

program can do itself. For example, why have a file system? Just let the user read and write a portion of the raw disk in a protected way. Of course, most users like having files, but the end-to-end argument says that the file system should be a library procedure linked with any program that needs to use files. This approach allows different programs to have different file systems. This line of reasoning says that all the operating system should do is securely allocate resources (e.g., the CPU and the disks) among the competing users. The Exokernel is an operating system built according to the end-to-end argument (Engler et al., 1995).

Client-Server Systems

A compromise between having the operating system do everything and the operating system do nothing is to have the operating system do a little bit. This design leads to a microkernel with much of the operating system running as user-level server processes as illustrated in Fig. 1-27. This is the most modular and flexible of all the designs. The ultimate in flexibility is to have each device driver also run as a user process, fully protected against the kernel and other drivers. Getting the drivers out of the kernel would eliminate the largest source of instability in any operating system—buggy third-party drivers—and would be a tremendous win in terms of reliability.

Of course, device drivers need to access the hardware device registers, so some mechanism is needed to provide this. If the hardware permits, each driver process could be given access to only those I/O devices it needs. For example, with memory-mapped I/O, each driver process could have the page for its device mapped in, but no other device pages. If the I/O port space can be partially protected, the correct portion of it could be made available to each driver.

Even if no hardware assistance is available, the idea can still be made to work. What is then needed is a new system call, available only to device driver processes, supplying a list of (port, value) pairs. What the kernel does is first check to see if the process owns all the ports in the list. If so, it then copies the corresponding values to the ports to initiate device I/O. A similar call can be used to read I/O ports in a protected way.

This approach keeps device drivers from examining (and damaging) kernel data structures, which is (for the most part) a good thing. An analogous set of calls could be made available to allow driver processes to read and write kernel tables, but only in a controlled way and with the approval of the kernel.

The main problem with this approach, and microkernels in general, is the performance hit all the extra context switches cause. However, virtually all work on microkernels was done many years ago when CPUs were much slower. Nowadays, applications that use every drop of CPU power and cannot tolerate a small loss of performance, are few and far between. After all, when running a word processor or Web browser, the CPU is probably idle 90% of the time. If a microkernel-based operating system turned an unreliable 900-MHz system into a

reliable 800-MHz system, probably few users would complain. After all, most of them were quite happy only a few years ago when they got their previous computer, at the then-stupendous speed of 100 MHz.

Extensible Systems

With the client-server systems discussed above, the idea was to get as much out of the kernel as possible. The opposite approach is to put more modules into the kernel, but in a protected way. The key word here is *protected*, of course. We studied some protection mechanisms in Sec. 9.5.6 that were initially intended for importing applets over the Internet, but are equally applicable to inserting foreign code into kernel. The most important ones are sandboxing and code signing as interpretation is not really practical for kernel code.

Of course, an extensible system by itself is not a way to structure an operating system. However, by starting with a minimal system consisting of little more than a protection mechanism and then adding protected modules to the kernel one at a time until reaching the functionality desired, a minimal system can be built for the application at hand. In this view, a new operating system can be tailored to each application by including only the parts it requires. Paramecium is an example of such a system (Van Doorn, 2001).

Kernel Threads

Another issue relevant here is that of system threads, no matter which structuring model is chosen. It is sometimes convenient to allow kernel threads to exist, separate from any user process. These threads can run in the background, writing dirty pages to disk, swapping processes between main memory and disk, and so on. In fact, the kernel itself can be structured entirely of such threads, so that when a user does a system call, instead of the user's thread executing in kernel mode, the user's thread blocks and passes control to a kernel thread that takes over to do the work.

In addition to kernel threads running in the background, most operating systems start up many daemon processes in the background, too. While these are not part of the operating system, they often perform "system" type activities. These might include getting and sending email and serving various kinds of requests for remote users, such as FTP and Web pages.

12.3.2 Mechanism versus Policy

Another principle that helps architectural coherence, along with keeping things small and well structured, is that of separating mechanism from policy. By putting the mechanism in the operating system and leaving the policy to user processes, the system itself can be left unmodified, even if there is a need to change policy. Even if the policy module has to be kept in the kernel, it should be

isolated from the mechanism, if possible, so that changes in the policy module do not affect the mechanism module.

To make the split between policy and mechanism clearer, let us consider two real-world examples. As a first example, consider a large company that has a payroll department, which is in charge of paying the employees' salaries. It has computers, software, blank checks, agreements with banks, and more mechanism for actually paying out the salaries. However, the policy—determining who gets paid how much—is completely separate and is decided by management. The payroll department just does what it is told to do.

As the second example, consider a restaurant. It has the mechanism for serving diners, including tables, plates, waiters, a kitchen full of equipment, agreements with credit card companies, and so on. The policy is set by the chef, namely, what is on the menu. If the chef decides that tofu is out and big steaks are in, this new policy can be handled by the existing mechanism.

Now let us consider some operating system examples. First, consider thread scheduling. The kernel could have a priority scheduler, with k priority levels. The mechanism is an array, indexed by priority level, as shown in Fig. 10-11 or Fig. 11-19. Each entry is the head of a list of ready threads at that priority level. The scheduler just searches the array from highest priority to lowest priority, selecting the first threads it hits. The policy is setting the priorities. The system may have different classes of users, each with a different priority, for example. It might also allow user processes to set the relative priority of its threads. Priorities might be increased after completing I/O or decreased after using up a quantum. There are numerous other policies that could be followed, but the idea here is the separation between setting policy and carrying it out.

A second example is paging. The mechanism involves MMU management, keeping lists of occupied pages and free pages, and code for shuttling pages to and from disk. The policy is deciding what to do when a page fault occurs. It could be local or global, LRU-based or FIFO-based, or something else, but this algorithm can (and should) be completely separate from the mechanics of actually managing the pages.

A third example is allowing modules to be loaded into the kernel. The mechanism concerns how they are inserted, how they are linked, what calls they can make, and what calls can be made on them. The policy is determining who is allowed to load a module into the kernel and which modules. Maybe only the superuser can load modules, but maybe any user can load a module that has been digitally signed by the appropriate authority.

12.3.3 Orthogonality

Good system design consists of separate concepts that can be combined independently. For example, in C, there are primitive data types including integers, characters, and floating-point numbers. There are also mechanisms for com-

binning data types, including arrays, structures, and unions. These ideas combine independently, allowing arrays of integers, arrays of characters, structure and union members that are floating-point numbers, etc. In fact, once a new data type has been defined, such as an array of integers, it can be used as if it were a primitive data type, for example as a member of a structure or a union. The ability to combine separate concepts independently is called **orthogonality**. It is direct consequence of the simplicity and completeness principles.

The concept of orthogonality also occurs in operating systems in various disguises. One example is the Linux clone system call, which creates a new thread. The call has a bitmap as a parameter, which allows the address space, working directory, file descriptors, and signals to shared or copied individually. If everything is copied, we have a new process, the same as fork. If nothing is copied, a new thread is created in the current process. However, it is also possible to create intermediate forms of sharing not possible in traditional UNIX systems. By separating out the various features and making them orthogonal, a finer degree of control is possible.

Another use of orthogonality is the separation of the process concept from the thread concept in Windows 2000. A process is a container for resources, nothing more and nothing less. A thread is a schedulable entity. When one process is given a handle for another process, it does not matter how many threads it has. When a thread is scheduled, it does not matter which process it belongs to. These concepts are orthogonal.

Our last example of orthogonality comes from UNIX. Process creation there is done in two steps: fork plus exec. Creating the new address space and loading it with a new memory image are separate, allowing things to be done in between (such as manipulating file descriptors). In Windows 2000, these two steps cannot be separated, that is, the concepts of making a new address space and filling it in are not orthogonal there. The Linux sequence of clone plus exec is yet more orthogonal, since there are even more fine-grained building blocks available. As a general rule, having a small number of orthogonal elements that can be combined in many ways leads to a small, simple, and elegant system.

12.3.4 Naming

Most long-lived data structures used by an operating system have some kind of name or identifier by which they can be referred. Obvious examples are login names, file names, device names, process IDs, and so on. How these names are constructed and managed is an important issue in system design and implementation.

Names designed for people to use are character-string names in ASCII or Unicode and are usually hierarchical. Directory paths, for example one such as */usr/ast/books/mos2/chap-12*, are clearly hierarchical, indicating a series of directories to search starting at the root. URLs are also hierarchical. For example,

`www.cs.vu.nl/~ast/` indicates a specific machine (*www*) in a specific department (*cs*) at specific university (*vu*) in a specific country (*nl*). The part after the slash indicates a specific file on the designated machine, in this case, by convention, `www/index.html` in *ast*'s home directory. Note that URLs (and DNS addresses in general, including email addresses) are “backward,” starting at the bottom of the tree and going up, unlike file names, which start at the top of the tree and go down. Another way of looking at this is whether the tree is written from the top starting at the left and going right or starting at the right and going left.

Often naming is done at two levels: external and internal. For example, files always have a character-string name for people to use. In addition, there is almost always an internal name that the system uses. In UNIX, the real name of a file is its i-node number; the ASCII name is not used at all internally. In fact, it is not even unique since a file may have multiple links to it. The analogous internal name in Windows 2000 is the file's index in the MFT. The job of the directory is to provide the mapping between the external name and the internal name, as shown in Fig. 12-3.

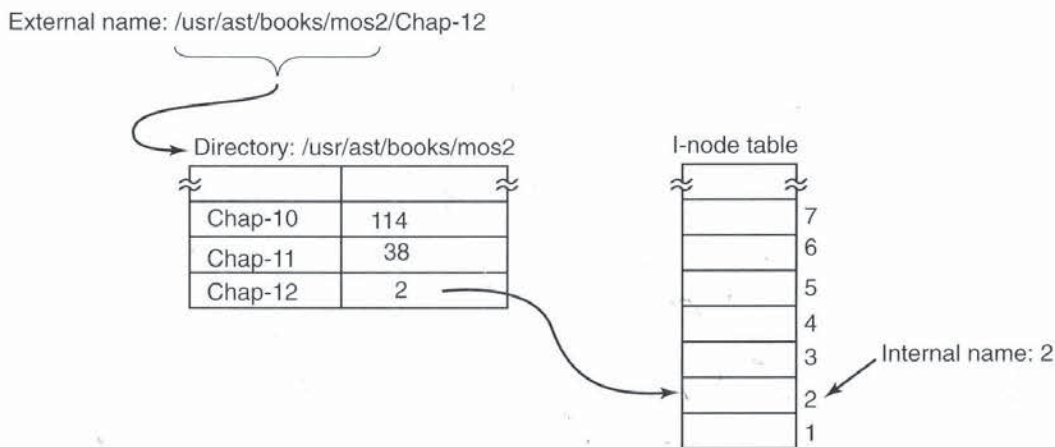


Figure 12-3. Directories are used to map external names onto internal names.

In many cases (such as the file name example given above), the internal name is an unsigned integer that serves as an index into a kernel table. Other examples of table-index names are file descriptors in UNIX and object handles in Windows 2000. Note that neither of these has any external representation. They are strictly for use by the system and running processes. In general, using table indices for transient names that are lost when the system is rebooted is a good idea.

Operating systems often support multiple name spaces, both external and internal. For example, in Chap. 11 we looked at three external name spaces supported by Windows 2000: file names, object names, and registry names (and there is also the Active Directory name space, which we did not look at). In addition, there are innumerable internal name spaces using unsigned integers, for example, object handles, MFT entries, etc. Although the names in the external name spaces

are all Unicode strings, looking up a file name in the registry will not work, just as using an MFT index in the object table will not work. In a good design, considerable thought is given to how many names spaces are needed, what the syntax of names is in each one, how they can be told apart, whether absolute and relative names exist, and so on.

12.3.5 Binding Time

As we have just seen, operating systems use various kinds of names to refer to objects. Sometimes the mapping between a name and an object is fixed, but sometimes it is not. In the latter case, it may matter when the name is bound to the object. In general, **early binding** is simple, but is not flexible, whereas **late binding** is more complicated but often more flexible.

To clarify the concept of binding time, let us look at some real-world examples. An example of early binding is the practice of some colleges to allow parents to enroll a baby at birth and prepay the current tuition. When the student shows up 18 years later, the tuition is fully paid up, no matter how high it may be at that moment.

In manufacturing, ordering parts in advance and maintaining an inventory of them is early binding. In contrast, just-in-time manufacturing requires suppliers to be able to provide parts on the spot, with no advance notice required. This is late binding.

Programming languages often support multiple binding times for variables. Global variables are bound to a particular virtual address by the compiler. This exemplifies early binding. Variables local to a procedure are assigned a virtual address (on the stack) at the time the procedure is invoked. This is intermediate binding. Variables stored on the heap (those allocated by *malloc* in C or *new* in Java) are assigned virtual addresses only at the time they are actually used. Here we have late binding.

Operating systems often use early binding for most data structures, but occasionally use late binding for flexibility. Memory allocation is a case in point. Early multiprogramming systems on machines lacking address relocation hardware had to load a program at some memory address and relocate it to run there. If it was ever swapped out, it had to be brought back at the same memory address or it would fail. In contrast, paged virtual memory is a form of late binding. The actual physical address corresponding to a given virtual address is not known until the page is touched and actually brought into memory.

Another example of late binding is window placement in a GUI. In contrast to the early graphical systems, in which the programmer had to specify the absolute screen coordinates for all images on the screen, in modern GUIs, the software uses coordinates relative to the window's origin, but that is not determined until the window is put on the screen, and it may even be changed later.

12.3.6 Static versus Dynamic Structures

Operating system designers are constantly forced to choose between static and dynamic data structures. Static ones are always simpler to understand, easier to program, and faster in use; dynamic ones are more flexible. An obvious example is the process table. Early systems simply allocated a fixed array of per-process structures. If the process table consisted of 256 entries, then only 256 processes could exist at any one instant. An attempt to create a 257th one would fail for lack of table space. Similar considerations held for the table of open files (both per user and system wide), and many other kernel tables.

An alternative strategy is to build the process table as a linked list of minitables, initially just one. If this table fills up, another one is allocated from a global storage pool and linked to the first one. In this way, the process table cannot fill up until all of kernel memory is exhausted.

On the other hand, the code for searching the table becomes more complicated. For example, the code for searching a static process table to see if a given PID, *pid*, is given in Fig. 12-4. It is simple and efficient. Doing the same thing for a linked list of minitables is more work.

```
found = 0;
for (p = &proc_table[0]; p < &proc_table[PROC_TABLE_SIZE]; p++) {
    if (p->proc_pid == pid) {
        found = 1;
        break;
    }
}
```

Figure 12-4. Code for searching the process table for a given PID.

Static tables are best when there is plenty of memory or table utilizations can be guessed fairly accurately. For example, in a single-user system, it is unlikely that the user will start up more than 32 processes at once and it is not a total disaster if an attempt to start a 33rd one fails.

Yet another alternative is to use a fixed-size table, but if it fills up, allocate a new fixed-size table, say, twice as big. The current entries are then copied over to the new table and the old table is returned to the free storage pool. In this way, the table is always contiguous rather than linked. The disadvantage here is that some storage management is needed and the address of the table is now a variable instead of a constant.

A similar issue holds for kernel stacks. When a thread switches to kernel mode, or a kernel-mode thread is run, it needs a stack in kernel space. For user threads, the stack can be initialized to run down from the top of the virtual address space, so the size need not be specified in advance. For kernel threads, the size must be specified in advance because the stack takes up some kernel virtual

address space and there may be many stacks. The question is: how much space should each one get? The trade-offs here are similar to those for the process table.

Another static-dynamic trade-off is process scheduling. In some systems, especially real-time ones, the scheduling can be done statically in advance. For example, an airline knows what time its flights will leave weeks before their departure. Similarly, multimedia systems know when to schedule audio, video, and other processes in advance. For general-purpose use, these considerations do not hold and scheduling must be dynamic.

Yet another static-dynamic issue is kernel structure. It is much simpler if the kernel is built as a single binary program and loaded into memory to run. The consequence of this design, however, is that adding a new I/O device requires a relinking of the kernel with the new device driver. Early versions of UNIX worked this way, and it was quite satisfactory in a minicomputer environment when adding new I/O devices was a rare occurrence. Nowadays, most operating systems allow code to be added to the kernel dynamically, with all the additional complexity that entails.

12.3.7 Top-Down versus Bottom-Up Implementation

While it is best to design the system top down, in theory it can be implemented top down or bottom up. In a top-down implementation, the implementers start with the system call handlers and see what mechanisms and data structures are needed to support them. These procedures are written and so on until the hardware is reached.

The problem with this approach is that it is hard to test anything with only the top-level procedures available. For this reason, many developers find it more practical to actually build the system bottom up. This approach entails first writing code that hides the low-level hardware, essentially the HAL in Fig. 11-7. Interrupt handling and the clock driver are also needed early on.

Then multiprogramming can be tackled, along with a simple scheduler (e.g., round-robin scheduling). At this point it should be possible to test the system to see if it can run multiple processes correctly. If that works, it is now time to begin the careful definition of the various tables and data structures needed throughout the system, especially those for process and thread management and later memory management. I/O and the file system can wait initially, except for a primitive way to read the keyboard and write to the screen for testing and debugging. In some cases, the key low-level data structures should be protected by allowing access only through specific access procedures—in effect, object-oriented programming, no matter what the programming language is. As lower layers are completed, they can be tested thoroughly. In this way, the system advances from the bottom up, much the way contractors build tall office buildings.

If a large team is available, an alternative approach is to first make a detailed design of the whole system, and then assign different groups to write different

modules. Each one tests its own work in isolation. When all the pieces are ready, they are integrated and tested. The problem with this line of attack is that if nothing works initially, it may be hard to isolate whether one or more modules are malfunctioning, or one group misunderstood what some other module was supposed to do. Nevertheless, with large teams, this approach is often used to maximize the amount of parallelism in the programming effort.

12.3.8 Useful Techniques

We have just looked at some abstract ideas for system design and implementation. Now we will examine a number of useful concrete techniques for system implementation. There are numerous other ones, of course, but space limitations restrict us to just a few of them.

Hiding the Hardware

A lot of hardware is ugly. It has to be hidden early on (unless it exposes power, which most hardware does not). Some of the very low-level details can be hidden by a HAL-type layer of the type shown in Fig. 12-2. However, many hardware details cannot be hidden this way.

One thing that deserves early attention is how to deal with interrupts. They make programming unpleasant, but operating systems have to deal with them. One approach is to turn them into something else immediately. For example, every interrupt could be turned into a pop-up thread instantly. At that point we are dealing with threads, rather than interrupts.

A second approach is to convert each interrupt into an unlock operation on a mutex that the corresponding driver is waiting on. Then the only effect of an interrupt is to cause some thread to become ready.

A third approach is convert an interrupt into a message to some thread. The low-level code just builds a message telling where the interrupt came from, enqueues it, and calls the scheduler to (potentially) run the handler, which was probably blocked waiting for the message. All these techniques, and other ones like them, all try to convert interrupts into thread synchronization operations. Having each interrupt handled by a proper thread in a proper context is easier to manage than running a handler in the arbitrary context that it happened to occur in. Of course, this must be done efficiently, but deep within the operating system, everything must be done efficiently.

Most operating systems are designed to run on multiple hardware platforms. These platforms can differ in terms of the CPU chip, MMU, word length, RAM size, and other features that cannot easily be masked by the HAL or equivalent. Nevertheless, it is highly desirable to have a single set of source files that are used to generate all versions; otherwise each bug that later turns up must be fixed multiple times in multiple sources, with the danger that the sources drift apart.

Some hardware differences, such as RAM size, can be dealt with by having the operating system determine the value at boot time and keep it in a variable. Memory allocators, for example, can use the RAM size variable to determine how big to make the block cache, page tables, etc. Even static tables such as the process table can be sized based on the total memory available.

However, other differences, such as different CPU chips, cannot be solved by having a single binary that determines at run time which CPU it is running on. One way to tackle the problem of one source and multiple targets is to use conditional compilation. In the source files, certain compile-time flags are defined for the different configurations and these are used to bracket code that is dependent on the CPU, word length, MMU, etc. For example, imagine an operating system that is to run on the Pentium and UltraSPARC chips, which need different initialization code. The *init* procedure could be written as illustrated in Fig. 12-5(a). Depending on the value of *CPU*, which is defined in the header file *config.h*, one kind of initialization or other is done. Because the actual binary contains only the code needed for the target machine, there is no loss of efficiency this way.

<pre>#include "config.h" init() { #if (CPU == PENTIUM) /* Pentium initialization here. */ #endif #if (CPU == ULTRASPARC) /* UltraSPARC initialization here. */ #endif }</pre>	<pre>#include "config.h" #if (WORD_LENGTH == 32) typedef int Register; #endif #if (WORD_LENGTH == 64) typedef long Register; #endif Register R0, R1, R2, R3;</pre>
(a)	(b)

Figure 12-5. (a) CPU-dependent conditional compilation. (b) Word-length dependent conditional compilation.

As a second example, suppose there is a need for a data type *Register*, which should be 32 bits on the Pentium and 64 bits on the UltraSPARC. This could be handled by the conditional code of Fig. 12-5(b) (assuming that the compiler produces 32-bit ints and 64-bit longs). Once this definition has been made (probably in a header file included everywhere), the programmer can just declare variables to be of type *Register* and know they will be the right length.

The header file, *config.h*, has to be defined correctly, of course. For the Pentium it might be something like this:

```
#define CPU PENTIUM
#define WORD_LENGTH 32
```

To compile the system for the UltraSPARC, a different *config.h* would be used, with the correct values for the UltraSPARC, probably something like

```
#define CPU ULTRASPARC
#define WORD_LENGTH 64
```

Some readers may be wondering why *CPU* and *WORD_LENGTH* are handled by different macros. We could easily have bracketed the definition of *Register* with a test on *CPU*, setting it to 32 bits for the Pentium and 64 bits for the UltraSPARC. However, this is not a good idea. Consider what happens when we later port the system to the 64-bit Intel Itanium. We would have to add a third conditional to Fig. 12-5(b) for the Itanium. By doing it as we have, all we have to do is include the line

```
#define WORD_LENGTH 64
```

to the *config.h* file for the Itanium.

This example illustrates the orthogonality principle we discussed earlier. Those items that are CPU-dependent should be conditionally compiled based on the *CPU* macro and those things that are word-length dependent should use the *WORD_LENGTH* macro. Similar considerations hold for many other parameters.

Indirection

It is sometimes said that there is no problem in computer science that cannot be solved with another level of indirection. While something of an exaggeration, there is definitely a grain of truth here. Let us consider some examples. On Pentium-based systems, when a key is depressed, the hardware generates an interrupt and puts the key number, rather than an ASCII character code, in a device register. Furthermore, when the key is released later, a second interrupt is generated, also with the key number. This indirection allows the operating system the possibility of using the key number to index into a table to get the ASCII character, which makes it easy to handle the many keyboards used around the world in different countries. Getting both the depress and release information makes it possible to use any key as a shift key since the operating system knows the exact sequence the keys were depressed and released.

Indirection is also used on output. Programs can write ASCII characters to the screen, but these are interpreted as indices into a table for the current output font. The table entry contains the bitmap for the character. This indirection makes it possible to separate characters from fonts.

Another example of indirection is the use of major device numbers in UNIX. Within the kernel there is a table indexed by major device number for the block devices and another one for the character devices. When a process opens a special file such as */dev/hd0*, the system extracts the type (block or character) and

major and minor device numbers from the i-node and indexes into the appropriate driver table to find the driver. This indirection makes it easy to reconfigure the system, because programs deal with symbolic device names, not actual driver names.

Yet another example of indirection occurs in message-passing systems that name a mailbox rather than a process as the message destination. By indirecting through mailboxes (as opposed to naming a process as the destination), considerable flexibility can be achieved (e.g., having a secretary handle her boss' messages).

In a sense, the use of macros, such as

```
#define PROC_TABLE_SIZE 256
```

is also a form of indirection, since the programmer can write code without having to know how big the table really is. It is good practice to give symbolic names to all constants (except sometimes -1 , 0 , and 1), and put these in headers with comments explaining what they are for.

Reusability

It is frequently possible to reuse the same code in slightly different contexts. Doing so is a good idea as it reduces the size of the binary and means that the code has to be debugged only once. For example, suppose that bitmaps are used to keep track of free blocks on the disk. Disk block management can be handled by having procedures *alloc* and *free* that manage the bitmaps.

As a bare minimum, these procedures should work for any disk. But we can go further than that. The same procedures can also work for managing memory blocks, blocks in the file system's block cache, and i-nodes. In fact, they can be used to allocate and deallocate any resources that can be numbered linearly.

Reentrancy

Reentrancy refers for the ability of code to be executed two or more times simultaneously. On a multiprocessor, there is always the danger that while one CPU is executing some procedure, another CPU will start executing it as well, before the first one has finished. In this case, two (or more) threads on different CPUs might be executing the same code at the same time. This situation must be protected against by using mutexes or some other means to protect critical regions.

However, the problem also exists on a uniprocessor. In particular, most of any operating system runs with interrupts enabled. To do otherwise, would lose many interrupts and make the system unreliable. While the operating system is busy executing some procedure, *P*, it is entirely possible that an interrupt occurs and that the interrupt handler also calls *P*. If the data structures of *P* were in an

inconsistent state at the time of the interrupt, the handler will see them in an inconsistent state and fail.

An obvious example where this can happen is if P is the scheduler. Suppose that some process used up its quantum and the operating system was moving it to the end of its queue. Part way through the list manipulation, the interrupt occurs, makes some process ready, and runs the scheduler. With the queues in an inconsistent state, the system will probably crash. As a consequence even on a uniprocessor, it is best that most of the operating system is reentrant, critical data structures are protected by mutexes, and interrupts are disabled at moments when they cannot be tolerated.

Brute Force

Using brute force to solve a problem has acquired a bad name over the years, but it is often the way to go in the name of simplicity. Every operating system has many procedures that are rarely called or operate with so little data that optimizing them is not worthwhile. For example, it is frequently necessary to search various tables and arrays within the system. The brute force algorithm is just leave the table in the order the entries are made and search it linearly when something has to be looked up. If the number of entries is small (say, under 100), the gain from sorting the table or hashing it is small, but the code is far more complicated and more likely to have bugs in it.

Of course, for functions that are on the critical path, say, context switching, everything should be done to make them fast, possibly even writing them in (heaven forbid) assembly language. But large parts of the system are not on the critical path. For example, many system calls are rarely called. If there is one fork every 10 sec, and it takes 10 msec to carry out, then even optimizing it to 0 wins only 0.1%. If the optimized code is bigger and buggier, a case can be made not to bother with the optimization.

Check for Errors First

Many system calls can potentially fail for a variety of reasons: the file to be opened belongs to someone else; process creation fails because the process table is full; or a signal cannot be sent because the target process does not exist. The operating system must painstakingly check for every possible error before carrying out the call.

Many system calls also require acquiring resources such as process table slots, i-node table slots, or file descriptors. A general piece of advice that can save a lot of grief is to first check to see if the system call can actually be carried out before acquiring any resources. This means putting all the tests at the beginning of the procedure that executes the system call. Each test should be of the form

```
if (error_condition) return(ERROR_CODE);
```

If the call gets all the way through the gauntlet of tests, then it is certain that it will succeed. At that point resources can be acquired.

Interspersing the tests with resource acquisition means that if some test fails along the way, all the resources acquired up to that point must be returned. If an error is made here and some resource is not returned, no damage is done immediately. For example, one process table entry may just become permanently unavailable. However, over a period of time, this bug may be triggered multiple times. Eventually, most or all the process table entries may become unavailable, leading to a system crash in an extremely unpredictable and difficult to debug way.

Many systems suffer from this problem in the form of memory leaks. Typically, the program calls *malloc* to allocate space but forgets to call *free* later to release it. Ever so gradually, all of memory disappears until the system is rebooted.

Engler et al. (2000) have proposed an interesting way to check for some of these errors at compile time. They observed that the programmer knows many invariants that the compiler does not know, such as when you lock a mutex, all paths starting at the lock must contain an unlock and no more locks of the same mutex. They have devised a way for the programmer to tell the compiler this fact and instruct it to check all the paths at compile time for violations of the invariant. The programmer can also specify that allocated memory must be released on all paths and many other conditions as well.

12.4 PERFORMANCE

All things being equal, a fast operating system is better than a slow one. However, a fast unreliable operating system is not as good as a reliable slow one. Since complex optimizations often lead to bugs, it is important to use them sparingly. This notwithstanding, there are places where performance is critical and optimizations are well worth the effort. In the following sections, we will look at some general techniques that can be used to improve performance in places where that is called for.

12.4.1 Why Are Operating Systems Slow?

Before talking about optimization techniques, it is worth pointing out that the slowness of many operating systems is to a large extent self-inflicted. For example, older operating systems, such as MS-DOS and UNIX Version 7, booted within a few seconds. Modern UNIX systems and Windows 2000 can take minutes to boot, despite running on hardware that is 100 times as fast. The reason is that they are doing much more, wanted or not. A case in point. Plug and play makes it somewhat easier to install a new hardware device, but the price paid is that on

every boot, the operating system has to go out and inspect all the hardware to see if there is anything new out there. This bus scan takes time.

An alternative (and in the author's opinion, better) approach would be to scrap plug-and-play altogether and have an icon on the screen labeled "Install new hardware." Upon installing a new hardware device, the user would click on this icon to start the bus scan, instead of doing it on every boot. The designers of current systems were well aware of this option, of course. They rejected it, basically, because they assumed that the users were too stupid to be able to do this correctly (although they would word it more kindly). This is only one example, but there are many more where the desire to make the system "user-friendly" (or "idiot-proof," depending on your viewpoint) slows the system down all the time for everyone.

Probably the biggest single thing system designers can do to improve performance is to be much more selective about adding new features. The question to ask is not: "Will the users like this?" but "Is this feature worth the inevitable price in code size, speed, complexity, and reliability?" Only if the advantages clearly outweigh the drawbacks should it be included. Programmers have a tendency to assume that code size and bug count will be 0 and speed will be infinite. Experience shows this view to be a bit optimistic.

Another factor that plays a role is product marketing. By the time version 4 or 5 of some product has hit the market, probably all the features that are actually useful have been included and most of the people who need the product already have it. To keep sales going, many manufacturers nevertheless continue to produce new versions, with more features, just so they can sell their existing customers upgrades. Adding new features just for the sake of adding new features may help sales but rarely helps performance.

12.4.2 What Should Be Optimized?

As a general rule, the first version of the system should be as straightforward as possible. The only optimizations should be things that are so obviously going to be a problem that they are unavoidable. Having a block cache for the file system is such an example. Once the system is up and running, careful measurements should be made to see where the time is *really* going. Based on these numbers, optimizations should be made where they will help most.

Here is a true story of where an optimization did more harm than good. One of the author's students (who shall here remain nameless) wrote the MINIX *mkfs* program. This program lays down a fresh file system on a newly formatted disk. The student spent about 6 months optimizing it, including putting in disk caching. When he turned it in, it did not work and it required several additional months of debugging. This program typically runs once on the hard disk during the life of the computer, when the system is installed. It also runs once for each floppy disk that is formatted. Each run takes about 2 sec. Even if the unoptimized version

had taken 1 minute, it was a poor use of resources to spend so much time optimizing a program that is used so infrequently.

A slogan that has considerable applicability to performance optimization is

Good enough is good enough.

By this we mean that once performance has achieved a reasonable level, it is probably not worth the effort and complexity to squeeze out the last few percent. If the scheduling algorithm is reasonably fair and keeps the CPU busy 90% of the time, it is doing its job. Devising a far more complex one that is 5% better is probably a bad idea. Similarly, if the page rate is low enough that it is not a bottleneck, jumping through hoops to get optimal performance is usually not worth it. Avoiding disaster is far more important than getting optimal performance, especially since what is optimal with one load may not be optimal with another.

12.4.3 Space-Time Trade-offs

One general approach to improving performance is to trade off time versus space. It frequently occurs in computer science that there is a choice between an algorithm that uses little memory but is slow and an algorithm that uses much more memory but is faster. When making an important optimization, it is worth looking for algorithms that gain speed by using more memory or conversely save precious memory by doing more computation.

One technique that is often helpful is to replace small procedures by macros. Using a macro eliminates the overhead normally associated with a procedure call. The gain is especially significant if the call occurs inside a loop. As an example, suppose we use bitmaps to keep track of resources and frequently need to know how many units are free in some portion of the bitmap. For this purpose we need a procedure, *bit_count*, that counts the number of 1 bits in a byte. The straightforward procedure is given in Fig. 12-6(a). It loops over the bits in a byte counting them one at a time.

This procedure has two sources of inefficiency. First, it must be called, stack space must be allocated for it, and it must return. Every procedure call has this overhead. Second, it contains a loop, and there is always some overhead associated with a loop.

A completely different approach is to use the macro of Fig. 12-6(b). It is an inline expression that computes the sum of the bits by successively shifting the argument, masking out everything but the low-order bit, and adding up the eight terms. The macro is hardly a work of art, but it appears in the code only once. When the macro is called, for example, by

```
sum = bit_count(table[i]);
```

the macro call looks identical to the call of the procedure. Thus other than one somewhat messy definition, the code does not look any worse in the macro case

```

#define BYTE_SIZE 8                /* A byte contains 8 bits */

int bit_count(int byte)
{
    int i, count = 0;
    for (i = 0; i < BYTE_SIZE; i++) /* loop over the bits in a byte */
        if ((byte >> i) & 1) count++; /* if this bit is a 1, add to count */
    return(count);                  /* return sum */
}

```

(a)

```

/*Macro to add up the bits in a byte and return the sum. */
#define bit_count(b) (b&1) + ((b>>1)&1) + ((b>>2)&1) + ((b>>3)&1) + \
    ((b>>4)&1) + ((b>>5)&1) + ((b>>6)&1) + ((b>>7)&1)

```

(b)

```

/*Macro to look up the bit count in a table. */
char bits[256] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, ...};
#define bit_count(b) (int) bits[b]

```

(c)

Figure 12-6. (a) A procedure for counting bits in a byte. (b) A macro to count the bits.

than in the procedure case, but it is much more efficient since it eliminates both the procedure call overhead and the loop overhead.

We can take this example one step further. Why compute the bit count at all? Why not look it up in a table? After all, there are only 256 different bytes, each with a unique value between 0 and 8. We can declare a 256-entry table, *bits*, with each entry initialized (at compile time) to the bit count corresponding to that byte value. With this approach no computation at all is needed at run time, just one indexing operation. A macro to do the job is given in Fig. 12-6(c).

This is a clear example of trading computation time against memory. However, we could go still further. If the bit counts for whole 32-bit words are needed, using our *bit_count* macro, we need to perform four lookups per word. If we expand the table to 65,536 entries, we can suffice with two lookups per word, at the price of a much bigger table.

Looking answers up in tables can be used in other ways. For example, in Chap. 7 we saw how JPEG image compression works, with fairly complex discrete cosine transformations. An alternative compression technique, GIF, uses table lookup to encode 24-bit RGB pixels. However, GIF only works on images with 256 or fewer colors. For each image to be compressed, a palette of 256 entries is constructed, each entry containing one 24-bit RGB value. The compressed

image then consists of an 8-bit index for each pixel instead of a 24-bit color value, a gain of a factor of three. This idea is illustrated for a 4×4 section of an image in Fig. 12-7. The original compressed image is shown in Fig. 12-7(a). Each value here is a 24-bit value, with 8 bits each giving the intensity for red, green, and blue. The GIF image is shown in Fig. 12-7(b). Here each value is an 8-bit index into the color palette. The color palette is stored as part of the image file, and is shown in Fig. 12-7(c). Actually, there is more to GIF, but the core of the idea is table lookup.

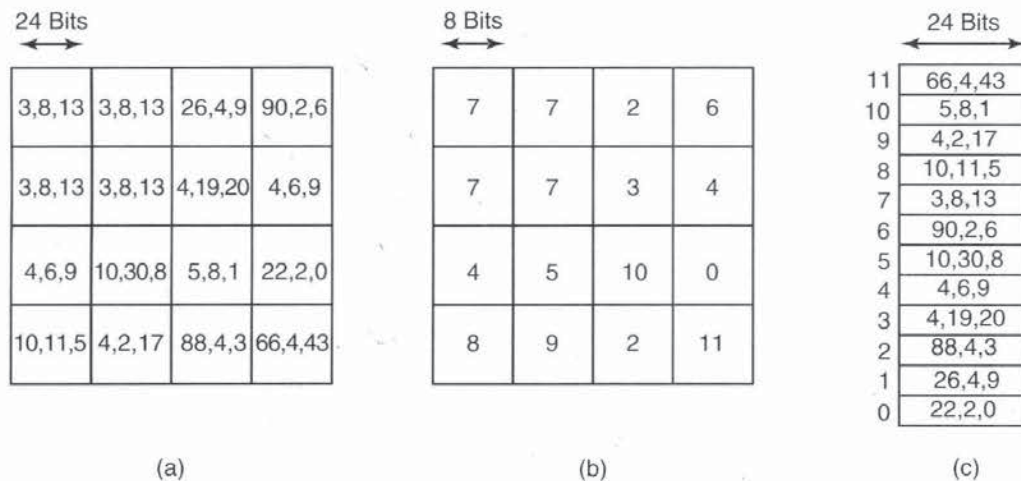


Figure 12-7. (a) Part of an uncompressed image with 24 bits per pixel. (b) The same part compressed with GIF, with 8 bits per pixel. (c) The color palette.

There is another way to reduce image size, and it illustrates a different trade-off. PostScript is a programming language that can be used to describe images. (Actually, any programming language can describe images, but PostScript is tuned for this purpose.) Many printers have a PostScript interpreter built into them to be able to run PostScript programs sent to them.

For example, if there is a rectangular block of pixels all the same color in an image, a PostScript program for the image would carry instructions to place a rectangle at a certain location and fill it with a certain color. Only a handful of bits are needed to issue this command. When the image is received at the printer, an interpreter there must run the program to construct the image. Thus PostScript achieves data compression at the expense of more computation, a different trade-off than table lookup, but a valuable one when memory or bandwidth is scarce.

Other trade-offs often involve data structures. Doubly-linked lists take up more memory than singly-linked lists, but often allow faster access to items. Hash tables are even more wasteful of space, but faster still. In short, one of the main things to consider when optimizing a piece of code is whether using different data structures would make the best time-space tradeoff.

12.4.4 Caching

A well-known technique for improving performance is caching. It is applicable any time it is likely the same result will be needed multiple times. The general approach is to do the full work the first time, and then save the result in a cache. On subsequent attempts, the cache is first checked. If the result is there, it is used. Otherwise, the full work is done again.

We have already seen the use of caching within the file system to hold some number of recently used disk blocks, thus saving a disk read on each hit. However, caching can be used for many other purposes as well. For example, parsing path names is surprisingly expensive. Consider the UNIX example of Fig. 6-39 again. To look up */usr/ast/mbox* requires the following disk accesses:

1. Read the i-node for the root directory (i-node 1).
2. Read the root directory (block 1).
3. Read the i-node for */usr* (i-node 6).
4. Read the */usr* directory (block 132).
5. Read the i-node for */usr/ast* (i-node 26).
6. Read the */usr/ast* directory (block 406).

It takes six disk accesses just to discover the i-node number of the file. Then the i-node itself has to be read to discover the disk block numbers. If the file is smaller than the block size (e.g., 1024 bytes), it takes eight disk accesses to read the data.

Some systems optimize path name parsing by caching (path, i-node) combinations. For the example of Fig. 6-39, the cache will certainly hold the first three entries of Fig. 12-8 after parsing */usr/ast/mbox*. The last three entries come from parsing other paths.

Path	I-node number
<i>/usr</i>	6
<i>/usr/ast</i>	26
<i>/usr/ast/mbox</i>	60
<i>/usr/ast/books</i>	92
<i>/usr/bal</i>	45
<i>/usr/bal/paper.ps</i>	85

Figure 12-8. Part of the i-node cache for Fig. 6-39.

When a path has to be looked up, the name parser first consults the cache and searches it for the longest substring present in the cache. For example, if the path

/usr/ast/grants/stw is presented, the cache returns the fact that */usr/ast* is i-node 26, so the search can start there, eliminating four disk accesses.

A problem with caching paths is that the mapping between file name and i-node number is not fixed for all time. Suppose that the file */usr/ast/mbox* is removed from the system and its i-node reused for a different file owned by a different user. Later, the file */usr/ast/mbox* is created again, and this time it gets i-node 106. If nothing is done to prevent it, the cache entry will now be wrong and subsequent lookups will return the wrong i-node number. For this reason, when a file or directory is deleted, its cache entry and (if it is a directory) all the entries below it must be purged from the cache.

Disk blocks and path names are not the only items that can be cached. I-nodes can be cached, too. If pop-up threads are used to handle interrupts, each one of them requires a stack and some additional machinery. These previously-used threads can also be cached, since refurbishing a used one is easier than creating a new one from scratch (to avoid having to allocate memory). Just about anything that is hard to produce can be cached.

12.4.5 Hints

Cache entries are always correct. A cache search may fail, but if it finds an entry, that entry is guaranteed to be correct and can be used without further ado. In some systems, it is convenient to have a table of **hints**. These are suggestions about the solution, but they are not guaranteed to be correct. The caller must verify the result itself.

A well-known example of hints are the URLs embedded on Web pages. Clicking on a link does not guarantee that the Web page pointed to is there. In fact, the page pointed to may have been removed years ago. Thus the information on the pointing page is really only a hint.

Hints are also used in connection with remote files. The information is the hint tells something about the remote file, such as where it is located. However, the file may have moved or been deleted since the hint was recorded, so a check is always needed to see if it is correct.

12.4.6 Exploiting Locality

Processes and programs do not act at random. They exhibit a fair amount of locality in time and space, and this information can be exploited in various ways to improve performance. One well-known example of spatial locality is the fact that processes do not jump around at random within their address spaces. They tend to use a relatively small number of pages during a given time interval. The pages that a process is actively using can be noted as its working set and the operating system can make sure that when the process is allowed to run, its working set is in memory, thus reducing the number of page faults.

The locality principle also holds for files. When a process has selected a particular working directory, it is likely that many of its future file references will be to files in that directory. By putting all the i-nodes and files for each directory close together on the disk, performance improvements can be obtained. This principle is what underlies the Berkeley Fast File System (McKusick et al., 1984).

Another area in which locality plays a role is in thread scheduling in multiprocessors. As we saw in Chap. 8, one way to schedule threads on a multiprocessor is to try to run each thread on the CPU it last used, in hopes that some of its memory blocks will still be in the memory cache.

12.4.7 Optimize the Common Case

It is frequently a good idea to distinguish between the most common case and the worst possible case and treat them differently. Often the code for the two is quite different. It is important to make the common case fast. For the worst case, if it occurs rarely, it is sufficient to make it correct.

As a first example, consider entering a critical region. Most of the time, the entry will succeed, especially if processes do not spend a lot of time inside critical regions. Windows 2000 takes advantage of this expectation by providing a Win32 API call `EnterCriticalSection` that atomically tests a flag in user mode (using TSL or equivalent). If the test succeeds, the process just enters the critical region and no kernel call is needed. If the test fails, the library procedure does a down on a semaphore to block the process. Thus in the normal case, no kernel call is needed.

As a second example, consider setting an alarm (using signals in UNIX). If no alarm is currently pending, it is straightforward to make an entry and put it on the timer queue. However, if an alarm is already pending, it has to be found and removed from the timer queue. Since the `alarm` call does not specify whether there is already an alarm set, the system has to assume worst case, that there is. However, since most of the time there is no alarm pending, and since removing an existing alarm is expensive, it is a good idea to distinguish these two cases.

One way to do this is to keep a bit in the process table that tells whether an alarm is pending. If the bit is off, the easy path is followed (just add a new timer queue entry without checking). If the bit is on, the timer queue must be checked.

12.5 PROJECT MANAGEMENT

Programmers are perpetual optimists. Most of them think that the way to write a program is to run to the keyboard and start typing. Shortly thereafter the fully debugged program is finished. For very large programs, it does not quite work like that. In the following sections we have a bit to say about managing large software projects, especially large operating system projects.

12.5.1 The Mythical Man Month

In his classic book, Fred Brooks, one of the designers of OS/360, who later moved to academia, addresses the question of why it is so hard to build big operating systems (Brooks, 1975, 1995). When most programmers see his claim that programmers can produce only 1000 lines of debugged code per *year* on large projects, they wonder whether Prof. Brooks is living in outer space, perhaps on Planet Bug. After all, most of them can remember an all nighter when they produced a 1000-line program in one night. How could this be the annual output of anybody with an $IQ > 50$?

What Brooks pointed out is that large projects, with hundreds of programmers, are completely different than small projects and that the results obtained from small projects do not scale to large ones. In a large project, a huge amount of time is consumed planning how to divide the work into modules, carefully specifying the modules and their interfaces, and trying to imagine how the modules will interact, even before coding begins. Then the modules have to be coded and debugged in isolation. Finally, the modules have to be integrated and system as a whole has to be tested. The normal case is that each module works perfectly when tested by itself, but the system crashes instantly when all the pieces are put together. Brooks estimated the work as being

- 1/3 Planning
- 1/6 Coding
- 1/4 Module testing
- 1/4 System testing

In other words, writing the code is the easy part. The hard part is figuring out what the modules should be and making module *A* correctly talk to module *B*. In a small program written by a single programmer, all that is left over is the easy part.

The title of Brooks' book comes from his assertion that people and time are not interchangeable. There is no such unit as a man-month (or a person-month). If a project takes 15 people 2 years to build, it is inconceivable that 360 people could do it in one month and probably not possible to have 60 people do it in 6 months.

There are three reasons for this effect. First, the work cannot be fully parallelized. Until the planning is done and it has been determined what modules are needed and what their interfaces will be, no coding can be even started. On a two-year project, the planning alone may take 8 months.

Second, to fully utilize a large number of programmers, the work must be partitioned into large numbers of modules so everyone has something to do. Since every module may potentially interact with every other module, the number of module-module interactions that need to be considered grows as the square of the number of modules, that is, as the square of the number of programmers. This

complexity quickly gets out of hand. Careful measurements of 63 software projects have confirmed that the trade-off between people and months is far from linear on large projects (Boehm, 1981).

Third, debugging is highly sequential. Setting ten debuggers on a problem does not find the bug ten times as fast. In fact, ten debuggers are probably slower than one because they will waste so much time talking to each other.

Brooks sums up his experience with trading-off people and time in Brooks' law:

Adding manpower to a late software project makes it later.

The problem with adding people is that they have to be trained in the project, the modules have to be redivided to match the larger number of programmers now available, many meetings will be needed to coordinate all the efforts and so on. Abdel-Hamid and Madnick (1991) confirmed this law experimentally. A slightly irreverent way of restating Brooks law is

It takes 9 months to bear a child, no matter how many women you assign to the job.

12.5.2 Team Structure

Commercial operating systems are large software projects and invariably require large teams of people. The quality of the people matters immensely. It has been known for decades that top programmers are 10× more productive than bad programmers (Sackman et al., 1968). The trouble is, when you need 200 programmers, it is hard to find 200 top programmers; you have to settle for a wide spectrum of qualities.

What is also important in any large design project, software or otherwise, is the need for architectural coherence. There should be one mind controlling the design. Brooks cites the Reims cathedral as an example of a large project that took decades to build, and in which the architects who came later subordinated their desire to put their stamp on the project to carry out the initial architect's plans. The result is an architectural coherence unmatched in other European cathedrals.

In the 1970s, Harlan Mills combined the observation that some programmers are much better than others with the need for architectural coherence to propose the **chief programmer team** paradigm (Baker, 1972). His idea was to organize a programming team like a surgical team rather than like a hog-butcher team. Instead of everyone hacking away like mad, one person wields the scalpel. Everyone else is there to provide support. For a 10-person project, Mills suggested the team structure of Fig. 12-9.

Three decades have gone by since this was proposed and put into production. Some things have changed (such as the need for a language lawyer—C is simpler

Title	Duties
Chief programmer	Performs the architectural design and writes the code
Copilot	Helps the chief programmer and serves as a sounding board
Administrator	Manages the people, budget, space, equipment, reporting, etc.
Editor	Edits the documentation, which must be written by the chief programmer
Secretaries	The administrator and editor each need a secretary
Program clerk	Maintains the code and documentation archives
Toolsmith	Provides any tools the chief programmer needs
Tester	Tests the chief programmer's code
Language lawyer	Part timer who can advise the chief programmer on the language

Figure 12-9. Mills' proposal for populating a 10-person chief programmer team.

than PL/I), but the need to have only one mind controlling the design is still true. And that one mind should be able to work 100% on designing and programming, hence the need for the support staff, although with help from the computer, a smaller staff will suffice now. But in its essence, the idea is still valid.

Any large project needs to be organized as a hierarchy. At the bottom level are many small teams, each headed by a chief programmer. At the next level, groups of teams must be coordinated by a manager. Experience shows that each person you manage costs you 10% of your time, so a full-time manager is needed for each group of 10 teams. These managers must be managed, and so on up the tree.

Brooks observed that bad news does not travel up the tree well. Jerry Saltzer of M.I.T. called this effect the **bad-news diode**. No chief programmer or manager wants to tell his boss that the project is 4 months late and has no chance whatsoever of meeting the deadline because there is a 1000-year old tradition of beheading the messenger who brings bad news. As a consequence, top management is generally in the dark about the state of the project. When it becomes obvious that the deadline cannot be met, top management responds by adding people, at which time Brooks' law kicks in.

In practice, large companies, which have had long experience producing software and know what happens if it is produced haphazardly, have a tendency to at least try to do it right. In contrast, smaller, newer companies, which are in a huge rush to get to market, do not always take the care to produce their software carefully. This haste often leads to far from optimal results.

Neither Brooks nor Mills foresaw the growth of the open source movement. Although it has had some successes, it remains to be seen if this is a viable model for producing large amounts of quality software once the novelty wears off. Recall that in its early days, radio was dominated by ham radio operators, but that

soon gave way to commercial radio and later to commercial television. What is noticeable is that the open source software projects that have been most successful have clearly used the chief programmer model of having one mind control the architectural design (e.g., Linus Torvalds for the Linux kernel and Richard Stallman for the GNU C compiler).

12.5.3 The Role of Experience

Having experienced designers is critical to an operating systems project. Brooks points out that most of the errors are not in the code, but in the design. The programmers correctly did what they were told to do. What they were told to do was wrong. No amount of test software will catch bad specifications.

Brooks' solution is to abandon the classical development model of Fig. 12-10(a) and use the model of Fig. 12-10(b). Here the idea is to first write a main program that merely calls the top-level procedures, which are initially dummies. Starting on day 1 of the project, the system will compile and run, although it does nothing. As time goes on, modules are inserted into the full system. The result of this approach is that system integration testing is performed continuously, so errors in the design show up much earlier. In effect, the learning process caused by bad design decisions starts much earlier in the cycle.

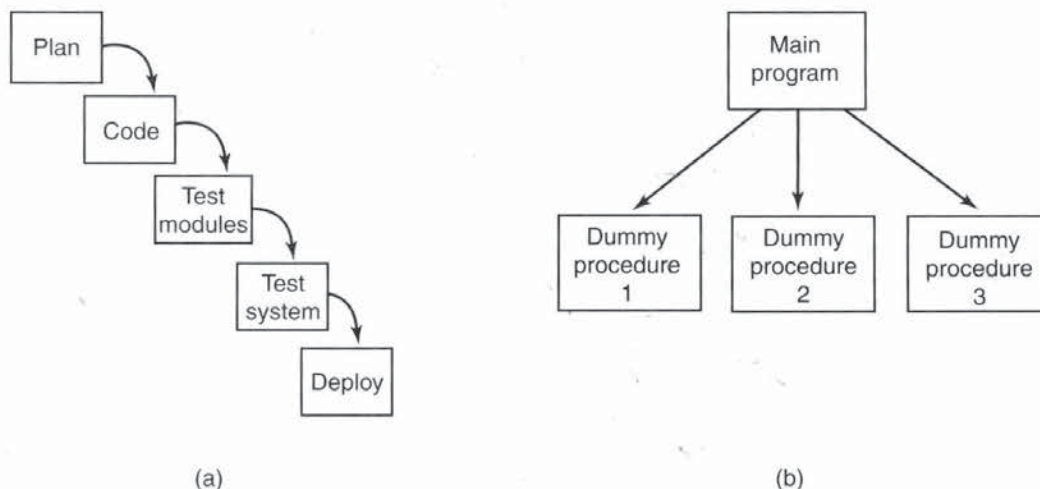


Figure 12-10. (a) Traditional software design progresses in stages. (b) Alternative design produces a working system (that does nothing) starting on day 1.

A little knowledge is a dangerous thing. Brooks observed what he called the **second system effect**. Often the first product produced by a design team is minimal because the designers are afraid it may not work at all. As a result, they are hesitant to put in many features. If the project succeeds, they build a follow-up system. Impressed by their own success, the second time the designers include all the bells and whistles that were intentionally left out the first time. As a result,

the second system is bloated and performs poorly. The third time around they are sobered by the failure of the second system and are cautious again.

The CTSS-MULTICS pair is a clear case in point. CTSS was the first general-purpose timesharing system and was a huge success despite having minimal functionality. Its successor, MULTICS was too ambitious and suffered badly for it. The ideas were good, but there were too many new things so the system performed poorly for years and was never a big commercial success. The third system in this line of development, UNIX, was much more cautious and much more successful.

12.5.4 No Silver Bullet

In addition to *The Mythical Man Month*, Brooks also wrote an influential paper called “No Silver Bullet” (Brooks, 1987). In it, he argued that none of the many nostrums being hawked by various people at the time was going to generate an order-of-magnitude improvement in software productivity within a decade. Experience shows that he was right.

Among the silver bullets that were proposed were better high-level languages, object-oriented programming, artificial intelligence, expert systems, automatic programming, graphical programming, program verification, and programming environments. Perhaps the next decade will see a silver bullet, but maybe we will have to settle for gradual, incremental improvements.

12.6 TRENDS IN OPERATING SYSTEM DESIGN

Making predictions is always difficult—especially about the future. For example, in 1899, the head of the U.S. Patent Office, Charles H. Duell, asked then-President McKinley to abolish the Patent Office (and his job!), because, as he put it: “Everything that can be invented, has been invented” (Cerf and Navasky, 1984). Nevertheless, Thomas Edison showed up on his doorstep within a few years with a couple of new items, including the electric light, the phonograph, and the movie projector. Let us put new batteries in our crystal ball and venture a guess on where operating systems are going in the near future.

12.6.1 Large Address Space Operating Systems

As machines move from 32-bit address spaces to 64-bit address spaces, major shifts in operating system design become possible. A 32-bit address space is not really that big. If you tried to divide up 2^{32} bytes by giving everybody on earth his or her own byte, there would not be enough bytes to go around. In contrast, 2^{64} is about 2×10^{19} . Now everybody gets his or her personal 3 GB chunk.

What could we do with an address space of 2×10^{19} bytes? For starters, we could eliminate the file system concept. Instead, all files could be conceptually held in (virtual) memory all the time. After all, there is enough room in there for over 1 billion full-length movies, each compressed to 4 GB.

Another possible use is a persistent object store. Objects could be created in the address space and kept there until all references to them were gone, at which time they would be automatically deleted. Such objects would be persistent in the address space, even over shutdowns and reboots of the computer. With a 64-bit address space, objects could be created at a rate of 100 MB/sec for 5000 years before we run out of address space. Of course, to actually store this amount of data, a lot of disk storage would be needed for the paging traffic, but for the first time in history, the limiting factor would be disk storage, not address space.

With large numbers of objects in the address space, it becomes interesting to allow multiple processes to run in the same address space at the same time, to share the objects in a general way. Such a design would clearly lead to very different operating systems than we now have. Some thoughts on this concept are contained in (Chase et al., 1994).

Another operating system issue that will have to be rethought with 64-bit addresses is virtual memory. With 2^{64} bytes of virtual address space and 8-KB pages we have 2^{51} pages. Conventional pages tables do not scale well to this size, so something else is needed. Inverted page tables are a possibility, but other ideas have been proposed as well (Talluri et al., 1995). In any event there is plenty of room for new research on 64-bit operating systems.

12.6.2 Networking

Current operating systems were designed for standalone computers. Networking was an afterthought and is generally accessed through special programs, such as Web browsers, FTP, or telnet. In the future, networking will probably be the basis for all operating systems. A standalone computer without a network connection will be as rare as a telephone without a connection to the network. And it is likely that multimegabit/sec connections will be the norm.

Operating systems will have to change to adapt to this paradigm shift. The difference between local data and remote data may blur to the point that virtually no one knows or cares where data are stored. Computers anywhere may be able to treat data anywhere as local data. To a limited extent, this is already true with NFS, but it is likely to become far more pervasive and better integrated.

Access to the Web, which now requires special programs (browsers) may also become completely integrated into the operating system in a seamless way. The standard way to store information may become Web pages, and these pages may contain a wide variety of nontext items, including audio, video, programs, and more, all managed as the operating system's fundamental data.

12.6.3 Parallel and Distributed Systems

Another area that is up and coming is parallel and distributed systems. Current operating systems for multiprocessors and multicomputers are just standard uniprocessor operating systems with minor adjustments to the scheduler to handle parallelism a bit better. In the future, we may see operating systems where parallelism is much more central than it now is. This effect will be enormously stimulated if desktop machines soon have two, four, or more CPUs in a multiprocessor configuration. This may lead to many application programs being designed for multiprocessors, with the concurrent demand for better operating system support for them.

Multicomputers are likely to dominate large-scale scientific and engineering supercomputers in the coming years, but the operating systems for them are still fairly primitive. Process placement, load balancing, and communication need a lot of work.

Current distributed systems are often built as middleware because existing operating systems do not provide the right facilities for distributed applications. Future ones may be designed with distributed systems in mind, so all the necessary features are already present in the operating system from the start.

12.6.4 Multimedia

Multimedia systems are clearly a rising star in the computer world. It would surprise no one if computers, stereos, televisions, and telephones all merged into a single device capable of supporting high-quality still images, audio and video, and connected to high-speed networks so these files could easily be downloaded, exchanged, and accessed remotely. The operating systems for these devices, or even for standalone audio and video devices, will have to be substantially different from current ones. In particular, real-time guarantees will be needed, and these will drive the system design. Also, consumers will be very intolerant of hourly crashes of their digital television sets, so better software quality and fault tolerance will be required. Also, multimedia files tend to be very long, so file systems will have to change to be able to handle them efficiently.

12.6.5 Battery-Powered Computers

Powerful desktop PCs, probably with 64-bit address spaces, high-bandwidth networking, multiple processors, and high-quality audio and video will no doubt be commonplace soon. Their operating systems will have to be appreciably different from current ones to handle all these demands. However, an even faster growing segment of the market is battery-powered computers, including laptops, palmtops, Webpads, and various telephone hybrids. Some of these will have wireless connections to the outside world; others will run in disconnected mode

when not docked at home. They will need different operating systems that are smaller, faster, more flexible, and more reliable than current ones. Various kinds of microkernel and extensible systems may form the basis here.

These operating systems will have to handle fully connected (i.e., wired), weakly connected (i.e., wireless), and disconnected operation, including data hoarding before going offline and consistency resolution when going back online better than current systems. They will also have to handle the problems of mobility better than current systems (e.g., find a laser printer, log onto it, and send it a file by radio). Power management, including extensive dialogs between the operating system and applications about how much battery power is left and how it can be best used, will be essential. Dynamic adaptation of applications to handle the limitations of tiny screens may become important. Finally, new input and output modes, including handwriting and speech, may require new techniques in the operating system to improve the quality. It is unlikely the operating system for a battery-powered, handheld wireless, voice-operated computer will have much in common with that of a desktop 64-bit four-CPU multiprocessor with a gigabit fiber-optic network connection. And of course, there will be innumerable hybrid machines with their own requirements.

12.6.6 Embedded Systems

One final area in which new operating systems will proliferate is embedded systems. The operating systems inside washing machines, microwave ovens, dolls, transistor (Internet?) radios, MP3 players, camcorders, elevators, and pacemakers will differ from all of the above and most likely from each other. Each one will probably be carefully tailored for its specific application, since it is unlikely anyone will ever stick a PCI card into a pacemaker to turn it into an elevator controller. Since all embedded systems run only a limited number of programs, known at design time, it may be possible to make optimizations not possible in general-purpose systems.

A promising idea for embedded systems are the extensible operating systems (e.g., Paramacium and Exokernel). These can be made as lightweight or heavyweight as the application in question demands, but in a consistent way across applications. Since embedded systems will be produced by the hundreds of millions, this will be a major market for new operating systems.

12.7 SUMMARY

Designing an operating system starts with determining what it should do. The interface should be simple, complete, and efficient. It should have a clear user interface paradigm, execution paradigm, and data paradigm.

The system should be well structured, using one of several known techniques, such as layering or client-server. The internal components should be orthogonal to one another and clearly separate policy from mechanism. Considerable thought should be given to issues such as static versus dynamic data structure, naming, binding time, and order of implementing modules.

Performance is important, but optimizations should be chosen carefully so as not to ruin the system's structure. Space-time trade-offs, caching, hints, exploiting locality, and optimizing the common case are often worth doing.

Writing a system with a couple of people is different than producing a big system with 300 people. In the latter case, team structure and project management play a crucial role in the success or failure of the project.

Finally, operating systems will have to change in the coming years to follow new trends and meet new challenges. These may include 64-bit address spaces, massive connectivity, desktop multiprocessors, multimedia, handheld wireless computers, and a huge variety of embedded systems. The coming years will be exciting times for operating system designers.

PROBLEMS

1. Moore's law describes a phenomenon of exponential growth similar to the population growth of an animal species introduced into a new environment with abundant food and no natural enemies. In nature, an exponential growth curve is likely eventually to become a sigmoid curve with an asymptotic limit when food supplies become limiting or predators learn to take advantage of new prey. Discuss some factors that may eventually limit the rate of improvement of computer hardware.
2. In Fig. 12-1, two paradigms are shown, algorithmic and event driven. For each of the following kinds of programs, which paradigm is likely to be easiest to use:
 - (a) A compiler.
 - (b) A photo editing program.
 - (c) A payroll program.
3. On some of the early Apple Macintoshes, the GUI code was in ROM. Why?
4. Hierarchical file names always start at the top of the tree. Consider, for example, the file name `/usr/ast/books/mos2/chap-12` rather than `chap-12/mos2/books/ast/usr`. In contrast, DNS names start at the bottom of the tree and work up. Is there some fundamental reason for this difference?
5. Corbató's dictum is that the system should provide minimal mechanism. Here is a list of POSIX calls that were also present in UNIX Version 7. Which ones are redundant, that is, could be removed with no loss of functionality because simple combinations of other ones could do the same job with about the same performance? Access, alarm, chdir, chmod, chown, chroot, close, creat, dup, exec, exit, fcntl, fork, fstat, ioctl, kill, link, lseek, mkdir, mknod, open, pause, pipe, read, stat, time, times, umask, unlink, utime, wait, and write.

6. Suppose that layers 3 and 4 in Fig. 12-2 were exchanged. What implications would that have for the design of the system?
7. In a microkernel-based client-server system, the microkernel just does message passing and nothing else. Is it possible for user processes to nevertheless create and use semaphores? If so, how? If not, why not?
8. Careful optimization can improve system call performance. Consider the case in which one system call is made every 10 msec. The average time of a call is 2 msec. If the system calls can be speeded up by a factor of two, how long does a process that took 10 sec to run now take?
9. Give a short discussion of mechanism versus policy in the context of retail stores.
10. Operating systems often do naming at two different levels: external and internal. What are the differences between these names with respect to
 - (a) Length
 - (b) Uniqueness
 - (c) Hierarchies
11. One way to handle tables whose size is not known in advance is to make them fixed, but when one fills up, to replace it with a bigger one, copy the old entries over to the new one, then release the old one. What are the advantages and disadvantages of making the new one $2\times$ the size of the original one, as compared to making it only $1.5\times$ as big?
12. In Fig. 12-4, a flag, *found*, is used to tell whether the PID was located. Would it have been possible to forget about *found* and just test *p* at the end of the loop to see whether it got to the end or not?
13. In Fig. 12-5, the differences between the Pentium and the UltraSPARC are hidden by conditional compilation. Could this same approach be used to hide the difference between Pentiums with an IDE disk as the only disk and Pentiums with a SCSI disk as the only disk? Would it be a good idea?
14. Indirection is a way of making an algorithm more flexible. Does it have any disadvantages, and if so, which ones?
15. Can reentrant procedures have private static global variables? Discuss your answer.
16. The macro of Fig. 12-6(b) is clearly much more efficient than the procedure of Fig. 12-6(a). One disadvantage, however, is that it is hard to read. Are there any other disadvantages? If so, what are they?
17. Suppose that we need a way of computing whether the number of bits in a 32-bit word is odd or even. Devise an algorithm for performing this computation as fast as possible. You may use up to 256 KB of RAM for tables if need be. Write a macro to carry out your algorithm. *Extra Credit:* Write a procedure to do the computation by looping over the 32 bits. Measure how many times faster your macro is than the procedure.
18. In Fig. 12-7, we saw how GIF files use 8-bit values to index into a color palette. The same idea can be used with a 16-bit-wide color palette. Under what circumstances, if any, might a 24-bit color palette be a good idea?

19. One disadvantage of GIF is that the image must include the color palette, which increases the file size. What is the minimum image size for which an 8-bit-wide color palette breaks even? Now repeat this question for a 16-bit-wide color palette.
20. In the text we showed how caching path names can result in a significant speedup when looking up path names. Another technique that is sometimes used is having a daemon program that opens all the files in the root directory and keeps them open permanently, in order to force their i-nodes to be in memory all the time. Does pinning the i-nodes like this improve the path lookup even more?
21. Even if a remote file has not been removed since a hint was recorded, it may have been changed since the last time it was referenced. What other information might it be useful to record?
22. Consider a system that hoards references to remote files as hints, for example as (name, remote-host, remote-name). It is possible that a remote file will quietly be removed and then replaced. The hint may then retrieve the wrong file. How can this problem be made less likely to occur?
23. In the text it is stated that locality can often be exploited to improve performance. But consider a case where a program reads input from one source and continuously outputs to two or more files. Can an attempt to take advantage of locality in the file system lead to a decrease in efficiency here? Is there a way around this?
24. Fred Brooks claims that a programmer can write 1000 lines of debugged code per year, yet the first version of MINIX (13,000 lines of code) was produced by one person in under three years. How do you explain this discrepancy?
25. Using Brooks' figure of 1000 lines of code per programmer per year, make an estimate of the amount of money it took to produce Windows 2000. Assume that a programmer costs \$100,000 per year (including overhead, such as computers, office space, secretarial support, and management overhead). Do you believe this answer? If not, what might be wrong with it?
26. As memory gets cheaper and cheaper, one could imagine a computer with a big battery-backed up RAM instead of a hard disk. At current prices, how much would a low-end RAM-only PC cost? Assume that a 1-GB RAM-disk is sufficient for a low-end machine. Is this machine likely to be competitive?
27. Name some features of a conventional operating system that are not needed in an embedded system used inside an appliance.
28. Write a procedure in C to do a double precision addition on two given parameters. Write the procedure using conditional compilation in such a way that it works on 16-bit machines and also on 32-bit machines.
29. Write programs that enter randomly generated short strings into an array and then can search the array for a given string using (a) a simple linear search (brute force), and (b) a more sophisticated method of your choice. Recompile your programs for array sizes ranging from small to as large as you can handle on your system. Evaluate the performance of both approaches). Where is the breakeven point?
30. Write a program to simulate an in-memory file system.

13

READING LIST AND BIBLIOGRAPHY

In the previous 12 chapters we have touched upon a variety of topics. This chapter is intended as an aid to readers interested in pursuing their study of operating systems further. Section 13.1 is a list of suggested readings. Section 13.2 is an alphabetical bibliography of all books and articles cited in this book.

In addition to the references given below, the *Proceedings of the n-th ACM Symposium on Operating Systems Principles* (ACM) held every other year and the *Proceedings of the n-th International Conference on Distributed Computing Systems* (IEEE) held every year are good places to look for recent papers on operating systems. So is the *USENIX Symposium on Operating Systems Design and Implementation*. Furthermore, *ACM Transactions on Computer Systems* and *Operating Systems Review* are two journals that often have relevant articles.

13.1 SUGGESTIONS FOR FURTHER READING

In the following sections, we give some suggestions for further reading. Unlike the papers cited in the sections entitled “RESEARCH ON ...” in the text, which were about current research, these references are mostly introductory or tutorial in nature. They can serve to present material present in this book from a different perspective or with a different emphasis, however.

13.1.1 Introduction and General Works

Kavi et al., “Computer Systems Research: The Pressure is On”

Where is systems research going? What is important now? How about developing robust, predictable systems? How about billion-transistor chips and billion-user worldwide systems? These are some of the questions and answers presented in this paper.

Milojicic, “Operating Systems: Now and in the Future,”

Suppose you were to ask six of the world’s leading experts in operating systems a series of questions about the field and where it was going. Would you get the same answers? *Hint*: No. Find out what they said here.

Silberschatz et al., *Applied Operating System Concepts*

A general textbook on operating systems. It covers processes, storage management, distributed systems, and protection. Three case studies are given: UNIX, Linux, and Windows NT. The cover is full of dinosaurs. What, if anything, this has to do with operating systems anno 2000 is unclear.

Stallings, *Operating Systems, 4th Ed.*

Still another textbook on operating systems. It covers all the traditional topics, and also includes a small amount of material on distributed systems.

Stevens, *Advanced Programming in the UNIX Environment*

This book tells how to write C programs that use the UNIX system call interface and the standard C library. Examples are based on the System V Release 4 and the 4.4BSD versions of UNIX. The relationship of these implementations to POSIX is described in detail.

13.1.2 Processes and Threads

Andrews and Schneider, “Concepts and Notations for Concurrent Programming”

A tutorial and survey of processes and interprocess communication, including busy waiting, semaphores, monitors, message passing, and other techniques. The article also shows how these concepts are embedded in various programming languages.

Ben-Ari, *Principles of Concurrent Programming*

This little book is entirely devoted to the problems of interprocess communication. There are chapters on mutual exclusion, semaphores, monitors, and the dining philosophers problem, among others.

Silberschatz et al., *Applied Operating System Concepts*

Chapters 4 through 6 cover processes and interprocess communication, including scheduling, critical sections, semaphores, monitors, and classical inter-process communication problems.

13.1.3 Deadlocks

Coffman et al., “System Deadlocks”

A short introduction to deadlocks, what causes them, and how they can be prevented or detected.

Holt, “Some Deadlock Properties of Computer Systems”

A discussion of deadlocks. Holt introduces a directed graph model that can be used to analyze some deadlock situations.

Isloor and Marsland, “The Deadlock Problem: An Overview”

A tutorial on deadlocks, with special emphasis on database systems. A variety of models and algorithms are covered.

13.1.4 Memory Management

Denning, “Virtual Memory”

A classic paper on many aspects of virtual memory. Denning was one of the pioneers in this field, and was the inventor of the working set concept.

Denning, “Working Sets Past and Present”

A good overview of numerous memory management and paging algorithms. A comprehensive bibliography is included.

Knuth, *The Art of Computer Programming* Vol. 1

First fit, best fit, and other memory management algorithms are discussed and compared in this book.

Silberschatz et al., *Applied Operating System Concepts*

Chapters 9 and 10 deal with memory management, including swapping, paging, and segmentation. A variety of paging algorithms are mentioned.

13.1.5 Input/Output

Chen et al., “RAID: High Performance Reliable Secondary Storage”

The use of multiple disk drives in parallel for fast I/O is a trend in high-end systems. The authors discuss this idea and examine different organizations in terms of performance, cost, and reliability.

Computer, March 1994

This issue of *Computer* contains eight articles on advanced I/O, and covers simulation, high performance storage, caching, I/O for parallel computers, and multimedia.

Geist and Daniel, "A Continuum of Disk Scheduling Algorithms"

A generalized disk arm scheduling algorithm is presented. Extensive simulation and experimental results are given.

Gibson and Van Meter, "Network Attached Storage"

The Internet is driving the need for vast amounts of storage for Web, database, and other servers. This situation has led to various designs in which free-standing storage can be attached to a network. In this paper, several architectures to accomplish this goal are discussed.

Ng, "Advances in Disk Technology: Performance Issues"

The various factors in disk performance, such as linear bit density, rpm, number of heads, and number of sectors/track can be traded off in various ways to affect system performance. These issues and their impact on performance are covered in this easy-to-read paper on disk technology.

Ruemmler and Wilkes, "An Introduction to Disk Drive Modeling"

The first part of this article is about modern disk drives and how they work inside, including topics such as seeking, zoning, track skewing, sparing, caching, read ahead, and much more. The second part is about modeling disk drives.

Walker and Cragon, "Interrupt Processing in Concurrent Processors"

Implementing precise interrupts on superscalar computers is a challenging activity. The trick is to serialize the state and do it quickly. A number of the design issues and trade-offs are discussed here.

13.1.6 File Systems

Harbron, *File Systems*

A book on file system design, applications, and performance. Both structure and algorithms are covered.

McKusick et al., "A Fast File System for UNIX"

The UNIX file system was completely redone for 4.2 BSD. This paper describes the design of the new file system, with emphasis on its performance.

Silberschatz et al., *Applied Operating System Concepts*

Chapter 11 is about file systems. It covers file operations, access methods, directories, and implementation, among other topics.

Stallings, *Operating Systems, 2nd Ed.*

Chapter 14 contains a fair amount of material about the security environment especially about hackers, viruses, and other threats.

13.1.7 Multimedia Operating Systems

ACM Computing Surveys, Dec. 1995

This issue of *ACM Computing Surveys* contains 21 short papers on various aspects of multimedia, ranging from low-level technical issues to high-level applications issues.

Computer, May 1995

The theme of this issue of *Computer* is multimedia and it contains six articles on the subject. After a brief introduction, the papers cover interactive television, multimedia servers, research management, and applications in medicine and education.

Lee, "Parallel Video Servers: A Tutorial"

Many organizations want to offer video on demand, which creates a need for scalable, fault-tolerant parallel video servers. The major issues of how to build them are covered here, including server architecture, striping, placement policies, load balancing, redundancy, protocols, and synchronization.

Leslie et al., "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications,"

Many attempts at implementing multimedia have been based on adding features to an existing operating system. An alternative approach is to start all over again, as described here, and build a new operating system for multimedia from scratch, with no need to be backward compatible for anything. The result is a fairly different design than conventional systems.

Reddy, "I/O Issues in a Multimedia System"

When people talk about computer performance, they usually mean CPU performance. However, for multimedia, I/O performance is at least as important and that is the subject of this paper. Disk scheduling, buffer space trade-offs, and admission control are among the topics covered here.

Sitaram and Dan, "Multimedia Servers"

Multimedia servers have many differences with regular file servers. The authors discuss the differences in detail, especially in the areas of scheduling, the storage subsystem, and caching.

13.1.8 Multiple Processor Systems

Ahmad, "Gigantic Clusters: Where Are They and What Are They Doing?"

To get an idea of the state-of-the-art in large multicomputers, this is a good place to look. It describes the idea and gives an overview of some of the larger systems currently in operation. Given the working of Moore's law, it is a reasonable bet that the sizes mentioned here will double about every 2 years or so.

Bhoedjang et al., "User-Level Network Interface Protocols"

More and more multicomputers are putting the network interface board in user space to improve performance. Doing so raises many design issues, eleven of which are discussed here. A number of actual systems are also compared.

Computer, Dec 1996

This issue of *Computer* contains eight papers on multiprocessors. One of them is a tutorial on the semantics of shared memory, but the other seven deal with multiprocessor applications and performance.

Dubois et al., "Synchronization, Coherence, and Event Ordering in Multiprocessors"

A tutorial on synchronization in shared-memory multiprocessor systems. However, some of the ideas are equally applicable to single processor and distributed memory systems as well.

Kwok and Ahmad "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors"

Optimal job scheduling of a multicomputer or multiprocessor is possible when the characteristics of all the jobs are known in advance. The problem is that optimal scheduling takes far too long to compute. In this paper, the authors discuss and compare 27 known algorithms for attacking this problem in different ways.

Langendoen et al., "Models for Asynchronous Message Handling"

High performance in multicomputers depends critically on the performance of the message-passing system, especially how incoming messages are handled. The main choices are active messages, upcalls, and pop-up threads. The authors describe all three of them in this paper and then present experimental results comparing them on the same hardware platform.

Protic et al., *Distributed Shared Memory: Concepts and Systems*

For an introduction to distributed shared memory, this collection of 28 previously published papers is a good start. It collects many of the classic papers on models, algorithms, and implementations into a single convenient book.

Stenstrom et al., "Trends in Shared Memory Multiprocessing,"

Which way are multiprocessors going? The authors believe the future lies more in small-scale multiprocessors than large-scale ones. They also discuss models, architectures, and parallel software.

Waldo, "Alive and Well: Jini Technology Today"

For a quick introduction to Jini, its components, and how they fit together, this paper is a good starting place. Perhaps indicative of how information will be disseminated in the future, it has no bibliography of published papers, but instead provides the URLs of many Web resources on Jini.

13.1.9 Security

Computer, Feb 2000

The theme of this issue of *Computer* is biometrics, with six papers on the subject. They range from an introduction to the subject, through various specific technologies, to a paper dealing with the legal and privacy issues.

Denning, "The United States vs. Craig Neidorf"

When a young hacker discovered and published information about how the telephone system works, he was indicted for computer fraud. This article describes the case, which involved many fundamental issues, including freedom of speech. The article is followed by some dissenting views and a rebuttal by Denning.

Denning, *Information Warfare and Security*

Information has become a weapon of war, both military and corporate. The participants not only try to attack the other side's information systems, but safeguard their own as well. In this fascinating book, the author covers every conceivable topic relating to offensive and defensive strategy, from data diddling to packet sniffers. A must read for anyone seriously interested in computer security.

Hafner and Markoff, *Cyberpunk*

Three compelling tales of young hackers breaking into computers around the world are told here by the *New York Times* computer reporter who broke the Internet worm story (Markoff). *Computer*, Feb 2000

Johnson and Jajodia, "Exploring Steganography: Seeing the Unseen"

Steganography has a long history, going back to the days when the writer would shave the head of a messenger, write a message on the shaved head, and send him off after the hair grew back. Although current techniques are often hairy, they are also digital. For a thorough introduction to the subject as currently practiced, this paper is the place to start.

Ludwig, *The Giant Black Book of Computer Viruses*, 2nd ed.

If you want to write antivirus software and need to understand how viruses work down to the bit level, this is the book for you. Every kind of virus is discussed at length and actual code for most of them is supplied on a floppy disk. A thorough knowledge of programming the Pentium in assembly language is a must, however.

Milojicic, "Security and Privacy,"

Security has many facets including operating systems, networks, implications for privacy, and more. In this article, six security experts are interviewed on their thoughts on the subject.

Nachenberg, "Computer Virus-Antivirus Coevolution"

As soon as the antivirus developers find a way to detect and neutralize some class of computer virus, the virus writers go them one better and improve the virus. The cat-and-mouse game played by the virus and antivirus sides is discussed here. The author is not optimistic about the antivirus writers winning the war, which is bad news for computer users.

Pfleeger, *Security in Computing*, 2nd ed.

Although a number of books on computer security have been published, most of them cover only network security. This book does that, but also has three chapters on operating systems security, which make it a good reference for more background on that subject.

13.1.10 UNIX and Linux

Bovet and Cesati, *Understanding the Linux Kernel*

This book is probably the best overall discussion of the Linux kernel. It covers processes, memory management, file systems, signals, and much more.

IEEE, *Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*

This is the standard. Some parts are actually quite readable, especially Annex B, "Rationale and Notes," which often sheds light on why things are done as they are. One advantage of referring to the standards document is that, by definition, there are no errors. If a typographical error in a macro name makes it through the editing process it is no longer an error, it is official.

Lewine, *POSIX Programmer's Guide*

This book describes the POSIX standard in more detail than the standards document, and includes discussions on how to convert older programs to POSIX and how to develop new programs for the POSIX environment, with examples.

Maxwell, *Linux Core Kernel Commentary*

The first 400 pages of this book contain a subset of the Linux kernel code. The last 150 pages consists of comments on the code, very much in the style of John Lions' classic book (1996). If you want to understand the Linux kernel in all its gory detail, this is the place to begin, but be warned: reading 40,000 lines of C is not for everyone.

McKusick et al., *The Design and Implementation of the 4.4 BSD Operating System*

The title says it all. But actually, the book is more general than that since all UNIX systems are fairly similar internally. This is an excellent reference for anyone wanting to learn about the internal workings of UNIX in general.

13.1.11 Windows 2000

Cusumano and Selby, "How Microsoft Builds Software"

Have you ever wondered how anyone could write a 29-million-line program (like Windows 2000) and have it work at all? To find out how Microsoft's build and test cycle is used to manage very large software projects, take a look at this paper.

Norton et al., *Complete Guide to Windows 2000 Professional*

If you are looking for a book that discusses how to set up and use Windows 2000 but also discusses many of the advanced features in some detail, such as the registry, FAT and NTFS file systems, ActiveX, DCOM, and networking this is a good choice. It lies in between the mass of books that tell where to click to get which effect and Solomon and Russinovich's book

Rector and Newcomer, *Win32 Programming*

If you are looking for one of those 1500-page books giving a summary of how to write Windows programs, this is not a bad start. It covers windows, devices, graphical output, keyboard and mouse input, printing, memory management, libraries, and synchronization, among many other topics. It requires knowledge of C or C++.

Solomon and Russinovich, *Inside Windows 2000, 3rd ed.*

If you want to learn how to use Windows 2000, there are hundreds of books out there. If you want to know how Windows 2000 works inside, this is your best bet. It covers numerous internal algorithms and data structures and in considerable technical detail. No other book comes close.

13.1.12 Design Principles

Brooks, “The Mythical Man Month: Essays on Software Engineering”

Fred Brooks was one of the designers of IBM’s OS/360. He learned the hard way what works and what does not work. The advice given in this witty, amusing, and informative book is as valid now as it was a quarter of a century ago when he first wrote it down.

Cooke et al., “UNIX and Beyond: An Interview with Ken Thompson”

Designing an operating system is much more of an art than a science. Consequently, listening to experts in the field is a good way to learn about the subject. They do not come much more expert than Ken Thompson, co-designer of UNIX, Inferno, and Plan 9. In this wide-ranging interview, Thompson gives his thoughts on where we came from and where we are going in the field.

Corbató, “On Building Systems That Will Fail”

In his Turing Award lecture, the father of timesharing addresses many of the same concerns that Brooks does in the *Mythical Man-Month*. His conclusion is that all complex systems will ultimately fail, and that to have any chance for success at all, it is absolutely essential to avoid complexity and strive for simplicity and elegance in design.

Crowley, *Operating Systems: A Design-Oriented Approach*

Most textbooks on operating systems just describe the basic concepts (processes, virtual memory, etc.) and a few examples, but say nothing about how to design an operating system. This one is unique in devoting four chapters to the subject.

Lampson, “Hints for Computer System Design”

Butler Lampson, one of the world’s leading designers of innovative operating systems, has collected many hints, suggestions, and guidelines from his years of experience and put them together in this entertaining and informative article. Like Brooks’ book, this is required reading for every aspiring operating system designer.

Wirth, “A Plea for Lean Software”

Niklaus Wirth, a famous and experienced system designer, makes the case here for lean and mean software based on a few simple concepts, instead of the bloated mess that much commercial software is. He makes his point by discussing his Oberon system, a network-oriented, GUI-based operating system that fits in 200 KB, including the Oberon compiler and text editor.

13.2 ALPHABETICAL BIBLIOGRAPHY

- ABDEL-HAMID, T., and MADNICK, S.: *Software Project Dynamics: An Integrated Approach*, Upper Saddle River, NJ: Prentice Hall, 1991.
- ABRAM-PROFETA, E.L., and SHIN, K.G.: "Providing Unrestricted VCR Functions in Multicast Video-on-Demand Servers," *Proc. Int'l Conf. on Multimedia Comp. Syst.*, IEEE, pp. 66-75, 1998.
- ABUTALEB, A., and LI, V.O.K.: "Paging Strategy Optimization in Personal Communication Systems Wireless Networks," *Wireless Networks*, vol. 3, pp. 195-204, Aug. 1997.
- ACCETTA, M., BARON, R., GOLUB, D., RASHID, R., TEVANIAN, A., and YOUNG, M.: "Mach: A New Kernel Foundation for UNIX Development," *Proc. Summer 1986 USENIX Conf.*, USENIX, pp. 93-112, 1986.
- ADAMS, G.B. III, AGRAWAL, D.P., and SIEGEL, H.J.: "A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks," *Computer*, vol. 20, pp. 14-27, June 1987.
- ADAMS, A., and SASSE, M.A.: "Taming the Wolf in Sheep's Clothing," *Proc. Seventh Int'l Conf. on Multimedia*, ACM, pp. 101-107, 1999.
- AHMAD, I.: "Gigantic Clusters: Where Are They and What Are They Doing?" *IEEE Concurrency*, vol. 8, pp. 83-85, April-June 2000.
- ALEXANDROV, A.D., IBEL, M., SCHAUSER, K.E., and SCHEIMAN, C.J.: "UFO: a Personal Global File System Based on User-Level Extensions to the Operating System," *Trans. on Computer Systems*, vol. 16, pp. 207-233, Aug. 1998.
- ALFIERI, R.A.: "An Efficient Kernel-Based Implementation of POSIX Threads," *Proc. Summer 1994 USENIX Tech. Conf.*, USENIX, pp. 59-72, June 1994.
- ALVAREZ, G.A., BURKHARD, W.A., and CRISTIAN, F.: "Tolerating Multiple Failures in RAID Architectures with Optimal Storage And Uniform Declustering," *Proc. 24th Int'l Symp. on Computer Architecture*, ACM, pp. 62-72, 1997.
- ANDERSON, T.E.: "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. on Parallel and Distr. Systems*, vol. 1, pp. 6-16, Jan. 1990.
- ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, E.D., and LEVY, H.M.: "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism," *ACM Trans. on Computer Systems*, vol. 10, pp. 53-79, Feb. 1992.
- ANDREWS, G.R.: *Concurrent Programming—Principles and Practice*, Redwood City, CA: Benjamin/Cummings, 1991.
- ANDREWS, G.R., and SCHNEIDER, F.B.: "Concepts and Notations for Concurrent Programming," *Computing Surveys*, vol. 15, pp. 3-43, March 1983.
- ARON, M., and DRUSCHEL, P.: "Soft Timers: Efficient Microsecond Software Timer Support for Network Processing," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 223-246, 1999.

- ARORA, A.S., BLUMOFÉ, R.D., and PLAXTON, C.G.:** "Thread Scheduling for Multiprogrammed Multiprocessors," *Proc. Tenth Symp. on Parallel Algorithms and Architectures*, ACM, pp. 119-129, 1998.
- BAKER, F.T.:** "Chief Programmer Team Management of Production Programming," *IBM Systems Journal*, vol. 11, pp. 1, 1972.
- BAKER-HARVEY, M.:** "ETI Resource Distributor: Guaranteed Resource Allocation and Scheduling in Multimedia Systems," *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 131-144, 1999.
- BALA, K., KAASHOEK, M.F., WEIHL, W.:** "Software Prefetching and Caching for Translation Lookaside Buffers," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 243-254, 1994.
- BALLINTIJN, G., VAN STEEN, M., and TANENBAUM, A.S.:** "Scalable Naming in Global Middleware," *Proc. 13th Int'l Conf. on Parallel and Distributed Systems*, ISCA, pp. 624-631, 2000.
- BAYS, C.:** "A Comparison of Next-Fit, First-Fit, and Best-Fit," *Commun. of the ACM*, vol. 20, pp. 191-192, March 1977.
- BECK, M., BOHME, H., DZIADZKA, M., KUNITZ, U., MAGNUS, R., VERWORNER, D.:** *Linux Kernel Internals*, 2nd ed., Reading, MA: Addison-Wesley, 1998.
- BELADY, L.A., NELSON, R.A., and SHEDLER, G.S.:** "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine," *Commun. of the ACM*, vol. 12, pp. 349-353, June 1969.
- BELL, D., and LA PADULA, L.:** "Secure Computer Systems: Mathematical Foundations and Model," Technical Report MTR 2547 v2, Mitre Corp., Nov. 1973.
- BEN-ARI, M.:** *Principles of Concurrent Programming*, Upper Saddle River, NJ: Prentice Hall International, 1982.
- BERNHARDT, C., and BIRSACK, E.W.:** "The Server Array: A Scalable Video Server Architecture," in *High-Speed Networking for Multimedia Applications*, Amsterdam: Kluwer Publishers, 1996.
- BERSHAD, B.N., SAVAGE, S., PARDYAK, P., BECKER, D., FIUCZYNSKI, M., and SIRER, E.G.:** "Protection is a Software Issue," *Proc. Hot Topics in Operating Systems V*, IEEE, pp. 62-65, 1995a.
- BERSHAD, B.N., SAVAGE, S., PARDYAK, P., SIRER, E.G., FIUCZYNSKI, M., BECKER, D., CHAMBERS, C., and EGGERS, S.:** "Extensibility, Safety, and Performance in the SPIN Operating System," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 267-284, 1995b.
- BHOEDJANG, R.A.F.:** *Communication Architectures for Parallel-Programming Systems*, Ph.D. Thesis, Vrije Universiteit, Amsterdam, The Netherlands, 2000.
- BHOEDJANG, R.A.F., RUHL, T., and BAL, H.E.:** "User-Level Network Interface Protocols," *Computer*, vol. 31, pp. 53-60, Nov. 1998.

- BHUYAN, L.N., YANG, Q., and AGRAWAL, D.P.:** "Performance of Multiprocessor Interconnection Networks," *Computer*, vol. 22, pp. 25-37, Feb. 1989.
- BIBA, K.:** "Integrity Considerations for Secure Computer Systems," Technical Report 76-371, U.S. Air Force Electronic Systems Division, 1977.
- BIRRELL, A.D., and NELSON, B.J.:** "Implementing Remote Procedure Calls," *ACM Trans. on Computer Systems*, vol. 2, pp. 39-59, Feb. 1984.
- BISDIKIAN, C.C., and PATEL, B.V.:** "Issues on Movie Allocation in Distributed Video-on-Demand Systems," *Proc. Int'l Conf. on Commun.*, IEEE, pp. 250-255, 1995.
- BLAUM, M., BRADY, J., BRUCK, J., and MENON, J.:** "EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures," *Proc. 21st Int'l Symp. on Computer Architecture*, ACM, pp. 245-254, 1994.
- BLUMOFE, R.D., and LEISERSON, C.E.:** "Scheduling Multithreaded Computations by Work Stealing," *Proc. 35th Annual Symp. on Foundations of Computer Science*, IEEE, pp. 356-368, Nov. 1994.
- BOEHM, B.W.:** *Software Engineering Economics*, Upper Saddle River, NJ: Prentice Hall, 1981.
- BOLOSKY, W.J., FITZGERALD, R.P., and DOUCEUR, J.R.:** "Distributed Schedule Management in the Tiger Video Fileserver," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 212-223, 1997.
- BORN, G.:** *Inside the Microsoft Windows 98 Registry*, Redmond, WA: Microsoft Press, 1998.
- BOVET, D.P., and CESATI, M.:** *Understanding the Linux Kernel*, Sebastopol, CA: O'Reilly & Associates, 2000.
- BRANDWEIN, R., KATSEFF, H., MARKOWITZ, R., MORTENSON, R., and ROBINSON, B.:** "Nemesis: Multimedia Information Delivery," *Proc. Second ACM Int'l Conf. on Multimedia*, ACM, pp. 473-481, 1994.
- BRICKER, A., GIEN, M., GUILLEMONT, M., LIPKIS, J., ORR, D., and ROZIER, M.:** "A New Look at Microkernel-Based UNIX Operating Systems: Lessons in Performance and Compatibility," *Proc. EurOpen Spring '91 Conf.*, EurOpen, pp. 13-32, 1991.
- BRINCH HANSEN, P.:** "The Programming Language Concurrent Pascal," *IEEE Trans. on Software Engineering*, vol. SE-1, pp. 199-207, June 1975.
- BROOKS, F. P., Jr.:** *The Mythical Man-Month: Essays on Software Engineering*, Reading, MA: Addison-Wesley, 1975.
- BROOKS, F.P., Jr.:** "No Silver Bullet—Essence and Accident in Software Engineering," *Computer*, vol. 20, pp. 10-19, April 1987.
- BROOKS, F. P., Jr.:** *The Mythical Man-Month: Essays on Software Engineering*, 20th Anniversary edition, Reading, MA: Addison-Wesley, 1995.

- BUCHANAN, M., and CHIEN, A.: "Coordinated Thread Scheduling for Workstation Clusters Under Windows NT," *The USENIX Windows NT Workshop*, USENIX, pp. 47-??, 1997.
- BUGNION, E., DEVINE, S., GOVIL, K., and ROSENBLUM, M.: "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *Trans. on Computer Systems*, vol. 15, pp. 412-447, Nov. 1997.
- BUZZARD, G., JACOBSON, D., MACKEY, M., MAROVICH, S., and WILKES, J.: "An Implementation of the Hamlyn Sender-Managed Interface Architecture," *Proc. Second Symp. on Operating System Design and Implementation*, USENIX, pp. 245-259, Oct. 1996.
- CANT, C.: *Writing Windows WDM Device Drivers*, Lawrence, KS: R&D Books, 1999.
- CAO, P., LIN, S.B., VENKATARAMAN, S., and WILKES, J.: "The TickerTAIP Parallel RAID Architecture," *Trans. on Computer Systems*, vol. 12, pp. 236-269, Aug. 1994.
- CAO, P., FELTEN, E.W., KARLIN, A.R., and LI, K.: "A Study of Integrated Prefetching and Caching Strategies," *Proc. SIGMETRICS Joint Int'l Conf. on Measurement and Modeling of Computer Systems*, ACM, pp. 188-197, 1995.
- CARLEY, L.R., GANGER, G.R., and NAGLE, D.F.: "MEMS-Based Integrated Circuit Mass Storage Systems," *Commun. of the ACM*, vol. 43, pp. 73-80, Nov. 2000.
- CARR, R.W., and HENNESSY, J.L.: "WSClock—A Simple and Effective Algorithm for Virtual Memory Management," *Proc. Eighth Symp. on Operating Systems Principles*, ACM, pp. 87-95, 1981.
- CARRIERO, N., and GELERNTER, D.: "The S/Net's Linda Kernel," *ACM Trans. on Computer Systems*, vol. 4, pp. 110-129, May 1986.
- CARRIERO, N., and GELERNTER, D.: "Linda in Context," *Commun. of the ACM*, vol. 32, pp. 444-458, April 1989.
- CARTER, J.B., BENNETT, J.K., and ZWAENEPOEL, W.: "Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems," *Trans. on Computer Systems*, vol. 13, pp. 205-243, Aug. 1995.
- CERF, C., and NAVASKY, V.: *The Experts Speak*, New York: Random House, 1984.
- CHANDRA, A., ADLER, M., GOYAL, P., and SHENOY, P.: "Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors," *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 45-58, 2000.
- CHASE, J.S., LEVY, H.M., FEELEY, M.J., and LAZOWSKA, E.D.: "Sharing and Protection in a Single-Address-Space Operating System," *Trans. on Computer Systems*, vol. 12, pp. 271-307, Nov. 1994.
- CHEN, P.M., LEE, E.K., GIBSON, G.A., KATZ, R.H., and PATTERSON, D.A.: "RAID: High Performance Reliable Storage," *Comp. Surv.*, vol. 26, pp. 145-185, June 1994.

- CHEN, P.M., NG, W.T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., and LOWELL, D.: "The Rio File Cache: Surviving Operating System Crashes," *Proc. Seventh Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 74-83, 1996.
- CHEN, S., and THAPAR, M.: "A Novel Video Layout Strategy for Near-Video-on-Demand Servers," *Proc. Int'l Conf. on Multimedia Computing and Systems*, IEEE, pp. 37-45, 1997.
- CHEN, S., and TOWSLEY, D.: "A Performance Evaluation of RAID Architectures," *IEEE Trans. on Computers*, vol. 45, pp. 1116-1130, Oct. 1996.
- CHERITON, D.R.: "An Experiment Using Registers for Fast Message-Based Interprocess Communication," *Operating Systems Review*, vol. 18, pp. 12-20, Oct. 1984.
- CHERITON, D.: "The V Distributed System," *Commun. of the ACM*, vol. 31, pp. 314-333, March 1988.
- CHERVENAK, A., VELLANKI, V., and KURMAS, Z.: "Protecting File Systems: A Survey of Backup Techniques," *Proc. 15th IEEE Symp. on Mass Storage Systems*, IEEE, 1998.
- CHOW, T.C.K., and ABRAHAM, J.A.: "Load Balancing in Distributed Systems," *IEEE Trans. on Software Engineering*, vol. SE-8, pp. 401-412, July 1982.
- CLARK, P.C., and HOFFMAN, L.J.: "BITS: A Smartcard Protected Operating System," *Commun. of the ACM*, vol. 37, pp. 66-70, Nov. 1994.
- COFFMAN, E.G., ELPHICK, M.J., and SHOSHANI, A.: "System Deadlocks," *Computing Surveys*, vol. 3, pp. 67-78, June 1971.
- COMER, D.: *Operating System Design. The Xinu Approach*, Upper Saddle River, NJ: Prentice Hall, 1984.
- COOKE, D., URBAN, J., and HAMILTON, S.: "Unix and Beyond: An Interview with Ken Thompson," *Computer*, vol. 32, pp. 58-64, May 1999.
- CORBALAN, J., MARTORELL, X., and LABARTA, J.: "Performance-Driven Processor Allocation," *Proc. Fourth Symp. on Operating Systems Design and Implementation, USENIX*, pp. 59-71, 2000.
- CORBATO, F.J.: "On Building Systems That Will Fail," *Commun. of the ACM*, vol. 34, pp. 72-81, June 1991.
- CORBATO, F.J., MERWIN-DAGGETT, M., and DALEY, R.C.: "An Experimental Time-Sharing System," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 335-344, 1962.
- CORBATO, F.J., SALTZER, J.H., and CLINGEN, C.T.: "MULTICS—The First Seven Years," *Proc. AFIPS Spring Joint Computer Conf.*, AFIPS, pp. 571-583, 1972.
- CORBATO, F.J., and VYSSOTSKY, V.A.: "Introduction and Overview of the MULTICS System," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 185-196, 1965.

- COURTOIS, P.J., HEYMANS, F., and PARNAS, D.L.: "Concurrent Control with Readers and Writers," *Commun. of the ACM*, vol. 10, pp. 667-668, Oct. 1971.
- CRANOR, C.D., and PARULKAR, G.M.: "The UVM Virtual Memory System," *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 117-130, 1999.
- CROWLEY, C.: *Operating Systems: A Design-Oriented Approach*, Chicago: Irwin, 1997.
- CUSUMANO, M.A., and SELBY, R.W.: "How Microsoft Builds Software," *Commun. of the ACM*, vol. 40, pp. 53-61, June 1997.
- DALEY, R.C., and DENNIS, J.B.: "Virtual Memory, Process, and Sharing in MULTICS," *Commun. of the ACM*, vol. 11, pp. 306-312, May 1968.
- DAN, A., SITARAM, D., and SHAHABUDDIN, P.: "Scheduling Policies for an On-Demand Video Server with Batching," *Proc. Second Int'l Conf. on Multimedia*, ACM, pp. 15-23, 1994.
- DANDAMUDI, S.P.: "Reducing Run Queue Contention in Shared Memory Multiprocessors," *Computer*, vol. 30, pp. 82-89, March 1997.
- DE JONGE, W., KAASHOEK, M.F., and HSIEH, W.C.: "The Logical Disk: A New Approach to Improving File Systems," *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 15-28, 1993.
- DENNING, D.: "A Lattice Model of Secure Information Flow," *Commun. of the ACM*, vol. 19, pp. 236-243, 1976.
- DENNING, D.: "The United States vs. Craig Neidorf," *Commun. of the ACM*, vol. 34, pp. 22-43, March 1991.
- DENNING, D.: *Information Warfare and Security*, Reading, MA: Addison-Wesley, 1999.
- DENNING, P.J.: "The Working Set Model for Program Behavior," *Commun. of the ACM*, vol. 11, pp. 323-333, 1968a.
- DENNING, P.J.: "Thrashing: Its Causes and Prevention," *Proc. AFIPS National Computer Conf.*, AFIPS, pp. 915-922, 1968b.
- DENNING, P.J.: "Virtual Memory," *Computing Surveys*, vol. 2, pp. 153-189, Sept. 1970.
- DENNING, P.J.: "Working Sets Past and Present," *IEEE Trans. on Software Engineering*, vol. SE-6, pp. 64-84, Jan. 1980.
- DENNIS, J.B., and VAN HORN, E.C.: "Programming Semantics for Multiprogrammed Computations," *Commun. of the ACM*, vol. 9, pp. 143-155, March 1966.
- DEVARAKONDA, M., KISH, B., and MOHINDRA, A.: "Recovery in the Calypso File System," *Trans. on Computer Systems*, vol. 14, pp. 287-310, Aug. 1996.
- DIFFIE, W., and HELLMAN, M.E.: "New Directions in Cryptography," *IEEE Trans. on Information Theory*, vol. IT-22, pp. 644-654, Nov. 1976.

- DIJKSTRA, E.W.: "Co-operating Sequential Processes," in *Programming Languages*, Genuys, F. (Ed.), London: Academic Press, 1965.
- DIJKSTRA, E.W.: "The Structure of THE Multiprogramming System," *Commun. of the ACM*, vol. 11, pp. 341-346, May 1968.
- DUBOIS, M., SCHEURICH, C., and BRIGGS, F.A.: "Synchronization, Coherence, and Event Ordering in Multiprocessors," *Computer*, vol. 21, pp. 9-21, Feb. 1988.
- DOUCEUR, J.R., and BOLOSKY, W.J.: "A Large-Scale Study of File-System Contents," *Proc. Int'l Conf. on Measurement and Modeling of Computer Systems*, ACM, pp. 59-70, 1999.
- DRUSCHEL, P., PAI, V.S., and ZWAENEPOEL, W.: "Extensible Systems are Leading OS Research Astray," *Proc. Hot Topics in Operating Systems VI*, IEEE, pp. 38-42, 1997.
- DUDA, K.J., and CHERITON, D.R.: "Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 261-276, 1999.
- EAGER, D.L., LAZOWSKA, E.D., and ZAHORJAN, J.: "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. on Software Engineering*, vol. SE-12, pp. 662-675, May 1986.
- EAGER, D.L., VERNON, M., and ZAHORJAN, J.: "Optimal and Efficient Merging Schedules for Video-on-Demand Servers," *Proc. Seventh Int'l Conf. on Multimedia*, ACM, pp. 199-202, 1999.
- EDLER, J., LIPKIS, J., and SCHONBERG, E.: "Process Management for Highly Parallel UNIX Systems," *Proc. USENIX Workshop on UNIX and Supercomputers*, USENIX, pp. 1-17, Sept. 1988.
- EGAN, J.I., and TEIXEIRA, T.J.: *Writing a UNIX Device Driver*, 2nd ed., New York: John Wiley, 1992.
- EL GAMAL, A.: "A Public Key Cryptosystem and Signature Scheme Based on Discrete Logarithms," *IEEE Trans. on Information Theory*, vol. IT-31, pp. 469-472, July 1985.
- ELLIS, C.S.: "The Case for Higher-Level Power Management," *Proc. Hot Topics in Operating Systems VII*, IEEE, pp. 162-167, 1999.
- ENGLER, D.R., GUPTA, S.K., and KAASHOEK, M.F.: "AVM: Application-Level Virtual Memory," *Proc. Hot Topics in Operating Systems V*, IEEE, pp. 72-77, 1995a.
- ENGLER, D., CHELF, B., CHOU, A., HALLEM, S.: "Checking System Rules Using System-Specific Programmer-Written Compiler Extensions," *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 1-16, 2000.
- ENGLER, D.R., and KAASHOEK, M.F.: "Exterminate All Operating System Abstractions," *Proc. Hot Topics in Operating Systems V*, IEEE, pp. 78-83, 1995.
- ENGLER, D.R., KAASHOEK, M.F., and O'TOOLE, J. Jr.: "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 251-266, 1995.

- EVEN, S.: *Graph Algorithms*, Potomac, MD: Computer Science Press, 1979.
- FABRY, R.S.: "Capability-Based Addressing," *Commun. of the ACM*, vol. 17, pp. 403-412, July 1974.
- FEELEY, M.J., MORGAN, W.E., PIGHIN, F.H., KARLIN, A.R., LEVY, H.M., and THEK-KATH, C.A.: "Implementing Global Memory Management in a Workstation Cluster," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 201-212, 1995.
- FERGUSON, D., YEMINI, Y., and NIKOLAOU, C.: "Microeconomic Algorithms for Load Balancing in Distributed Computer Systems," *Proc. Eighth Int'l Conf. on Distributed Computing Systems*, IEEE, pp. 491-499, 1988.
- FEUSTAL, E.A.: "The Rice Research Computer—A Tagged Architecture," *Proc. AFIPS Conf.*, AFIPS, 1972.
- FLINN, J., and SATYANARAYANAN, M.: "Energy-Aware Adaptation for Mobile Applications," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 48-63, 1999.
- FLUCKIGER, F.: *Understanding Networked Multimedia*, Upper Saddle River, NJ: Prentice Hall, 1995.
- FORD, B., HIBLER, M., LEPREAU, J., TULLMAN, P., BACK, G., CLAWSON, S.: "Micro-kernels Meet Recursive Virtual Machines," *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, pp. 137-151, 1996.
- FORD, B., and SUSARLA, S.: "CPU Inheritance Scheduling," *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, pp. 91-105, 1996.
- FORD, B., BACK, G., BENSON, G., LEPREAU, J., LIN, A., SHIVERS, O.: "The Flux OSkit: A Substrate for Kernel and Language Research," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 38-51, 1997.
- FOTHERINGHAM, J.: "Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store," *Commun. of the ACM*, vol. 4, pp. 435-436, Oct. 1961.
- GAFSI, J., and BIERSECK, E.W.: "A Novel Replica Placement Strategy for Video Servers," *Proc. Sixth Int'l Workshop on Interactive and Distrib. Multimedia Systems*, ACM, pp. 321-335.
- GEIST, R., and DANIEL, S.: "A Continuum of Disk Scheduling Algorithms," *ACM Trans. on Computer Systems*, vol. 5, pp. 77-92, Feb. 1987.
- GELERNTER, D.: "Generative Communication in Linda," *ACM Trans. on Programming Languages and Systems*, vol. 7, pp. 80-112, Jan. 1985.
- GHORMLEY, D., PETROU, D., RODRIGUES, S., VAHDAT, A., and ANDERSON, T.E.: "SLIC: An Extensible System for Commodity Operating Systems," *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 39-46, 1998.
- GIBSON, G.A., and VAN METER, R.: "Network Attached Storage," *Commun. of the ACM*, vol. 43, pp. 37-45, Nov. 2000.

- GILL, D.S., ZHOU, S., and SANDHU, H.S.: "A Case Study of File System Workload in a Large-Scale Distributed Environment," *Proc. 1994 Conf. on Measurement and Modeling of Computer Systems*, ACM, pp. 276-277, 1994.
- GOLDBERG, L.A., GOLDBERG, P.W., PHILLIPS, C.A., and SORKIN, G.B.: "Constructing Virus Phylogenies," *Journal of Algorithms*, vol. 26, pp. 188-208, Jan. 1998.
- GOLDEN, D., and PECHURA, M.: "The Structure of Microcomputer File Systems," *Commun. of the ACM*, vol. 29, pp. 222-230, March 1986.
- GONG, L.: *Inside Java 2 Platform Security*, Reading, MA: Addison-Wesley, 1999.
- GOODHEART, B., and COX, J.: *The Magic Garden Explained*, Upper Saddle River, NJ: Prentice Hall, 1994.
- GOVIL, K., TEODOSIU, D., HUANG, Y., and ROSENBLUM, M.: "Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors," *Proc. 17th Symp. on Operating Systems Principles*, ACM, 1999, pp. 154-169.
- GOYAL, P., GUO, X., and VIN, H.M.: "A Hierarchical CPU Scheduler for Multimedia Operating Systems," *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, pp. 107-121, 1996.
- GRAHAM, R.: "Use of High-Level Languages for System Programming," Project MAC Report TM-13, M.I.T., Sept. 1970.
- GRIFFIN, J.L., SCHLOSSER, S.W., GANGER, G.R., and NAGLE, D.F.: "Operating System Management of MEMS-based Storage Devices," *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 87-102, 2000.
- GRIMM, R., and BERSHAD, B.: "Security for Extensible Systems," *Proc. Hot Topics in Operating Systems VI*, IEEE, pp. 62-66, 1997.
- GRIWODZ, C., BAR, M., and WOLF, L.C.: "Long-Term Movie Popularity Models in Video-on-Demand Systems," *Proc. Fifth Int'l Conf. on Multimedia*, ACM, pp. 349-357, 1997.
- GROPP, W., LUSK, E., and SKJELLUM, A.: *Using MPI: Portable Parallel Programming with the Message Passing Interface*, Cambridge, MA: M.I.T. Press, 1994.
- GROSSMAN, D., and SILVERMAN, H.: "Placement of Records on a Secondary Storage Device to Minimize Access Time," *Journal of the ACM*, vol. 20, pp. 429-438, 1973.
- HAFNER, K., and MARKOFF, J.: *Cyberpunk*, New York: Simon and Schuster, 1991.
- HAND, S.M.: "Self-Paging in the Nemesis Operating System," *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 73-86, 1999.
- HARBON, T.R.: *File Systems*, Upper Saddle River, NJ: Prentice Hall, 1988.
- HARCHOL-BALTER, M., and DOWNEY, A.B.: "Exploiting Process Lifetime Distributions for Dynamic Load Balancing," *Proc. SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, ACM, pp. 13-24, 1996.

- HARRISON, M.A., RUZZO, W.L., and ULLMAN, J.D.:** "Protection in Operating Systems," *Commun. of the ACM*, vol. 19, pp. 461-471, Aug. 1976.
- HART, J.M.:** *Win32 System Programming*, Reading, MA: Addison-Wesley, 1997.
- HARTIG, H., HOHMUTH, M., LIEDTKE, J., and SCHONBERG, S.:** "The Performance of Kernel-Based Systems," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 66-77, 1997.
- HARTMAN, J.H., and OUSTERHOUT, J.K.:** "The Zebra Striped Network File System," *Trans. on Computer Systems*, vol. 13, pp. 274-310, Aug. 1995.
- HAUSER, C., JACOBI, C., THEIMER, M., WELCH, B., and WEISER, M.:** "Using Threads in Interactive Systems: A Case Study," *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 94-105, 1993.
- HAVENDER, J.W.:** "Avoiding Deadlock in Multitasking Systems," *IBM Systems Journal*, vol. 7, pp. 74-84, 1968.
- HEBBARD, B., et al.:** "A Penetration Analysis of the Michigan Terminal System," *Operating Systems Review*, vol. 14, pp. 7-20, Jan. 1980.
- HEIDEMANN, J.S., and POPEK, G.J.:** "File-System Development with Stackable Layers," *Trans. on Computer Systems*, vol. 12, pp. 58-89, Feb. 1994.
- HEYBEY, A., SULLIVAN, M., ENGLAND, P.:** "Calliope: A Distributed Scalable Multimedia Server," *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 75-86, 1996.
- HIPSON, P.D.:** *Mastering Windows 2000 Registry*, Alameda, CA: Sybex, 2000.
- HOARE, C.A.R.:** "Monitors, An Operating System Structuring Concept," *Commun. of the ACM*, vol. 17, pp. 549-557, Oct. 1974; Erratum in *Commun. of the ACM*, vol. 18, p. 95, Feb. 1975.
- HOLT, R.C.:** "Some Deadlock Properties of Computer Systems," *Computing Surveys*, vol. 4, pp. 179-196, Sept. 1972.
- HONEYMAN, P., ADAMSON, A., COFFMAN, K., JANAKIRAMAN, J., JERDONEK, R., and REES, J.:** "Secure Videoconferencing," *The Seventh USENIX Security Symp.*, USENIX, pp. 123-133, 1998.
- HOWARD, J.H., KAZAR, M.J., MENEES, S.G., NICHOLS, D.A., SATYANARAYANAN, M., SIDEBOTHAM, R.N., and WEST, M.J.:** "Scale and Performance in a Distributed File System," *ACM Trans. on Computer Systems*, vol. 6, pp. 55-81, Feb. 1988.
- HUCK, J., and HAYS, J.:** "Architectural Support for Translation Table Management in Large Address Space Machines," *Proc. 20th Int'l Symp. on Computer Architecture*, ACM, pp. 39-50, 1993.
- HUTCHINSON, N.C., MANLEY, S., FEDERWISCH, M., HARRIS, G., HITZ, D., KLEIMAN, S., and O'MALLEY, S.:** "Logical vs. Physical File System Backup," *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 239-249, 1999.

- IEEE:** *Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, New York: Institute of Electrical and Electronics Engineers, Inc., 1990.
- ISLOOR, S.S., and MARSLAND, T.A.:** “The Deadlock Problem: An Overview,” *Computer*, vol. 13, pp. 58-78, Sept. 1980.
- ITZKOVITZ, A., and SCHUSTER, A.:** “MultiView and Millipage—Fine-Grain Sharing in Page-Based DSMs,” *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 215-228, 1999.
- IVENS, K.:** *Optimizing the Windows Registry*, Foster City, CA: IDG Books Worldwide, 1998.
- JOHNSON, K.L., KAASHOEK, M.F., and WALLACH, D.A.:** “CRL: High-Performance All-Software Distributed Shared Memory,” *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 213-226, 1995.
- JOHNSON, N.F., and JAJODIA, S.:** “Exploring Steganography: Seeing the Unseen,” *Computer*, vol. 31, pp. 26-34, Feb. 1998.
- JONES, M.B., ROSU, D., and ROSU, M.-C.:** “CPU Reservations and Time Constraints Efficient, Predictable Scheduling of Independent Activities,” *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 198-211, 1997.
- KAASHOEK, M.F., ENGLER, D.R., GANGER, G.R., BRICENO, H., HUNT, R., MAZIERES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., and MACKENZIE, K.:** “Application Performance and Flexibility on Exokernel Systems,” *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 52-65, 1997.
- KABAY, M.:** “Flashes from the Past,” *Information Security*, p. 17, 1997.
- KALLAHALLA, M., and VARMAN, P.J.:** “Optimal Read-Once Parallel Disk Scheduling,” *Proc. Sixth Workshop on I/O in Parallel and Distributed Systems*, ACM, pp. 68-77, 1999.
- KARACALI, B., TAI, K.C., and VOUK, M.A.:** “Deadlock Detection of EFSMs Using Simultaneous Reachability Analysis,” *Proc. Int’l Conference on Dependable Systems and Networks (DSN 2000)*, IEEE, pp. 315-324, 2000.
- KARLIN, A.R., LI, K., MANASSE, M.S., and OWICKI, S.:** “Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor,” *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 41-54, 1991.
- KARLIN, A.R., MANASSE, M.S., MCGEOCH, L., and OWICKI, S.:** “Competitive Randomized Algorithms for Non-Uniform Problems,” *Proc. First Annual ACM Symp. on Discrete Algorithms*, ACM, pp. 301-309, 1989.
- KARPOVICH, J.F., GRIMSHAW, A.S., and FRENCH, J.C.:** “Extensible File System (ELFS): An Object-Oriented Approach to High Performance File I/O,” *Proc. Ninth Annual Conf. on Object-Oriented Programming Systems, Language, and Applications*, ACM, pp. 191-204, 1994.

- KATCHER, D.I., KETTLER, K.A., and STROSNIDER, J.K.:** "Real-Time Operating Systems for Multimedia Processing," *Proc. Hot Topics in Operating Systems V*, IEEE, 1995.
- KAUFMAN, C., PERLMAN, R., and SPECINER, M.:** *Network Security*, Upper Saddle River, NJ: Prentice Hall, 1995.
- KAVI, K., BROWNE, J.C., and TRIPATHI, A.:** "Computer Systems Research: The Pressure is On," *Computer*, vol. 32, pp. 30-39, Jan 1999.
- KELEHER, P., COX, A., DWARKADAS, S., and ZWAENEPOEL, W.:** "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," *Proc. USENIX Winter 1994 Conf.*, USENIX, pp. 115-132, 1994.
- KERNIGHAN, B.W., and PIKE, R.:** *The UNIX Programming Environment*, Upper Saddle River, NJ: Prentice Hall, 1984.
- KHALIDI, Y.A., and NELSON, M.N.:** "Extensible File Systems in Spring," *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 1-14, 1993.
- KLEIN, D.V.:** "Foiling the Cracker: A Survey of, and Improvements to, Password Security," *Proc. UNIX Security Workshop II*, USENIX, Summer 1990.
- KLEINROCK, L.:** *Queueing Systems. Vol. I*, New York: John Wiley, 1975.
- KLINE, R.L., and GLINERT, E.P.:** "Improving GUI Accessibility for People with Low Vision," *Proc. Conf. on Human Factors in Computing Systems*, ACM, pp. 114-121, 1995.
- KNUTH, D.E.:** *The Art of Computer Programming, Volume 1: Fundamental Algorithms, 2nd Ed.*, Reading, MA: Addison-Wesley, 1973.
- KOCHAN, S.G., and WOOD, P.H.:** *UNIX Shell Programming*, Indianapolis, IN: Hayden Books, 1990.
- KRAVETS, R., and KRISHNAN, P.:** "Power Management Techniques for Mobile Communication," *Proc. Fourth ACM/IEEE Int'l Conf. on Mobile Computing and Networking*, ACM/IEEE, pp. 157-168, 1998.
- KRISHNAN, R.:** "Timeshared Video-on-Demand: A Workable Solution," *IEEE Multimedia*, vol. 6, Jan.-March 1999, pp. 77-79.
- KRUEGER, P., LAI, T.-H., and DIXIT-RADIYA, V.A.:** "Job Scheduling is More Important than Processor Allocation for Hypercube Computers," *IEEE Trans. on Parallel and Distr. Systems*, vol. 5, pp. 488-497, May 1994.
- KUMAR, V.P., and REDDY, S.M.:** "Augmented Shuffle-Exchange Multistage Interconnection Networks," *Computer*, vol. 20, pp. 30-40, June 1987.
- KWOK, Y.-K., AHMAD, I.:** "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *Computing Surveys*, vol. 31, pp. 406-471, Dec. 1999.
- LAMPORT, L.:** "Password Authentication with Insecure Communication," *Commun. of the ACM*, vol. 24, pp. 770-772, Nov. 1981.

- LAMPSON, B.W.: "A Scheduling Philosophy for Multiprogramming Systems," *Commun. of the ACM*, vol. 11, pp. 347-360, May 1968.
- LAMPSON, B.W.: "A Note on the Confinement Problem," *Commun. of the ACM*, vol. 10, pp. 613-615, Oct. 1973.
- LAMPSON, B.W.: "Hints for Computer System Design," *IEEE Software*, vol. 1, pp. 11-28, Jan. 1984.
- LAMPSON, B.W., and STURGIS, H.E.: "Crash Recovery in a Distributed Data Storage System," Xerox Palo Alto Research Center Technical Report, June 1979.
- LANDWEHR, C.E.: "Formal Models of Computer Security," *Computing Surveys*, vol. 13, pp. 247-278, Sept. 1981.
- LANGENDOEN, K., BHOEDJANG, R., and BAL, H.E.: "Models for Asynchronous Message Passing," *IEEE Concurrency*, vol. 5, pp. 28-37, April-June 1997.
- LEBECK, A.R., FAN, X., ZENG, H., ELLIS, C.S.: "Power Aware Page Allocation," *Proc. Ninth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM, 2000.
- LEE, J.Y.B.: "Parallel Video Servers: A Tutorial," *IEEE Multimedia*, vol. 5, pp. 20-28, April-June 1998.
- LEE, W., SU, D., WIJESEKERA, D., SRIVASTAVA, J., KENCHAMMANA-HOSEKOTE, D., and FORESTI, M.: "Experimental Evaluation of PFS Continuous Media File System," *Proc. Sixth Int'l Conf. on Information and Knowledge Management*, ACM, pp. 246-253, 1997.
- LESLIE, I., McAULEY, D., BLACK, R., ROSCOE, T., BARHAM, P., EVERS, D., FAIRBAIRNS, R., and HYDEN, E.: "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," *IEEE Journal on Selected Areas in Commun.*, vol. 14, pp. 1280-1297, July 1996.
- LEVIN, R., COHEN, E.S., CORWIN, W.M., POLLACK, F.J., and WULF, W.A.: "Policy/Mechanism Separation in Hydra," *Proc. Fifth Symp. on Operating Systems Principles*, ACM, pp. 132-140, 1975.
- LEWINE, D.: *POSIX Programmer's Guide*, Sebastopol, CA: O'Reilly & Associates, 1991.
- LI, K.: "Shared Virtual Memory on Loosely Coupled Multiprocessors," Ph.D. Thesis, Yale Univ., 1986.
- LI, K., and HUDAK, P.: "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. on Computer Systems*, vol. 7, pp. 321-359, Nov. 1989.
- LI, K., KUMPF, R., HORTON, P., and ANDERSON, T.: "A Quantitative Analysis of Disk Drive Power Management in Portable Computers," *Proc. 1994 Winter Conf., USENIX*, pp. 279-291, 1994.
- LIEDTKE, J.: "Improving IPC by Kernel Design," *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 175-188, 1993.

- LIEDTKE, J.: "On Micro-Kernel Construction," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 237-250, 1995.
- LIEDTKE, J.: "Toward Real Microkernels," *Commun. of the ACM*, vol. 39, pp. 70-77, Sept. 1996.
- LINDE, R.R.: "Operating System Penetration," *Proc. AFIPS National Computer Conf.*, AFIPS, pp. 361-368, 1975.
- LIONS, J.: *Lions' Commentary on Unix 6th Edition, with Source Code*, San Jose, CA: Peer-to-Peer Communications, 1996.
- LIU, C.L., and LAYLAND, J.W.: "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, vol. 20, pp. 46-61, Jan. 1973.
- LO, V.M.: "Heuristic Algorithms for Task Assignment in Distributed Systems," *Proc. Fourth Int'l Conf. on Distributed Computing Systems*, IEEE, pp. 30-39, 1984.
- LORCH, J.R., and SMITH, A.J.: "Reducing Processor Power Consumption by Improving Processor Time Management In a Single-User Operating System," *Proc. Second Int'l Conf. on Mobile Computing and Networking*, ACM, pp. 143-154, 1996.
- LORCH, J., and SMITH, A.J.: "Apple Macintosh's Energy Consumption," *IEEE Micro*, vol. 18, pp. 54-63, Nov./Dec. 1998.
- LOUGHER, P., PEGLER, D., and SHEPHERD, D.: "Scalable Storage Servers for Digital Audio and Video," *Proc. IEE Int'l Conf. on Storage and Recording Systems*, London: IEE, pp. 140-143, 1994.
- LU, Y.-H., SIMUNIC, T., DE MICHELI, G.: "Software Controlled Power Management," *Proc. Seventh Int'l Workshop on Hardware/Software Codesign*, ACM, pp. 157-161, 1999.
- LUDWIG, M.A.: *The Giant Black Book of Computer Viruses*, 2nd ed., Show Low, AZ: American Eagle Publications, 1998.
- LUMB, C.R., SCHINDLER, J., GANGER, G.R., and NAGLE, D.F.: "Towards Higher Disk Head Utilization: Extracting Free Bandwidth from Busy Disk Drives," *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 87-102, 2000.
- MAEKAWA, M., OLDEHOEFT, A.E., and OLDEHOEFT, R.R.: *Operating Systems: Advanced Concepts*, Menlo Park, CA: Benjamin/Cummings, 1987.
- MALKEWITZ, R.: "Head Pointing and Speech Control as a hands-free interface to Desktop Computing," *Proc. Third Int'l Conf. on Assistive Technologies*, ACM, pp. 182-188, 1998.
- MANARIS, B., and HARKREADER, A.: "SUITEKeys: A Speech Understanding Interface for the Motor-Control Challenged," *Proc. Third Int'l Conf. on Assistive Technologies*, ACM, pp. 108-115, 1998.
- MARK, A.R.: "The Development of Destination-Specific Biometric Authentication," *Proc. Tenth Conf. on Computers, Freedom and Privacy*, ACM, pp. 77-80, 2000.

- MARKOWITZ, J.A.: "Voice Biometrics," *Commun. of the ACM*, vol. 43, pp. 66-73, Sept. 2000.
- MARSH, B.D., SCOTT, M.L., LEBLANC, T.J., and MARKATOS, E.P.: "First-Class User-Level Threads," *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 110-121, 1991.
- MATTHEWS, J.N., ROSELLI, D., COSTELLO, A.M., WANG, R.Y., and ANDERSON, T.E.: "Improving the Performance of Log-Structured File Systems with Adaptive Methods," *Proc. 16th Symp. on Operating Systems Prin.*, ACM, pp. 238-251, 1997.
- MAXWELL, S.E.: *Linux Core Kernel Commentary*, Scottsdale, AZ: Coriolis, 1999.
- MAZIERES, D., KAMINSKY, M., KAASHOEK, M.F., and WITCHEL, E.: "Separating Key Management from File System Security," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 124-139, 1999.
- MCDANIEL, T.: "Magneto-Optical Data Storage," *Commun. of the ACM*, vol. 43, pp. 57-63, Nov. 2000.
- McKUSICK, M.K., BOSTIC, K., KARELS, M.J., and QUARTERMAN, J.S.: *The Design and Implementation of the 4.4 BSD Operating System*, Reading, MA: Addison-Wesley, 1996.
- McKUSICK, M.J., JOY, W.N., LEFFLER, S.J., and FABRY, R.S.: "A Fast File System for UNIX," *ACM Trans. on Computer Systems*, vol. 2, pp. 181-197, Aug. 1984.
- MEDINETS, D.: *UNIX Shell Programming Tools*, New York, NY: McGraw-Hill, 1999.
- MELLOR-CRUMMEY, J.M., and SCOTT, M.L.: "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Trans. on Computer Systems*, vol. 9, pp. 21-65, Feb. 1991.
- MERCER, C.W.: "Operating System Support for Multimedia Applications," *Proc. Second Int'l Conf on Multimedia*, ACM, pp. 492-493, 1994.
- MILLER, F.W.: "pk: A POSIX Threads Kernel," *FREENIX Track: USENIX Annual Technical Conference*, USENIX, pp. 179-182, 1999.
- MILOJICIC, D.: "Operating Systems: Now and in the Future," *IEEE Concurrency*, vol. 7, pp. 12-21, Jan.-March 1999.
- MILOJICIC, D.: "Security and Privacy," *IEEE Concurrency*, vol. 8, pp. 70-79, April-June 2000.
- MONROSE, F., and RUBIN, A.: "Authentication Via Keystroke Dynamics," *Proc. Conf. on Computer and Communications Security*, ACM, pp. 48-56, 1997.
- MORRIS, J.H., SATYANARAYANAN, M., CONNER, M.H., HOWARD, J.H., ROSENTHAL, D.S., and SMITH, F.D.: "Andrew: A Distributed Personal Computing Environment," *Commun. of the ACM*, vol. 29, pp. 184-201, March 1986.
- MORRIS, R., and THOMPSON, K.: "Password Security: A Case History," *Commun. of the ACM*, vol. 22, pp. 594-597, Nov. 1979.

- MULLENDER, S.J., and TANENBAUM, A.S.: "Immediate Files," *Software—Practice and Experience*, vol. 14, pp. 365-368, April 1984.
- MYERS, A.C., and LISKOV, B.: "A Decentralized Model for Information Flow Control," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 129-142, 1997.
- NACHENBERG, C.: "Computer Virus-Antivirus Coevolution" *Commun. of the ACM*, vol. 40, pp. 46-51, Jan. 1997.
- NEMETH, E., SNYDER, G., SEEBASS, S., and HEIN, T.R.: *UNIX System Administration Handbook, 2nd ed.*, Upper Saddle River, NJ: Prentice Hall, 2000.
- NEWHAM, C., and ROSENBLATT, B.: *Learning the Bash Shell*, Sebastopol, CA: O'Reilly & Associates, 1998.
- NEWTON, G.: "Deadlock Prevention, Detection, and Resolution: An Annotated Bibliography," *Operating Systems Review*, vol. 13, pp. 33-44, April 1979.
- NIEH, J., and LAM, M.S.: "The Design, Implementation and Evaluation of SMART a Scheduler for Multimedia Applications," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 184-197, 1997.
- NIST (National Institute of Standards and Technology): FIPS Pub. 180-1, 1995.
- NG, S.W.: "Advances in Disk Technology: Performance Issues," *Computer*, vol. 31, pp. 75-81, May 1998.
- NORTON, P., MUELLER, J., and MANSFIELD, R.: *Complete Guide to Windows 2000 Professional*, Indianapolis, IN: Sams, 2000.
- OKI, B., PFLUEGL, M., SIEGEL, A., and SKEEN, D.: "The Information Bus—An Architecture for Extensible Distributed Systems," *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 58-68, 1993.
- ONEY, W.: *Programming the Microsoft Windows Driver Model*, Redmond, WA: Microsoft Press, 1999.
- ORGANICK, E.I.: *The Multics System*, Cambridge, MA: M.I.T. Press, 1972.
- ORLOV, S.S.: "Volume Holographic Data Storage," *Commun. of the ACM*, vol. 43, pp. 47-54, Nov. 2000.
- OUSTERHOUT, J.K.: "Scheduling Techniques for Concurrent Systems," *Proc. Third Int'l Conf. on Distrib. Computing Systems*, IEEE, pp. 22-30, 1982.
- PAI, V.S., DRUSCHEL, P., and ZWAENEPOEL, W.: "IO-Lite: A Unified I/O Buffering and Caching System," *Trans. on Computer Systems*, vol. 18, pp. 37-66, Feb. 2000.
- PAKIN, S., KARAMCHETI, V., CHIEN, A.A.: "Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively Parallel Processors," *IEEE Concurrency*, vol. 5, pp. 60-73, April-June 1997.
- PANKANTI, S., BOLLE, R.M., and JAIN, A.: "Biometrics: The Future of Identification," *Computer*, vol. 33, pp. 46-49, Feb. 2000.

- PATE, S.D.:** *UNIX Internals A Practical Approach*, Reading, MA: Addison-Wesley, 1996.
- PATTERSON, R.H., GIBSON, G.A., GINTING, E., STODOLSKY, D., and ZELENKA, J.:** "Informed Prefetching and Caching," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 79-95, 1995.
- PATTERSON, D.A., GIBSON, G., and KATZ, R.:** "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, ACM, pp. 109-166, 1988.
- PENTLAND, A., and CHOUDHURY, T.:** "Face Recognition for Smart Environments," *Computer*, vol. 33, pp. 50-55, Feb. 2000.
- PETERSON, G.L.:** "Myths about the Mutual Exclusion Problem," *Information Processing Letters*, vol. 12, pp. 115-116, June 1981.
- PETROU, D., MILFORD, J., and GIBSON, G.:** "Implementing Lottery Scheduling," *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 1-14, 1999.
- PETZOLD, C.:** *Programming Windows, 5th ed.*, Redmond, WA: Microsoft Press, 1999.
- PFLEEGER, C.P.:** *Security in Computing, 2nd ed.*, Upper Saddle River, NJ: Prentice Hall, 1997.
- PHILBIN, J., EDLER, J., ANSHUS, O.J., DOUGLAS, C.C., and LI, K.:** "Thread Scheduling for Cache Locality," *Proc. Seventh Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 60-71, 1996.
- PRECHELT, L.:** "An Empirical Comparison of Seven Programming Languages," *Computer*, vol. 33, pp. 23-29, Oct. 2000.
- PROTIC, J., TOMASEVIC, M., and MILUTINOVIC, V.:** *Distributed Shared Memory: Concepts and Systems*, Los Alamitos, CA: IEEE Computer Society, 1998.
- RAWSON, F.L. III:** "Experience with the Development of a Microkernel-Based, Multi-Server Operating System," *Proc. Hot Topics in Operating Systems VI*, IEEE, pp. 2-7, 1997.
- RECTOR, B.E., and NEWCOMER, J.M.:** *Win32 Programming*, Reading, MA: Addison-Wesley, 1997.
- REDDY, A.L.N., and WYLLIE, J.C.:** "Disk Scheduling in a Multimedia I/O System," *Proc. ACM Multimedia Conf.*, ACM, pp. 225-233, 1992.
- REDDY, A.L.N., and WYLLIE, J.C.:** "I/O Issues in a Multimedia System," *Computer*, vol. 27, pp. 69-74, March 1994.
- RITCHIE, D.M.:** "Reflections on Software Research," *Commun. of the ACM*, vol. 27, pp. 758-760, Aug. 1984.
- RITCHIE, D.M., and THOMPSON, K.:** "The UNIX Timesharing System," *Commun. of the ACM*, vol. 17, pp. 365-375, July 1974.
- RIVEST, R.L.:** "The MD5 Message-Digest Algorithm," RFC 1320, April 1992.

- RIVEST, R.L., SHAMIR, A., and ADLEMAN, L.:** "On a Method for Obtaining Digital Signatures and Public Key Cryptosystems," *Commun. of the ACM*, vol. 21, pp. 120-126, Feb. 1978.
- ROBBINS, A.:** *UNIX in a Nutshell: A Desktop Quick Reference for SVR4 and Solaris 7*, Sebastopol, CA: O'Reilly & Associates, 1999.
- ROMPOGIANNAKIS, Y., NERJES, G., MUTH, P., PATERAKIS, M., TRIANTAFILLOU, P., and WEIKUM, G.:** "Disk Scheduling for Mixed-Media Workloads in a Multimedia Server," *Proc. Sixth Int'l Conf. on Multimedia*, ACM, pp. 297-302, 1998.
- ROSENBLUM, M., and OUSTERHOUT, J.K.:** "The Design and Implementation of a Log-Structured File System," *Proc. 13th Symp. on Oper. Sys. Prin.*, ACM, pp. 1-15, 1991.
- ROSELLI, D., and LORCH, J.R.:** "A Comparison of File System Workloads," *Proc. USENIX Annual Tech. Conf.*, USENIX, pp. 41-54, 2000.
- ROZIER, M., ABBROSSIMOV, V., ARMAND, F., BOULE, I., GIEN, M., GUILLEMONT, M., HERRMANN, F., KAISER, C., LEONARD, P., LANGLOIS, S., and NEUHAUSER, W.:** "Chorus Distributed Operating Systems," *Computing Systems*, vol. 1, pp. 305-379, Oct. 1988.
- RUEMMLER, C., and WILKES, J.:** "An Introduction to Disk Drive Modeling," *Computer*, vol. 27, pp. 17-28, March 1994.
- SACKMAN, H., ERIKSON, W.J., and GRANT, E.E.:** "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," *Commun. of the ACM*, vol. 11, pp. 3-11, Jan. 1968.
- SALTZER, J.H.:** "Protection and Control of Information Sharing in MULTICS," *Commun. of the ACM*, vol. 17, pp. 388-402, July 1974.
- SALTZER, J.H., REED, D.P., and CLARK, D.D.:** "End-to-End Arguments in System Design," *Trans. on Computer Systems*, vol. 2, pp. 277-277, Nov. 1984.
- SALTZER, J.H., and SCHROEDER, M.D.:** "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 63, pp. 1278-1308, Sept. 1975.
- SALUS, P.H.:** "UNIX At 25," *Byte*, vol. 19, pp. 75-82, Oct. 1994.
- SANDHU, R.S.:** "Lattice-Based Access Control Models," *Computer*, vol. 26, pp. 9-19, Nov. 1993.
- SANTRY, D., FEELEY, M.J., HUTCHINSON, N.C., VEITCH, A.C. :** "Elephant: The File System That Never Forgets," *Proc. Hot Topics in Operating Systems VII*, IEEE, pp. 2-7, 1999a.
- SANTRY, D., FEELEY, M., HUTCHINSON, N., VEITCH, A.C., CARTON, R.W., and OFIR, J.:** "Deciding When to Forget in the Elephant File System," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 110-123, 1999b.
- SATYANARAYANAN, M., HOWARD, J.H., NICHOLS, D.N., SIDEBOTHAM, R.N., SPECTOR, A.Z., and WEST, M.J.:** "The ITC Distributed File System: Principles and Design," *Proc. of the Tenth Symp. on Operating System Prin.*, ACM, pp. 35-50, 1985.

- SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., and ANDERSON, T.: "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *Trans. on Computer Systems*, vol. 15, pp. 391-411, Nov. 1997.
- SCALES, D.J., and GHARACHORLOO, K.: "Towards Transparent and Efficient Software Distributed Shared Memory," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 157-169, 1997.
- SCHMIDT, B.K., LAM, M.S., and NORTHCUTT, J.D.: "The Interactive Performance of Slim: A Stateless, Thin-Client Architecture," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 32-47, 1999.
- SCOTT, M., LEBLANC, T., and MARSH, B.: "Multi-model Parallel Programming in Psyche," *Proc. Second ACM Symp. on Principles and Practice of Parallel Programming*, ACM, pp. 70-78, 1990.
- SEAWRIGHT, L.H., and MACKINNON, R.A.: "VM/370—A Study of Multiplicity and Usefulness," *IBM Systems Journal*, vol. 18, pp. 4-17, 1979.
- SEGARRA, M.-T., and ANDRI, F.: "MFS: a Mobile File System Using Generic System Services," *Proc. 1999 Symp. on Applied Computing*, ACM, pp. 419-420, 1999.
- SELTZER, M., ENDO, Y., SMALL, C., and SMITH, K.: "Dealing with Disaster: Surviving Misbehaved Kernel Extensionsm". *Proc. Second Symp. on Operating System Design and Implementation*, USENIX, pp. 213-227, 1994.
- SHENOY, P.J., and VIN, H.M.: "Efficient Striping Techniques for Variable Bit Rate Continuous Media File Servers," *Perf. Eval. Journal*, vol. 38, pp. 175-199, 1999.
- SHENOY, P.J., GOYAL, P., and VIN, H.M.: "Architectural Considerations for Next Generation File Systems," *Proc. Seventh Int'l Conf. on Multimedia*, ACM, pp. 457-467, 1999.
- SILBERSCHATZ, A., GALVIN, P.B., and GAGNE, G: *Applied Operating System Concepts*, New York: Wiley, 2000.
- SIMON, R.J.: *Windows NT Win32 API SuperBible*, Corte Madera, CA: Sams Publishing, 1997.
- SITARAM, D., and DAN, A.: *Multimedia Servers*, San Francisco, CA: Morgan Kaufman, 2000.
- SLAUGHTER, L., OARD, D.W., WARNICK, V.L., HARDING, J.L., and WILKERSON, G.J.: "A Graphical Interface for Speech-Based Retrieval," *Proc. Third ACM Conf. on Digital Libraries*, ACM, pp. 305-306, 1998.
- SMALL, C., and SELTZER, M.: "MiSFIT: constructing Safe Extensible Systems," *IEEE Concurrency*, vol. 6, pp. 34-41, July-Sept. 1998.
- SMITH, D.K., and ALEXANDER, R.C.: *Fumbling the Future: How Xerox Invented, Then Ignored, the First Personal Computer*, New York: William Morrow, 1988.
- SNIR, M., OTTO, S.W., HUSS-LEDERMAN, S., WALKER, D.W., and DONGARRA, J.: *MPI: The Complete Reference Manual*, Cambridge, MA: M.I.T. Press, 1996.

- SOLOMON, D.A., and RUSSINOVICH, M.E.: *Inside Windows 2000*, 3rd ed., Redmond, WA: Microsoft Press, 2000.
- SPAFFORD, E., HEAPHY, K., and FERBRACHE, D.: *Computer Viruses*, Arlington, VA: ADAPSO, 1989.
- STALLINGS, W.: *Operating Systems*, 4th Ed., Upper Saddle River, NJ: Prentice Hall, 2001.
- STEENKISTE, P.A.: "A Systematic Approach to Host Interface Design for High-Speed Networks," *Computer*, vol. 27, pp. 47-57, March 1994.
- STEINMETZ, R., and NAHRSTEDT, K.: *Multimedia: Computing, Communications and Applications*, Upper Saddle River, NJ: Prentice Hall, 1995.
- STENSTROM, P., HAGERSTEN, E., LILJA, D.J., MARTONOSI, M., and VENUGOPAL, M.: "Trends in Shared Memory Multiprocessing," *Computer*, vol. 30, pp. 44-50, 1997.
- STETS, R., DWARKADAS, S., HARDAVELLAS, N., HUNT, G., KONTOTHANASSIS, L., PARTHASARATHY, S., and SCOTT, M.: "Cashmere—2L Software Coherent Shared Memory on a Clustered Remote-Write Network," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 170-183, 1997.
- STEVENS, W.R.: *Advanced Programming in the UNIX Environment*, Reading, MA: Addison-Wesley, 1992.
- STOLL, C.: *The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage*, New York: Doubleday, 1989.
- STONE, H.S., and BOKHARI, S.H.: "Control of Distributed Processes," *Computer*, vol. 11, pp. 97-106, July 1978.
- TAI, K.C., and CARVER, R.H.: "VP: A New Operation for Semaphores," *Operating Systems Review*, vol. 30, pp. 5-11, July 1996.
- TALLURI, M., and HILL, M.D.: "Surpassing the TLB Performance of Superpages with Less Operating System Support," *Proc. Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 171-182, 1994.
- TALLURI, M., HILL, M.D., and KHALIDI, Y.A.: "A New Page Table for 64-bit Address Spaces," *Proc. 15th Symp. on Operating Systems Prin.*, ACM, pp. 184-200, 1995.
- TANENBAUM, A.S.: *Operating Systems: Design and Implementation*, Upper Saddle River, NJ: Prentice Hall, 1987.
- TANENBAUM, A.S.: *Computer Networks*, Upper Saddle River, NJ: Prentice Hall, 1996.
- TANENBAUM, A.S., VAN RENESSE, R., VAN STAVEREN, H., SHARP, G.J., MULLENDER, S.J., JANSEN, J., and VAN ROSSUM, G.: "Experiences with the Amoeba Distributed Operating System," *Commun. of the ACM*, vol. 33, pp. 46-63, Dec. 1990.
- TANENBAUM, A.S., and VAN STEEN, M.R.: *Distributed Systems*, Upper Saddle River, NJ: Prentice Hall, 2002.

- TANENBAUM, A.S., and WOODHULL, A.S.:** *Operating Systems: Design and Implementation*, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 1997.
- TAYLOR, R.N, MEDVIDOVIC, N., ANDERSON, K.M., WHITEHEAD, E.J., and ROBBINS, J.E.:** "A Component- and Message-Based Architectural Style for GUI Software," *Proc. 17th Int'l Conf. on Software Engineering*, ACM, pp. 295-304, 1995.
- TEORY, T.J.:** "Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 1-11, 1972.
- THEKKATH, C.A., MANN, T., and LEE, E.K.:** "Frangipani: A Scalable Distributed File System," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 224-237, 1997.
- THOMPSON, K.:** "Reflections on Trusting Trust," *Commun. of the ACM*, vol. 27, pp. 761-763, Aug. 1984.
- TRONO, J.A.:** "Comments on Tagged Semaphores," *Operating Systems Review*, vol. 34, pp. 7-11, Oct. 2000.
- TUCKER, A., and GUPTA, A.:** "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors," *Proc. 12th Symp. on Operating Systems Principles*, ACM, pp. 159-166, 1989.
- UHLIG, R., NAGLE, D., STANLEY, T., MUDGE, T., SECREST, S., and BROWN, R.:** "Design Tradeoffs for Software-Managed TLBs," *ACM Trans. on Computer Systems*, vol. 12, pp. 175-205, Aug. 1994.
- VAHALIA, U.:** *UNIX Internals—The New Frontiers*, Upper Saddle River, NJ: Prentice Hall, 1996.
- VAN BUSKIRK, R., and LALOMIA, M.:** "A Comparison of Speech and Mouse/Keyboard GUI Navigation," *Conference Companion on Human Factors in Computing Systems*, ACM, p. 96, 1995.
- VAN DOORN, L.:** *The Design and Application of an Extensible Operating System*, Ph.D. Thesis, Vrije Universiteit, Amsterdam, The Netherlands, 2001.
- VAN DOORN, L., HOMBURG, P., and TANENBAUM, A.S.:** "Paramecium: An Extensible Object-Based Kernel," *Proc. Hot Topics in Operating Systems V*, IEEE, pp. 86-89, 1995.
- VAN STEEN, M., HAUCK, F.J., BALLINTIJN, G., and TANENBAUM, A.S.:** "Algorithmic Design of the Globe Wide-Area Location Service," *Computer Journal*, vol. 41, pp. 207-310, 1998a.
- VAN STEEN, M., HAUCK, F.J., HOMBURG, P., and TANENBAUM, A.S.:** "Locating Objects in Wide-Area Systems," *IEEE Communications Magazine*, vol. 36, pp. 104-109, Jan. 1998b.
- VAN STEEN, M., HOMBURG, P., and TANENBAUM, A.S.:** "Globe: A Wide-Area Distributed System," *IEEE Concurrency*, vol. 7, pp. 70-78, Jan.-March 1999a.

- VAN STEEN, M., TANENBAUM, A.S., KUZ, I., and SIPS, H.J.: "A Scalable Middleware Solution for Advanced Wide-Area Web Services," *Distributed Systems Engineering*, vol. 7, pp. 34-42, 1999b.
- VASWANI, R., and ZAHORJAN, J.: "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared-Memory Multiprocessors," *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 26-40, 1991.
- VENKATASUBRAMANIAN, N., and RAMANATHAN, S.: "Load Management in Distributed Video Servers," *Proc. 17th Int'l Conf. on Distrib. Computing Systems*, IEEE, pp. 528-535, 1997.
- VINOSKI, S.: "CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 35, pp. 46-56, Feb 1997.
- VISCAROLA, P.G., and MASON, W.A.: *Windows NT Device Driver Development*, Indianapolis, IN: Macmillan Technical Publishing, 1999.
- VOGELS, W.: "File System Usage in Windows NT 4.0," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 93-109, 1999.
- VON EICKEN, T., BASU, A., BUCH, V., and VOGELS, W.: "U-Net: a User-Level Network Interface for Parallel and Distributed Computing," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 40-53, 1995.
- VON EICKEN, T., CULLER, D., GOLDSTEIN, S.C., SCHAUER, K.E.: "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Int'l Symp. on Computer Architecture*, ACM, pp. 256-266, 1992.
- WAHBE, R., LUCCO, S., ANDERSON, T., and GRAHAM, S.: "Efficient Software-Based Fault Isolation," *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 203-216, 1993.
- WALDO, J.: "The Jini Architecture for Network-Centric Computing," *Commun. of the ACM*, vol. 42, pp. 76-82, July 1999.
- WALDO, J.: "Alive and Well: Jini Technology Today," *Computer*, vol. 33, pp. 107-109, June 2000.
- "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. First Symp. on Operating System Design and Implementation, USENIX*, pp. 1-12, 1994.
- WALKER, W., and CRAGON, H.G.: "Interrupt Processing in Concurrent Processors," *Computer*, vol. 28, pp. 36-46, June 1995.
- WAN, G., and LIN, E.: "A Dynamic Paging Scheme for Wireless Communication Systems," *Proc. Third Int'l Conf. on Mobile Computing and Networking*, ACM/IEEE, pp. 195-203, 1997.
- WANG, C., GOEBEL, V., and PLAGEMANN, T.: "Techniques to Increase Disk Access Locality in the Minorca Multimedia File System," *Proc. Seventh Int'l Conf. on Multimedia*, ACM, pp. 147-150, 1999.

- WANG, R.Y., ANDERSON, T.E., and PATTERSON, D.A.: "Virtual Log Based File Systems for a Programmable Disk," *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 29-43, 1999.
- WEISER, M., WELCH, B., DEMERS, A., and SHENKER, S.: "Scheduling for Reduced CPU Energy," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 13-23, 1994.
- WILKES, J., GOLDING, R., STAELIN, C., and SULLIVAN, T.: "The HP AutoRAID Hierarchical Storage System," *ACM Trans. on Computer Systems*, vol. 14, pp. 108-136, Feb. 1996.
- WIRTH, N.: "A Plea for Lean Software," *Computer*, vol. 28, pp. 64-68, Feb. 1995.
- WOLMAN, A., VOELKER, M., SHARMA, N., CARDWELL, N., KARLIN, A., and LEVY, H.M.: "On the Scale and Performance of Cooperative Web Proxy Caching," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 16-31, 1999.
- WONG, C.K.: *Algorithmic Studies in Mass Storage Systems*, New York: Computer Science Press, 1983.
- WONG, P.C., and LEE, Y.B.: "Redundant Array of Inexpensive Servers (RAIS)," *Proc. Int'l Conf. on Commun.*, IEEE, pp. 787-792, 1997.
- WORTHINGTON, B.L., GANGER, G.R., and PATT, Y.N.: "Scheduling Algorithms for Modern Disk Drives," *Proc. 1994 Conf. on Measurement and Modeling of Computer Systems*, pp. 241-251, 1994.
- WU, M., and SHU, W.: "Scheduling for Large-Scale Parallel Video Servers," *Proc. Sixth Symp. on Frontiers of Massively Parallel Computation*, IEEE, pp. 126-133, 1996.
- WULF, W.A., COHEN, E.S., CORWIN, W.M., JONES, A.K., LEVIN, R., PIERSON, C., and POLLACK, F.J.: "HYDRA: The Kernel of a Multiprocessor Operating System," *Commun. of the ACM*, vol. 17, pp. 337-345, June 1974.
- YOUNG, M., TEVANIAN, A., Jr., RASHID, R., GOLUB, D., EPPINGER, J., CHEW, J., BOLOSKY, W., BLACK, D., and BARON, R.: "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proc. 11th Symp. on Operating Systems Principles*, ACM, pp. 63-76, 1987.
- ZACHARY, G.P.: *Showstopper*, New York: Maxwell MacMillan, 1994.
- ZAHORJAN, J., LAZOWSKA, E.D., and EAGER, D.L.: "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems," *IEEE Trans. on Parallel and Distr. Systems*, vol. 2, pp. 180-198, April 1991.
- ZEKAUSKAS, M.J., SAWDON, W.A., and BERSHAD, B.N.: "Software Write Detection for a Distributed Shared Memory," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 87-100, 1994.
- ZOBEL, D.: "The Deadlock Problem: A Classifying Bibliography," *Operating Systems Review*, vol. 17, pp. 6-16, Oct. 1983.

ZUBERI, K.M., PILLAI, P., and SHIN, K.G.: "EMERALDS: A Small-Memory Real-Time Microkernel," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 277-299, 1999.

ZWICKY, E.D.: "Torture-Testing Backup and Archive Programs: Things You Ought to Know but Probably Would Rather Not," *Proc. Fifth Conf. on Large Installation Systems Admin.*, USENIX, pp. 181-190, 1991.

INDEX

A

- Absolute path, 395, 733
- Access control element, 846-847
- Access control list, 647-650, 846-847
- Access token, 846
- ACE (*see* Access Control Element)
- Acknowledged datagram service, 555
- Acknowledgement, 121, 554
- ACL (*see* Access Control List)
- ACPI, 370
- Active message, 537
- Adapter, 271
- ADC (*see* Analog digital converter)
- Address space, 34
- Address windowing extensions, 816
- Admission control algorithm, 458
- Admission scheduler, 140
- ADSL (*see* Asymmetric Digital Subscriber Line)
- Adversary, 585
- Affinity scheduling, 523
- AFS (*see* Andrew File System)
- Agent, 637
- Aging, 146, 220-222
- Alarm signal, 36
- Analog digital converter, 460
- Andrew file system, 564-565
- Antivirus techniques, 628-635
 - Behavioral checkers, 633
 - Integrity checkers, 632
 - Virus scanners, 629-632
- APC (*see* Asynchronous Procedure Call)
- Aperiodic process, 149
- API (*see* Application Program Interface)
- Applet, 637
- Application program interface, 54
- Architectural coherence, 861
- Architecture, 3
- Associative memory, 211
- Asymmetric digital subscriber line, 454, 727
- Asynchronous call, 535
- Asynchronous procedure call, 783
- Atomic action, 110
- Attribute, file, 386-387
- Audio encoding, 459-461
- Authentication, 121
 - challenge-response, 600
 - user, 591-606
- Automounting, 749
- AV disk, 324

B

Backing store, 246
 Backup, file system, 416-421
 Bad block, 322-324, 418
 Bad-news diode, 892
 Balance set manager, 820
 Banker's algorithm, 178-180
 multiple resources, 179-180
 single resource, 178-179
 Bank switching, 816
 Barrier, 123-124
 Base record, 834
 Base register, 26, 196
 Basic input output system, 33, 190
 Batch system, 7
 Battery management, 369
 Battery-powered computers, 896-897
 Behavioral checker, 633
 Belady's anomaly, 229
 Bell-La Padula model, 657-658
 Berkeley fast file system, 744-745
 Berkeley UNIX, 675-676
 Best fit algorithm, 201
 Biba model, 658-659
 Binary exponential backoff, 552
 Binary semaphore, 111
 Binding time, 874
 BIOS (*see* Basic Input Output System)
 Bitmap, 199-200, 352-354
 Bitmap graphics, 344
 Block cache, 424-426, 492-494
 Block device, 270, 290
 Blocking call, 535
 Blocking network, 510
 Blocking versus nonblocking call, 535-537
 Block size, 298, 411-412
 Block special file, 40, 383, 724
 Boot block, 399
 Boot sector, 810
 Booting UNIX, 708-710
 Booting Windows 2000, 810-811
 Bottom-up implementation, 876-877
 Bounded-buffer problem, 108
 Bridge, 552
 Brute force solution, 881
 BSD, 13
 BSS segment, 711
 Buddy algorithm, 721
 Buffer cache, 424-426, 728

Buffering, 294-296
 double, 295-296
 Buffer overflow attack, 610-612
 Burst mode, 278
 Bus, 31-34
 Bus-based multiprocessor, 506-507
 Busy waiting, 30, 104, 285
 Byte code, JVM, 643

C

Cache, 87
 Cache-Coherent NUMA, 511
 Cache hit, 24
 Cache line, 23
 Cache manager, Windows 2000, 785
 Caching, 887-888
 file, 424-426
 multimedia, 492-494
 Windows 2000, 849-851
 Call gate, 261-262
 Canonical mode, 337-340
 Capability, 650-653
 Capability list, 650
 CC-NUMA, 511
 CD-recordable, 310-312
 CD-Rewritable, 312-313
 CD-ROM, 33-34, 306-310
 file system, 430-435
 multisession, 312
 session, 312
 VTOC, 311-312
 XA, 311
 Central processing unit, 21-23
 CERT (*see* Computer Emergency Response Team)
 Certificate, 591
 Challenge-response, 600
 Character device, 270, 291
 Character special file, 40, 383, 724
 Checkerboarding, 253
 Checkpointing, 174
 Chief programmer team, 891
 Child process, 35, 691
 Chrominance signal, 462
 Ciphertext, 587
 Circuit switching, 529
 Circular wait condition, 164-165, 182-183

- Classical IPC problems, 124-132
 - dining philosophers, 125-128
 - readers and writers, 128-129
 - sleeping barber, 129-132
 - Cleaner, 430
 - Cleaning policy, 241
 - Client process, 62
 - Client-server model, 61-63
 - Client-server system, 869-870
 - Client stub, 538
 - C-list, 650, 730
 - Clock, 327-333
 - square-wave mode, 328
 - one-shot mode, 328
 - Clock hardware, 328-329
 - Clock paging algorithm, 218-219
 - Clock software, 329-332
 - Clock tick, 328
 - Cluster computer, 526
 - Cluster of workstations, 526
 - Cluster size, 440
 - CMOS memory, 25-26
 - CMS (*see* Conversational Monitor System)
 - Code page, 347
 - Code review, 610
 - Code signing, 641-642
 - Color palette, 346
 - Command interpreter, 35
 - Compaction, memory, 197
 - Compatible Time Sharing System, 11
 - Composite video, 462
 - Computer
 - first generation, 6-7
 - fourth generation, 14-16
 - second generation, 7-9
 - third generation, 9-13
 - Computer emergency response team, 637
 - Computer hardware, 20-34
 - Condition variable, 116, 698
 - Configuration manager, Windows 2000, 785
 - Confinement problem, 661-663
 - Constant data length, 485
 - Constant time length, 485
 - Context switch, 28, 142
 - Control object, Windows 2000, 783
 - Conversational monitor system, 59-60
 - Cooked character stream, 731
 - Cooked mode, 336
 - Copy on write, 241, 701, 815
 - CORBA, 565-567
 - Core image, 35
 - Core map, 716
 - Co-scheduling, 525
 - Countermeasures, 606
 - Covert channel, 661-665
 - COW (*see* Cluster Of Workstations)
 - CP/M, 14, 435-438
 - CPU (*see* Central Processing Unit)
 - CPU scheduler (*see* Scheduling algorithm)
 - Cracker, 592, 593-596
 - Crash, 290, 417, 678, 814
 - Critical region, 102-103
 - Critical section, 102
 - Crossbar switch, 507
 - Crosspoint, 508
 - Cryptography, 587-591
 - CTSS (*see* Compatible Time Sharing System)
 - Cube, 527
 - Current directory, 396
 - Cycle stealing, 278
 - Cylinder, 24
 - Cylinder group, 745
 - Cylinder skew, 316
- D**
- DACL (*see* Discretionary ACL)
 - Daemon, 74, 299, 690
 - printer, 101
 - DAG (*see* Directed Acyclic Graph)
 - Data confidentiality, 584
 - Data encryption standard, 843
 - Datagram service, 555
 - Data integrity, 584
 - Data segment, 50, 711
 - Deadlock, 36-37, 159-185
 - conditions, 164
 - definition, 163
 - Deadlock avoidance, 175-180
 - multiple resources, 179-180
 - single resource, 178-179
 - Deadlock detection, 168-175
 - Deadlock modeling, 164-167
 - Deadlock prevention, 180-185
 - attacking circular wait, 182-183
 - attacking hold and wait, 181
 - attacking mutual exclusion, 180-181
 - attacking no preemption, 182

- Deadlock recovery, 173-175
 - using killing processes, 174-175
 - using preemption, 173-174
 - using rollback, 174
- Decibel, 459
- Dedicated I/O device, 297
- Deferred procedure call, 783
- Degree of multiprogramming, 141
- Demand paging, 222
- Denial of service attack, 584
- DES (*see* Data Encryption Standard)
- Design, operating system, 855-898
- Design principles, 859-861
- Device context, 351
- Device controller, 271-272
- Device driver, 29, 289-292
 - uniform interfacing, 293
 - Windows 2000, 827-830
- Device independence, 283
- Device independent bitmap, 354
- Device object, 828
- Device register, 2
- Diameter, network, 527
- DIB (*see* Device independent bitmap)
- Digital signature, 590-591
- Digital versatile disk, 313, 454
- Digital video, 467
- Digital video disk, 313
- Digram, 588
- Dining philosophers problem, 125-128
- Directed acyclic graph, 408
- Direct memory access, 31, 276-279, 287
- Directory-based multiprocessor, 511
- Directory, 38, 383, 393-399
 - CP/M, 437-438
 - current, 396
 - distributed system, 560-561
 - hierarchical, 395
 - implementation, 405-408
 - ISO 9660, 432-433
 - MS-DOS, 440
 - root, 393
 - single level, 393-394
 - UNIX version 7, 445-446
 - Windows 98, 443-445
 - Windows 2000, 839-840
 - working, 396
- Directory operation, 398-399
- DirectX, 781
- Dirty bit, 210
- Disabling interrupts, 103-104
- Discretionary access control, 657
- Discretionary ACL, 846
- Disk, 300-327
 - CD-ROM, 306-310
 - CD-recordable, 310-312
 - CD-rewritable, 312-313
 - DVD, 313-315
 - IDE, 300
 - magnetic, 300-302
- Disk arm scheduling, 318-322
 - elevator algorithm, 319
 - shortest seek first, 319
- Disk backup, 416-421
- Disk farm, 490
- Disk formatting, 315-318
- Disk hardware, 300
- Disk interleaving, 317-318
- Disk quota, 415-416
- Disk scheduling, 318-322
 - dynamic, 496-498
 - multimedia, 494-498
 - static, 495-496
- Disk space management, 410-411
- Dispatcher, 87
- Dispatcher object, 783
- Distance string, 232
- Distributed operating system, 16
- Distributed shared memory, 242, 540-544, 568
- Distributed system, 505, 549-577
- DLL (*see* Dynamic Link Library)
- DMA (*see* Direct Memory Access)
- DNS (*see* Domain Name System)
- Domain name system, 557
- Domain, protection, 645
- DOS (*see* MS-DOS)
- Dotted decimal notation, 594
- Double buffering, 295
- Double indirect block, 446, 744
- Double torus, 527
- DPC (*see* Deferred Procedure Call)
- Driver interface, 370
- Driver-Kernel Interface, 728
- Driver object, 824
- DSM (*see* Distributed Shared Memory)
- D-space, 239
- DVD (*see* Digital Versatile Disk)
- DV (*see see* Digital Video)
- Dynamic disk, 825
- Dynamic link library, 794

E

Earliest deadline first, 473-476
 Early binding, 874
 Echoing, 337-338
 EDF (*see* Earliest Deadline First)
 EEPROM (*see* Electrically erasable ROM)
 Effective UID, 755
 EFS (*see* Encrypting File System)
 Electrically Erasable ROM, 25
 Elevator algorithm, 320
 Embedded systems, 897
 Encryption, 587-589, 842-844
 Encrypting file system, 842
 End-to-end argument, 868-869
 Environment Subsystem, 792
 Error-correcting code, 272
 Error handling, 322-324
 Error reporting, 297
 Escape character, 340
 Escape sequence, 341
 Ethernet, 551-552
 Event-driven paradigm, 862
 Event, Windows 2000, 801-802
 Exclusive lock, 736
 Executive, Windows 2000, 784-787
 Exokernel, 61, 868-869
 Experience, role of in project mgmt, 893-894
 Extended machine, 4
 Extensible system, 870
 External fragmentation, 253

F

Fair share scheduling, 148
 False sharing, 541-543
 FAT (*see* File Allocation Table)
 Fiber, Windows 2000, 798
 Field, 461
 FIFO page replacement algorithm, 217
 File, 38-41

- block size, 411-412
- contiguous, 400-402
- shared, 408
- special, 40, 383, 724

 File access, 385-386

File allocation

- using FAT, 403-404
- using linked list, 402

 File allocation table, 403-404

- FAT-16 file system, 830
- FAT-32 file system, 830

 File attribute, 386-387
 File caching, 494
 File compression, 841-842
 File descriptor, 39, 389, 737
 File encryption, 842-844
 File extension, 381
 File handle, 749
 File implementation, 400-405
 File management, 413-414
 File name lookup, 839-841
 File naming, 380-382
 File operation, 387-388
 File placement

- multimedia, 481-492
- near video on demand, 486-487

 File read ahead, 427
 File sharing, 562-564
 File structure, 382-383
 File system, 379-449

- backup, 416-421
- CD-ROM, 430-435
- consistency, 421-424
- CP/M, 435-438
- disk quotas, 415-416
- distributed, 559-564
- examples, 430-448
- FAT-16, 830
- FAT-32, 830
- implementation, 399-430
- ISO 9660, 430-435
- layout, 399-400
- Linux, 746-747
- log structured, 428-430
- MD-DOS, 438-442
- multimedia, 475-481
- performance, 424-428
- reliability, 416-424
- UNIX, 732-753
- UNIX system calls, 736-740
- UNIX Version 7, 445-448
- Win32 API calls, 831-832
- Windows 2000, 830-844
- Windows 98, 442-445

 File type, 383-385

Filter, 685
 Filter driver, 830
 Finger daemon, 636
 Finger length identification, 604
 Fingerprint identification, 605
 Finite-state machine, 89
 First-come first-served scheduling, 139
 First fit algorithm, 200
 First-in First-out page replacement, 217
 Fixed partitions, 191-192
 Flash RAM, 25
 Fly-by mode, 278
 FMS (*see* FORTRAN Monitor System)
 Folder, 393
 Font, 354-355
 Formal Models of Secure Systems, 655-647
 FORTRAN monitor system, 8-9
 Fragmentation
 external, 253
 internal, 237
 Frame, 461
 Free disk block management, 413-414

G

Gang scheduling, 524-526
 GDI (*see* Graphics Device Interface)
 GDT (*see* Global Descriptor Table)
 Generic rights, 652
 Generic security attacks, 613
 GID (*see* Group ID)
 Global descriptor table, 257
 Globe, 567-571
 GNU public license, 679
 Goat file, 629
 GPL (*see* GNU Public License)
 Graphical user interface, 15, 342-355
 input software, 347
 output software, 347-355
 Graphics adapter, 344
 Graphics device interface, 351, 786
 Green book, 309
 Grid, 527
 Group, 648
 Group ID, 36, 753
 Guaranteed scheduling, 146-147
 GUI (*see* Graphical User Interface)

H

Hacker, 592, (*see also* cracker)
 HAL (*see* Hardware Abstraction Layer)
 Handle, 77, 773
 Hard link, 398
 Hard real time system, 148
 Hardware abstraction layer, 779-782
 Header file, 687
 Head skew, 316
 Hiding the hardware, 877-879
 High sierra, 309
 Hint, 888
 Hive, registry, 777
 Host, 553
 Hungarian notation, 350
 Hypercube, 528
 Hyperlink, 558

I

IBM 1401, 7-10
 IBM 7084, 8-10
 IBM PC, 14
 IBM System/360, 9-10
 IDE Disk, 28
 IDL (*see* Interface Definition Language)
 IEEE 1394, 32
 IEEE standard 1003.1, 676
 IIOP (*see* Internet InterOrb Protocol)
 Immediate file, 837
 Impersonation, 846
 Imprecise interrupt, 282
 Incremental dump, 417
 Indirect block, 446, 744
 Indirection, 879-880
 I-node, 51, 404-405, 740
 I-node table, 741
 Input software
 GUI, 347
 terminal, 336-340
 Instruction backup, 244-245
 Instruction set architecture, 2
 Integrated drive electronics, 28
 Integrity checker, 632
 Interface definition language, 566
 Interlaced video, 462
 Interleaving, disk, 317-318

Internal fragmentation, 237
 Internet, 552-553
 Internet InterOrb Protocol, 566
 Internet protocol, 556, 726
 Internet worm, 635-637
 Interpretation, 640-641
 Interprocess communication, 35, 100-124
 Interrupt, 30, 279-282
 imprecise, 282
 precise, 281
 Interrupt-driven I/O, 286
 Interrupt handler, 287
 Interrupt vector, 30, 79, 280
 Intrinsic, 357
 Intruder, 585
 I/O, 38, 269-372
 DMA, 287, 276-279
 memory-mapped, 272-276
 I/O device, 28-31, 270-271
 I/O hardware, 269-282
 clock, 328-329
 disk, 300-315
 display, 344-346
 keyboard, 335
 mouse, 343
 I/O manager, Windows 2000, 784
 I/O port, 273
 I/O request packet, 827
 I/O software, 282-287
 buffering, 283-284, 294-296
 device independent, 283, 292-298
 error handling, 283
 error reporting, 297
 input, 336-340, 347
 layers, 287
 output, 341-342, 347-355
 synchronous, 283
 naming, 283
 UNIX, 723-732
 Windows 2000, 824-830
 IP address, 556
 IPC (*see* Interprocess communication)
 IP (*see* Internet Protocol)
 IRP (*see* I/O Request Packet)
 ISA bus, 31
 ISA (*see* Instruction Set Architecture)
 ISO 9660 file system, 309-310, 430-435
 Joliet extensions, 435
 Rock Ridge extensions, 434-435
 I-space, 239

J

Java development kit, 643
 Java security, 642-644
 Java virtual machine, 61, 640-641
 JavaSpace, 576
 JDK (*see* Java Development Kit)
 Jiffy, 707
 Jini, 575-577
 discovery protocol, 575
 lease, 575
 lookup service, 575
 Jitter, 457
 Job, 7
 Windows 2000, 796
 Joint photographic experts group, 464
 Joliet extensions, 435
 JPEG standard, 464-467
 JVM (*see* Java Virtual Machine)

K

Kernel
 UNIX, 687-690
 Windows 2000, 782-784
 Kernel mode, 2
 Kernel thread, 870
 Key
 cryptographic, 587
 file, 383
 registry, 775
 Keymap, 347

L

Land, 307
 LAN (*see* Local Area Network)
 Large address space, 894-895
 Late binding, 874
 Layered system, 57-59, 867
 LDT (*see* Local Descriptor Table)
 Least recently used page replacement, 218-220
 Legacy device, 33
 LFS (*see* Log-Structured File system)
 Lightweight process, 81
 Limit register, 26, 196

Linda, 572-573
 template, 573
 tuple space, 572
 Linear address, 259-260
 Line discipline, 730
 Link, 408, 734
 Linked list, 200-202
 Linux, 13, 678-680
 file system, 746-747
 memory management, 719-723
 scheduling algorithm, 706-708
 threads, 702-704
 Loadable module, 730
 Load balancing algorithm, 545
 bidding, 548
 graph-theoretic, 545-546
 receiver-initiated, 547-548
 sender-initiated, 546-547
 Local Area Network, 551
 Local descriptor table, 257
 Locality, 888-889
 Locality of reference, 222
 Local procedure call, Windows 2000, 785
 Location independence, 562
 Location service, 571
 Location transparency, 561
 Lock, file, 735
 Lock variable, 104
 Logical block addressing, 302
 Logical dump, 419
 Logic bomb, 609
 Login spoofing, 608-609
 Log-structured file system, 428-430
 Loosely-coupled system, 505
 Lottery scheduling, 147
 LRU page replacement algorithm, 218-220
 Luminance signal, 462

M

Machine language, 2
 Macroblock, 468
 Magic character, shell, 684
 Magic number, 384
 Mailbox, 122
 Mainframe, 7
 Major device, 293, 724
 Mandatory access controls, 657

Mapped page writer, 822
 Master boot record, 317, 399
 Master file table, 834-839
 MBR (*see* Master Boot Record)
 MD5, 590
 Mechanism versus policy, 62, 149-150,
 247-249, 870-871
 Memory, 23-28
 Memory compaction, 197
 Memory hierarchy, 189
 Memory management, 37, 189-263
 bitmaps, 199-200
 design issues, 234-242
 implementation, 242-249
 linked lists, 200-202
 Linux, 719-723
 segmentation, 249-262
 system calls, 714-715
 swapping, 196-202
 UNIX, 710-723
 virtual memory, 202-214
 Windows 2000, 811-823
 Memory management unit, 27, 203
 Memory manager, 189
 Windows 2000, 785
 Memory-mapped file, 391-393, 713
 Memory-mapped I/O, 272-276
 Memory scheduler, 141
 Mesh, 527
 Message digest, 590
 Message passing, 119-123
 design issues, 121
 producer-consumer, 121-123
 Message-passing interface, 123
 Metafile, 353
 Method, 349, 565
 Metric units, 66-67
 MFT (*see* Master File Table)
 Microarchitecture level, 1
 Microcomputer, 14
 Microkernel, 62
 Microprogram, 1
 Middleware, 550
 coordination-based, 572-577
 document-based, 558-559
 file-system-based, 559-564
 shared-object-based, 565-571
 MINIX, 13, 677-678
 Minor device, 53, 294, 724
 Missing disk block, 422

MMU (*see* Memory Management Unit)
 Mobile code, 637-642
 Modeling paging systems, 228-234
 Modified page writer, 822
 Monitor, 115-119
 Monoalphabetic substitution, 588
 Monolithic system, 56-57
 Monoprogramming, 190-191
 Motif, 16, 357
 Motion picture experts group, 467
 Mounted file system, 52
 MPEG standard, 467-469
 MPI (*see* Message-Passing Interface)
 MS-DOS, 14, 763-764
 emulation on Windows 2000, 809-810
 file system, 438-442
 Multicomputer, 526-548
 communication software, 531-540
 hardware, 527-531
 interconnection, 527-531
 scheduling, 544-545
 software, 531-548
 MULTICS, 12, 672, 254-257
 Multilevel security, 657-659
 Multimedia system, 453-499, 896
 audio encoding, 459-461
 disk scheduling, 494-498
 file, 458-463
 file placement, 481-492
 file system paradigms, 475-481
 process scheduling, 469-475
 server, 475
 video compression, 463-469
 video encoding, 461-463
 Multiple queue scheduling, 144-146
 Multiprocessor, 71, 506-526
 bus-based, 506-507
 CC-NUMA, 511
 crossbar-based, 507-508
 directory-based, 511
 hardware, 506-513
 multistage network, 509-510
 NC-NUMA, 511
 operating system, 513-516
 scheduling, 521-526
 synchronization, 516-521
 Multiprogramming, 10, 72
 degree, 193
 fixed partitions, 191-192
 modeling, 192-194

Multistage switching networks, 509
 Multithreaded code, 81, 97-100
 Mutation engine, 631
 Mutex, 113-114, 698
 Mutual exclusion, 102, 103-108
 busy waiting, 103
 disabling interrupts, 103-104
 lock variables, 104
 Peterson's algorithm, 105-106
 strict alternation, 104-105
 TSL instruction, 107-108
 Mythical man month, 890-891

N

Naming, 872-874
 Naming transparency, 561-562
 Narrow striping, 492
 NC-NUMA, 511
 Near video on demand, 478-481
 file placement, 486-487
 Network file system, 747-753
 architecture, 747-748
 implementation, 750-753
 protocols, 749-750
 Network hardware, 551-553
 Networking, 895
 UNIX, 725-726
 Network interface, 529
 Network operating system, 16
 Network protocol, 555-558
 Network service, 553-555
 acknowledged datagram, 555
 connectionless, 554
 connection-oriented, 554
 datagram, 555
 request-reply service, 555
 Network terminal, 355-363
 SLIM, 360-363
 Next fit algorithm, 201
 NFS (*see* Network File System)
 NFU page replacement algorithm, 220
 Nonblocking system call, 535
 Nonblocking network, 508
 Noncanonical mode, 337
 Nonpreemptive scheduling, 135
 Nonresident attribute, 836
 Nonresource deadlocks, 184

Nonuniform memory access machine, 511-513
 Nonvolatile RAM, 327
 Not frequently used page replacement, 220
 Not recently used page replacement, 216-217
 NRU page replacement algorithm, 216-217
 NT file system, 830-844)
 NTFS (*see* NT File System)
 NUMA multiprocessor, 511-513

O

Object, 565
 security, 647
 Windows 2000, 787-792
 Object adapter, 566
 Object manager, Windows 2000, 784
 Object name space, Windows 2000, 791-792
 Object request broker, 566
 Omega network, 509
 One-time password, 599-600
 One-way function, 589
 Ontogeny recapitulates phylogeny, 16-18
 Open file description, 743
 Open software foundation, 677
 Operating system, 1
 batch, 7
 concepts, 34-44
 design, 855-898
 embedded, 20
 extended machine, 3-4
 history, 6-18
 implementation, 867-882
 multimedia, 453-499
 multiprocessor, 513-516
 real-time, 19
 smart card, 20
 performance, 882-889
 resource manager, 5-6
 structure, 56-65, 867-871
 trends, 894-897
 types, 18-20
 Optical disk, 306-315
 Optimal page replacement algorithm, 215-216
 Optimization, common case, 889
 Orange book, 311, 659-661
 ORB (*see* Object Request Broker)
 Organ-pipe algorithm, 488
 Orthogonality, 871-872

OS/360, 10-11
 OSF (*see* Open Software Foundation)
 Ostrich algorithm, 167-168
 Output software
 GUI, 347-355
 terminal, 341
 Windows, 347
 Overlapped seek, 300
 Overlay, 202

P

P and V primitives, 111
 Packet, 528
 Packet sniffer, 596
 Page, 203-210
 pinned, 246
 Page daemon, 716
 Page directory, 259-260
 Page fault, 205
 Page fault frequency replacement, 235-236
 Page fault handling, 243-244
 Windows 2000, 818-819
 Page fault rate, 233-234
 Page frame, 203
 Page replacement algorithm, 214-228
 aging, 220-222
 clock, 218-219
 FIFO, 217
 global, 234
 Linux, 719-723
 local, 234
 LRU, 218-220
 NFU, 220
 NRU, 216-217
 optimal, 215-216
 PFF, 235-236
 second chance, 217-218
 summary, 227-228
 two-handed clock, 718
 UNIX, 717-719
 Windows 2000, 819-823
 working set, 222-225
 WSclock, 225-227
 Page size, 237-238
 Page table, 205-210
 inverted, 213-214
 multilevel, 207-209

- Page table entry, 209-210
- Page table structure, 209-210
- Paging, 202
 - demand, 222
 - design issues, 234-242
 - global, 234
 - implementation issues, 242-249
 - load control, 236-237
 - local, 234
 - modeling, 228-23
 - UNIX, 716-717
 - Windows 2000, 819-823
- Paging daemon, 241
- Paradigm,
 - data, 863-864
 - execution, 862-863
 - user interface, 861-862
- Parallel and distributed systems, 896
- Parent process, 691
- Partition, disk, 53, 317
- Password, 592-600
 - one-time, 599-600
- Path name, 38, 395-398
 - absolute, 395
 - relative, 396
- PCI bus, 31
- PDA (*see* Personal Digital Assistant)
- PDP-11, 13
- Penetration team, 613
- Pentium, segmentation, 257-262
- Perfect shuffle network, 509
- Performance, operating system, 882-889
- Periodic process, 149
- Personal digital assistant, 20
- Peterson's algorithm, 105-106
- PFF page replacement algorithm, 235-236
- Physical address, 27
- Physical dump, 418
- Physical memory management, 821
- PID (*see* Process ID)
- Pinned page, 246
- Pipe, 40-41, 692
- Pipeline, 22, 685
- Pipe symbol, 685
- Pit, 307
- Pixel, 344, 463
- Plaintext, 587
- Play point, 480
- Plug and play, 33
- Plug-and-play manager, 785
- Policy versus mechanism, 62, 149-150, 247-249, 870-871
- Polling, 285
- Pop-up thread, 537
- Portable C compiler, 674
- Position independent code, 816
- POSIX, 13, 46, 676
 - thread calls, 697-698
- Power management, 363-371
 - degraded operation, 370-371
 - hardware issues, 364-365
 - operating system issues, 365-370
 - Windows 2000, 785
- Preamble, disk, 272
- Precise interrupt, 281
- Preemptive scheduling, 135
- Prepaging, 222
- Present/absent bit, 205
- Primary volume descriptor, 431
- Principal, security, 647
- Principle of least privilege, 653
- Printer daemon, 101
- Priority inversion, 108, 808
- Priority scheduling, 143-144
- Privacy, 585
- Process, 34-36, 71-153
 - aperiodic, 149
 - child, 691
 - classical IPC problems, 124-132
 - interprocess communication, 100-124
 - parent, 691
 - periodic, 149
 - UNIX, 690-710
 - Windows 2000, 796-811
- Process control block, 79
- Process creation, 73-75
- Process group, 692
- Process hierarchy, 76-77
- Process ID, 48, 691
- Process implementation, 79-80
- Process management, UNIX, 692-698
- Process manager, Windows 2000, 784
- Process model, 72-73
- Process scheduling, 132-152
 - multimedia, 469-475
- Process state, 77-79
- Process switch, 142
- Process table, 35, 79-80
 - UNIX, 699
 - Windows 2000, 796-811

Process termination, 75-76
 Processor allocation algorithm, 545
 Processor sharing, 155
 Producer-consumer problem, 108-110
 message passing, 121-123
 monitors, 117-118
 semaphores, 11-113
 Program counter, 21
 Programmed I/O, 284-286
 Program status word, 21
 Progressive video, 462
 Project management, 889-894
 Prompt, 42, 683
 Proportionality, 138
 Protection, memory, 194-196
 Protection command, 655-656
 Protection domain, 645-647
 Protection matrix, 645-647
 Protection mechanism, 584, 645-653
 Protection ring, 262
 Protocol, 356, 602
 Protocol stack, 556
 Pseudoparallelism, 71
 PSW (*see* Program Status Word)
 Public-key cryptography, 588-589
 Publish/Subscribe, 574-575
 Pull server, 475
 Pulse code modulation, 460
 Push server, 475

Q

Quality of service, 457, 554
 Quantization noise, 460
 Quantum, 142
 Quick fit algorithm, 201
 Quota, disk, 415-416

R

Race condition, 100-102
 RAID, 302-306
 striping, 303
 RAM (*see* Random Access Memory)
 Random access file, 385
 Random access memory, 24
 Raster graphics, 344

Rate monotonic scheduling, 472-473
 Raw mode, 336
 Read ahead, file, 427, 752
 Readers and writers problem, 128-129
 Read-only memory, 25
 Real time, 329
 Real-time operating system, 20
 Real-time scheduling, 470-475
 Receive primitive, 534-535
 Recycle bin, 417
 Recycling of concepts, 43-44
 Red book, 306
 Redirection, I/O, 42, 692
 Reentrancy, 880-881
 Reentrant code, 292
 Reference monitor, 640, 645, 655
 Reference string, 230-231
 Registry, Windows, 774-777
 Regular file, 383
 Relative path, 396, 733
 Reliability, file system, 416-424
 Relocation, 194-196
 Remote procedure call, 537-540
 Rendezvous, 123
 Reparse point, 841
 Request-reply service, 555
 Resource, 160-163, 358
 nonpreemptable, 161
 preemptable, 160
 Resource acquisition, 161-163
 Resource trajectories, 175-176
 Resource vector, 171
 Response time, 138
 Reusability, 880
 Right, 645
 RMS (*see* Rate Monotonic Scheduling)
 R-node, 751
 Rock Ridge extensions, 434-435
 Role, 648
 ROM (*see* Read-Only Memory)
 Root, 754
 Root directory, 38
 Root file system, 40
 Root key, registry, 775
 Round, 495
 Round robin scheduling, 142-143
 Router, 553
 RPC (*see* Remote Procedure Call)
 RSA algorithm, 589
 Rwx bits, 41

S

- SACL (*see* System ACL)
- Safe state, 176-178
- Salt, 597
- Sandboxing, 638-640
- Scan-EDF algorithm, 497
- Schedulable system, 149
- Scheduler, 132
- Scheduler activations, 94
- Scheduling, process, 132-152
- Scheduling algorithm, 132-152
 - affinity, 523
 - batch systems, 138-139
 - categories, 135-136
 - earliest deadline first, 473-476
 - fair share, 148
 - first-come, first-served, 139
 - goals, 136-138
 - guaranteed, 146-147
 - interactive, 142-148
 - Linux, 706-708
 - lottery, 147
 - multicomputer, 544-545
 - multiprocessor, 521-526
 - multimedia, 469-475
 - multiple queues, 144-146
 - multiprocessor, 521-526
 - priority, 143-144
 - rate monotonic, 472-473
 - real-time, 148-149
 - real-time, 470-475
 - round robin, 142-143
 - shortest job first, 139-140
 - shortest process next, 146
 - shortest remaining time next, 140
 - thread, 150-151
 - three-level scheduling, 140-142
 - two-level, 523
 - UNIX, 704-706
 - Windows 2000, 804-809
- Scheduling mechanism, 149
- Scheduling policy, 149
- Script kiddy, 596
- SCSI bus, 32
- Second chance page replacement, 217-218
- Second system effect, 893
- Secret-key cryptography, 588
- Secure hash algorithm, 590
- Security, 41, 583-666
 - Security attacks, 606-644
 - buffer overflow, 610-612
 - logic bomb, 609
 - login spoofing, 608-609
 - trap door, 610
 - Trojan horse, 607-608
 - UNIX, 753-757
 - Win32 API calls, 847-848
 - Windows 2000, 844-849
 - Security flaws, 614-616
 - OS/360, 615
 - TENEX, 614
 - UNIX, 614
 - Security by obscurity, 587
 - Security descriptor, 846
 - Security design principles, 616-617
 - Security environment, 584-586
 - Security ID, 845-847
 - Security manager, Windows 2000, 785
 - Segment, 250-251
 - Segmentation, 249-262
 - implementation, 253
 - MULTICS, 254-257
 - Pentium, 257-262
 - Selector, Pentium, 258
 - Semantics of file sharing, 562
 - Semaphore, 110-113
 - binary, 111
 - Send primitive, 534-535
 - Sequencer, 569
 - Sequential access file, 385
 - Sequential consistency, 543-544, 562
 - Sequential process, 72
 - Serial line, 334
 - Server process, 62
 - Server stub, 538
 - Session manager, 811
 - Session semantics, 563
 - Set-top box, 455
 - SETUID bit, 755
 - SHA (*see* Secure Hash Algorithm)
 - Shared library, 251
 - Shared lock, 736
 - Shared-memory multiprocessor, 506-526
 - Shared pages, 239-241
 - Shared text segment, 712
 - Shell, 35, 41-42, 683-686, 693-694
 - Shell script, 686
 - Shortest job first scheduling, 139-140
 - Shortest process next scheduling, 146

- Shortest remaining time next, 140
 - Shortest seek first, 319
 - SID (*see* Security ID)
 - Signal, 692
 - Signature block, 590
 - Silver bullet, lack of, 894
 - Single indirect block, 446, 744
 - Skeleton, 566
 - SLED, 303
 - Sleep primitive, 108
 - Sleeping barber problem, 129-132
 - SLIM network terminal, 360-363
 - Smart card, 20, 601
 - Smart scheduling, 522
 - SMP (*see* Symmetric MultiProcessor)
 - Socket, 725-726
 - Soft real time system, 148
 - Soft timer, 332-333
 - Space sharing, 523
 - Space-time trade-off, 884-886
 - Special file, 40, 383, 724
 - Spin lock, 105, 518-519
 - Spinning versus switching, 519-521
 - Spooler directory, 101
 - Spooling, 11, 298
 - Spooling directory, 299
 - Stable storage, 324-327
 - Stack algorithm, 229-232
 - Stack pointer, 21
 - Stack segment, 50
 - Standard error, 684
 - Standard input, 684
 - Standard output, 684
 - Starvation, 126, 184-185
 - Stateless file system, 750
 - Static versus dynamic structures, 875-876
 - Steganography, 664
 - Store-and-forward packet switching, 528-529
 - Stored value card, 601
 - Stream, 730-732
 - Striping, disk, 492
 - Stub, 538
 - Subject, security, 647
 - Subkeys, registry, 775
 - Superblock, 399, 740
 - Superscalar CPU, 22
 - Superuser, 36, 754
 - Supervisor mode, 2
 - SVID (*see* System V Interface Definition)
 - Swapper, 715
 - Swapper thread, 821
 - Swapping, 196-202
 - UNIX, 715-716
 - Symbolic link, 409-410
 - Symmetric-key cryptography, 588
 - Symmetric multiprocessor, 515-516
 - Synchronization, 113
 - multiprocessor, 516-521
 - Synchronous call, 535
 - System/360, 9-10
 - System ACL, 847
 - System call, 23, 44-56
 - example of use, 389-391
 - UNIX file system, 736-740
 - UNIX I/O, 726-727
 - UNIX memory management, 714-715
 - UNIX process management, 692-696
 - UNIX security, 755-756
 - UNIX thread management, 696-698
 - Win32 file system, 831-833
 - Win32 I/O, 825-827
 - Win32 memory management, 816-817
 - Win32 process management, 799-804
 - Win32 security, 847-849
 - Win32 thread management, 799-804
 - System call interface, 864-866
 - System services, 786
 - System V, 13
 - System V interface definition, 676
-
- ## T
- Tagged architecture, 650-651
 - TCB (*see* Trusted Computing Base)
 - TCP (*see* Transmission Control Protocol)
 - TCP/IP, 676
 - Team structure, 891-893
 - Template, 573
 - TENEX security flaw, 614
 - Termcap, 341
 - Terminal
 - character-oriented, 333-342
 - input software, 336-340
 - network, 355-363
 - output software, 341
 - RS-232, 334-336
 - Text segment, 50, 711
 - Thermal management, 369

Thin client, 360-363
 Thompson, Ken, 13
 Thrashing, 222
 Thread, 81-100
 hybrid, 94
 kernel, 93
 Linux, 702-704
 pop-up, 96-97
 UNIX, 701-702
 user, 90-93
 Windows 2000, 804-809
 worker, 87
 Thread implementation, 90-93
 Thread jacket, 92
 Thread model, 81-85
 Thread scheduling, 150-151
 Thread system calls, 696-698
 Thread table, 90
 Thread usage, 85-90
 Thread wrapper, 92
 Threat, 584-585
 Three-level scheduling, 140-142
 Tiger team, 613
 Tightly-coupled system, 505
 Timer, 327
 Timesharing, 11, 521-523
 Time-space trade-off, 884-886
 TLB (*see* Translation Lookaside Buffer)
 Top-down implementation, 876-877
 TOPS-20, 384
 Torvalds, Linus, 13
 Track, 24
 Trampoline, 810
 Translation lookaside buffer, 211
 software management, 212-213
 Transmission control protocol, 557, 676, 726
 TRAP door, 610
 Trap instruction, 45
 Trends in operating system design, 894-897
 Triple indirect block, 446, 744
 Trojan horse, 607-608, 663
 Trusted computing base, 654-655
 Trusted system, 653-665
 TSL instruction, 107-108
 Tuple, 572
 Tuple space, 572
 Turnaround time, 137
 Two-level directory, 394
 Two-level scheduling, 523
 Two-phase locking, 183-184

U

UART, 335
 UDP (*see* User Datagram Protocol)
 UID (*see* User ID)
 UI (*see* UNIX International)
 UMA multiprocessor, 506-510
 UNICS, 672-673
 Uniform resource locator, 558
 Universal coordinated time, 329
 Universal serial bus, 32
 UNIX, 13 15-16, 671-758
 Berkeley, 675-676
 booting, 708-710
 file system, 732-753
 goals, 681-682
 history, 672-680
 interfaces, 682-683
 I/O, 723-732
 kernel structure, 687-690
 memory management, 710-723
 overview, 681-690
 paging, 716-717
 password security, 596-598
 PDP-11, 673-674
 portable, 674-675
 processes, 690-710
 scheduling algorithm, 704-706
 security, 753-757
 security flaws, 614
 shell, 683-686, 693-694
 standard, 676-677
 system calls (*see* System calls)
 threads, 701-702
 utility programs, 686-688
 version 7 file system, 445-448
 UNIX International, 677
 Unsafe state, 176-178
 Upcall, 95
 URL (*see* Uniform Resource Locator)
 USB (*see* Universal serial bus)
 User authentication, 591-606
 using password, 592-600
 using biometrics, 603-605
 using physical object, 601-603
 User datagram protocol, 726
 User friendly, 15
 User ID, 36, 753
 User interface paradigm, 861
 User mode, 3

User-space I/O software, 298-300
 User structure, UNIX, 699
 UTC (*see* Universal Coordinated Time)

V

VAD (*see* Virtual Address Descriptor)
 Value, registry, 775
 Vampire tap, 551
 VCR control function, 476
 Vector graphics, 344
 Venus, 564
 Vice, 564
 Video compression, 463-469
 Video controller, 344
 Video encoding, 461
 Video on demand, 454
 Video RAM, 344
 Video server, 454
 Virtual address, 27, 203
 Virtual address space, 203
 Virtual address descriptor, 818
 Virtual machine, 4, 59-61
 Virtual machine monitor, 59
 Virtual memory, 197, 202-214
 Virtual memory interface, 241-242
 Virus, 617-635
 avoidance, 633-634
 boot sector, 623-624
 cavity, 622
 companion, 619-620
 damage scenarios, 618-619
 denial of service attack, 618
 device driver, 625
 dropper, 619
 executable program, 620-622
 how spread, 626-628
 how to fight, 628-635
 macro, 625
 memory resident, 623
 overwriting, 620
 parasitic, 622
 payload, 619
 polymorphic, 631
 recovery, 635
 source code, 626
 Virus scanner, 629-632
 V-node, 750-751

W

Wakeup primitive, 108
 Wakeup waiting bit, 110
 WAN (*see* Wide Area Network)
 War dialer, 594
 Watchdog timer, 331
 Web browser, 558
 Web page, 558
 Wide area network, 551
 Wide striping, 492
 Widget, 357
 Wildcard, 649
 Wild card, shell, 684
 Win32 API calls, 53-56, 772-774
 file system, 831-833
 I/O, 825-827
 memory management, 816-817
 process management, 799-804
 security, 847-849
 thread management, 799-804
 Window, 347
 Window manager, 357
 Windows 95, history, 764-765
 Windows 98,
 file system, 442-445
 history, 764-765
 Windows 2000, 763-852
 booting, 810-811
 critical section, 801
 device drivers, 827-830
 executive, 784-787
 file system, 830-844
 history, 767-771
 I/O, 824-830
 IPC, 800-802
 kernel, 782-784
 master file table, 834-839
 memory management, 811-823
 objects, 787-792
 page fault handling, 818-819
 page replacement algorithm, 819-823
 processes, 796-811
 programming, 771-777
 registry, 774-777
 scheduling, 804-809
 security, 844-849
 socket, 801
 system calls (*see* Win32 API calls)
 system structure, 778-796

Windows driver model, 827
Windows Me, 15, 764-765
Windows Millennium edition, 15, 764-765
Windows NT, 15, 765-766
Wireless communication, 368
VM/370, 59-60
Working directory, 39, 396, 733
Working set, 222-225
Working set manager, 820
Wormhole routing, 529
Worm, Internet, 635
Worst fit algorithm, 201
Write-through cache, 426
WSClock, 225-227

X

X client, 356
X server, 356
X terminal, 356
X Window System, 16, 356-360
Xlib, 356

Y

Yellow book, 308

Z

Zero page thread, 822
Zipf's law, 488
Zombie state, 695

ABOUT THE AUTHOR

Andrew S. Tanenbaum has an S.B. degree from M.I.T. and a Ph.D. from the University of California at Berkeley. He is currently a Professor of Computer Science at the Vrije Universiteit in Amsterdam, The Netherlands, where he heads the Computer Systems Group. He is also Dean of the Advanced School for Computing and Imaging, an interuniversity graduate school doing research on advanced parallel, distributed, and imaging systems. Nevertheless, he is trying very hard to avoid turning into a bureaucrat.

In the past, he has done research on compilers, operating systems, networking, and local-area distributed systems. His current research focuses primarily on the design of wide-area distributed systems that scale to a billion users. These research projects have led to over 85 refereed papers in journals and conference proceedings and five books.

Prof. Tanenbaum has also produced a considerable volume of software. He was the principal architect of the Amsterdam Compiler Kit, a widely-used toolkit for writing portable compilers, as well as of MINIX, a small UNIX clone intended for use in student programming labs. Together with his Ph.D. students and programmers, he helped design the Amoeba distributed operating system, a high-performance microkernel-based distributed operating system. The MINIX and Amoeba systems are now available for free via the Internet.

His Ph.D. students have gone on to greater glory after getting their degrees. He is very proud of them. In this respect he resembles a mother hen.

Prof. Tanenbaum is a Fellow of the ACM, a Fellow of the the IEEE, a member of the Royal Netherlands Academy of Arts and Sciences, winner of the 1994 ACM Karl V. Karlstrom Outstanding Educator Award, and winner of the 1997 ACM/SIGCSE Award for Outstanding Contributions to Computer Science Education. He is also listed in *Who's Who in the World*. His home page on the World Wide Web can be found at URL <http://www.cs.vu.nl/~ast/>.

Document generated by Anna's Archive around 2023-2024 as part of the DuXiu collection
(<https://annas-blog.org/duxiu-exclusive.html>).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):

```
{
  "filename": "TW9kZXJue9wZXJhdGluZyBTeXN0ZW1zIFNlY29uZCBFZGll0aW9uXzQwMzU5MTk3LnppcA==",
  "filename_decoded": "Modern Operating Systems Second Edition_40359197.zip",
  "filesize": 275204542,
  "md5": "c98d84534b83b583e62ec3dd5d4db6f8",
  "header_md5": "b90446ff713a8767f4e86229dc1d0cb1",
  "sha1": "6fe68fc0b841619cba608289fa897bc252cd5ed1",
  "sha256": "a2348cf99e6ea388166e7bbae5e9e7d406d27cd39aa0f5c9cacf7dba1f7ceffd",
  "crc32": 3191181425,
  "zip_password": "",
  "uncompressed_size": 306214878,
  "pdg_dir_name": "Modern Operating Systems Second Edition_40359197",
  "pdg_main_pages_found": 952,
  "pdg_main_pages_max": 952,
  "total_pages": 977,
  "total_pixels": 5295501624,
  "pdf_generation_missing_pages": false
}
```