek, Walker, Chow, Edwards, Kline, Rudisin & Thiel, 1981; Walker Popek, English, Kline & Thiel, 1983; Weinstein, Page, Livesey & Popek, 1985].

The convenience of having a single global name space is obvious. In addition, this approach means that the operating system is free to move files around between machines to keep all the disks equally full and busy, and that the system can maintain replicated copies of files if it so chooses. When the user or program must specify the machine name, the system cannot decide on its own to move a file to a new machine because that would change the (user visible) name used to access the file. Thus in a network operating system, control over file placement must be done manually by the users, whereas in a distributed operating system it can be done automatically, by the system itself.

4.2. Protection

Closely related to the transparency of the file system is the issue of protection. UNIX, and many other operating systems, assign a unique internal identifier to each user. Each file in the file system has a little table associated with it (the i-node in UNIX), telling who the owner is, where the disk blocks are located, etc. If two previously independent machines are now connected, it may turn out that some internal User IDentifier (UID), e.g., number 12, has been assigned to a different user on each machine. Consequently, when user 12 tries to access a remote file, the remote file system cannot see whether the access is permitted, since two different users have the same UID.

One solution to this problem is to require all remote users wanting to access files on machine *C* to first log onto *C* using a user name that is local to *C*. When used this way, the network is just being used as a fancy switch to allow users at any terminal to log onto any computer, just as a telephone company switching center allows any subscriber to call any other subscriber.

This solution is usually inconvenient for people and impractical for programs, so something better is needed. The next step up is to allow any user to access files on any machine without having to log in, but to have the remote user appear to have the UID corresponding to 'GUEST' or 'DEMO' or some other publicly known login name. Generally such names have little authority, and can only access files that have been designated as readable or writable by all users.

A better approach is to have the operating system provide a mapping between UIDs, so when a user with UID 12 on his home machine accesses a remote machine on which his UID is 15, the remote machine treats all accesses as though they were done by user 15. This approach implies that sufficient tables are provided to map each user from his home (machine, UID)-pair to the appropriate UID for any other machine (and that messages cannot be tampered with).

In a true distributed system, there should be a unique UID for every user, and that UID should be valid on all machines without any mapping. In this way no protection problems arise on remote accesses to files; as far as protection goes, a remote access can be treated like a local access with the same UID. The protection issue makes the difference between a network operating system and a distributed one clear: in one case there are various machines, each with its own user-to-UID mapping, and in the other there is a single, system-wide mapping that is valid everywhere.

4.3. Execution location

Program execution is the third area in which machine boundaries are visible in network operating systems. When a user or a running program wants to create a new process, where is the process created? At least four schemes have been used so far. The first of these is that the user simply says 'CREATE PROCESS' in one way or another, and specifies nothing about where. Depending on the implementation, this can be the best way or the worst way to do it. In the most distributed case, the system chooses a CPU by looking at the load, location of files to be used, etc. In the least distributed case, the system always runs the process on one specific machine (usually the machine on which the user is logged in).

The second approach to process location is to allow users to run jobs on any machine by first logging in there. In this model, processes on different machines cannot communicate or exchange data, but a simple manual load balancing is possible.

The third approach is to use a special command that the user types at a terminal to cause a program to be executed on a specific machine. A typical command might be

```
remote vax4 who
```

to run the *who* program on machine *vax4*. In this arrangement, the environment of the new process is the remote machine. In other words, if that process tries to read or write files from its current working directory, it will discover that its working directory is on the remote machine, and files that were in the parent process' directory are no longer present. Similarly, files written in the working directory will appear on the remote machine, not the local one.

The fourth approach is to provide the 'CREATE PROCESS' system call with a parameter specifying where to run the new process, possibly with a new system call for specifying the default site. As with the previous method, the environment will generally be the remote machine. In many cases, signals and other forms of interprocess communication between processes do not work properly between processes on different machines.

4.4. The OSI model

Network operating systems communicate by sending messages over a network. The content, format, and rules by which these messages are sent is called the network *protocol*. When computer networks were first established, each manufacturer had its own protocols, with the result that machines from one company could not talk to those of another. To alleviate this situation, a model was developed in which international standards could take the place of proprietary standards. In this section we will give a brief overview of this model.

The model was developed by ISO and is known as *OSI*, for *Open Systems Interconnection*. It consists of 7 layers (layer 7 on top, layer 1 on the bottom):

7. Application layer
6. Presentation layer
5. Session Layer
4. Transport Layer
3. Network Layer
2. Data Link Layer
1. Physical Layer

Each layer has a well-defined function and its own protocols. The idea is to start with the base network, and enhance its services by adding the physical layer to it. Then the data link layer is added, to enhance the services more, and so on. In the following paragraphs we will briefly summarize the function of each layer.

The *physical layer* is concerned with transmitting raw bits over a communication channel. The design issues have to do with making such that when one side sends a 1 bit, it is received by the other side as a 1 bit, not as a 0 bit. Typical questions here are how many volts should be used to represent a 1 and how many for a 0, how many microseconds a bit occupies, whether transmission may proceed simultaneously in both directions, how the initial connection is established and how it is torn down when both sides are finished, how many pins the network connector has and what each pin is used for. In some cases a transmission facility consists of multiple physical channels, in which case the physical layer can make them look like a single channel, although higher layers can also perform this function. The design issues here largely deal with mechanical, electrical, and procedural interfacing to the subnet.

The task of the *data link layer* is to take a raw transmission facility and transform it into a line that appears free of transmission errors to the network layer. It accomplishes this task done by breaking the input data up into *data frames*, transmitting the frames sequentially, and processing the *acknowledgement frames* sent back by the receiver. Since layer 1 merely accepts and transmits a stream of bits without any regard to meaning or structure, it is up to the data link layer to create and recognize frame boundaries. This can be accomplished by attaching special bit patterns to the beginning and end of the frame. These bit patterns can accidentally occur in the data, so special care must be taken to avoid confusion.

A noise burst on the line can destroy a frame completely. In this case, the layer 2 software on the source machine must retransmit the frame. However, multiple transmissions of the same frame introduce the possibility of duplicate frames. A duplicate frame could be sent, for example, if the acknowledgement frame from the receiver back to the sender was destroyed. It is up to this layer to solve the problems caused by damaged, lost, and duplicate frames, so that layer 3 can assume it is working with an error-free (virtual) line. Layer 2 may offer several different services classes to layer 3, each of a different quality and with a different price.

Another issue that arises at layer 2 (and at most of the higher layers as well) is how to keep a fast transmitter from drowning a slow receiver in data. Some mechanism must be employed to let the transmitter know how much buffer space the receiver has at the moment. Typically, this mechanism and the error handling are integrated together.

The *network layer* is concerned with controlling the operation of the network. A key design issue is how packet routes are determined. Routes could be based on static tables that are 'wired into' the network and rarely changed. They could also be determined at the start of each conversation, for example, a terminal session. Finally, they could be highly dynamic, being determined anew for each packet, to reflect the current network load.

If too many packets are present in the network at the same time, they will get in each others' way, forming bottlenecks. The control of such congestion also belongs to layer 3.

Since the operators of the network may well expect remuneration for their efforts, there is often some accounting function built into layer 3. At the very least, the software must count how many packets or characters or bits are sent by each customer, to produce billing information. When a packet crosses a national border, with different rates on each side, the accounting can become complicated.

When a packet has to travel from one network to another to get to its destination, many problems can arise. The addressing used by the second network may be different from the first one. The second one may not accept the packet at all because it is too large. The protocols may differ, and so on. It is up to the network layer to overcome all these problems to allow heterogeneous networks to be connected together.

The basic function of the *transport layer*, is to accept data from the session layer, split it up into smaller units if need be, pass these to the network layer, and ensure that the pieces all arrive correctly at the other end. Furthermore, all this must be done in the most efficient possible way, and in a way that isolates the session layer from the inevitable changes in the hardware technology.

Under normal conditions, the transport layer creates a distinct network (i.e., layer 3) connection for each transport (i.e., layer 4) connection required by the session layer. However, if the transport connection requires a high throughput, the transport layer might create multiple network connections, dividing the data among the network connections to improve throughput. On the other hand, if creating or maintaining a network connection is expensive, the transport layer might multiplex several transport connections onto the same network connection, to reduce the cost. In all cases, the transport layer is required to make the multiplexing transparent to the session layer.

The transport layer also determines what type of service to provide the session layer, and ultimately, the users of the network. The most popular type of transport connection is an error-free (virtual) point-to-point channel that delivers messages in the order in which they were sent. However, other possible kinds of transport service are transport of isolated messages with no guarantee about the order of delivery, and broadcasting of messages to multiple destinations. The type of service is determined when the connection is established.

The *session layer* allows users on different machines to establish *sessions* between them. A session allows ordinary data transport, as does the transport layer, but it also provides some enhanced services useful in a few applications. A session might be used to allow a user to log into a remote time-sharing system or to transfer a file between two machines.

One of the services of the session layer is to manage dialog control. Sessions can allow traffic to go in both directions at the same time, or in only one direction at a time. If traffic can only go one way at a time (analogous to a single railroad track), the session layer can help keep track of whose turn it is.

A related session service is *token management*. For certain protocols, it is essential that both sides do not attempt the same operation at the same time. To manage these activities, the session layer provides tokens that can be exchanged. Only the side holding the token may perform the critical operation.

Another session service is *synchronization*. Consider the problems that might occur when trying to do a 4 hour file transfer between two machines on a network with a 1 hour mean time between crashes. After each transfer was aborted, the whole transfer would have to start over again, and would probably fail again when the network next crashed. To eliminate this problem, the session layer can built in checkpoints, so that after a crash, only the data after the last checkpoint has to be repeated.

Another service is *quarantine service*, which has to do with telling the session layer on the receiving side not to deliver any data to the user process until all of it has arrived. The feature is useful to make sure that no work is started until it is known for sure that all the data has already arrived safely.

The *presentation layer* performs certain functions that are requested sufficiently often to warrant finding a general solution for them, rather than letting each user solve the problems. In particular, unlike all the lower layers, which are just interested in moving bits reliably from here to there, the presentation layer is concerned with the syntax and semantics of the information transmitted. These functions can often be performed by library routines called by the user.

A typical example of a presentation service is encoding data in a standard way that all machines can understand. Most user programs do not exchange random binary bit strings. They exchange things such as people's names, dates, amounts of money, and invoices. These items are represented as character strings, integers, floating point numbers, and structured values composed of several simpler items. Different computers have different codes for representing character strings (e.g., ASCII and EBCDIC), different systems for representing integers (e.g., one's complement and two's complement), and so on. In order to make it possible for computers with different representations to communicate, various conversions must take place. These conversions are done in the presentation layer.

The *application layer* contains a variety of protocols that are commonly needed. For example, there are hundreds of incompatible terminal types in the world. Consider the plight of a full screen editor that is supposed to work with many different terminal types, each with different screen layouts, escape sequences for inserting and deleting text, moving the cursor, and so on.

One way to solve this problem is to have a standard *network virtual terminal* that editors and other programs can be written to deal with. To attach a real terminal to the network, a piece of software must be written to map the functions of the network virtual terminal onto the real terminal. For example, when the editor moves the virtual terminal's cursor to the upper left-hand corner of the screen, this software must issue the proper command sequence to the real terminal to get its cursor there too. All the virtual terminal software is in the application layer.

Another application layer function is file transfer. Different file systems have different file naming conventions, different ways of representing text lines, and so on. Transferring a file between two different systems requires handling these and other incompatibilities. This work, too, belongs to the application layer, as do remote job entry and various other general-purpose and special-purpose facilities.

5. Distributed operating systems

Network operating systems are a step forward from single user systems. They allow users to harness the power of multiple machines instead of just one. However, they have the disadvantage of burdening the users with knowing about the details of what is located where. It would be far better to have the entire collection of machines act to the users as if it was a single machine, a *virtual uniprocessor.*

Needless to say, arranging for the illusion that there is really only one computer, when in fact there are many, is not easy. This difficult task is the job of the distributed operating system. A great deal of research in this area is presently in progress. In this section we will look at some of the principal design issues involved in the design of distributed operating systems. In this section we will look at five issues that distributed systems' designers are faced with:
- communication primitives,
- naming and protection,
- resource management,
- fault tolerance,
- services to provide.

While no list could possibly be exhaustive at this early stage of development, these topics should provide a reasonable impression of the areas in which current research is proceeding.

5.1. Communication primitives

The computers forming a distributed system normally do not share primary memory, so communication via shared memory techniques such as semaphores and monitors are generally not applicable. Instead, message passing in one form or another is used. One widely discussed framework for message-passing systems is the OSI reference model, discussed above. Unfortunately, the overhead created by all these layers is substantial. In a distributed systems consisting primarily of huge mainframes from different manufacturers, connected by slow leased lines (say, 56 kbps), the overhead might be tolerable.

Plenty of computing capacity would be available for running complex protocols, and the narrow bandwidth means that close coupling between the systems would be impossible anyway. On the other hand, in a distributed system consisting of identical microcomputers connected by a 10 Mbps or faster local network, the price of the OSI model is generally too high. Nearly all the experimental distributed systems discussed in the literature so far have opted for a different, much simpler model, so we will not mention the OSI model further in this paper.

5.1.1. Message passing

The model that is favored by researchers in this area is the *client-server model*, in which a client process wanting some service (e.g., reading some data from a file) sends a message to the server and then waits for a reply message. In the most naked form, the system just provides two primitives: SEND and RECEIVE. The SEND primitive specifies the destination and provides a message; the RECEIVE primitive tells from whom a message is desired (including 'anyone') and provides a buffer where the incoming message is to be stored. No initial setup is required, and no connection is established, hence no teardown is required.

Precisely what semantics these primitives ought to have has been a subject of much controversy among researchers. Two of the fundamental decision that must be made are unreliable vs. reliable and nonblocking vs. blocking primitives. At one extreme, SEND can put a message out onto the network and wish it good luck. No guarantee of delivery is provided, and no automatic retransmission is attempted by the system if the message is lost. At the other extreme, SEND can handle lost messages, retransmissions, and acknowledgements internally, so that when SEND terminates, the program is sure that the message has been received and acknowledged.

Blocking vs. nonblocking primitives. The other choice is between nonblocking and blocking primitives. With nonblocking primitives, SEND returns control to the user program as soon as the message has been queued for subsequent transmission (or a copy made). If no copy is made, any changes the program makes to the data before or (heaven forbid) *while* it is being sent, are made at the program's peril. When the message has been transmitted (or copied to a safe place for subsequent transmission), the program is interrupted to inform it that the buffer may be reused. The corresponding RECEIVE primitive signals a willingness to receive a message, and provides a buffer for it to be put into. When a message has arrived, the program is informed by interrupt or it can poll for status continuously, or go to sleep until the interrupt arrives. The advantage of these nonblocking primitives is that they provide the maximum flexibility: programs can compute and perform message I/O in parallel any way they want to.

Nonblocking primitives also have a disadvantage: they make programming tricky and difficult. Irreproducible, timing-dependent programs are painful to write and awful to debug. Consequently, many people advocate sacrificing some flexibility and efficiency by using blocking primitives. A blocking SEND does not return control to the user until the message has been sent (unreliable blocking primitive) or until the message has been sent and an acknowledgement received (reliable blocking primitive). Either way, the program may immediately modify the buffer without danger. A blocking RECEIVE does not return control until a message has been placed in the buffer. Reliable and unreliable RECEIVEs differ in that the former automatically acknowledges receipt of message, whereas the latter does not. It is not reasonable to combine a reliable SEND with an unreliable RECEIVE or vice versa, so the system designers must make a choice and provide one set or the other. Blocking and nonblocking primitives do not conflict, so there is no harm done if the sender uses one and the receiver the other.

Buffered vs. unbuffered primitives. Another design decision that must be made is whether or not to buffer messages. The simplest strategy is not to buffer. When a sender has a message for a receiver that has not (yet) executed a RECEIVE primitive, the sender is blocked until a RECEIVE has been done, at which time the message is copied from sender to receiver. This strategy is sometimes referred to as a *rendezvous*.

A slight variation on this theme is to copy the message to an internal buffer on the sender's machine, thus providing for a nonblocking version of the same scheme. As long as the sender does not do any more SENDs before the RECEIVE occurs, no problem occurs.

A more general solution is to have a buffering mechanism, usually in the operating system kernel, which allows senders to have multiple SENDs outstanding even without any interest on the part of the receiver. Although buffered message passing can be implemented in many ways, a typical approach is to provide users with a system call CREATEBUF, which creates a kernel buffer, sometimes called a *mail-box*, of a user-specified size. To communicate, a sender can now send messages to the receiver's mailbox, where they will be buffered until requested by the receiver. Buffering is not only more complex (creating, destroying, and generally managing the mailboxes), but also raises issues of protection, the need for special high-priority interrupt messages, what to do with mailboxes owned by processes that have been killed or died of natural cases, and more.

A more structured form of communication is achieved by distinguishing requests from replies. With this approach, one typically has three primitives: SEND_GET, GET_REQUEST, and SEND_REPLY. SEND_GET is used by clients to send requests and get replies. It combines a SEND to a server with a RECEIVE to get the server's reply. GET_REQUEST is done by servers to acquire messages containing work for them to do. When a server has carried out the work, it sends a reply with SEND_REPLY. By thus restricting the message traffic, and by using reliable, blocking primitives, one can create some order in the chaos.

5.1.2. Remote procedure call

The next step forward in message-passing systems is the realization that the model of 'client sends request and blocks until server sends reply' looks very similar to a traditional procedure call from the client to the server. This model has become known in the literature as *remote procedure call* and has been widely discussed [Birrell & Nelson, 1984; Nelson, 1981; Spector, 1982]. The idea is to make the semantics of intermachine communication as similar as possible to normal procedure calls because the latter is familiar, well understood, and has proved its worth over the years as a tool for dealing with abstraction. It can be viewed as a refinement of the reliable, blocking SEND_GET, GET_REQUEST SEND_REPLY primitives, with a more user-friendly syntax.

The remote procedure call can be organized as follows. The client (calling program) makes a normal procedure call, say $p(x, y)$, on its machine, with the intention of invoking the remote procedure p on some other machine. A dummy or *stub* procedure p must be included in the caller's address space, or at least be dynamically linked to it upon call. This procedure, which may be automatically generated by the compiler, collects the parameters and packs them into a message in a standard format. It then sends the message to the remote machine (using SEND_GET) and blocks, waiting for an answer.

At the remote machine, another stub procedure should be waiting for a message using GET_REQUEST. When a message comes in, the parameters are unpacked by an input handling procedure, which then makes the local call $p(x, y)$. The remote procedure p is thus called locally, so its normal assumptions about where to find parameters, the state of the stack, etc., are identical to the case of a purely local call. The only procedures that know that the call is remote are the stubs, which build and send the message on the client side and disassemble and make the call on the server size. The result of the procedure call follows an analogous path in the reverse direction.

5.1.3. Error handling

In error handling, the communication primitives of distributed systems differ radically from those of centralized systems. In a centralized system, a system crash means that the client, server, and communication channel are all completely destroyed, and no attempt is made to revive them. In a distributed system, matters are more complex. If a client has initiated a remote procedure call with a server that has crashed, the client may just be left hanging forever unless a timeout is built in. However, such a timeout introduces race conditions in the form of clients that time out too quickly, thinking that the server is down, when in fact, it is merely very slow.

Client crashes can also cause trouble for servers. Consider for example, the case of processes A and B communicating via the UNIX pipe model $A \mid B$ with A the server and B the client. B asks A for data and gets a reply, but unless that reply is acknowledged somehow, A does not know when it can safely discard data that it may not be able to reproduce. If *B* crashes, how long should *A* hold onto the data? (Hint: if the answer is less than infinity, problems will be introduced whenever *B* is slow in sending an acknowledgement.)

Closely related to this is the problem of what happens if a client cannot tell when a server has crashed. Simply waiting until the server is rebooted and trying again sometimes works and sometimes does not. A case where it works: client asks to read block 7 of some file. A case where it does not work: client says transfer a million dollars from one bank account to another. In the former case, it does not matter whether or not the server carried out the request before crashing; carrying it out a second time does no harm. In the latter case, one would definitely prefer the call to be carried out exactly once, no more and no less. Calls that may be repeated without harm (like the first example) are said to be *idempotent*. Unfortunately, it is not always possible to arrange for all calls to have this property. Any call that causes action to occur in the outside world, such as transferring money, printing lines, or opening a valve in an automated chocolate factory just long enough to fill exactly one vat, is likely to cause trouble if performed twice.

Spector [1982] and Nelson [1981] have looked at the problem of trying to make sure remote procedure calls are executed exactly once, and have developed taxonomies for classifying the semantics of different systems. These vary from systems that offer no guarantee at all (zero or more executions), to those that guarantee at most one execution (zero or one), to those that guarantee at least one execution (one or more).

Getting it right (exactly one) is probably impossible, because even if the remote execution can be reduced to one instruction (e.g., setting a bit in a device register that opens the chocolate valve), one can never be sure after a crash if the system went down a microsecond before, or a microsecond after, the one critical instruction. Sometimes one can make a guess based on observing external events (e.g., looking to see if the factory floor is covered with a sticky, brown material), but in general there is no way of knowing. Note that the problem of creating stable storage [Lampson, 1981] is fundamentally different, since remote procedure calls to the stable storage server in that model never causes events external to the computer.

5.2. Naming and protection

All operating systems objects such as files, directories, segments, mailboxes, processes, services, servers, nodes, and I/O devices. When a process wants to access one of these objects, it must present some kind of name to the operating system to specify which object it wants to access. In some instances these names are ASCII strings designed for human use, in others they are binary numbers used only internally. In all cases they have to be managed and protected from misuse.

5.2.1. Name servers

In centralized systems, the problem of naming can be effectively handled in a straightforward way. The system maintains a table or data base providing the necessary name-to-object mappings. The most straightforward generalization of this approach to distributed systems is the single name server model. In this model, a server accepts names in one domain and maps them onto names in another domain. For example, to locate services in some distributed systems, one sends the service name in ASCII to the name server, and it replies with the node number where that service can be found, or with the process name of the server process, or perhaps with the name of a mailbox to which requests for service can be sent. The name server's data base is built up by registering services, processes, etc., that want to be publicly known. File directories can be regarded as a special case of name service.

Although this model is often acceptable in a small distributed system located at a single site, in a large system it is undesirable to have a single centralized component (the name server) whose demise can bring the whole system to a grinding halt. In addition, if it becomes overloaded, performance will degrade. Furthermore, in a geographically distributed system that may have nodes in different cities or even countries, having a single name server will be inefficient due to the long delays in accessing it.

The next approach is to partition the system into domains, each with its own name server. If the system is composed of multiple local networks connected by gateways and bridges, it seems natural to have one name server per local network. One way to organize such a system is to have a global naming tree, with files and other objects having names of the form: /country/city/network/pathname. When such a name is presented to any name server, it can immediately route the request to some name server in the designated country, which then sends it to a name server in the designated city, and so on until it reaches the name server in the network where the object is located, where the mapping can be done. Telephone numbers use such a hierarchy, composed of country code, area code, exchange code (first 3 digits of telephone number in North America), and subscriber line number.

Having multiple name servers does not necessarily require having a single, global naming hierarchy. Another way to organize the name servers is to have each one effectively maintain a table of, for example, (ASCII string, pointer)-pairs, where the pointer is really a kind of capability for any object or domain in the system. When a name, say *a*/*b*/*c*, is looked up by the local name server, it may well yield a pointer to another domain (name server), to which the rest of the name, *b*/*c*, is sent for further processing. This facility can be used to provide links (in the UNIX sense) to files or objects whose precise whereabouts is managed by a remote name server. Thus if a file *foobar* is located in another local network, *n*, with name server *n.s*, one can make an entry in the local name server's table for the pair (*x*, *n.s*) and then access *x*/*foobar* as though it were a local object. Any appropriately authorized user or process knowing the name *x*/*foobar* could make its own synonym *s* and then perform accesses using *s/x/foobar*. Each name server parsing a name that involves multiple name servers just strips off the first component and passes the rest of the name to the name server found by looking up the first component locally.

A more extreme way of distributing the name server is to have each machine manage its own names. To look up a name, one broadcasts it on the network. At each machine, the incoming request is passed to the local name server, which replies only if it finds a match. Although broadcasting is easiest over a local network such as a ring net or CSMA net (e.g., Ethernet), it is also possible over store-and-forward packet switching networks such as the ARPAnet [Dalal, 1977].

Although the normal use of a name server is to map an ASCII string onto a binary number used internally to the system, such as a process identifier or machine number, once in a while the inverse mapping is also useful. For example, if a machine crashes, upon rebooting it could present its (hardwired) node number to the name server to ask what it was doing before the crash, i.e., ask for the ASCII string corresponding to the service it is supposed to be offering so it can figure out what program to reboot.

5.3. Resource management

Resource management in a distributed system differs from that in a centralized system in a fundamental way. Centralized systems always have tables that give complete and up-to-date status information about all the resources being managed; distributed systems do not. For example, the process manager in a traditional centralized operating system normally uses a 'process table' with one entry per potential process. When a new process has to be started, it is simple enough to scan the whole table to see if a slot is free. A distributed operating system, on the other hand, has a much harder job of finding out if a processor is free, especially if the system designers have rejected the idea of having any central tables at all, for reasons of reliability. Furthermore, even if there is a central table, recent events on outlying processors may have made some table entries obsolete without the table manager knowing it.

The problem of managing resources without having accurate global state information is very difficult. Relatively little work has been done in this area. In the following sections we will look at some work that has been done, including distributed process management and scheduling.

5.3.1. Scheduling

The hierarchical model provides a general model for resource control, but does not provide any specific guidance on how to do scheduling. If each process uses an entire processor (i.e., no multiprogramming), and each process is independent of all the others, any process can be assigned to any processor at random. However, if it is common that several processes are working together and must communicate frequently with each other, as in UNIX pipelines or in cascaded (nested) remote procedure calls, then it is desirable to make sure the whole group runs at once. In this section we will address that issue.

Let us assume that each processor can handle up to N processes. If there are plenty of machines and N is reasonably large, the problem is not finding a free machine (i.e., a free slot in some process table), but something more subtle. The basic difficulty can be illustrated by an example in which processes A and B run on one machine and processes C and D run on another. Each machine is time-shared in, say, 100 msec time slices, with A and C running in the even slices, and B and D running in the odd ones. Suppose that A sends many messages or makes many remote procedure calls of 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.

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

Ousterhout [1982] has proposed several algorithms based on the concept of *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 machine. Thus, column 4 consists of all the processes that run on machine 4. Row 3 is the collection of all processes that are in slot 3 of some machine, starting with the process in slot 3 of machine 0, then the process in slot 3 of machine 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 machines, 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 the four processes that must communicate should be put into slot 3, on machines 1, 2, 3, and 4 for optimum performance.

5.3.2. Load balancing

The goal of Ousterhout's work is to place processes that work together on different processors, so that they can all run in parallel. Other researchers have tried to do precisely the opposite, namely, to find subsets of all the processes in the system that are working together, so closely related groups of processes can be placed on the same machine to reduce interprocess communication costs [Chu, Holloway, Min-Tsung & Efe, 1980; Chow & Abraham, 1982; Gylys & Edwards, 1976; Stone, 1977, 1978; Stone & Bokhari, 1978; Lo, 1984]. Yet other researches have been concerned primarily with load balancing, to prevent a situation in which some processors are overloaded while others are empty [Barak & Shiloh, 1985; Efe, 1982; Krueger & Finkel, 1983; Stankovic & Sidhu, 1984]. Of course, the goals of maximizing throughput, minimizing response time, and keeping the load uniform, are to some extent in conflict, so many of the researchers try to evaluate different compromises and tradeoffs.

Each of these different approaches to scheduling makes different assumptions about what is known and what is most important. The people trying to cluster processes to minimize communication costs, for example, assume that any process can run on any machine, that the computing needs of each process are known in advance, and that the interprocess communication traffic between each pair of processes is also known in advance. The people doing load balancing typically make the realistic assumption that nothing about the future behavior of a process is known. The minimizers are generally theorists, whereas the load balancers tend to be people making real systems who care less about optimality than devising algorithms that can actually be used. Let us now briefly look at each of these approaches.

Graph theoretic models. If the system consists of a fixed number of processes, each with known CPU and memory requirements, and a known matrix giving the average amount of traffic between each pair of processes, scheduling can be attacked as a graph-theoretic problem. The system can be represented as a graph, with each process a node, and each pair of communicating processes connected by an arc labeled with the data rate between them.

The problem of allocating all the processes to k processors then reduces to the problem of partitioning the graph into k disjoint subgraphs, such that each subgraph meets certain constraints (e.g., total CPU and memory requirements below some limit). Arcs that are entirely within one subgraph represent internal communication within a single processor (= fast), whereas arcs that cut across subgraph boundaries represent communication between two processors (= slow). The idea is to find a partitioning of the graph that meets the constraints and minimizes the network traffic, or some variation of this idea. Many papers have been written on this subject, for example, Chow & Abraham [1982], Stone [1977, 1978], Stone and Bokhari [1978], Lo [1984]. The results are somewhat academic, since in real systems virtually none of the assumptions (fixed number of processes with static requirements, known traffic matrix, error-free processors and communication) are ever met.

Heuristic load balancing. When the goal of the scheduling algorithm is dynamic, heuristic, load balancing, rather than finding related clusters, a different approach is taken. Here the idea is for each processor to continually estimate its own load, for processors to exchange load information, and for process creation and migration to utilize this information.

Various methods of load estimation are possible. One way is just to measure the number of runnable processes on each CPU periodically, and take the average of the last *n* measurements as the load. Another way [Bryant & Finkel, 1981] is to estimate the residual running times of all the processes and define the load on a processor as the number of CPU seconds all its processes will need to finish. The residual time can be estimated mostly simply by assuming it is equal to the CPU time already consumed. Bryant and Finkel also discuss other estimation techniques in which both the number of processes and length of remaining time are important. When round robin scheduling is used, it is better to be competing against one process that needs 100 sec than against 100 processes that each need 1 sec.

Once each processor has computed its load, a way is needed for each processor to find out how everyone else is doing. One way is for each processor to just broadcast its load periodically. After receiving a broadcast from a lightly loaded machine, a processor should shed some of its load by giving it to the lightly loaded processor. This algorithm has several problems. First, it requires a broadcast facility, which may not be available. Second, it consumes considerable bandwidth for all the 'Here is my load' messages. Third, there is a great danger that many processors will try to shed load to the same (previously) lightly loaded processor at once.

A different strategy [Smith, 1979; Barak & Shiloh, 1985] if for each processor to periodically pick another processor (possibly a neighbor, possibly at random), and exchange load information with it. After the exchange, the more heavily loaded processor can send processes to the other one until they are equally loaded. In this model, if 100 processes are suddenly created in an otherwise empty system, after one exchange we will have two machines with 50 processes, and after two exchanges most probably four machines with 25 processes. Processes diffuse around the network like a cloud of gas.

Actually migrating running processes is trivial in theory but close to impossible in practice. The hard part is not moving the code, data, and registers, but moving the environment, such as the current position within all the open files, the current values of any running timers, pointers or file descriptors for communicating with tape drives or other I/O devices, etc. All of these problems relate to moving variables and data structures related to the process that are scattered about inside the operating system. What is feasible in practice is to use the load information to create new processes on lightly loaded machines, rather than trying to move running processes.

If one has adopted the idea of creating new processes only on lightly loaded machines, another approach, called bidding, is possible [Faber & Larson, 1972; Stankovic & Sidhu, 1984]. When a process wants some work done, it broadcasts a request for bids, telling what it needs (e.g., a 68000 CPU 512 K memory, floating point, and a tape drive).

Other processors can then bid for the work, telling what their workload is, how much memory they have available, etc. The process making the request then chooses the most suitable machine and creates the process there. If multiple request-for-bid messages are outstanding at the same time, a processor accepting a bid may discover that the workload on the bidding machine is not what is expected because that processor has bid for and won other work in the meantime.

5.4. Fault tolerance

Proponents of distributed systems often claim that such systems can be more reliable than centralized systems. Actually, there are at least two issues involved here: reliability and availability. Reliability has to do with the system not corrupting or losing your data. Availability has to do with the system being up when you need it. A system could be highly reliable in the sense that it never loses data, but at the same time be down most of the time and hence hardly usable. However, many people use the term 'reliability' to cover availability as well, and we will not make the distinction either in the rest of the paper.

The reason why distributed systems are potentially more reliable than a centralized system is that if a system only has one instance of some critical component, such as a CPU, disk, or network interface, and that component fails, the system will go down. When there are multiple instances, the system may be able to continue in spite of occasional failures. In addition to hardware failures, one can also consider software failures. These are of two types: the software failed to meet the formal specification (implementation error), or the specification does not correctly model what the customer wanted (specification error). All work on program verification is aimed at the former, but the latter is also an issue. Distributed systems allow both hardware and software errors to be dealt with, albeit in somewhat different ways.

An important distinction should be made between systems that are fault tolerant and those that are fault intolerant. A fault tolerant system is one that can continue functioning (perhaps in a degraded form) even if something goes wrong. A fault intolerant system collapses as soon as any error occurs. Biological systems are highly fault tolerant; if you cut your finger, you probably will not die. If a memory failure garbles 1/10 of 1 percent of the program code or stack of a running program, the program will almost certainly crash instantly upon encountering the error.

It is sometimes useful to distinguish between expected faults and unexpected faults. When the ARPAnet was designed, people expected to lose packets from time to time. This particular error was expected and precautions were taken to deal with it. On the other hand, no one expected a memory error in one of the packet switching machines to cause that machine to tell the world that it had a delay time of zero to every machine in the network, which resulted in all network traffic being rerouted to the broken machine.

One of the key advantages of distributed systems is that there are enough resources to achieve fault tolerance, at least with respect to expected errors. The system can be made to tolerate both hardware and software errors, although it should be emphasized that in both cases it is the software, not the hardware, that cleans up the mess when an error occurs. In the past few years, two approaches to making distributed systems fault tolerant have emerged. They differ radically in orientation, goals, and attitude toward the theologically sensitive issue of the perfectability of mankind (programmers in particular). One approach is based on redundancy and the other is based on the notion of an atomic transaction. Both are described briefly below.

5.4.1. Redundancy techniques

All the redundancy techniques that have emerged take advantage of the existence of multiple processors by duplicating critical processes on two or more machines. A particularly simple, but effective, technique is to provide every process with a backup process on a different processor. All processes communicate by message passing. Whenever anyone sends a message to a process, it also sends the same message to the backup process. The system ensures that neither the primary nor the backup can continue running until it has been verified that both have correctly received the message.

Thus, if one process crashes due to any hardware fault, the other one can continue. Furthermore, the remaining process can then clone itself, making a new backup to maintain the fault tolerance in the future. Borg, Baumbach & Glazer [1983] have described a system using these principles.

One disadvantage of duplicating every process is the extra processors required, but another, more subtle problem, is that if processes exchange messages at a high rate, a considerable amount of CPU time may go into keeping the processes synchronized at each message exchange. Powell & Presotto [1983] have described a redundant system that puts almost no additional load on the processes being backed up. In their system, all messages sent on the network are recorded by a special 'recorder' process. From time to time, each process checkpoints itself onto a remote disk.

If a process crashes, recovery is done by sending the most recent checkpoint to an idle processor and telling it to start running. The recorder process then spoon-feeds it all the messages that the original process received between the checkpoint and the crash. Messages sent by the newly restarted process are discarded. Once the new process has worked its way up to the point of crash, it begins sending and receiving messages normally, without help from the recording process.

The beauty of this scheme is that the only additional work a process must do to become immortal is to checkpoint itself from time to time. In theory, even the checkpoints can be disposed with, if the recorder process has enough disk space to store all the messages sent by all the currently running processes. If no checkpoints are made, when a process crashes, the recorder will have to replay the process's whole history.

Both of the above techniques only apply to tolerance of hardware errors.

However, it is also possible to use redundancy in distributed systems to make systems tolerant of software errors. One approach is to structure each program as a collection of modules, each one with a well-defined function and a precisely specified interface to the other modules. Instead of writing a module only once, N programmers are asked to program it, yielding N functionally identical modules.

During execution, the program runs on N machines in parallel. After each module finishes, the machines compare their results and vote on the answer. If a majority of the machines say that the answer is X, then all of them use X as the answer, and all continue in parallel with the next module. In this manner, the effects of an occasional software bug can be voted down. If formal specifications for any of the modules are available, the answers can also be checked against the specifications to guard against the possibility of accepting an answer that is clearly wrong.

A variation of this idea can be used to improve system performance. Instead of always waiting for all the processes to finish, as soon as k of them agree on an answer, those that have not yet finished are told to drop what they are doing, accept the value found by the k processes, and continue with the next module. Some work in this area is discussed by Avizienis & Chen [1977], Avizienis & Kelly [1984], and Anderson & Lee [1981].

5.4.2. Atomic transactions

When multiple users on several machines are concurrently updating a distributed data base and one or more machines crash, the potential for chaos is truly impressive. In a certain sense, the current situation is a step backward from the technology of the 1950s, when the normal way of updating a data base was to have one magnetic tape, called the 'master file', and one or more tapes with updates (e.g., daily sales reports from all of a company's stores). The master tape and updates were brought to the computer center, which then mounted the master tape and one update tape, and ran the update program to produce a new master tape. This new tape was then used as the 'master' for use with the next update tape.

This scheme had the very real advantage that if the update program crashed, one could always fall back on the previous master tape and the update tapes. In other words, an update run could be viewed as either running correctly to completion (and producing a new master tape), or having no effect at all (crash part way through, new tape discarded). Furthermore, update jobs from different sources always ran in some (undefined) sequential order. It never happened that two users would concurrently read a field in a record (e.g., 6), each add 1 to the value, and each store a 7 in that field, instead of the first one storing a 7 and the second storing an 8.

The property of run-to-completion or do-nothing is called an *atomic update*. The property of not interleaving two jobs is called *serializability*. The goal of people working on the atomic transaction approach to fault tolerance has been to regain the advantages of the old tape system without giving up the convenience of data bases on disk that can be modified in place, and to be able to do everything in a distributed way.

Lampson [1981] has described a way of achieving atomic transactions by building up a hierarchy of abstractions. We will summarize his model below. Real disks can crash during READ and WRITE operations in unpredictable ways. Furthermore, even if a disk block is correctly written, there is a small (but nonzero) probability of it subsequently being corrupted by a newly developed bad spot on the disk surface. The model assumes that spontaneous block corruptions are sufficiently infrequent that the probability of *two* such events happening within some predetermined time, T, is negligible. To deal with real disks, the system software must be able to tell if a block is valid or not, for example, by using a checksum.

The first layer of abstraction on top of the real disk is the 'careful disk', in which every CAREFUL-WRITE is read back immediately to verify that it is correct. If the CAREFUL-WRITE persistently fails, the system marks the block as "bad" and then intentionally crashes. Since CAREFUL-WRITEs are verified, CAREFUL-READs will always be good, unless a block has gone bad after being written and verified.

The next layer of abstraction is *stable storage*. A stable storage block consists of an ordered pair of careful blocks, which are typically corresponding careful blocks on different drives, to minimize the chance of both being damaged by a hardware failure. The stable storage algorithm guarantees that at least one of the blocks is always valid. The STABLE-WRITE primitive first does a CAREFUL-WRITE on one block of the pair, and then the other. If the first one fails, a crash is forced, as mentioned above, and the second one is left untouched.

After every crash, and at least once every time period T, a special cleanup process is run to examine each stable block. If both blocks are 'good' and identical, nothing has to be done. If one is 'good' and one is 'bad' (failure during a CAREFUL-WRITE), the 'bad' one is replaced by the 'good' one. If both are 'good' but different (crash between two CAREFUL-WRITEs), the second one is replaced by a copy of the first one. This algorithm allows individual disk blocks to be updated atomically and survive infrequent crashes.

Stable storage can be used to create 'stable processors' [Lampson, 1981]. To make itself crashproof, a CPU must checkpoint itself on stable storage periodically. If it subsequently crashes, it can always restart itself from the last checkpoint. Stable storage can also be used to create stable monitors, in order to ensure that two concurrent processes never enter the same critical region at the same time, even if they are running on different machines.

Given a way to implement crashproof processors (stable processors) and crashproof disks (stable storage), it is possible to implement multicomputer atomic transactions. Before updating any part of the data in place, a stable processor first writes an intentions list to stable storage, providing the new value for each datum to be changed. Then it sets a commit flag to indicate that the intentions list is complete. The commit flag is set by atomically updating a special block on stable storage. Finally it begins making all the changes called for in the intentions list. Crashes during this phase have no serious consequences because the intentions list is stored in stable storage. Furthermore, the actual making of the changes is idempotent, so repeated crashes and restarts during this phase are not harmful.

Atomic actions have been implemented in a number of systems [see, e.g., Fridrich & Older, 1981; Mitchell & Dion, 1982; Brown, Kolling & Taft, 1985; Popek, Walker, Chow, Edwards, Kline, Rudisin & Thiel, 1981; Reed & Svobodova, 1981].

5.5. Services

In a distributed system, it is natural to provide functions by user-level server processes that have traditionally been provided by the operating system. This approach leads to a smaller (hence more reliable) kernel and makes it easier to provide, modify, and test new services. In the following sections, we will look at some of these services, but first we look at how services and servers can be structured.

5.5.1. File service

There is little doubt that the most important service in any distributed system is the file service. Many file services and file servers have been designed and implemented, so a certain amount of experience is available [e.g., Birrell & Needham, 1980; Dellar, 1982; Dion, 1980; Fridrich & Older, 1981; 1984; Mitchell & Dion, 1982; Mullender & Tanenbaum, 1985; Reed & Svobodova, 1981; Satyanarayanan, Howard, Nichols, Sidebotham, Spector & West, 1985; Schroeder, Gifford & Needham, 1985; Sturgis, Mitchell & Israel, 1980; Svobodova, 1981; Swinehart, McDaniel & Boggs, 1979]. A survey about file servers can be found in Svobodova [1984].

File services can be roughly classified into two kinds, 'traditional' and 'robust'. Traditional file service is offered by nearly all centralized operating systems (e.g., the UNIX file system). Files can be opened, read, and rewritten in place. In particular, a program can open a file, seek to the middle of the file, and update blocks of data within the file. The file server implements these updates by simply overwriting the relevant disk blocks. Concurrency control, if there is any, usually involves locking entire files before updating them.

Robust file service, on the other hand, is aimed at those applications that require extremely high reliability and whose users are prepared to pay a significant penalty in performance to achieve it. These file services generally offer atomic updates and similar features lacking in the traditional file service.

In the following paragraphs, we discuss some of the issues relating to traditional file service (and file servers) and then look at those issues that specifically relate to robust file service and servers. Since robust file service normally includes traditional file service as a subset, the issues covered in the first part also apply.

Conceptually, there are three components that a traditional file service normally has:

- disk service,
- flat file service,
- directory service.

The disk service is concerned with reading and writing raw disk blocks, without regard to how they are organized. A typical command to the disk service is to allocate and write a disk block, and return a capability or address (suitably protected) so the block can be read later.

The flat file service is concerned with providing its clients with an abstraction consisting of files, each of which is a linear sequence of records, possibly 1-byte records (as in UNIX) or client-defined records. The operations are reading and writing records, starting at some particular place in the file. The client need not be concerned with how or where the data in the file are stored.

The directory service provides a mechanism for naming and protecting files, so they can be accessed conveniently and safely. The directory service typically provides objects called directories that map ASCII names onto the internal identification used by the file service.

5.5.2. Print service

Compared to file service, on which a great deal of time and energy has been expended by a large number of people, the other services seem rather meager. Still, it is worth saying at least a little bit about a few of the more interesting ones.

Nearly all distributed systems have some kind of print service, to which clients can send files or file names or capabilities for files with instructions to print them on one of the available printers, possibly with some text justification or other formatting beforehand. In some cases, the whole file is sent to the print server in advance, and the server must buffer it. In other cases, only the file name or capability is sent, and the print server reads the file block by block as needed. The latter strategy eliminates the need for buffering (read: a disk) on the server side, but can cause problems if the file is modified after the print command is given but prior to the actual printing. Users generally prefer 'call by value' rather than 'call by reference' semantics for printers.

One way to achieve the 'call by value' semantics is to have a printer spooler server. To print a file, the client process sends the file to the spooler. When the file has been copied to the spooler's directory, an acknowledgement is sent back to the client.

The actual print server is then implemented as a print client. Whenever the print client has nothing to print, it requests another file or block of a file from the print spooler, prints it, and then requests the next one. In this way the print spooler is a server to both the client and the printing device.

Printer service is discussed by Needham & Herbert [1982].

5.5.3. Process service

Every distributed operating system needs some mechanism for creating new processes. At the lowest level, deep inside the system kernel, there must be a way of creating a new process from scratch. One way is to have a FORK call, as UNIX does, but other approaches are also possible. For example, in Amoeba, it is possible to ask the kernel to allocate chunks of memory of given sizes. The caller can then read and write these chunks, loading them with the text, data, and stack segments for a new process. Finally, the caller can give the filled-in segments back to the kernel and ask for a new process built up from these pieces. This scheme allows processes to be created remotely or locally, as desired.

At a higher level, it is frequently useful to have a process server that one can ask whether there is a Pascal, troff, or some other service, in the system. If there is, the request is forwarded to the relevant server. If not, it is the job of the process server to build a process somewhere and give it the request. After, say, a VLSI design rule checking server has been created and has done its work, it may or may not be a good idea to keep it is the machine where it was created, depending on how much work (e.g., network traffic) is required to load it, and how often it is called. The process server could easily manage a server cache on a least recently used basis, so that servers for common applications are usually preloaded and ready to go. As special-purpose VLSI processors become available for compilers and other applications, the process server should be given the job of managing them in a way that is transparent to the system's users.

5.5.4. Terminal service

How the terminals are tied to the system obviously depends to a large extent on the system architecture. If the system consists of a small number of minicomputers, each with a well-defined and stable user population, then each terminal can be hardwired to the computer its user normally logs on to. If, however, the system consists entirely of a pool of processors that are dynamically allocated as needed, it is better to connect all the terminals to one or more terminal servers that serve as concentrators.

The terminal servers can also provide such features as local echoing, intraline editing, and window management, if desired. Furthermore, the terminal server can also hide the idiosyncracies of the various terminals in use by mapping them all onto a standard virtual terminal. In this way, the rest of the software deals only with the virtual terminal characteristics and the terminal server takes care of the mappings to and from all the real terminals. The terminal server can also be used to support multiple windows per terminal, with each window acting as a virtual terminal.

5.5.5. Mail service

Electronic mail is a popular application of computers these days. Practically every university computer science department in the Western world is on at least one international network for sending and receiving electronic mail. When a site consists of only one computer, keeping track of the mail is easy. However, when a site has dozens of computers spread over multiple local networks, users often want to be able to read their mail on any machine they happen to be logged on to. This desire gives rise to the need for a machine-independent mail service, rather like a print service that can be accessed system wide. Almes, Black, Lazowska & Noe [1985] discuss how mail is handled in the Eden system.

5.5.6. Time service

There are two ways to organize a time service. In the simplest way, clients can just ask the service what time it is. In the other way, the time service can broadcast the correct time periodically, to keep all the clocks on the other machines in sync. The time server can be equipped with a radio receiver tuned to WWV or some other transmitter that provides the exact time down to the microsecond.

Even with these two mechanisms, it is impossible to have all processes exactly synchronized. Consider what happens when a process requests the time-of-day from the time server. The request message comes in to the server, and a reply is sent back immediately. That reply must propagate back to the requesting process, cause an interrupt on its machine, have the kernel started up, and finally have the time recorded somewhere. Each of these steps introduces an unknown, variable delay.

On an Ethernet, for example, the amount of time required for the time server to put the reply message onto the network is nondeterministic and depends on the number of machines contending for access at that instant. If a large distributed system has only one time server, messages to and from it may have to travel a long distance and pass over store-and-forward gateways with variable queueing delays. If there are multiple time servers, they may get out of synchronization because their crystals run at slightly different rates. Einstein's special theory of relativity also puts constraints on synchronizing remote clocks.

The result of all these problems is that having a single, global time is impossible. Distributed algorithms that depend on being able to find a unique global ordering of widely separated events may not work as expected. A number of researchers have tried to find solutions to the various problems caused by the lack of global time. [See, e.g., Jefferson, 1985; Lamport, 1984, 1978; Marzullo & Owicki, 1985; Reed, 1983; Reif & Spirakis, 1984].

5.5.7. Boot service

The boot service has two functions: bringing up the system from scratch when the power is turned on, and helping important services survive crashes. In both cases, it is helpful if the boot server has a hardware mechanism for forcing a recalcitrant machine to jump to a program in its own ROM, in order to reset it. The ROM program could simply sit in a loop waiting for a message from the boot service. The message would then be loaded into that machine's memory and executed as a program.

The second function alluded to above is the 'immortality service'. An important service could register with the boot service, which would then poll it periodically to see if it were still functioning. If not, the boot service could initiate measures to patch things up, for example, forcibly reboot it or allocate another processor to take over its work. To provide high reliability, the boot service should itself consist of multiple processors, each of which keeps checking that the other ones are still working properly.

6. Suggested readings

This chapter has only scratched the surface of the operating systems literature. For more complete introductions, a number of textbooks are available, including Bach [1986], Deitel [1983], Finkel [1986], Peterson & Silberschatz [1985], and Tanenbaum [1987, 1992]. Recent papers can often be found in the journal *ACM Transactions on Computer Systems*. There are two conferences that also have many papers on operating systems: *ACM Symposium on Operating Systems Principles* (held biannually in odd-numbered years), and IEEE's *Distributed Computer Systems* (held every year).

References

Almes, G.T., A.P. Black, E.D. Lazowska, J.D. Noe (1985). The Eden system: A technical review. *IEEE Trans. Software Engrg.* SE-11, 43–59.

Anderson, T., P.A. Lee (1981). *Fault Tolerance, Principles and Practice*, Prentice-Hall, Int'l, London.

Avizienis, A., L. Chen (1977). On the implementation of *N*-version programming for software fault-tolerance during execution, *Proc. COMPSAC*, IEEE, pp. 149–155.

Avizienis A., J. Kelly (1984). Fault tolerance by design diversity. *Computer* 17, 66–80.

Bach, M. (1986). *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ.

Barak, A., A. Shiloh (1985). A distributed load-balancing policy for a multicomputer. *Software – Practice & Experience* 15, 901–913.

Birell, A.D., R.M. Needham (1980). A universal file server. *IEEE Trans. Software Engrg.* SE-6, 450–453.

Birell, A.D., B.J. Nelson (1984). Implementing remote procedure calls, *ACM Trans. Comput. Systems* 2, 39–59.

Borg, A., J. Baumbach, S. Glazer (1983). A message system supporting fault tolerance, *Proc. Ninth Symp. Operating Syst. Prin.*, ACM, pp. 90–99.

Brown, M.R., K.N. Kolling, E.A. Taft (1985). The Alpine file system. *ACM Trans. Comput. Syst.* 3, 261–293.

Brownbridge, D.R., L.F. Marshall, B. Randell (1982). The newcastle connection- or UNIXES or the World Unite! *Software – Practice & Experience* 12, 1147–1162.

Bryant, R.M., R.A. Finkel (1981). A stable distributed scheduling algorithm, *Proc. 2nd Int'l Conf. on Distributed Comput. Syst.*, IEEE, pp. 314–323.

Cadow, H. (1970). *OS/360 Job Control Language*, Prentice-Hall, Englewood Cliffs, NJ.

Chow, T.C.K., J.A. Abraham (1982). Load balancing in distributed systems, *IEEE Trans. Software Engrg.* SE-8, 401–412.

Chu, W.W., L.J. Holloway, L. Min-Tsung, K. Efe (1980). Task allocation in distributed data processing. *Computer* 13, 57–69.

Corbato, F.J., M. Merwin-Dagett, R.C. Daley (1962). An experimental time-sharing system, *Proc. AFIPS Fall Joint Computer Conf.*, 335–344.

Corbato, F.J., V.A. Vyssotsky (1965). Introduction and overview of the MULTICS system, *Proc. AFIPS Fall Joint Computer Conf.*, 185–196.

Dalal, Y.K. (1977). Broadcast protocols in packet switched computer networks, Ph.D. Thesis, Stanford Univ.

Daley, R.C., J.B. Dennis (1968). Virtual memory, process, and sharing in MULTICS. *Commun. of the ACM* 11, 306–312.

Deitel, H.M. (1983). *An Introduction to Operating Systems*, Addison-Wesley, Reading, MA.

Dellar, C. (1982). A file servers for a network of low-cost personal microcomputers. *Software – Practice & Experience* 12, 1051–1068.

Denning, P.J. (1970). Virtual memory. *Comput. Surveys* 2, 153–189.

Dion, J. (1980). The Cambridge file server. *Operating Syst. Rev.* 14, 41–49.

Efe, K. (1982). Heuristic models of task assignment scheduling in distributed systems, *Computer* 15, 50–56.

Farber, D.J., K.C. Larson (1972). The system architecture of the distributed computer system – The communications system, *Symp. Computer Netw.*, Polytechnic Institute of Brooklyn.

Finkel, R.A. (1986). *An Operating Systems Vade Mecum*, Prentice-Hall, Englewood Cliffs, NJ.

Fridrich, M., W. Older (1981). The Felix file server, *Proc. Eighth Symp. Operating Syst. Prin.*, ACM, pp. 37–44.

Fridrich, M., W. Older (1984). HELIX: The architecture of a distributed file system, *Proc. Fourth Int'l. Conf. on Distributed Comput. Syst.*, IEEE, pp. 422–431.

Gylys, V.B., J.A. Edwards (1976). Optimal partitioning of workload for distributed systems. *COMP-CON*, pp. 353–357.

Jefferson, D.R. (1985). Virtual time. *ACM Trans. Program. Lang. Syst.* 7, 404–425.

Kernighan, B.W., J.R. Mashey (1979). The UNIX programming environment. *Software – Practice & Experience* 9, 1–16.

Krueger, P., R.A. Finkel (1983). An adaptive load balancing algorithm for a multicomputer, Computer Science Dept., Univ. of Wisconsin.

Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system. *Commun. ACM* 21, 558–565.

Lamport, L. (1984). Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.* 6, 254–280.

Lampson, B.W. (1981). Atomic transactions, in: *Distributed Systems – Architecture and Implementation*, Springer-Verlag, Berlin, pp. 246–265.

Lo, V.M. (1984). Heuristic algorithms for task assignment in distributed systems, *Proc. Fourth Int'l Conf. on Distributed Comput. Syst.*, IEEE, pp. 30–39.

Marzullo, K., S. Owicki (1985). Maintaining the time in a distributed system. *Operating Syst. Rev.* 19, 44–54.

Mitchell, J.G., J. Dion (1982). A comparison of two network-based file servers. *Commun. ACM* 25, 233–245.

Mullender, S.J., A.S. Tanenbaum (1985). A distributed file service based on optimistic concurrency control, *Proc. Tenth Symp. Operating Syst. Prin.*, ACM, pp. 51–62.

Needham, R.M., A.J. Herbert (1982). *The Cambridge Distributed Computing System*, Addison-Wesley, Reading, MA.

Nelson, B.J. (1981). Remote procedure call, Techn. Rep. CSL-81-9, Xerox PARC.

Organick, E.I. (1972). *The Multics System*, M.I.T. Press, Cambridge, MA.

Ousterhout, J.K. (1982). Scheduling techniques for concurrent systems, *Proc. 3rd Int'l Conf. on Distributed Comput. Syst.*, IEEE, pp. 22–30.

Peterson, J.L., A. Silberschatz (1985). *Operating Systems Concepts, 2nd edition*, Addison-Wesley, Reading, MA.

Popek, G., B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel (1981). LOCUS: A network transparent, high reliability distributed system, *Proc. Eighth Symp. Operating Syst. Prin.*, ACM, pp. 160–168.

Powell, M.L., D.L. Presotto (1983). Publishing – A reliable broadcast communication mechanism, *Proc. Ninth Symp. Operating Syst. Prin.*, ACM, pp. 100–109.

Reed, D.P. (1983). Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.* 1, 3–23.

Reed, D.P., L. Svobodova (1981). SWALLOW: A distributed data storage system for a local network, in: A. West, P. Janson (eds.), *Local Networks for Computer Communications*, North-Holland, Amsterdam, pp. 355–373.

Reif, J.H., P.G. Spirakis (1984). Real-time synchronization of interprocess communications, *ACM Trans. Program. Lang. Syst.* 6, 215–238.

Ritchie, D.M., K. Thompson (1974). The UNIX time-sharing system. *Commun. ACM* 17, 365–375.

Saltzer, J.H. (1974). Protection and control of information sharing in MULTICS. *Commun. ACM* 17, 388–402.

Satyanarayanan, M., J. Howard, D. Nichols, R. Sidebotham, A. Spector, M. West (1985). The ITC distributed file system: Principles and design, *Proc. Tenth Symp. Operating Syst. Prin.*, ACM, pp. 35–50.

Schroeder, M., D. Gifford, R. Needham (1985). A caching file system for a programmer's workstation, *Proc. Tenth Symp. Operating Syst. Prin.*, ACM, pp. 25–34.

Smith, R. (1979). The contract net protocol: High-level communication and control in a distributed problem solver, *Proc. 1st Int'l Conf. Distributed Comput. Syst.*, IEEE, pp. 185–192.

Spector, A.Z. (1982). Performing remote operations efficiently on a local computer network. *Commun. ACM* 25, 246–260.

Stankovic, J.A., I.S. Sidhu (1984). An adaptive bidding algorithm for processes, clusters, and distributed ups, *Proc. Fourth Int'l Conf. on Distributed Comput. Syst.*, IEEE, pp. 49–59.

Stone, H.S. (1977). Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Software Engineering* SE-3, 88–93.

Stone, H.S. (1978). Critical load factors in distributed computer systems. *IEEE Trans. Software Engrg.* SE-4, 254–258.

Stone, H.S., S.H. Bokhari (1978). Control of distributed processes. *Computer* 11, 97–106.

Sturgis, H.E., J.G. Mitchell, J. Israel (1980). Issues in the design and use of a distributed file system. *Operating Systems Rev.* 14, 55–69.

Svobodova, L. (1981). A reliable object-oriented data repository for a distributed computer system, *Proc. Eight Symp. Operating Syst. Prin.*, ACM, pp. 47–58.

Svobodova, L. (1984). File servers for network-based distributed systems. *Comput. Surveys* 16, 353–398.

Swinehart, D., G. McDaniel, D. Boggs (1979). WFS: A simple shared file system for a distributed environment, *Proc. Seventh Symp. Operating Syst. Prin.*, ACM, pp. 9–17.

Tanenbaum, A.S. (1987). *Operating Systems: Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ.

Tanenbanm, A.S. (1992). *Modern Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ.

Walker, B., G. Popek, R. English, C. Kline, G. Thiel (1983). The LOCUS distributed operating system, *Proc. Ninth Symp. Operating Syst. Prin.*, ACM, pp. 49–70.

Wambecq, A. (1983). NETIX: A network-using operating system, based on UNIX software, Proc. NFWO-ENRS Contact Group, Leuven, Belgium.

Weinstein, M.J., T.W. Page, Jr., B.K. Livesey, G.J. Popek (1985). Transactions and synchronization in a distributed operating system, *Proc. 10th Symp. Oper. Syst. Prin.*, pp. 115–125.