initialized using InitializeSecurityDescriptor. This call fills in the header. If the owner SID is not known, it can be looked up by name using LookupAccountSid. It can then be inserted into the security descriptor. The same holds for the group SID, if any. Normally, these will be the caller's own SID and one of the called's groups, but the system administrator can fill in any SIDs.

Win32 API function	Description
InitializeSecurityDescriptor	Prepare a new security descriptor for use
LookupAccountSid	Look up the SID for a given user name
SetSecurityDescriptorOwner	Enter the owner SID in the security descriptor
SetSecurityDescriptorGroup	Enter a group SID in the security descriptor
InitializeAcl	Initialize a DACL or SACL
AddAccessAllowedAce	Add a new ACE to a DACL or SACL allowing access
AddAccessDeniedAce	Add a new ACE to a DACL or SACL denying access
DeleteAce	Remove an ACE from a DACL or SACL
SetSecurityDescriptorDacl	Attach a DACL to a security descriptor

Figure 11-47. The principal Win32 API functions for security.

At this point the security descriptor's DACL (or SACL) can be initialized with InitializeAcl. ACL entries can be added using AddAccessAllowedAce and AddAccessDeniedAce. These calls can be repeated multiple times to add as many ACE entries as are needed. DeleteAce can be used to remove an entry, that is, when modifying an existing ACL rather than when constructing a new ACL. When the ACL is ready, SetSecurityDescriptorDacl can be used to attach it to the security descriptor. Finally, when the object is created, the newly minted security descriptor can be passed as a parameter to have it attached to the object.

11.10.3 Implementation of Security

Security in a stand-alone Windows system is implemented by a number of components, most of which we have already seen (networking is a whole other story and beyond the scope of this book). Logging in is handled by *winlogon* and authentication is handled by *lsass*. The result of a successful login is a new GUI shell (*explorer.exe*) with its associated access token. This process uses the SECURITY and SAM hives in the registry. The former sets the general security policy and the latter contains the security information for the individual users, as discussed in Sec. 11.2.3.

Once a user is logged in, security operations happen when an object is opened for access. Every OpenXXX call requires the name of the object being opened and the set of rights needed. During processing of the open, the security reference monitor (see Fig. 11-11) checks to see if the caller has all the rights required. It

performs this check by looking at the caller's access token and the DACL associated with the object. It goes down the list of ACEs in the ACL in order. As soon as it finds an entry that matches the caller's SID or one of the caller's groups, the access found there is taken as definitive. If all the rights the caller needs are available, the open succeeds; otherwise it fails.

DACLs can have Deny entries as well as Allow entries, as we have seen. For this reason, it is usual to put entries denying access in front of entries granting access in the ACL, so that a user who is specifically denied access cannot get in via a back door by being a member of a group that has legitimate access.

After an object has been opened, a handle to it is returned to the caller. On subsequent calls, the only check that is made is whether the operation now being tried was in the set of operations requested at open time, to prevent a caller from opening a file for reading and then trying to write on it. Additionally, calls on handles may result in entries in the audit logs, as required by the SACL.

Windows added another security facility to deal with common problems securing the system by ACLs. There are new mandatory **integrity-level SIDs** in the process token, and objects specify an integrity-level ACE in the SACL. The integrity level prevents write-access to objects no matter what ACEs are in the DACL. In particular, the integrity-level scheme is used to protect against an Internet Explorer process that has been compromised by an attacker (perhaps by the user ill-advisedly downloading code from an unknown Website). **Low-rights IE**, as it is called, runs with an integrity level set to *low*. By default all files and registry keys in the system have an integrity level of *medium*, so IE running with low-integrity level cannot modify them.

A number of other security features have been added to Windows in recent years. Starting with service pack 2 of Windows XP, much of the system was compiled with a flag (/GS) that did validation against many kinds of stack buffer overflows. Additionally a facility in the AMD64 architecture, called NX, was used to limit execution of code on stacks. The NX bit in the processor is available even when running in x86 mode. NX stands for *no execute* and allows pages to be marked so that code cannot be executed from them. Thus, if an attacker uses a buffer-overflow vulnerability to insert code into a process, it is not so easy to jump to the code and start executing it.

Windows Vista introduced even more security features to foil attackers. Code loaded into kernel mode is checked (by default on x64 systems) and only loaded if it is properly signed by a known and trusted authority. The addresses that DLLs and EXEs are loaded at, as well as stack allocations, are shuffled quite a bit on each system to make it less likely that an attacker can successfully use buffer overflows to branch into a well-known address and begin executing sequences of code that can be weaved into an elevation of privilege. A much smaller fraction of systems will be able to be attacked by relying on binaries being at standard addresses. Systems are far more likely to just crash, converting a potential elevation attack into a less dangerous denial-of-service attack.

Yet another change was the introduction of what Microsoft calls **UAC** (**User Account Control**). This is to address the chronic problem in Windows where most users run as administrators. The design of Windows does not require users to run as administrators, but neglect over many releases had made it just about impossible to use Windows successfully if you were not an administrator. Being an administrator all the time is dangerous. Not only can user errors easily damage the system, but if the user is somehow fooled or attacked and runs code that is trying to compromise the system, the code will have administrative access, and can bury itself deep in the system.

With UAC, if an attempt is made to perform an operation requiring administrator access, the system overlays a special desktop and takes control so that only input from the user can authorize the access (similarly to how CTRL-ALT-DEL works for C2 security). Of course, without becoming administrator it is possible for an attacker to destroy what the user really cares about, namely his personal files. But UAC does help foil existing types of attacks, and it is always easier to recover a compromised system if the attacker was unable to modify any of the system data or files.

The final security feature in Windows is one we have already mentioned. There is support to create *protected processes* which provide a security boundary. Normally, the user (as represented by a token object) defines the privilege boundary in the system. When a process is created, the user has access to process through any number of kernel facilities for process creation, debugging, path names, thread injection, and so on. Protected processes are shut off from user access. The original use of this facility in Windows was to allow digital rights management software to better protect content. In Windows 8.1, protected processes were expanded to more user-friendly purposes, like securing the system against attackers rather than securing content against attacks by the system owner.

Microsoft's efforts to improve the security of Windows have accelerated in recent years as more and more attacks have been launched against systems around the world. Some of these attacks have been very successful, taking entire countries and major corporations offline, and incurring costs of billions of dollars. Most of the attacks exploit small coding errors that lead to buffer overruns or using memory after it is freed, allowing the attacker to insert code by overwriting return addresses, exception pointers, virtual function pointers, and other data that control the execution of programs. Many of these problems could be avoided if type-safe languages were used instead of C and C++. And even with these unsafe languages many vulnerabilities could be avoided if students were better trained to understand the pitfalls of parameter and data validation, and the many dangers inherent in memory allocation APIs. After all, many of the software engineers who write code at Microsoft today were students a few years earlier, just as many of you reading this case study are now. Many books are available on the kinds of small coding errors that are exploitable in pointer-based languages and how to avoid them (e.g., Howard and LeBlank, 2009).

11.10.4 Security Mitigations

It would be great for users if computer software did not have any bugs, particularly bugs that are exploitable by hackers to take control of their computer and steal their information, or use their computer for illegal purposes such as distributed denial-of-service attacks, compromising other computers, and distribution of spam or other illicit materials. Unfortunately, this is not *yet* feasible in practice, and computers continue to have security vulnerabilities. Operating system developers have expended incredible efforts to minimize the number of bugs, with enough success that attackers are increasing their focus on application software, or browser plug-ins, like Adobe Flash, rather than the operating system itself.

Computer systems can still be made more secure through **mitigation** techniques that make it more difficult to exploit vulnerabilities when they are found. Windows has continually added improvements to its mitigation techniques in the ten years leading up to Windows 8.1.

Mitigation	Description
/GS compiler flag	Add canary to stack frames to protect branch targets
Exception hardening	Restrict what code can be invoked as exception handlers
NX MMU protection	Mark code as non-executable to hinder attack payloads
ASLR	Randomize address space to make ROP attacks difficult
Heap hardening	Check for common heap usage errors
VTGuard	Add checks to validate virtual function tables
Code Integrity	Verify that libraries and drivers are properly cryptographically signed
Patchguard	Detect attempts to modify kernel data, e.g. by rootkits
Windows Update	Provide regular security patches to remove vulnerabilities
Windows Defender	Built-in basic antivirus capability

Figure 11-48. Some of the principal security mitigations in Windows.

The mitigations listed undermine different steps required for successful widespread exploitation of Windows systems. Some provide **defense-in-depth** against attacks that are able to work around other mitigations. /GS protects against stack overflow attacks that might allow attackers to modify return addresses, function pointers, and exception handlers. Exception hardening adds additional checks to verify that exception handler address chains are not overwritten. No-eXecute protection requires that successful attackers point the program counter not just at a data payload, but at code that the system has marked as executable. Often attackers attempt to circumvent NX protections using **return-oriented-programming** or **return to libC** techniques that point the program counter at fragments of code that allow them to build up an attack. **ASLR** (**Address Space Layout Randomization**) foils such attacks by making it difficult for an attacker to know ahead of time just exactly where the code, stacks, and other data structures are loaded in

the address space. Recent work shows how running programs can be rerandomized every few seconds, making attacks even more difficult (Giuffrida et al., 2012).

Heap hardening is a series of mitigations added to the Windows implementation of the heap that make it more difficult to exploit vulnerabilities such as writing beyond the boundaries of a heap allocation, or some cases of continuing to use a heap block after freeing it. VTGuard adds additional checks in particularly sensitive code that prevent exploitation of use-after-free vulnerabilities related to virtual-function tables in C++.

Code integrity is kernel-level protection against loading arbitrary executable code into processes. It checks that programs and libraries were cryptographically signed by a trustworthy publisher. These checks work with the memory manager to verify the code on a page-by-page basis whenever individual pages are retrieved from disk. **Patchguard** is a kernel-level mitigation that attempts to detect rootkits designed to hide a successful exploitation from detection.

Windows Update is an automated service providing fixes to security vulnerabilities by patching the affected programs and libraries within Windows. Many of the vulnerabilities fixed were reported by security researchers, and their contributions are acknowledged in the notes attached to each fix. Ironically the security updates themselves pose a significant risk. Almost all vulnerabilities used by attackers are exploited only after a fix has been published by Microsoft. This is because reverse engineering the fixes themselves is the primary way most hackers discover vulnerabilities in systems. Systems that did not have all known updates immediately applied are thus susceptible to attack. The security research community is usually insistent that companies patch all vulnerabilities found within a reasonable time. The current monthly patch frequency used by Microsoft is a compromise between keeping the community happy and how often users must deal with patching to keep their systems safe.

The exception to this are the so-called **zero day** vulnerabilities. These are exploitable bugs that are not known to exist until after their use is detected. Fortunately, zero day vulnerabilities are considered to be rare, and reliably exploitable zero days are even rarer due to the effectiveness of the mitigation measures described above. There is a black market in such vulnerabilities. The mitigations in the most recent versions of Windows are believed to be causing the market price for a useful zero day to rise very steeply.

Finally, antivirus software has become such a critical tool for combating malware that Windows includes a basic version within Windows, called **Windows Defender**. Antivirus software hooks into kernel operations to detect malware inside files, as well as recognize the behavioral patterns that are used by specific instances (or general categories) of malware. These behaviors include the techniques used to survive reboots, modify the registry to alter system behavior, and launching particular processes and services needed to implement an attack. Though Windows Defender provides reasonably good protection against common malware, many users prefer to purchase third-party antivirus software.

Many of these mitigations are under the control of compiler and linker flags. If applications, kernel device drivers, or plug-in libraries read data into executable memory or include code without /GS and ASLR enabled, the mitigations are not present and any vulnerabilities in the programs are much easier to exploit. Fortunately, in recent years the risks of not enabling mitigations are becoming widely understood by software developers, and mitigations are generally enabled.

The final two mitigations on the list are under the control of the user or administrator of each computer system. Allowing Windows Update to patch software and making sure that updated antivirus software is installed on systems are the best techniques for protecting systems from exploitation. The versions of Windows used by enterprise customers include features that make it easier for administrators to ensure that the systems connected to their networks are fully patched and correctly configured with antivirus software.

11.11 SUMMARY

Kernel mode in Windows is structured in the HAL, the kernel and executive layers of NTOS, and a large number of device drivers implementing everything from device services to file systems and networking to graphics. The HAL hides certain differences in hardware from the other components. The kernel layer manages the CPUs to support multithreading and synchronization, and the executive implements most kernel-mode services.

The executive is based on kernel-mode objects that represent the key executive data structures, including processes, threads, memory sections, drivers, devices, and synchronization objects—to mention a few. User processes create objects by calling system services and get back handle references which can be used in subsequent system calls to the executive components. The operating system also creates objects internally. The object manager maintains a namespace into which objects can be inserted for subsequent lookup.

The most important objects in Windows are processes, threads, and sections. Processes have virtual address spaces and are containers for resources. Threads are the unit of execution and are scheduled by the kernel layer using a priority algorithm in which the highest-priority ready thread always runs, preempting lower-priority threads as necessary. Sections represent memory objects, like files, that can be mapped into the address spaces of processes. EXE and DLL program images are represented as sections, as is shared memory.

Windows supports demand-paged virtual memory. The paging algorithm is based on the working-set concept. The system maintains several types of page lists, to optimize the use of memory. The various page lists are fed by trimming the working sets using complex formulas that try to reuse physical pages that have not been referenced in a long time. The cache manager manages virtual addresses in the kernel that can be used to map files into memory, dramatically improving

I/O performance for many applications because read operations can be satisfied without accessing the disk.

I/O is performed by device drivers, which follow the Windows Driver Model. Each driver starts out by initializing a driver object that contains the addresses of the procedures that the system can call to manipulate devices. The actual devices are represented by device objects, which are created from the configuration description of the system or by the plug-and-play manager as it discovers devices when enumerating the system buses. Devices are stacked and I/O request packets are passed down the stack and serviced by the drivers for each device in the device stack. I/O is inherently asynchronous, and drivers commonly queue requests for further work and return back to their caller. File-system volumes are implemented as devices in the I/O system.

The NTFS file system is based on a master file table, which has one record per file or directory. All the metadata in an NTFS file system is itself part of an NTFS file. Each file has multiple attributes, which can be either in the MFT record or nonresident (stored in blocks outside the MFT). NTFS supports Unicode, compression, journaling, and encryption among many other features.

Finally, Windows has a sophisticated security system based on access control lists and integrity levels. Each process has an authentication token that tells the identity of the user and what special privileges the process has, if any. Each object has a security descriptor associated with it. The security descriptor points to a discretionary access control list that contains access control entries that can allow or deny access to individuals or groups. Windows has added numerous security features in recent releases, including BitLocker for encrypting entire volumes, and address-space randomization, nonexecutable stacks, and other measures to make successful attacks more difficult.

PROBLEMS

1. Give one advantage and one disadvantage of the registry vs. having individual *.ini* files.

2. A mouse can have one, two, or three buttons. All three types are in use. Does the HAL hide this difference from the rest of the operating system? Why or why not?

3. The HAL keeps track of time starting in the year 1601. Give an example of an application where this feature is useful.

4. In Sec. 11.3.3 we described the problems caused by multithreaded applications closing handles in one thread while still using them in another. One possibility for fixing this would be to insert a sequence field. How could this help? What changes to the system would be required?

5. Many components of the executive (Fig. 11-11) call other components of the executive. Give three examples of one component calling another one, but use (six) different components in all.

6. Win32 does not have signals. If they were to be introduced, they could be per process, per thread, both, or neither. Make a proposal and explain why it is a good idea.

7. An alternative to using DLLs is to statically link each program with precisely those library procedures it actually calls, no more and no less. If this scheme were to be introduced, would it make more sense on client machines or on server machines?

8. The discussion of Windows User-Mode Scheduling mentioned that user-mode and kernel-mode threads had different stacks. What are some reasons why separate stacks are needed?

9. Windows uses 2-MB large pages because it improves the effectiveness of the TLB, which can have a profound impact on performance. Why is this? Why are 2-MB large pages not used all the time?

10. Is there any limit on the number of different operations that can be defined on an executive object? If so, where does this limit come from? If not, why not?

11. The Win32 API call WaitForMultipleObjects allows a thread to block on a set of synchronization objects whose handles are passed as parameters. As soon as any one of them is signaled, the calling thread is released. Is it possible to have the set of synchronization objects include two semaphores, one mutex, and one critical section? Why or why not? (*Hint*: This is not a trick question but it does require some careful thought.)

12. When initializing a global variable in a multithreaded program, a common programming error is to allow a race condition where the variable can be initialized twice. Why could this be a problem? Windows provides the InitOnceExecuteOnce API to prevent such races. How might it be implemented?

13. Name three reasons why a desktop process might be terminated. What additional reason might cause a process running a modern application to be terminated?

14. Modern applications must save their state to disk every time the user switches away from the application. This seems inefficient, as users may switch back to an application many times and the application simply resumes running. Why does the operating system require applications to save their state so often rather than just giving them a chance at the point the application is actually going to be terminated?

15. As described in Sec. 11.4, there is a special handle table used to allocate IDs for processes and threads. The algorithms for handle tables normally allocate the first available handle (maintaining the free list in LIFO order). In recent releases of Windows this was changed so that the ID table always keeps the free list in FIFO order. What is the problem that the LIFO ordering potentially causes for allocating process IDs, and why does not UNIX have this problem?

16. Suppose that the quantum is set to 20 msec and the current thread, at priority 24, has just started a quantum. Suddenly an I/O operation completes and a priority 28 thread is made ready. About how long does it have to wait to get to run on the CPU?

17. In Windows, the current priority is always greater than or equal to the base priority. Are there any circumstances in which it would make sense to have the current priority be lower than the base priority? If so, give an example. If not, why not?

18. Windows uses a facility called Autoboost to temporarily raise the priority of a thread that holds the resource that is required by a higher-priority thread. How do you think this works?

19. In Windows it is easy to implement a facility where threads running in the kernel can temporarily attach to the address space of a different process. Why is this so much harder to implement in user mode? Why might it be interesting to do so?

20. Name two ways to give better response time to the threads in important processes.

21. Even when there is plenty of free memory available, and the memory manager does not need to trim working sets, the paging system can still frequently be writing to disk. Why?

22. Windows swaps the processes for modern applications rather than reducing their working set and paging them. Why would this be more efficient? (Hint: It makes much less of a difference when the disk is an SSD.)

23. Why does the self-map used to access the physical pages of the page directory and page tables for a process always occupy the same 8 MB of kernel virtual addresses (on the x86)?

24. The x86 can use either 64-bit or 32-bit page table entries. Windows uses 64-bit PTEs so the system can access more than 4 GB of memory. With 32-bit PTEs, the self-map uses only one PDE in the page directory, and thus occupies only 4 MB of addresses rather than 8 MB. Why is this?

25. If a region of virtual address space is reserved but not committed, do you think a VAD is created for it? Defend your answer.

26. Which of the transitions shown in Fig. 11-34 are policy decisions, as opposed to required moves forced by system events (e.g., a process exiting and freeing its pages)?

27. Suppose that a page is shared and in two working sets at once. If it is evicted from one of the working sets, where does it go in Fig. 11-34? What happens when it is evicted from the second working set?

28. When a process unmaps a clean stack page, it makes the transition (5) in Fig. 11-34. Where does a dirty stack page go when unmapped? Why is there no transition to the modified list when a dirty stack page is unmapped?

29. Suppose that a dispatcher object representing some type of exclusive lock (like a mutex) is marked to use a notification event instead of a synchronization event to announce that the lock has been released. Why would this be bad? How much would the answer depend on lock hold times, the length of quantum, and whether the system was a multiprocessor?

30. To support POSIX, the native NtCreateProcess API supports duplicating a process in order to support fork. In UNIX fork is shortly followed by an exec most of the time. One example where this was used historically was in the Berkeley dump(8S) program which would backup disks to magnetic tape. Fork was used as a way of checkpointing the dump program so it could be restarted if there was an error with the tape device.

Give an example of how Windows might do something similar using NtCreateProcess. (*Hint*: Consider processes that host DLLs to implement functionality provided by a third party).

31. A file has the following mapping. Give the MFT run entries.

Offset	0	1	2	3	4	5	6	7	8	9	10
Disk address	50	51	52	22	24	25	26	53	54	-	60

32. Consider the MFT record of Fig. 11-41. Suppose that the file grew and a 10th block was assigned to the end of the file. The number of this block is 66. What would the MFT record look like now?

33. In Fig. 11-44(b), the first two runs are each of length 8 blocks. Is it just an accident that they are equal, or does this have to do with the way compression works? Explain your answer.

34. Suppose that you wanted to build Windows Lite. Which of the fields of Fig. 11-45 could be removed without weakening the security of the system?

35. The mitigation strategy for improving security despite the continuing presence of vulnerabilities has been very successful. Modern attacks are very sophisticated, often requiring the presence of multiple vulnerabilities to build a reliable exploit. One of the vulnerabilities that is usually required is an *information leak*. Explain how an information leak can be used to defeat address-space randomization in order to launch an attack based on return-oriented programming.

36. An extension model used by many programs (Web browsers, Office, COM servers) involves *hosting* DLLs to hook and extend their underlying functionality. Is this a reasonable model for an RPC-based service to use as long as it is careful to impersonate clients before loading the DLL? Why not?

37. When running on a NUMA machine, whenever the Windows memory manager needs to allocate a physical page to handle a page fault it attempts to use a page from the NUMA node for the current thread's ideal processor. Why? What if the thread is currently running on a different processor?

38. Give a couple of examples where an application might be able to recover easily from a backup based on a volume shadow copy rather the state of the disk after a system crash.

39. In Sec. 11.10, providing new memory to the process heap was mentioned as one of the scenarios that require a supply of zeroed pages in order to satisfy security requirements. Give one or more other examples of virtual memory operations that require zeroed pages.

40. Windows contains a hypervisor which allows multiple operating systems to run simultaneously. This is available on clients, but is far more important in cloud computing. When a security update is applied to a guest operating system, it is not much different than patching a server. However, when a security update is applied to the root operating system, this can be a big problem for the users of cloud computing. What is the nature of the problem? What can be done about it?

41. The *regedit* command can be used to export part or all of the registry to a text file under all current versions of Windows. Save the registry several times during a work session and see what changes. If you have access to a Windows computer on which you can install software or hardware, find out what changes when a program or device is added or removed.

42. Write a UNIX program that simulates writing an NTFS file with multiple streams. It should accept a list of one or more files as arguments and write an output file that contains one stream with the attributes of all arguments and additional streams with the contents of each of the arguments. Now write a second program for reporting on the attributes and streams and extracting all the components.

12

OPERATING SYSTEM DESIGN

In the past 11 chapters, we have covered a lot of ground and taken a look at many concepts and examples relating to operating systems. But studying existing operating systems is different from designing a new one. In this chapter we will take a quick look at some of the issues and trade-offs that operating systems designers have to consider when designing and implementing a new system.

There is a certain amount of folklore about what is good and what is bad floating around in the operating systems community, but surprisingly little has been written down. Probably the most important book is Fred Brooks' classic *The Mythical Man Month* in which he relates his experiences in designing and implementing IBM's OS/360. The 20th anniversary edition revises some of that material and adds four new chapters (Brooks, 1995).

Three classic papers on operating system design are "Hints for Computer System Design" (Lampson, 1984), "On Building Systems That Will Fail" (Corbató, 1991), and "End-to-End Arguments in System Design" (Saltzer et al., 1984). Like Brooks' book, all three papers have survived the years extremely well; most of their insights are still as valid now as when they were first published.

This chapter draws upon these sources as well as on personal experience as designer or codesigner of two operating systems: Amoeba (Tanenbaum et al., 1990) and MINIX (Tanenbaum and Woodhull, 2006). Since no consensus exists among operating system designers about the best way to design an operating system, this chapter will thus be more personal, speculative, and undoubtedly more controversial than the previous ones.

12.1 THE NATURE OF THE DESIGN PROBLEM

Operating system design is more of an engineering project than an exact science. It is hard to set clear goals and meet them. Let us start with these points.

12.1.1 Goals

In order to design a successful operating system, the designers must have a clear idea of what they want. Lack of a goal makes it very hard to make subsequent decisions. To make this point clearer, it is instructive to take a look at two programming languages, PL/I and C. PL/I was designed by IBM in the 1960s because it was a nuisance to have to support both FORTRAN and COBOL, and embarrassing to have academics yapping in the background that Algol was better than both of them. So a committee was set up to produce a language that would be all things to all people: PL/I. It had a little bit of FORTRAN, a little bit of COBOL, and a little bit of Algol. It failed because it lacked any unifying vision. It was simply a collection of features at war with one another, and too cumbersome to be compiled efficiently, to boot.

Now consider C. It was designed by one person (Dennis Ritchie) for one purpose (system programming). It was a huge success, in no small part because Ritchie knew what he wanted and did not want. As a result, it is still in widespread use more than three decades after its appearance. Having a clear vision of what you want is crucial.

What do operating system designers want? It obviously varies from system to system, being different for embedded systems than for server systems. However, for general-purpose operating systems four main items come to mind:

1. Define abstractions.
2. Provide primitive operations.
3. Ensure isolation.
4. Manage the hardware.

Each of these items will be discussed below.

The most important, but probably hardest task of an operating system is to define the right abstractions. Some of them, such as processes, address spaces, and files, have been around so long that they may seem obvious. Others, such as threads, are newer, and are less mature. For example, if a multithreaded process that has one thread blocked waiting for keyboard input forks, is there a thread in the new process also waiting for keyboard input? Other abstractions relate to synchronization, signals, the memory model, modeling of I/O, and many other areas.

Each of the abstractions can be instantiated in the form of concrete data structures. Users can create processes, files, pipes, and more. The primitive operations

manipulate these data structures. For example, users can read and write files. The primitive operations are implemented in the form of system calls. From the user's point of view, the heart of the operating system is formed by the abstractions and the operations on them available via the system calls.

Since on some computers multiple users can be logged into a computer at the same time, the operating system needs to provide mechanisms to keep them separated. One user may not interfere with another. The process concept is widely used to group resources together for protection purposes. Files and other data structures generally are protected as well. Another place where separation is crucial is in virtualization: the hypervisor must ensure that the virtual machines keep out of each other's hair. Making sure each user can perform only authorized operations on authorized data is a key goal of system design. However, users also want to share data and resources, so the isolation has to be selective and under user control. This makes it much harder. The email program should not be able to clobber the Web browser. Even when there is only a single user, different processes need to be isolated. Some systems, like Android, will start each process that belongs to the same user with a different user ID, to protect the processes from each other.

Closely related to this point is the need to isolate failures. If some part of the system goes down, most commonly a user process, it should not be able to take the rest of the system down with it. The system design should make sure that the various parts are well isolated from one another. Ideally, parts of the operating system should also be isolated from one another to allow independent failures. Going even further, maybe the operating system should be fault tolerant and self healing?

Finally, the operating system has to manage the hardware. In particular, it has to take care of all the low-level chips, such as interrupt controllers and bus controllers. It also has to provide a framework for allowing device drivers to manage the larger I/O devices, such as disks, printers, and the display.

12.1.2 Why Is It Hard to Design an Operating System?

Moore's Law says that computer hardware improves by a factor of 100 every decade. Nobody has a law saying that operating systems improve by a factor of 100 every decade. Or even get better at all. In fact, a case can be made that some of them are worse in key respects (such as reliability) than UNIX Version 7 was back in the 1970s.

Why? Inertia and the desire for backward compatibility often get much of the blame, and the failure to adhere to good design principles is also a culprit. But there is more to it. Operating systems are fundamentally different in certain ways from small application programs you can download for $49. Let us look at eight of the issues that make designing an operating system much harder than designing an application program.

First, operating systems have become extremely large programs. No one person can sit down at a PC and dash off a serious operating system in a few months.

Or even a few years. All current versions of UNIX contain millions of lines of code; Linux has hit 15 million, for example. Windows 8 is probably in the range of 50–100 million lines of code, depending on what you count (Vista was 70 million, but changes since then have both added code and removed it). No one person can understand a million lines of code, let alone 50 or 100 million. When you have a product that none of the designers can hope to fully understand, it should be no surprise that the results are often far from optimal.

Operating systems are not the most complex systems around. Aircraft carriers are far more complicated, for example, but they partition into isolated subsystems much better. The people designing the toilets on a aircraft carrier do not have to worry about the radar system. The two subsystems do not interact much. There are no known cases of a clogged toilet on an aircraft carrier causing the ship to start firing missiles. In an operating system, the file system often interacts with the memory system in unexpected and unforeseen ways.

Second, operating systems have to deal with concurrency. There are multiple users and multiple I/O devices all active at once. Managing concurrency is inherently much harder than managing a single sequential activity. Race conditions and deadlocks are just two of the problems that come up.

Third, operating systems have to deal with potentially hostile users—users who want to interfere with system operation or do things that they are forbidden from doing, such as stealing another user's files. The operating system needs to take measures to prevent these users from behaving improperly. Word-processing programs and photo editors do not have this problem.

Fourth, despite the fact that not all users trust each other, many users do want to share some of their information and resources with selected other users. The operating system has to make this possible, but in such a way that malicious users cannot interfere. Again, application programs do not face anything like this challenge.

Fifth, operating systems live for a very long time. UNIX has been around for 40 years. Windows has been around for about 30 years and shows no signs of vanishing. Consequently, the designers have to think about how hardware and applications may change in the distant future and how they should prepare for it. Systems that are locked too closely into one particular vision of the world usually die off.

Sixth, operating system designers really do not have a good idea of how their systems will be used, so they need to provide for considerable generality. Neither UNIX nor Windows was designed with a Web browser or streaming HD video in mind, yet many computers running these systems do little else. Nobody tells a ship designer to build a ship without specifying whether they want a fishing vessel, a cruise ship, or a battleship. And even fewer change their minds after the product has arrived.

Seventh, modern operating systems are generally designed to be portable, meaning they have to run on multiple hardware platforms. They also have to support thousands of I/O devices, all of which are independently designed with no

regard to one another. An example of where this diversity causes problems is the need for an operating system to run on both little-endian and big-endian machines. A second example was seen constantly under MS-DOS when users attempted to install, say, a sound card and a modem that used the same I/O ports or interrupt request lines. Few programs other than operating systems have to deal with sorting out problems caused by conflicting pieces of hardware.

Eighth, and last in our list, is the frequent need to be backward compatible with some previous operating system. That system may have restrictions on word lengths, file names, or other aspects that the designers now regard as obsolete, but are stuck with. It is like converting a factory to produce next year's cars instead of this year's cars, but while continuing to produce this year's cars at full capacity.

12.2 INTERFACE DESIGN

It should be clear by now that writing a modern operating system is not easy. But where does one begin? Probably the best place to begin is to think about the interfaces it provides. An operating system provides a set of abstractions, mostly implemented by data types (e.g., files) and operations on them (e.g., read). Together, these form the interface to its users. Note that in this context the users of the operating system are programmers who write code that use system calls, not people running application programs.

In addition to the main system-call interface, most operating systems have additional interfaces. For example, some programmers need to write device drivers to insert into the operating system. These drivers see certain features and can make certain procedure calls. These features and calls also define an interface, but a very different one from one application programmers see. All of these interfaces must be carefully designed if the system is to succeed.

12.2.1 Guiding Principles

Are there any principles that can guide interface design? We believe there are. Briefly summarized, they are simplicity, completeness, and the ability to be implemented efficiently.

Principle 1: Simplicity

A simple interface is easier to understand and implement in a bug-free way. All system designers should memorize this famous quote from the pioneer French aviator and writer, Antoine de St. Exupéry:

> *Perfection is reached not when there is no longer anything to add, but when there is no longer anything to take away.*

If you want to get really picky, he didn't say that. He said:

> *Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.*

But you get the idea. Memorize it either way.

This principle says that less is better than more, at least in the operating system itself. Another way to say this is the KISS principle: Keep It Simple, Stupid.

Principle 2: Completeness

Of course, the interface must make it possible to do everything that the users need to do, that is, it must be complete. This brings us to another famous quote, this one from Albert Einstein:

> *Everything should be as simple as possible, but no simpler.*

In other words, the operating system should do exactly what is needed of it and no more. If users need to store data, it must provide some mechanism for storing data. If users need to communicate with each other, the operating system has to provide a communication mechanism, and so on. In his 1991 Turing Award lecture, Fernando Corbató, one of the designers of CTSS and MULTICS, combined the concepts of simplicity and completeness and said:

> *First, it is important to emphasize the value of simplicity and elegance, for complexity has a way of compounding difficulties and as we have seen, creating mistakes. My definition of elegance is the achievement of a given functionality with a minimum of mechanism and a maximum of clarity.*

The key idea here is *minimum of mechanism*. In other words, every feature, function, and system call should carry its own weight. It should do one thing and do it well. When a member of the design team proposes extending a system call or adding some new feature, the others should ask whether something awful would happen if it were left out. If the answer is: "No, but somebody might find this feature useful some day," put it in a user-level library, not in the operating system, even if it is slower that way. Not every feature has to be faster than a speeding bullet. The goal is to preserve what Corbató called minimum of mechanism.

Let us briefly consider two examples from our own experience: MINIX (Tanenbaum and Woodhull, 2006) and Amoeba (Tanenbaum et al., 1990). For all intents and purposes, MINIX until very recently had only three kernel calls: send, receive, and sendrec. The system is structured as a collection of processes, with the memory manager, the file system, and each device driver being a separate schedulable process. To a first approximation, all the kernel does is schedule processes and handle message passing between them. Consequently, only two system calls were needed: send, to send a message, and receive, to receive one. The third call, sendrec, is simply an optimization for efficiency reasons to allow a message

to be sent and the reply to be requested with only one kernel trap. Everything else is done by requesting some other process (e.g., the file-system process or the disk driver) to do the work. The most recent version of MINIX added two additional calls, both for asynchronous communication. The senda call sends an asynchronous message. The kernel will attempt to deliver the message, but the application does not wait for this; it just keeps running. Similarly, the system uses the notify call to deliver short notifications. For instance, the kernel can notify a device driver in user space that something happened—much like an interrupt. There is no message associated with a notification. When the kernel delivers a notification to process, all it does is flip a bit in a per-process bitmap indicating that something happened. Because it is so simple, it can be fast and the kernel does not need to worry about what message to deliver if the process receives the same notification twice. It is worth observing that while the number of calls is still very small, it is growing. Bloat is inevitable. Resistance is futile.

Of course, these are just the kernel calls. Running a POSIX compliant system on top of it, requires implementing a lot of POSIX system calls. But the beauty of it is that they all map on just a tiny set of kernel calls. With a system that is (still) so simple, there is a chance we may even get it right.

Amoeba is even simpler. It has only one system call: perform remote procedure call. This call sends a message and waits for a reply. It is essentially the same as MINIX' sendrec. Everything else is built on this one call. Whether or not synchronous communication is the way to go is another matter, one that we will return to in Sec. 12.3.

Principle 3: Efficiency

The third guideline is efficiency of implementation. If a feature or system call cannot be implemented efficiently, it is probably not worth having. It should also be intuitively obvious to the programmer about how much a system call costs. For example, UNIX programmers expect the lseek system call to be cheaper than the read system call because the former just changes a pointer in memory while the latter performs disk I/O. If the intuitive costs are wrong, programmers will write inefficient programs.

12.2.2 Paradigms

Once the goals have been established, the design can begin. A good starting place is thinking about how the customers will view the system. One of the most important issues is how to make all the features of the system hang together well and present what is often called **architectural coherence**. In this regard, it is important to distinguish two kinds of operating system "customers." On the one hand, there are the *users*, who interact with application programs; on the other are the *programmers*, who write them. The former mostly deal with the GUI; the latter

mostly deal with the system call interface. If the intention is to have a single GUI that pervades the complete system, as in the Macintosh, the design should begin there. If, on the other hand, the intention is to support many possible GUIs, such as in UNIX, the system-call interface should be designed first. Doing the GUI first is essentially a top-down design. The issues are what features it will have, how the user will interact with it, and how the system should be designed to support it. For example, if most programs display icons on the screen and then wait for the user to click on one of them, this suggests an event-driven model for the GUI and probably also for the operating system. On the other hand, if the screen is mostly full of text windows, then a model in which processes read from the keyboard is probably better.

Doing the system-call interface first is a bottom-up design. Here the issues are what kinds of features programmers in general need. Actually, not many special features are needed to support a GUI. For example, the UNIX windowing system, X, is just a big C program that does reads and writes on the keyboard, mouse, and screen. X was developed long after UNIX and did not require many changes to the operating system to get it to work. This experience validated the fact that UNIX was sufficiently complete.

User-Interface Paradigms

For both the GUI-level interface and the system-call interface, the most important aspect is having a good paradigm (sometimes called a metaphor) to provide a way of looking at the interface. Many GUIs for desktop machines use the WIMP paradigm that we discussed in Chap. 5. This paradigm uses point-and-click, point-and-double-click, dragging, and other idioms throughout the interface to provide an architectural coherence to the whole. Often there are additional requirements for programs, such as having a menu bar with FILE, EDIT, and other entries, each of which has certain well-known menu items. In this way, users who know one program can quickly learn another.

However, the WIMP user interface is not the only one possible. Tablets, smartphones and some laptops use touch screens to allow users to interact more directly and more intuitively with the device. Some palmtop computers use a stylized handwriting interface. Dedicated multimedia devices may use a VCR-like interface. And of course, voice input has a completely different paradigm. What is important is not so much the paradigm chosen, but the fact that there is a single overriding paradigm that unifies the entire user interface.

Whatever paradigm is chosen, it is important that all application programs use it. Consequently, the system designers need to provide libraries and tool kits to application developers that give them access to procedures that produce the uniform look-and-feel. Without tools, application developers will all do something different. User interface design is important, but it is not the subject of this book, so we will now drop back down to the subject of the operating system interface.

Execution Paradigms

Architectural coherence is important at the user level, but equally important at the system-call interface level. It is often useful to distinguish between the execution paradigm and the data paradigm, so we will do both, starting with the former.

Two execution paradigms are widespread: algorithmic and event driven. The **algorithmic paradigm** is based on the idea that a program is started to perform some function that it knows in advance or gets from its parameters. That function might be to compile a program, do the payroll, or fly an airplane to San Francisco. The basic logic is hardwired into the code, with the program making system calls from time to time to get user input, obtain operating system services, and so on. This approach is outlined in Fig. 12-1(a).

```
main( )
{
      int ... ;

      init( );
      do_something( );
      read(...);
      do_something_else( );
      write(...);
      keep_going( );
      exit(0);
}
```

(a)

```
main( )
{
      mess_t msg;

      init( );
      while (get_message(&msg)) {
            switch (msg.type) {
                  case 1: ... ;
                  case 2: ... ;
                  case 3: ... ;
            }
      }
}
```

(b)

Figure 12-1. (a) Algorithmic code. (b) Event-driven code.

The other execution paradigm is the **event-driven paradigm** of Fig. 12-1(b). Here the program performs some kind of initialization, for example by displaying a certain screen, and then waits for the operating system to tell it about the first event. The event is often a key being struck or a mouse movement. This design is useful for highly interactive programs.

Each of these ways of doing business engenders its own programming style. In the algorithmic paradigm, algorithms are central and the operating system is regarded as a service provider. In the event-driven paradigm, the operating system also provides services, but this role is overshadowed by its role as a coordinator of user activities and a generator of events that are consumed by processes.

Data Paradigms

The execution paradigm is not the only one exported by the operating system. An equally important one is the data paradigm. The key question here is how system structures and devices are presented to the programmer. In early FORTRAN

batch systems, everything was modeled as a sequential magnetic tape. Card decks read in were treated as input tapes, card decks to be punched were treated as output tapes, and output for the printer was treated as an output tape. Disk files were also treated as tapes. Random access to a file was possible only by rewinding the tape corresponding to the file and reading it again.

The mapping was done using job control cards like these:

```
MOUNT(TAPE08, REEL781)
RUN(INPUT, MYDATA, OUTPUT, PUNCH, TAPE08)
```

The first card instructed the operator to go get tape reel 781 from the tape rack and mount it on tape drive 8. The second card instructed the operating system to run the just-compiled FORTRAN program, mapping *INPUT* (meaning the card reader) to logical tape 1, disk file *MYDATA* to logical tape 2, the printer (called *OUTPUT*) to logical tape 3, the card punch (called *PUNCH*) to logical tape 4. and physical tape drive 8 to logical tape 5.

FORTRAN had a well-defined syntax for reading and writing logical tapes. By reading from logical tape 1, the program got card input. By writing to logical tape 3, output would later appear on the printer. By reading from logical tape 5, tape reel 781 could be read in, and so on. Note that the tape idea was just a paradigm to integrate the card reader, printer, punch, disk files, and tapes. In this example, only logical tape 5 was a physical tape; the rest were ordinary (spooled) disk files. It was a primitive paradigm, but it was a start in the right direction.

Later came UNIX, which goes much further using the model of "everything is a file." Using this paradigm, all I/O devices are treated as files and can be opened and manipulated as ordinary files. The C statements

```
fd1 = open("file1", O_RDWR);
fd2 = open("/dev/tty", O_RDWR)'
```

open a true disk file and the user's terminal (keyboard + display). Subsequent statements can use *fd1* and *fd2* to read and write them, respectively. From that point on, there is no difference between accessing the file and accessing the terminal, except that seeks on the terminal are not allowed.

Not only does UNIX unify files and I/O devices, but it also allows other processes to be accessed over pipes as files. Furthermore, when mapped files are supported, a process can get at its own virtual memory as though it were a file. Finally, in versions of UNIX that support the */proc* file system, the C statement

```
fd3 = open("/proc/501", O_RDWR);
```

allows the process to (try to) access process 501's memory for reading and writing using file descriptor *fd3*, something useful for, say, a debugger.

Of course, just because someone says that everything is a file does not mean it is true—for everything. For instance, UNIX network sockets may resemble files somewhat, but they have their own, fairly different, socket API. Another operating

system, Plan 9 from Bell Labs, has not compromised and does not provide specialized interfaces for network sockets and such. As a result, the Plan 9 design is arguably cleaner.

Windows tries to make everything look like an object. Once a process has acquired a valid handle to a file, process, semaphore, mailbox, or other kernel object, it can perform operations on it. This paradigm is even more general than that of UNIX and much more general than that of FORTRAN.

Unifying paradigms occur in other contexts as well. One of them is worth mentioning here: the Web. The paradigm behind the Web is that cyberspace is full of documents, each of which has a URL. By typing in a URL or clicking on an entry backed by a URL, you get the document. In reality, many "documents" are not documents at all, but are generated by a program or shell script when a request comes in. For example, when a user asks an online store for a list of CDs by a particular artist, the document is generated on-the-fly by a program; it certainly did not exist before the query was made.

We have now seen four cases: namely, everything is a tape, file, object, or document. In all four cases, the intention is to unify data, devices, and other resources to make them easier to deal with. Every operating system should have such a unifying data paradigm.

12.2.3 The System-Call Interface

If one believes in Corbató's dictum of minimal mechanism, then the operating system should provide as few system calls as it can get away with, and each one should be as simple as possible (but no simpler). A unifying data paradigm can play a major role in helping here. For example, if files, processes, I/O devices, and much more all look like files or objects, then they can all be read with a single `read` system call. Otherwise it may be necessary to have separate calls for `read_file`, `read_proc`, and `read_tty`, among others.

Sometimes, system calls may need several variants, but it is often good practice to have one call that handles the general case, with different library procedures to hide this fact from the programmers. For example, UNIX has a system call for overlaying a process' virtual address space, `exec`. The most general call is

```
exec(name, argp, envp);
```

which loads the executable file *name* and gives it arguments pointed to by *argp* and environment variables pointed to by *envp*. Sometimes it is convenient to list the arguments explicitly, so the library contains procedures that are called as follows:

```
execl(name, arg0, arg1, ..., argn, 0);
execle(name, arg0, arg1, ..., argn, envp);
```

All these procedures do is stick the arguments in an array and then call `exec` to do the real work. This arrangement is the best of both worlds: a single straightforward

system call keeps the operating system simple, yet the programmer gets the convenience of various ways to call exec.

Of course, trying to have one call to handle every possible case can easily get out of hand. In UNIX creating a process requires two calls: fork followed by exec. The former has no parameters; the latter has three. In contrast, the WinAPI call for creating a process, CreateProcess, has 10 parameters, one of which is a pointer to a structure with an additional 18 parameters.

A long time ago, someone should have asked whether something awful would happen if some of these had been omitted. The truthful answer would have been in some cases programmers might have to do more work to achieve a particular effect, but the net result would have been a simpler, smaller, and more reliable operating system. Of course, the person proposing the 10 + 18 parameter version might have added: "But users like all these features." The rejoinder might have been they like systems that use little memory and never crash even more. Trade-offs between more functionality at the cost of more memory are at least visible and can be given a price tag (since the price of memory is known). However, it is hard to estimate the additional crashes per year some feature will add and whether the users would make the same choice if they knew the hidden price. This effect can be summarized in Tanenbaum's first law of software:

Adding more code adds more bugs.

Adding more features adds more code and thus adds more bugs. Programmers who believe adding new features does not add new bugs either are new to computers or believe the tooth fairy is out there watching over them.

Simplicity is not the only issue that comes out when designing system calls. An important consideration is Lampson's (1984) slogan:

Don't hide power.

If the hardware has an extremely efficient way of doing something, it should be exposed to the programmers in a simple way and not buried inside some other abstraction. The purpose of abstractions is to hide undesirable properties, not hide desirable ones. For example, suppose the hardware has a special way to move large bitmaps around the screen (i.e., the video RAM) at high speed. It would be justified to have a new system call to get at this mechanism, rather than just provide ways to read video RAM into main memory and write it back again. The new call should just move bits and nothing else. If a system call is fast, users can always build more convenient interfaces on top of it. If it is slow, nobody will use it.

Another design issue is connection-oriented vs. connectionless calls. The Windows and UNIX system calls for reading a file are connection-oriented, like using the telephone. First you open a file, then you read it, finally you close it. Some remote file-access protocols are also connection-oriented. For example, to use FTP, the user first logs in to the remote machine, reads the files, and then logs out.

On the other hand, some remote file-access protocols are connectionless. The Web protocol (HTTP) is connectionless. To read a Web page you just ask for it; there is no advance setup required (a TCP connection *is* required, but this is at a lower level of protocol. HTTP itself is connectionless).

The trade-off between any connection-oriented mechanism and a connectionless one is the additional work required to set up the mechanism (e.g., open the file), and the gain from not having to do it on (possibly many) subsequent calls. For file I/O on a single machine, where the setup cost is low, probably the standard way (first open, then use) is the best way. For remote file systems, a case can be made both ways.

Another issue relating to the system-call interface is its visibility. The list of POSIX-mandated system calls is easy to find. All UNIX systems support these, as well as a small number of other calls, but the complete list is always public. In contrast, Microsoft has never made the list of Windows system calls public. Instead the WinAPI and other APIs have been made public, but these contain vast numbers of library calls (over 10,000) but only a small number are true system calls. The argument for making all the system calls public is that it lets programmers know what is cheap (functions performed in user space) and what is expensive (kernel calls). The argument for not making them public is that it gives the implementers the flexibility of changing the actual underlying system calls to make them better without breaking user programs. As we saw in Sec. 9.7.7, the original designers simply got it wrong with the access system call, but now we are stuck with it.

12.3 IMPLEMENTATION

Turning away from the user and system-call interfaces, let us now look at how to implement an operating system. In the following sections we will examine some general conceptual issues relating to implementation strategies. After that we will look at some low-level techniques that are often helpful.

12.3.1 System Structure

Probably the first decision the implementers have to make is what the system structure should be. We examined the main possibilities in Sec. 1.7, but will review them here. An unstructured monolithic design is not a good idea, except maybe for a tiny operating system in, say, a toaster, but even there it is arguable.

Layered Systems

A reasonable approach that has been well established over the years is a layered system. Dijkstra's THE system (Fig. 1-25) was the first layered operating system. UNIX and Windows 8 also have a layered structure, but the layering in both

of them is more a way of trying to describe the system than a real guiding principle that was used in building the system.

For a new system, designers choosing to go this route should *first* very carefully choose the layers and define the functionality of each one. The bottom layer should always try to hide the worst idiosyncracies of the hardware, as the HAL does in Fig. 11-4. Probably the next layer should handle interrupts, context switching, and the MMU, so above this level the code is mostly machine independent. Above this, different designers will have different tastes (and biases). One possibility is to have layer 3 manage threads, including scheduling and interthread synchronization, as shown in Fig. 12-2. The idea here is that starting at layer 4 we have proper threads that are scheduled normally and synchronize using a standard mechanism (e.g., mutexes).

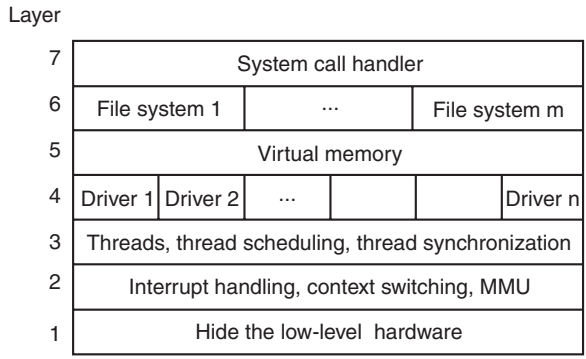


Figure 12-2. One possible design for a modern layered operating system.

In layer 4 we might find the device drivers, each one running as a separate thread, with its own state, program counter, registers, and so on, possibly (but not necessarily) within the kernel address space. Such a design can greatly simplify the I/O structure because when an interrupt occurs, it can be converted into an unlock on a mutex and a call to the scheduler to (potentially) schedule the newly readied thread that was blocked on the mutex. MINIX 3 uses this approach, but in UNIX, Linux, and Windows 8, the interrupt handlers run in a kind of no-man's land, rather than as proper threads like other threads that can be scheduled, suspended, and the like. Since a huge amount of the complexity of any operating system is in the I/O, any technique for making it more tractable and encapsulated is worth considering.

Above layer 4, we would expect to find virtual memory, one or more file systems, and the system-call handlers. These layers are focused on providing services to applications. If the virtual memory is at a lower level than the file systems, then the block cache can be paged out, allowing the virtual memory manager to dynamically determine how the real memory should be divided among user pages and kernel pages, including the cache. Windows 8 works this way.

Exokernels

While layering has its supporters among system designers, another camp has precisely the opposite view (Engler et al., 1995). Their view is based on the **end-to-end argument** (Saltzer et al., 1984). This concept says that if something has to be done by the user program itself, it is wasteful to do it in a lower layer as well.

Consider an application of that principle to remote file access. If a system is worried about data being corrupted in transit, it should arrange for each file to be checksummed at the time it is written and the checksum stored along with the file. When a file is transferred over a network from the source disk to the destination process, the checksum is transferred, too, and also recomputed at the receiving end. If the two disagree, the file is discarded and transferred again.

This check is more accurate than using a reliable network protocol since it also catches disk errors, memory errors, software errors in the routers, and other errors besides bit transmission errors. The end-to-end argument says that using a reliable network protocol is then not necessary, since the endpoint (the receiving process) has enough information to verify the correctness of the file. The only reason for using a reliable network protocol in this view is for efficiency, that is, catching and repairing transmission errors earlier.

The end-to-end argument can be extended to almost all of the operating system. It argues for not having the operating system do anything that the user 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).

Microkernel-Based 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. 12-3. 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, but even having the device drivers run in the kernel adds to the modularity.

When the device drivers are in the kernel, they can access the hardware device registers directly. When they are not, some mechanism is needed to provide access to them. 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

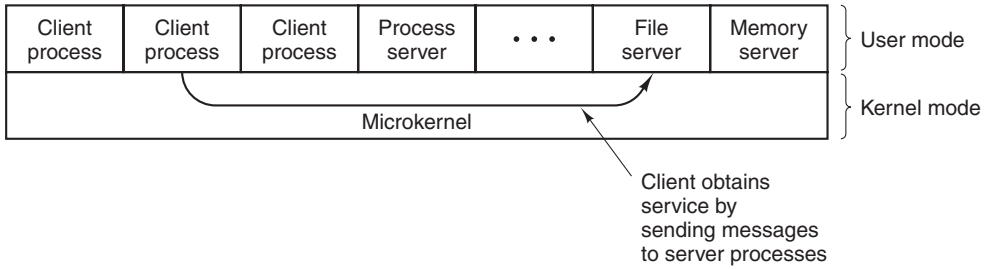


Figure 12-3. Client-server computing based on a microkernel.

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.

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 with 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 95% of the time. If a microkernel-based operating system turned an unreliable 3.5-GHz system into a reliable 3.0-GHz system, probably few users would complain. Or even notice. 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 1 GHz. Also, it is not clear whether the cost of interprocess communication is still as much of an issue if cores are no longer a scarce resource. If each device driver and each component of the operating system has its own dedicated core, there is no context switching during interprocess communication. In addition, the caches, branch predictors and TLBs will be all warmed up and ready to run at full speed. Some experimental work on a high-performance operating system based on a microkernel was presented by Hruby et al. (2013).

It is noteworthy that while microkernels are not popular on the desktop, they are very widely used in cell phones, industrial systems, embedded systems, and

military systems, where very high reliability is absolutely essential. Also, Apple's OS X, which runs on all Macs and Macbooks, consists of a modified version of FreeBSD running on top of a modified version of the Mach microkernel.

Extensible Systems

With the client-server systems discussed above, the idea was to remove 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 the 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 no matter which structuring model is chosen is that of system threads. 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 forth. 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. While these are not part of the operating system, they often perform "system" type activities. These might including getting and sending email and serving various kinds of requests for remote users, such as FTP and Web pages.

12.3.2 Mechanism vs. 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 mechanisms 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 food suppliers and 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, let us consider thread scheduling. The kernel could have a priority scheduler, with k priority levels. The mechanism is an array, indexed by priority level, as is the case in UNIX and Windows 8. 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 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 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 combining

data types, including arrays, structures, and unions. These ideas combine independently, allowing arrays of integers, arrays of characters, structures and union members that are floating-point numbers, and so forth. 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 a 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 be 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 8. 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 8, 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 even more fine-grained building blocks are 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 to. 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 that were primarily designed for human beings to use are character-string names in ASCII or Unicode and are usually hierarchical. Directory paths, such as */usr/ast/books/mos4/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 in ASCII or Unicode 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 8 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. 12-4.

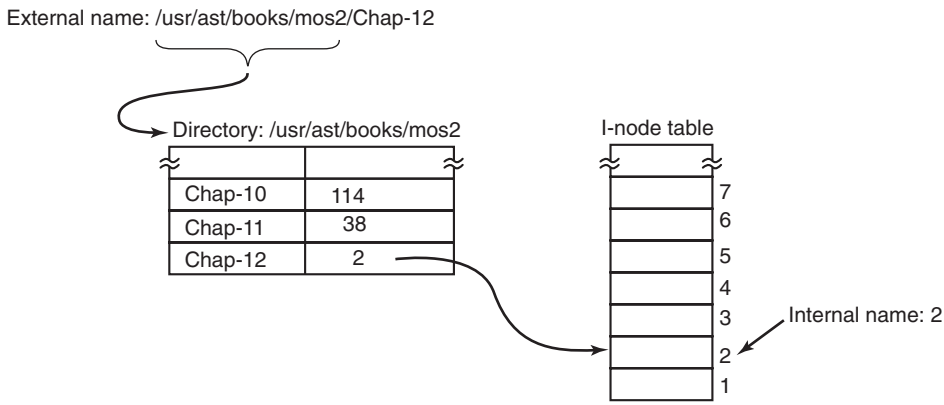


Figure 12-4. Directories are used to map external names onto internal names.

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 8. 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 commonly support multiple namespaces, both external and internal. For example, in Chap. 11 we looked at three external namespaces supported by Windows 8: file names, object names, and registry names (and there is also the Active Directory namespace, which we did not look at). In addition, there are innumerable internal namespaces using unsigned integers, for example, object

handles and MFT entries. Although the names in the external namespaces 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 namespaces 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, when the name is bound to the object may matter. In general, **early binding** is simple, but 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, 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 vs. 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 systemwide), 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 for a given PID, *pid*, is given in Fig. 12-5. 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 12-5. 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 128 processes at once, and it is not a total disaster if an attempt to start a 129th 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 from user mode 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. Making key data structures like these dynamic is possible, but complicated.

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 vs. 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. 11-4. 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 of programmers 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 Synchronous vs. Asynchronous Communication

Another issue that often creeps up in conversations between operating system designers is whether the interactions between the system components should be synchronous or asynchronous (and, related, whether threads are better than events). The issue frequently leads to heated arguments between proponents of the two camps, although it does not leave them foaming at the mouth quite as much as when deciding really important matters—like which is the best editor, *vi* or *emacs*. We use the term "synchronous" in the (loose) sense of Sec. 8.2 to denote calls that block until completion. Conversely, with "asynchronous" calls the caller keeps running. There are advantages and disadvantages to either model.

Some systems, like Amoeba, really embrace the synchronous design and implement communication between processes as blocking client-server calls. Fully synchronous communication is conceptually very simple. A process sends a request and blocks waiting until the reply arrives—what could be simpler? It becomes a little more complicated when there are many clients all crying for the server's attention. Each individual request may block for a long time waiting for other requests to complete first. This can be solved by making the server multi-threaded so that each thread can handle one client. The model is tried and tested in many real-world implementations, in operating systems as well as user applications.

Things get more complicated still if the threads frequently read and write shared data structures. In that case, locking is unavoidable. Unfortunately, getting the locks right is not easy. The simplest solution is to throw a single big lock on all shared data structures (similar to the big kernel lock). Whenever a thread wants to access the shared data structures, it has to grab the lock first. For performance reasons, a single big lock is a bad idea, because threads end up waiting for each other all the time even if they do not conflict at all. The other extreme, lots of micro locks for (parts) of individual data structures, is much faster, but conflicts with our guiding principle number one: simplicity.

Other operating systems build their interprocess communication using asynchronous primitives. In a way, asynchronous communication is even simpler than its synchronous cousin. A client process sends a message to a server, but rather than wait for the message to be delivered or a reply to be sent back, it just continues executing. Of course, this means that it also receives the reply asynchronously and should remember which request corresponded to it when it arrives. The server typically processes the requests (events) as a single thread in an event loop.

Whenever the request requires the server to contact other servers for further processing it sends an asynchronous message of its own and, rather than block, continues with the next request. Multiple threads are not needed. With only a single thread processing events, the problem of multiple threads accessing shared data structures cannot occur. On the other hand, a long-running event handler makes the single-threaded server's response sluggish.

Whether threads or events are the better programming model is a long-standing controversial issue that has stirred the hearts of zealots on either side ever since John Ousterhout's classic paper: "Why threads are a bad idea (for most purposes)" (1996). Ousterhout argues that threads make everything needlessly complicated: locking, debugging, callbacks, performance—you name it. Of course, it would not be a controversy if everybody agreed. A few years after Ousterhout's paper, Von Behren et al. (2003) published a paper titled "Why events are a bad idea (for high-concurrency servers)." Thus, deciding on the right programming model is a hard, but important decision for system designers. There is no slam-dunk winner. Web servers like *apache* firmly embrace synchronous communication and threads, but others like *lighttpd* are based on the **event-driven paradigm**. Both are very popular. In our opinion, events are often easier to understand and debug than threads. As long as there is no need for per-core concurrency, they are probably a good choice.

12.3.9 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 others, of course, but space limitations restrict us to just a few.

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 as layer 1. 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 to immediately 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 others 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, and the like. 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, and so on. For example, imagine an operating system that is to run on the IA32 line of x86 chips (sometimes referred to as x86-32), or on UltraSPARC chips, which need different initialization code. The *init* procedure could be written as illustrated in Fig. 12-6(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.

As a second example, suppose there is a need for a data type *Register*, which should be 32 bits on the IA32 and 64 bits on the UltraSPARC. This could be handled by the conditional code of Fig. 12-6(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 IA32 it might be something like this:

```
#define CPU IA32
#define WORD_LENGTH 32
```

```
#include "config.h"

init( )
{
#if (CPU == IA32)
/* IA32 initialization here. */
#endif

#if (CPU == ULTRASPARC)
/* UltraSPARC initialization here. */
#endif

}
```

(a)

```
#include "config.h"

#if (WORD_LENGTH == 32)
typedef int Register;
#endif

#if (WORD_LENGTH == 64)
typedef long Register;
#endif

Register R0, R1, R2, R3;
```

(b)

Figure 12-6. (a) CPU-dependent conditional compilation. (b) Word-length-dependent conditional compilation.

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 IA32 and 64 bits for the UltraSPARC. However, this is not a good idea. Consider what happens when we later port the system to the 32-bit ARM. We would have to add a third conditional to Fig. 12-6(b) for the ARM. By doing it as we have, all we have to do is include the line

```
#define WORD_LENGTH 32
```

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

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 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 x86-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 in which 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 to the ability of code to be executed two or more times simultaneously. On a multiprocessor, there is always the danger than 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 has used up its quantum and the operating system is moving it to the end of its queue. Partway 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 few 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 to 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 1000), 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. Sorting or hashing the mount table (which keeps track of mounted file systems in UNIX systems) really is not a good idea.

Of course, for functions that are on the critical path, say, context switching, everything should be done to make them very 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 invoked. If there is one fork every second, and it takes 1 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 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 gamut 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 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. No big deal. However, over a period of time, this bug may be triggered multiple times. Eventually, most or all of 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 a 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 worth the effort. In the following sections, we will look at some 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 8 can take several minutes to boot, despite running on hardware that is 1000 times faster. 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 authors' 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 it 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 linguistic preferences) 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 whether some users like it, but whether it is 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 wee 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 a steady stream of 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 authors (AST) had a former student (who shall here remain nameless) who wrote the original 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 on the hard disk once during the life of the computer, when the system is installed. It also runs once for each 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 the 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.

Another concern is what to optimize when. Some programmers have a tendency to optimize to death whatever they develop, as soon as it is appears to work. The problem is that after optimization, the system may be less clean, making it harder to maintain and debug. Also, it makes it harder to adapt it, and perhaps do more fruitful optimization later. The problem is known as premature optimization. Donald Knuth, sometimes referred to as the father of the analysis of algorithms, once said that "premature optimization is the root of all evil."

12.4.3 Space-Time Trade-offs

One general approach to improving performance is to trade off time vs. 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 sometimes helpful is to replace small procedures by macros. Using a macro eliminates the overhead that is 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 will need a procedure, *bit_count*, that counts the number of 1 bits in a byte. The

obvious procedure is given in Fig. 12-7(a). It loops over the bits in a byte, counting them one at a time. It is pretty simple and straightforward.

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

int bit_count(int byte)
{                                            /* Count the bits in a 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 12-7. (a) A procedure for counting bits in a byte. (b) A macro to count the bits. (c) A macro that counts bits by table lookup.

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. 12-7(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 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. 12-7(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 also be used in other ways. Anwell-known image-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. 12-8. The original compressed image is shown in Fig. 12-8(a). Each value is a 24-bit value, with 8 bits for the intensity of red, green, and blue, respectively. The GIF image is shown in Fig. 12-8(b). 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. 12-8(c). Actually, there is more to GIF, but the core idea is table lookup.

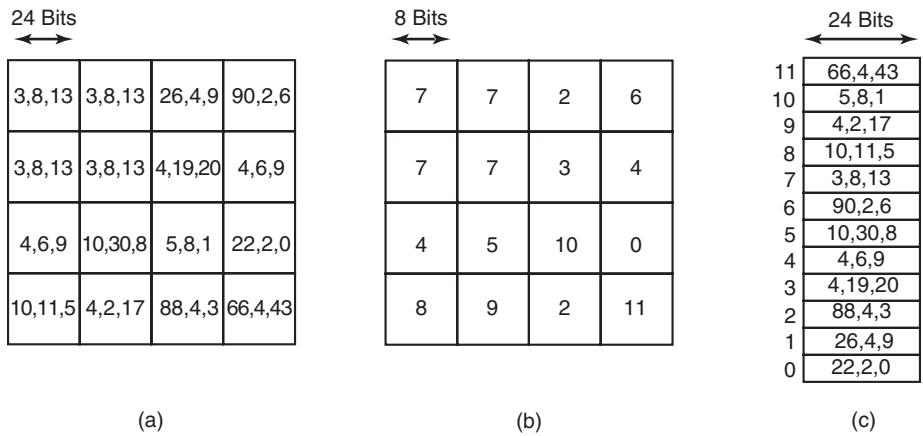


Figure 12-8. (a) Part of an uncompressed image with 24 bits per pixel. (b) The same part compressed with GIF, with 8 bits per pixel. (c) The color palette.

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 trade-off.

12.4.4 Caching

A well-known technique for improving performance is caching. It is applicable whenever 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. 4-34 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. 4-34, the cache will certainly hold the first three entries of Fig. 12-9 after parsing */usr/ast/mbox*. The last three entries come from parsing other paths.

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

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

Figure 12-9. Part of the i-node cache for Fig. 4-34.

/usr/ast/grants/erc 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 are cacheable. 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 called 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 10 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 in 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 8 takes advantage of this expectation by providing a WinAPI 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. In Chap. 2 we saw that futexes on Linux likewise optimize for the common case of no contention.

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, *The Mythical Man Month*, 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 the 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 1 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 even be 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 that everyone has something to do. Since every module might potentially interact with every other one, 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 10 debuggers on a problem does not find the bug 10 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 in France 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-butchering 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. 12-10.

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

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 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.

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 his manager wants to tell the big boss that the project is 4 months late and has no chance whatsoever of meeting the deadline because there is a 2000-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 undeniably obvious that the deadline cannot be met under any conditions, top management panics and 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. While many expressed doubt (especially those leading large closed-source software companies), open source software has been a tremendous success. From large servers to embedded devices, and from industrial control systems to handheld smartphones, open source software is everywhere. Large companies like Google and IBM are throwing their weight behind Linux now and contribute heavily in code. 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 absolutely critical to any software 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 illustrated in Fig. 12-11(a) and use the model of Fig. 12-11(b). Here the idea is to first write a main program that merely calls the top-level procedures, initially dummies. Starting on day 1 of the project, the system will compile and run, although it does nothing. As time goes on, real modules replace the dummies. The result is that system integration testing is performed continuously, so errors in the design show up much earlier, so the learning process caused by bad design starts earlier.

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 commercial success. The third system in this line of development, UNIX, was much more cautious and much more successful.

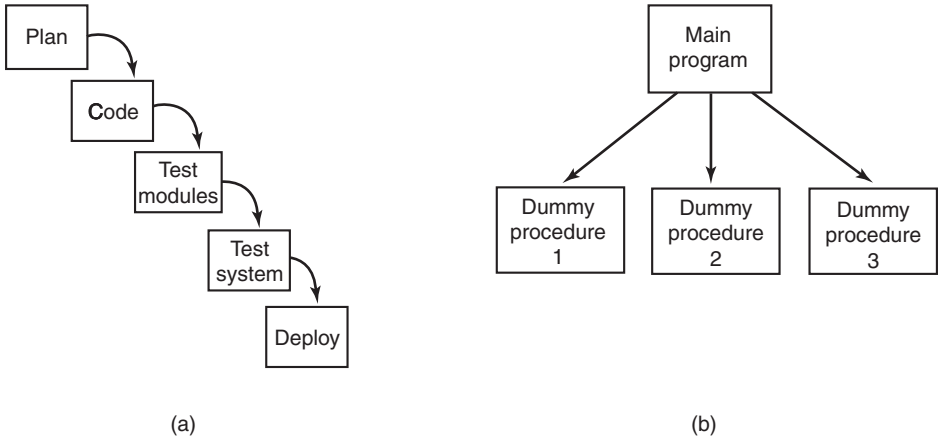


Figure 12-11. (a) Traditional software design progresses in stages. (b) Alternative design produces a working system (that does nothing) starting on day 1.

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

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. The point is that the world is constantly changing and operating systems must adapt to the new reality all the time. In this section, we mention a few trends that are relevant for operating system designers today.

To avoid confusion, the **hardware developments** mentioned below are here already. What is not here is the operating system software to use them effectively.

Generally, when new hardware arrives, what everyone does is just plop the old software (Linux, Windows, etc.) down on it and call it a day. In the long run, this is a bad idea. What we need is innovative software to deal with innovative hardware. If you are a computer science or engineering student or an ICT professional, your homework assignment is to think up this software.

12.6.1 Virtualization and the Cloud

Virtualization is an idea whose time has definitely come—again. It first surfaced in 1967 with the IBM CP/CMS system, but now it is back in full force on the x86 platform. Many computers are now running hypervisors on the bare hardware, as illustrated in Fig. 12-12. The hypervisor creates a number of virtual machines, each with its own operating system. This phenomenon was discussed in Chap. 7 and appears to be the wave of the future. Nowadays, many companies are taking the idea further by virtualizing other resources also. For instance, there is much interest in virtualizing the control of network equipment, even going so far as running the control of their networks in the cloud also. In addition, vendors and researchers constantly work on making hypervisors better for some notion of better: smaller, faster, or with provable isolation properties.

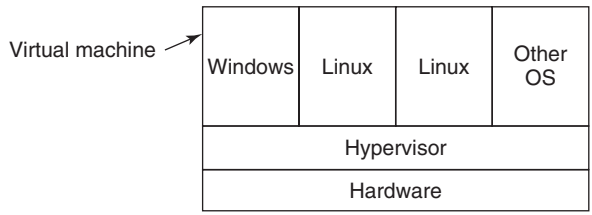


Figure 12-12. A hypervisor running four virtual machines.

12.6.2 Manycore Chips

There used to be a time that memory was so scarce that a programmer knew every byte in person and celebrated its birthday. Nowaways, programmers rarely worry about wasting a few megabytes here and there. For most applications, memory is no longer a scarce resource. What will happen when cores become equally plentiful? Phrased differently, as manufacturers are putting more and more cores on a die, what happens if there are so many that a programmers stops worrying about wasting a few cores here and there?

Manycore chips are here already, but the operating systems for them do not use them well. In fact, stock operating systems often do not even scale beyond a few dozens of cores and developers are constantly struggling to remove all the bottlenecks that limit scalability.

One obvious question is: what do you do with all the cores? If you run a popular server handling many thousands of client requests per second, the answer may be relatively simple. For instance, you may decide to dedicate a core to each request. Assuming you do not run into locking issues too much, this may work. But what do we do with all those cores on tablets?

Another question is: what *sort* of cores do we want? Deeply pipelined, superscalar cores with fancy out-of-order and speculative execution at high clock rates may be great for sequential code, but not for your energy bill. They also do not help much if your job exhibits a lot of parallelism. Many applications are better off with smaller and simpler cores, if they get more of them. Some experts argue for heterogeneous multicores, but the questions remain the same: what cores, how many, and at what speeds? And we have not even begun to mention the issue of running an operating system and all of its applications. Will the operating system run on all cores or only some? Will there be one or more network stacks? How much sharing is needed? Do we dedicate certain cores to specific operating system functions (like the network or storage stack)? If so, do we replicate such functions for better scalability?

Exploring many different directions, the operating system world is currently trying to formulate answers to these questions. While researchers may disagree on the answers, most of them agree on one thing: these are exciting times for systems research!

12.6.3 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 a 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 ran 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.

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 page 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.

12.6.4 Seamless Data Access

Ever since the dawn of computing, there has been a strong distinction between *this* machine and *that* machine. If the data was on *this* machine, you could not access it from *that* machine, unless you explicitly transferred it first. Similarly, even if you had the data, you could not use it unless you had the right software installed. This model is changing.

Nowadays, users expect much of the data to be accessible from anywhere at any time. Typically, this is accomplished by storing the data in the cloud using storage services like Dropbox, GoogleDrive, iCloud, and SkyDrive. All files stored there can be accessed from any device that has a network connection. Moreover, the programs to access the data often reside in the cloud too, so you do not even have to have all the programs installed either. It allows people to read and modify word-processor files, spreadsheets, and presentations using a smartphone on the toilet. This is generally regarded as progress.

To make this happen seamlessly is tricky and requires a lot of clever systems' solutions under the hood. For instance, what to do if there is no network connection? Clearly, you do not want to stop people from working. Of course, you could buffer changes locally and update the master document when the connection was re-established, but what if multiple devices have made conflicting changes? This is a very common problem if multiple users share data, but it could even happen with a single user. Moreover, if the file is large, you do not want to wait a long time until you can access it. Caching, preloading and synchronization are key issues here. Current operating systems deal with merging multiple machines in a seamful way (assuming that "seamful" is the opposite of "seamless") We can surely do a lot better.

12.6.5 Battery-Powered Computers

Powerful PCs with 64-bit address spaces, high-bandwidth networking, multiple processors, and high-quality audio and video, are now standard on desktop systems and moving rapidly into notebooks, tablets, and even smartphones. As this trend

continues, their operating systems will have to be appreciably different from current ones to handle all these demands. In addition, they must balance the power budget and "keep cool." Heat dissipation and power consumption are some of the most important challenges even in high-end computers.

However, an even faster growing segment of the market is battery-powered computers, including notebooks, tablets, $100 laptops, and smartphones. Most of these have wireless connections to the outside world. They demand operating systems that are smaller, faster, more flexible, and more reliable than operating systems on high-end devices. Many of these devices today are based on traditional operating systems like Linux, Windows and OS X, but with significant modification. In addition, they frequently use a microkernel/hypervisor-based solution to manage the radio stack.

These operating systems 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. In the future, 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 likely that the operating system for a battery-powered, handheld wireless, voice-operated computer will be appreciably different from that of a desktop 64-bit 16-core CPU 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, 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 PCIe 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 is the extensible operating system (e.g., Paramecium 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 vs. 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 are changing to adapt to new trends and meet new challenges. These include hypervisor-based systems, multicore systems, 64-bit address spaces, handheld wireless computers, and embedded systems. There is no doubt that 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 of the following paradigms is likely to be easiest to use?

(a) A compiler.
(b) A photo-editing program.
(c) A payroll program.

3. 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?

4. 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.

5. Suppose that layers 3 and 4 in Fig. 12-2 were exchanged. What implications would that have for the design of the system?

6. 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?

7. 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?

8. 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?

9. 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× the size of the original one, as compared to making it only 1.5× as big?

10. In Fig. 12-5, a flag, *found*, is used to tell whether the PID was located. Would it b 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?

11. In Fig. 12-6, the differences between the x86 and the UltraSPARC are hidden by conditional compilation. Could the same approach be used to hide the difference between x86 machines with an IDE disk as the only disk and x86 machines with a SCSI disk as the only disk? Would it be a good idea?

12. Indirection is a way of making an algorithm more flexible. Does it have any disadvantages, and if so, what are they?

13. Can reentrant procedures have private static global variables? Discuss your answer.

14. The macro of Fig. 12-7(b) is clearly much more efficient than the procedure of Fig. 12-7(a). One disadvantage, however, is that it is hard to read. Are there any other disadvantages? If so, what are they?

15. 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.

16. In Fig. 12-8, 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?

17. 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.

18. 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?

19. 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?

20. 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?

21. 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?

22. 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?

23. 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 8. 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?

24. 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 100-GB RAM-disk is sufficient for a low-end machine. Is this machine likely to be competitive?

25. Name some features of a conventional operating system that are not needed in an embedded system used inside an appliance.

26. 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.

27. 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 break-even point?

28. 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 to aid 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 *ACM Symposium on Operating Systems Principles* (SOSP) held in odd-numbered years and the *USENIX Symposium on Operating Systems Design and Implementation* (OSDI) held in even numbered years are good sources for ongoing work on operating systems. The *Eurosys Conference*, held annually is also a source of top-flight papers. Furthermore, the journals *ACM Transactions on Computer Systems* and *ACM SIGOPS Operating Systems Review*, often have relevant articles. Many other ACM, IEEE, and USENIX conferences deal with specialized topics.

13.1 SUGGESTIONS FOR FURTHER READING

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

13.1.1 Introduction

Silberschatz et al., *Operating System Concepts*, 9th ed.,

A general textbook on operating systems. It covers processes, memory management, storage management, protection and security, distributed systems, and some special-purpose systems. Two case studies are given: Linux and Windows 7. The cover is full of dinosaurs. These are legacy animals, to empahsize that operating systems also carry a lot of legacy.

Stallings, *Operating Systems*, 7th 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 and Rago, *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.

Tanenbaum and Woodhull, *Operating Systems Design and Implementation*

A hands-on way to learn about operating systems. This book discusses the usual principles, but in addition discusses an actual operating system, MINIX 3, in great detail, and contains a listing of that system as an appendix.

13.1.2 Processes and Threads

Arpaci-Dusseau and Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*

The entire first part of this book is dedicated to virtualization of the CPU to share it with multiple processes. What is nice about this book (besides the fact that there is a free online version), is that it introduces not only the concepts of processing and scheduling techniques, but also the APIs and systems calls like `fork` and `exec` in some detail.

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. The article is old, but it has stood the test of time very well.

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. It, too, has stood up very well over the years.

Zhuravlev et al., "Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors"

Multicore systems have started to dominate the field of general-purpose computing world. One of the most important challenges is shared resource contention. In this survey, the authors present different scheduling techniques for handling such contention.

Silberschatz et al., *Operating System Concepts*, 9th ed.,

Chapters 3 through 6 cover processes and interprocess communication, including scheduling, critical sections, semaphores, monitors, and classical interprocess communication problems.

Stratton et al., "Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems"

Programming a system with half a dozen threads is hard enough. But what happens when you have thousands of them? To say it gets tricky is to put it mildly. This article talks about approaches that are being taken.

13.1.3 Memory Management

Denning, "Virtual Memory"

A classic paper on many aspects of virtual memory. Peter 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. Although many of the papers are old, the principles really have not changed at all.

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

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

Arpaci-Dusseau and Arpaci-Dusseaum "Operating Systems: Three Easy Pieces"

This book has a rich section on virtual memory in Chapters 12 to 23 and includes a nice overview of page replacement policies.

13.1.4 File Systems

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., *Operating System Concepts*, 9th ed.,

Chapters 10–12 are about storage hardware and file systems. They cover file operations, interfaces, access methods, directories, and implementation, among other topics.

Stallings, *Operating Systems*, 7th ed.,

Chapter 12 contains a fair amount of material about file systems and little bit about their security.

Cornwell, "Anatomy of a Solid-state Drive"

If you are interested in solid state drives, Michael Cornwell's introduction is a good starting point. In particular, the author succinctly describes the way in way traditional hard drives and SSDs differ.

13.1.5 Input/Output

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

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

Scheible, "A Survey of Storage Options"

There are many ways to store bits these days: DRAM, SRAM, SDRAM, flash memory, hard disk, floppy disk, CD-ROM, DVD, and tape, to name a few. In this article, the various technologies are surveyed and their strengths and weaknesses highlighted.

Stan and Skadron, "Power-Aware Computing"

Until someone manages to get Moore's Law to apply to batteries, energy usage is going to continue to be a major issue in mobile devices. Power and heat are so critical these days that operating systems are aware of the CPU temperature and adapt their behavior to it. This article surveys some of the issues and serves as an introduction to five other articles in this special issue of *Computer* on power-aware computing.

Swanson and Caulfield, "Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage"

Disks exist for two reasons: when power is turned off, RAM loses its contents. Also, disks are very big. But suppose RAM did not lose its contents when powered off? How would that change the I/O stack? Nonvolatile memory is here and this article looks at how it changes systems.

Ion, "From Touch Displays to the Surface: A Brief History of Touchscreen Technology,"

Touch screens have become ubiquitous in a short time span. This article traces the history of the touch screen through history with easy-to-understand explanations and nice vintage pictures and videos. Fascinating stuff!

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 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.

Levine, "Defining Deadlock"

In Chap. 6 of this book, we focused on resource deadlocks and barely touched on other kinds. This short paper points out that in the literature, various definitions have been used, differing in subtle ways. The author then looks at communication, scheduling, and interleaved deadlocks and comes up with a new model that tries to cover all of them.

Shub, "A Unified Treatment of Deadlock"

This short tutorial summarizes the causes and solutions to deadlocks and suggests what to emphasize when teaching it to students.

13.1.7 Virtualization and the Cloud

Portnoy, "Virtualization Essentials"

A gentle introduction to virtualization. It covers the context (including the relation between virtualization and the cloud), and covers a variety of solutions (with a bit more emphasis on VMware).

Erl et al., *Cloud Computing: Concepts, Technology & Architecture*

A book devoted to cloud computing in a broad sense. The authors explain in

detail what is hidden behind acronyms like IAAS, PAAS, SAAS, and similar "X" As A Service family members.

Rosenblum and Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends"

Starting with a history of virtual machine monitors, this article then goes on to discuss the current state of CPU, memory, and I/O virtualization. In particular, it covers problem areas relating to all three and how future hardware may alleviate the problems.

Whitaker et al., "Rethinking the Design of Virtual Machine Monitors"

Most computers have some bizarre and difficult to virtualize aspects. In this paper, the authors of the Denali system argue for paravirtualization, that is, changing the guest operating systems to avoid using the bizarre features so that they need not be emulated.

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 two years or so.

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.

Geer, "For Programmers, Multicore Chips Mean Multiple Challenges"

Multicore chips are happening—whether the software folks are ready or not. As it turns out, they are not ready, and programming these chips offers many challenges, from getting the right tools, to dividing up the work into little pieces, to testing the results.

Kant and Mohapatra, "Internet Data Centers"

Internet data centers are massive multicomputers on steroids. They often contain tens or hundreds of thousands of computers working on a single application. Scalability, maintenance, and energy use are major issues here. This article forms an introduction to the subject and introduces four additional articles on the subject.

Kumar et al., "Heterogeneous Chip Multiprocessors"

The multicore chips used for desktop computers are symmetric—all the cores are identical. However, for some applications, heterogeneous CMPs are widespread, with cores for computing, video decoding, audio decoding, and so on. This paper discusses some issues related to heterogeneous CMPs.

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 too long to compute. In this paper, the authors discuss and compare 27 known algorithms for attacking this problem in different ways.

Zhuravlev et al., "Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors"

As mentioned earlier, one of the most important challenges in multiprocessor systems is shared resource contention. This survey presents different scheduling techniques for handling such contention.

13.1.9 Security

Anderson, *Security Engineering, 2nd Edition*

A wonderful book that explains very clearly how to build dependable and secure systems by one of the best-known researchers in the field. Not only is this a fascinating look at many aspects of security (including techniques, applications, and organizational issues), it is also freely available online. No excuse for not reading it.

Van der Veen et al., "Memory Errors: the Past, the Present, and the Future"

A historical view on memory errors (including buffer overflows, format string attacks, dangling pointers, and many others) that includes attacks and defenses, attacks that evade those defenses, new defenses that stop the attacks that evaded the earlier defenses, and ..., well, anyway, you get the idea. The authors show that despite their old age and the rise of other types of attack, memory errors remain an extremely important attack vector. Moreover, they argue that this situation is not likely to change any time soon.

Bratus, "What Hackers Learn That the Rest of Us Don't"

What makes hackers different? What do they care about that regular programmers do not? Do they have different attitudes toward APIs? Are corner cases important? Curious? Read it.

Bratus et al., "From Buffer Overflows to Weird Machines and Theory of Computation"

Connecting the humble buffer overflow to Alan Turing. The authors show that hackers program vulnerable programs like *weird machines* with strange-looking instruction sets. In doing so, they come full circle to Turing's seminal research on "What is computable?"

Denning, *Information Warfare and Security*

Information has become a weapon of war, both military and corporate. The participants try not only to attack the other side's information systems, but to safeguard their own, too. 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.

Ford and Allen, "How Not to Be Seen"

Viruses, spyware, rootkits, and digital rights management systems all have a great interest in hiding things. This article provides a brief introduction to stealth in its various forms.

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).

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, tattoo 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 and have lower latency. For a thorough introduction to the subject as currently practiced, this paper is the place to start.

Ludwig, "The Little Black Book of Email Viruses"

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 many of them is supplied as well. A thorough knowledge of programming the x86 in assembly language is a must, however.

Mead, "Who is Liable for Insecure Systems?"

Although most work on computer security approaches it from a technical perspective, that is not the only one. Suppose software vendors were legally liable for the damages caused by their faulty software. Chances are security would get a lot more attention from vendors than it does now? Intrigued by this idea? Read this article.

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.

Sasse, "Red-Eye Blink, Bendy Shuffle, and the Yuck Factor: A User Experience of Biometric Airport Systems"

The author discusses his experiences with the iris recognition system used at a number of large airports. Not all of them are positive.

Thibadeau, "Trusted Computing for Disk Drives and Other Peripherals"

If you thought a disk drive was just a place where bits are stored, think again. A modern disk drive has a powerful CPU, megabytes of RAM, multiple communication channels, and even its own boot ROM. In short, it is a complete computer system ripe for attack and in need of its own protection system. This paper discusses securing the disk drive.

13.1.10 Case Study 1: UNIX, Linux, and Android

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.

Fusco, *The Linux Programmer's Toolbox*

This book describes how to use Linux for the intermediate user, one who knows the basics and wants to start exploring how the many Linux programs work. It is intended for C programmers.

Maxwell, *Linux Core Kernel Commentary*

The first 400 pages of this book contain a subset of the Linux kernel code. The last 150 pages consist of comments on the code, very much in the style of John Lions' classic book. 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.

13.1.11 Case Study 2: Windows 8

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. The procedure is quite instructive.

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++.

Russinovich and Solomon, *Windows Internals, Part 1*

If you want to learn how to use Windows, there are hundreds of books out there. If you want to know how Windows 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 Operating System Design

Saltzer and Kaashoek, *Principles of Computer System Design: An Introduction*

This books looks at computer systems in general, rather than operating systems per se, but the principles they identify apply very much to operating systems also. What is interesting about this work is that it carefully identifies "the ideas that worked," such as names, file systems, read-write coherence, authenticated and confidential messages, etc. Principles that, in our opinion, all computer scientists in the world should recite every day, before going to work.

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 give 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.

ACCETTA, M., BARON, R., GOLUB, D., RASHID, R., TEVANIAN, A., and YOUNG, M.: "Mach: A New Kernel Foundation for UNIX Development," *Proc. USENIX Summer 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, K., and AGESEN, O.: "A Comparison of Software and Hardware Technqiues for X86 Virtualization," *Proc. 12th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, ACM, pp. 2–13, 2006.

AGESEN, O., MATTSON, J., RUGINA, R., and SHELDON, J.: "Software Techniques for Avoiding Hardware Virtualization Exits," *Proc. USENIX Ann. Tech. Conf.*, USENIX, 2012.

AHMAD, I.: "Gigantic Clusters: Where Are They and What Are They Doing?" *IEEE Concurrency*, vol. 8, pp. 83–85, April-June 2000.

AHN, B.-S., SOHN, S.-H., KIM, S.-Y., CHA, G.-I., BAEK, Y.-C., JUNG, S.-I., and KIM, M.-J.: "Implementation and Evaluation of EXT3NS Multimedia File System," *Proc. 12th Ann. Int'l Conf. on Multimedia*, ACM, pp. 588–595, 2004.

ALBATH, J., THAKUR, M., and MADRIA, S.: "Energy Constraint Clustering Algorithms for Wireless Sensor Networks," *J. Ad Hoc Networks*, vol. 11, pp. 2512–2525, Nov. 2013.

AMSDEN, Z., ARAI, D., HECHT, D., HOLLER, A., and SUBRAHMANYAM, P.: "VMI: An Interface for Paravirtualization," *Proc. 2006 Linux Symp.*, 2006.

ANDERSON, D.: *SATA Storage Technology: Serial ATA*, Mindshare, 2007.

ANDERSON, R.: *Security Engineering*, 2nd ed., Hoboken, NJ: John Wiley & Sons, 2008.

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.

APPUSWAMY, R., VAN MOOLENBROEK, D.C., and TANENBAUM, A.S.: "Flexible, Modular File Volume Virtualization in Loris," *Proc. 27th Symp. on Mass Storage Systems and Tech.*, IEEE, pp. 1–14, 2011.

ARNAB, A., and HUTCHISON, A.: "Piracy and Content Protection in the Broadband Age," *Proc. S. African Telecomm. Netw. and Appl. Conf*, 2006.

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.

ARPACI-DUSSEAU, R. and ARPACI-DUSSEAU, A.: *Operating Systems: Three Easy Pieces*, Madison, WI: Arpacci-Dusseau, 2013.

BAKER, F.T.: "Chief Programmer Team Management of Production Programming," *IBM Systems J.*, vol. 11, pp. 1, 1972.

BAKER, M., SHAH, M., ROSENTHAL, D.S.H., ROUSSOPOULOS, M., MANIATIS, P., GIULI, T.J., and BUNGALE, P.: "A Fresh Look at the Reliability of Long-Term Digital Storage," *Proc. First European Conf. on Computer Systems (EUROSYS)*, ACM, pp. 221–234, 2006.

BALA, K., KAASHOEK, M.F., and WEIHL, W.: "Software Prefetching and Caching for Translation Lookaside Buffers," *Proc. First Symp. on Operating Systems Design and Implementation*, USENIX, pp. 243–254, 1994.

BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., and WARFIELD, A.: "Xen and the Art of Virtualization," *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 164–177, 2003.

BARNI, M.: "Processing Encrypted Signals: A New Frontier for Multimedia Security," *Proc. Eighth Workshop on Multimedia and Security*, ACM, pp. 1–10, 2006.

BARR, K., BUNGALE, P., DEASY, S., GYURIS, V., HUNG, P., NEWELL, C., TUCH, H., and ZOPPIS, B.: "The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket?" *ACM SIGOPS Operating Systems Rev.*, vol. 44, pp. 124–135, Dec. 2010.

BARWINSKI, M., IRVINE, C., and LEVIN, T.: "Empirical Study of Drive-By-Download Spyware," *Proc. Int'l Conf. on I-Warfare and Security*, Academic Confs. Int'l, 2006.

BASILLI, V.R., and PERRICONE, B.T.: "Software Errors and Complexity: An Empirical Study," *Commun. of the ACM*, vol. 27, pp. 42–52, Jan. 1984.

BAUMANN, A., BARHAM, P., DAGAND, P., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHUPBACH, A., and SINGHANIA, A.: "The Multikernel: A New OS Architecture for Scalable Multicore Systems," *Proc. 22nd Symp. on Operating Systems Principles*, ACM, pp. 29–44, 2009.

BAYS, C.: "A Comparison of Next-Fit, First-Fit, and Best-Fit," *Commun. of the ACM*, vol. 20, pp. 191–192, March 1977.

BEHAM, M., VLAD, M., and REISER, H.: "Intrusion Detection and Honeypots in Nested Virtualization Environments," *Proc. 43rd Conf. on Dependable Systems and Networks*, IEEE, pp. 1–6, 2013.

BELAY, A., BITTAU, A., MASHTIZADEH, A., TEREI, D., MAZIERES, D., and KOZYRAKIS, C.: "Dune: Safe User-level Access to Privileged CPU Features," *Proc. Ninth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 335–348, 2010.

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 and Distributed Programming*, Upper Saddle River, NJ: Prentice Hall, 2006.

BEN-YEHUDA, M., D. DAY, M., DUBITZKY, Z., FACTOR, M., HAR'EL, N., GORDON, A., LIGUORI, A., WASSERMAN, O., and YASSOUR, B.: "The Turtles Project: Design and Implementation of Nested Virtualization," *Proc. Ninth Symp. on Operating Systems Design and Implementation*, USENIX, Art. 1–6, 2010.

BHEDA, R.A., BEU, J.G., RAILING, B.P., and CONTE, T.M.: "Extrapolation Pitfalls When Evaluating Limited Endurance Memory," *Proc. 20th Int'l Symp. on Modeling, Analysis, & Simulation of Computer and Telecomm. Systems*, IEEE, pp. 261–268, 2012.

BHEDA, R.A., POOVEY, J.A., BEU, J.G., and CONTE, T.M.: "Energy Efficient Phase Change Memory Based Main Memory for Future High Performance Systems," *Proc. Int'l Green Computing Conf.*, IEEE, pp. 1–8, 2011.

BHOEDJANG, R.A.F., RUHL, T., and BAL, H.E.: "User-Level Network Interface Protocols," *Computer*, vol. 31, pp. 53–60, Nov. 1998.

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.

BISHOP, M., and FRINCKE, D.A.: "Who Owns Your Computer?" *IEEE Security and Privacy*, vol. 4, pp. 61–63, 2006.

BLACKHAM, B, SHI, Y. and HEISER, G.: "Improving Interrupt Response Time in a Verifiable Protected Microkernel," *Proc. Seventh European Conf. on Computer Systems (EUROSYS)*, April, 2012.

BOEHM, B.: *Software Engineering Economics*, Upper Saddle River, NJ: Prentice Hall, 1981.

BOGDANOV, A., AND LEE, C.H.: "Limits of Provable Security for Homomorphic Encryption," *Proc. 33rd Int'l Cryptology Conf.*, Springer, 2013.

BORN, G: *Inside the Windows 98 Registry*, Redmond, WA: Microsoft Press, 1998.

BOTELHO, F.C., SHILANE, P., GARG, N., and HSU, W.: "Memory Efficient Sanitization of a Deduplicated Storage System," *Proc. 11th USENIX Conf. on File and Storage Tech.*, USENIX, pp. 81–94, 2013.

BOTERO, J. F., and HESSELBACH, X.: "Greener Networking in a Network Virtualization Environment," *Computer Networks*, vol. 57, pp. 2021–2039, June 2013.

BOULGOURIS, N.V., PLATANIOTIS, K.N., and MICHELI-TZANAKOU, E.: *Biometics: Theory Methods, and Applications*, Hoboken, NJ: John Wiley & Sons, 2010.

BOVET, D.P., and CESATI, M.: *Understanding the Linux Kernel*, Sebastopol, CA: O'Reilly & Associates, 2005.

BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., and ZHANG, Z.: "Corey: an Operating System for Many Cores," *Proc. Eighth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 43–57, 2008.

BOYD-WICKIZER, S., CLEMENTS A.T., MAO, Y., PESTEREV, A., KAASHOEK, F.M., MORRIS, R., and ZELDOVICH, N.: "An Analysis of Linux Scalability to Many Cores," *Proc. Ninth Symp. on Operating Systems Design and Implementation*, USENIX, 2010.

BRATUS, S.: "What Hackers Learn That the Rest of Us Don't: Notes on Hacker Curriculum," *IEEE Security and Privacy*, vol. 5, pp. 72–75, July/Aug. 2007.

BRATUS, S., LOCASTO, M.E., PATTERSON, M., SASSAMAN, L., SHUBINA, A.: "From Buffer Overflows to Weird Machines and Theory of Computation," *;Login:*, USENIX, pp. 11–21, December 2011.

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.: "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, Boston: Addison-Wesley, 1995.

BRUSCHI, D., MARTIGNONI, L., and MONGA, M.: "Code Normalization for Self-Mutating Malware," *IEEE Security and Privacy*, vol. 5, pp. 46–54, March/April 2007.

BUGNION, E., DEVINE, S., GOVIL, K., and ROSENBLUM, M.: "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *ACM Trans. on Computer Systems*, vol. 15, pp. 412–447, Nov. 1997.

BUGNION, E., DEVINE, S., ROSENBLUM, M., SUGERMAN, J., and WANG, E.: "Bringing Virtualization to the x86 Architecture with the Original VMware Workstation," *ACM Trans. on Computer Systems*, vol. 30, number 4, pp.12:1–12:51, Nov. 2012.

BULPIN, J.R., and PRATT, I.A.: "Hyperthreading-Aware Process Scheduling Heuristics," *Proc. USENIX Ann. Tech. Conf.*, USENIX, pp. 399–403, 2005.

CAI, J., and STRAZDINS, P.E.: "An Accurate Prefetch Technique for Dynamic Paging Behaviour for Software Distributed Shared Memory," *Proc. 41st Int'l Conf. on Parallel Processing*, IEEE., pp. 209–218, 2012.

CAI, Y., and CHAN, W.K.: "MagicFuzzer: Scalable Deadlock Detection for Large-scale Applications," *Proc. 2012 Int'l Conf. on Software Engineering*, IEEE, pp. 606–616, 2012.

CAMPISI, P.: *Security and Privacy in Biometrics*, New York: Springer, 2013.

CARPENTER, M., LISTON, T., and SKOUDIS, E.: "Hiding Virtualization from Attackers and Malware," *IEEE Security and Privacy*, vol. 5, pp. 62–65, May/June 2007.

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.

CERF, C., and NAVASKY, V.: *The Experts Speak*, New York: Random House, 1984.

CHEN, M.-S., YANG, B.-Y., and CHENG, C.-M.: "RAIDq: A Software-Friendly, Multiple-Parity RAID," *Proc. Fifth Workshop on Hot Topics in File and Storage Systems*, USENIX, 2013.

CHEN, Z., XIAO, N., and LIU, F.: "SAC: Rethinking the Cache Replacement Policy for SSD-Based Storage Systems," *Proc. Fifth Int'l Systems and Storage Conf.*, ACM, Art. 13, 2012.

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.

CHIDAMBARAM, V., PILLAI, T.S., ARPACI-DUSSEAU, A.C., and ARPACI-DUSSEAU, R.H.: "Optimistic Crash Consistency," *Proc. 24th Symp. on Operating System Principles*, ACM, pp. 228–243, 2013.

CHOI, S., and JUNG, S.: "A Locality-Aware Home Migration for Software Distributed Shared Memory," *Proc. 2013 Conf. on Research in Adaptive and Convergent Systems*, ACM, pp. 79–81, 2013.

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.

CLEMENTS, A.T, KAASHOEK, M.F., ZELDOVICH, N., MORRIS, R.T., and KOHLER, E.: "The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors," *Proc. 24th Symp. on Operating Systems Principles*, ACM, pp. 1–17, 2013.

COFFMAN, E.G., ELPHICK, M.J., and SHOSHANI, A.: "System Deadlocks," *Computing Surveys*, vol. 3, pp. 67–78, June 1971.

COLP, P., NANAVATI, M., ZHU, J., AIELLO, W., COKER, G., DEEGAN, T., LOSCOCCO, P., and WARFIELD, A.: "Breaking Up Is Hard to Do: Security and Functionality in a Commodity Hypervisor," *Proc. 23rd Symp. of Operating Systems Principles*, ACM, pp. 189–202, 2011.

COOKE, D., URBAN, J., and HAMILTON, S.: "UNIX and Beyond: An Interview with Ken Thompson," *Computer*, vol. 32, pp. 58–64, May 1999.

COOPERSTEIN, J.: *Writing Linux Device Drivers: A Guide with Exercises*, Seattle: CreateSpace, 2009.

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., and VYSSOTSKY, V.A.: "Introduction and Overview of the MULTICS System," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 185–196, 1965.

CORBET, J., RUBINI, A., and KROAH-HARTMAN, G.: *Linux Device Drivers*, Sebastopol, CA: O'Reilly & Associates, 2009.

CORNWELL, M.: "Anatomy of a Solid-State Drive," *ACM Queue 10* 10, pp. 30–37, 2012.

CORREIA, M., GOMEZ FERRO, D., JUNQUEIRA, F.P., and SERAFINI, M.: "Practical Hardening of Crash-Tolerant Systems," *Proc. USENIX Ann. Tech. Conf.*, USENIX, 2012.

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.

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.

DABEK, F., KAASHOEK, M.F., KARGET, D., MORRIS, R., and STOICA, I.: "Wide-Area Cooperative Storage with CFS," *Proc. 18th Symp. on Operating Systems Principles*, ACM, pp. 202–215, 2001.

DAI, Y., QI, Y., REN, J., SHI, Y., WANG, X., and YU, X.: "A Lightweight VMM on Many Core for High Performance Computing," *Proc. Ninth Int'l Conf. on Virtual Execution Environments*, ACM, pp. 111–120, 2013.

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.

DASHTI, M., FEDOROVA, A., FUNSTON, J., GAUD, F., LACHAIZE, R., LEPERS, B., QUEMA, V., and ROTH, M.: "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," *Proc. 18th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, ACM, pp. 381–394, 2013.

DAUGMAN, J.: "How Iris Recognition Works," *IEEE Trans. on Circuits and Systems for Video Tech.*, vol. 14, pp. 21–30, Jan. 2004.

DAWSON-HAGGERTY, S., KRIOUKOV, A., TANEJA, J., KARANDIKAR, S., FIERRO, G., and CULLER, D: "BOSS: Building Operating System Services," *Proc. 10th Symp. on Networked Systems Design and Implementation*, USENIX, pp. 443–457, 2013.

DAYAN, N., SVENDSEN, M.K., BJORING, M., BONNET, P., and BOUGANIM, L.: "Eagle-Tree: Exploring the Design Space of SSD-based Algorithms," *Proc. VLDB Endowment*, vol. 6, pp. 1290–1293, Aug. 2013.

DE BRUIJN, W., BOS, H., and BAL, H.: "Application-Tailored I/O with Streamline," *ACM Trans. on Computer Syst.*, vol. 29, number 2, pp.1–33, May 2011.

DE BRUIJN, W., and BOS, H.: "Beltway Buffers: Avoiding the OS Traffic Jam," *Proc. 27th Int'l Conf. on Computer Commun.*, April 2008.

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, D.: *Information Warfare and Security*, Boston: Addison-Wesley, 1999.

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.

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.

DUNN, A., LEE, M.Z., JANA, S., KIM, S., SILBERSTEIN, M., XU, Y., SHMATIKOV, V., and WITCHEL, E.: "Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels," *Proc. 10th Symp. on Operating Systems Design and Implementation*, USENIX, pp. 61–75, 2012.

DUTTA, K., SINGH, V.K., and VANDERMEER, D.: "Estimating Operating System Process Energy Consumption in Real Time," *Proc. Eighth Int'l Conf. on Design Science at the Intersection of Physical and Virtual Design*, Springer-Verlag, pp. 400–404, 2013.

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.

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.

EL FERKOUSS, O., SNAIKI, I., MOUNAOUAR, O., DAHMOUNI, H., BEN ALI, R., LEMIEUX, Y., and OMAR, C.: "A 100Gig Network Processor Platform for Openflow," *Proc. Seventh Int'l Conf. on Network Services and Management*, IFIP, pp. 286–289, 2011.

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.

ELNABLY., A., and WANG, H.: "Efficient QoS for Multi-Tiered Storage Systems," *Proc. Fourth USENIX Workshop on Hot Topics in Storage and File Systems*, USENIX, 2012.

ELPHINSTONE, K., KLEIN, G., DERRIN, P., ROSCOE, T., and HEISER, G.: "Towards a Practical, Verified, Kernel," *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 117–122, 2007.

ENGLER, D.R., CHELF, B., CHOU, A., and 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., 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.

ERL, T., PUTTINI, R., and MAHMOOD, Z.: "Cloud Computing: Concepts, Technology & Architecture," Upper Saddle River, NJ: Prentice Hall, 2013.

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.

FANDRICH, M., AIKEN, M., HAWBLITZEL, C., HODSON, O., HUNT, G., LARUS, J.R., and LEVI, S.: "Language Support for Fast and Reliable Message-Based Communication in Singularity OS," *Proc. First European Conf. on Computer Systems (EUROSYS)*, ACM, pp. 177–190, 2006.

FEELEY, M.J., MORGAN, W.E., PIGHIN, F.H., KARLIN, A.R., LEVY, H.M., and THEKKATH, C.A.: "Implementing Global Memory Management in a Workstation Cluster," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 201–212, 1995.

FELTEN, E.W., and HALDERMAN, J.A.: "Digital Rights Management, Spyware, and Security," *IEEE Security and Privacy*, vol. 4, pp. 18–23, Jan./Feb. 2006.

FETZER, C., and KNAUTH, T.: "Energy-Aware Scheduling for Infrastructure Clouds," *Proc. Fourth Int'l Conf. on Cloud Computing Tech. and Science*, IEEE, pp. 58–65, 2012.

FEUSTAL, E.A.: "The Rice Research Computer—A Tagged Architecture," *Proc. AFIPS Conf.*, AFIPS, 1972.

FLINN, J., and SATYANARAYANAN, M.: "Managing Battery Lifetime with Energy-Aware Adaptation," *ACM Trans. on Computer Systems*, vol. 22, pp. 137–179, May 2004.

FLORENCIO, D., and HERLEY, C.: "A Large-Scale Study of Web Password Habits," *Proc. 16th Int'l Conf. on the World Wide Web*, ACM, pp. 657–666, 2007.

FORD, R., and ALLEN, W.H.: "How Not To Be Seen," *IEEE Security and Privacy*, vol. 5, pp. 67–69, Jan./Feb. 2007.

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.

FRYER, D., SUN, K., MAHMOOD, R., CHENG, T., BENJAMIN, S., GOEL, A., and DEMKE BROWN, A.: "ReCon: Verifying File System Consistency at Runtime," *Proc. 10th USENIX Conf. on File and Storage Tech.*, USENIX, pp. 73–86, 2012.

FUKSIS, R., GREITANS, M., and PUDZS, M.: "Processing of Palm Print and Blood Vessel Images for Multimodal Biometrics," *Proc. COST1011 European Conf. on Biometrics and ID Mgt.*, Springer-Verlag, pp. 238–249, 2011.

FURBER, S.B., LESTER, D.R., PLANA, L.A., GARSIDE, J.D., PAINKRAS, E., TEMPLE, S., and BROWN, A.D.: "Overview of the SpiNNaker System Architecture," *Trans. on Computers*, vol. 62, pp. 2454–2467, Dec. 2013.

FUSCO, J.: *The Linux Programmer's Toolbox*, Upper Saddle River, NJ: Prentice Hall, 2007.

GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., and BONEH, D.: "Terra: A Virtual Machine-Based Platform for Trusted Computing," *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 193–206, 2003.

GAROFALAKIS, J., and STERGIOU, E.: "An Analytical Model for the Performance Evaluation of Multistage Interconnection Networks with Two Class Priorities," *Future Generation Computer Systems*, vol. 29, pp. 114–129, Jan. 2013.

GEER, D.: "For Programmers, Multicore Chips Mean Multiple Challenges," *Computer*, vol. 40, pp. 17–19, Sept. 2007.

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.

GHOSHAL, D., and PLALE, B: "Provenance from Log Files: a BigData Problem," *Proc. Joint EDBT/ICDT Workshops*, ACM, pp. 290–297, 2013.

GIFFIN, D, LEVY, A., STEFAN, D., TEREI, D., MAZIERES, D.: "Hails: Protecting Data Privacy in Untrusted Web Applications," *Proc. 10th Symp. on Operating Systems Design and Implementation*, USENIX, 2012.

GIUFFRIDA, C., KUIJSTEN, A., and TANENBAUM, A.S.: "Enhanced Operating System Security through Efficient and Fine-Grained Address Space Randomization," *Proc. 21st USENIX Security Symp.*, USENIX, 2012.

GIUFFRIDA, C., KUIJSTEN, A., and TANENBAUM, A.S.: "Safe and Automatic Live Update for Operating Systems," *Proc. 18th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, ACM, pp. 279–292, 2013.

GOLDBERG, R.P:: *Architectural Principles for Virtual Computer Systems*, Ph.D. thesis, Harvard University, Cambridge, MA, 1972.

GOLLMAN, D.: *Computer Security*, West Sussex, UK: John Wiley & Sons, 2011.

GONG, L.: *Inside Java 2 Platform Security*, Boston: Addison-Wesley, 1999.

GONZALEZ-FEREZ, P., PIERNAS, J., and CORTES, T.: "DADS: Dynamic and Automatic Disk Scheduling," *Proc. 27th Symp. on Appl. Computing*, ACM, pp. 1759–1764, 2012.

GORDON, M.S., JAMSHIDI, D.A., MAHLKE, S., and MAO, Z.M.: "COMET: Code Offload by Migrating Execution Transparently," *Proc. 10th Symp. on Operating Systems Design and Implementation*, USENIX, 2012.

GRAHAM, R.: "Use of High-Level Languages for System Programming," Project MAC Report TM-13, M.I.T., Sept. 1970.

GROPP, W., LUSK, E., and SKJELLUM, A.: *Using MPI: Portable Parallel Programming with the Message Passing Interface*, Cambridge, MA: M.I.T. Press, 1994.

GUPTA, L.: "QoS in Interconnection of Next Generation Networks," *Proc. Fifth Int'l Conf. on Computational Intelligence and Commun. Networks*, IEEE, pp. 91–96, 2013.

HAERTIG, 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.

HAFNER, K., and MARKOFF, J.: *Cyberpunk*, New York: Simon and Schuster, 1991.

HAITJEMA, M.A.: *Delivering Consistent Network Performance in Multi-Tenant Data Centers*, Ph.D. thesis, Washington Univ., 2013.

HALDERMAN, J.A., and FELTEN, E.W.: "Lessons from the Sony CD DRM Episode," *Proc. 15th USENIX Security Symp.*, USENIX, pp. 77–92, 2006.

HAN, S., MARSHALL, S., CHUN, B.-G., and RATNASAMY, S.: "MegaPipe: A New Programming Interface for Scalable Network I/O," *Proc. USENIX Ann. Tech. Conf.*, USENIX, pp. 135–148, 2012.

HAND, S.M., WARFIELD, A., FRASER, K., KOTTSOVINOS, E., and MAGENHEIMER, D.: "Are Virtual Machine Monitors Microkernels Done Right?," *Proc. 10th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 1–6, 2005.

HARNIK, D., KAT, R., MARGALIT, O., SOTNIKOV, D., and TRAEGER, A.: "To Zip or Not to Zip: Effective Resource Usage for Real-Time Compression," *Proc. 11th USENIX Conf. on File and Storage Tech.*, USENIX, pp. 229–241, 2013.

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*, Boston: Addison-Wesley, 1997.

HARTER, T., DRAGGA, C., VAUGHN, M., ARPACI-DUSSEAU, A.C., and ARPACI-DUSSEAU, R.H.: "A File Is Not a File: Understanding the I/O Behavior of Apple Desktop Applications," *ACM Trans. on Computer Systems*, vol. 30, Art. 10, pp. 71–83, Aug. 2012.

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 J.*, vol. 7, pp. 74–84, 1968.

HEISER, G., UHLIG, V., and LEVASSEUR, J.: "Are Virtual Machine Monitors Microkernels Done Right?" *ACM SIGOPS Operating Systems Rev.*, vol. 40, pp. 95–99, 2006.

HEMKUMAR, D., and VINAYKUMAR, K.: "Aggregate TCP Congestion Management for Internet QoS," *Proc. 2012 Int'l Conf. on Computing Sciences*, IEEE, pp. 375–378, 2012.

HERDER, J.N., BOS, H., GRAS, B., HOMBURG, P., and TANENBAUM, A.S.: "Construction of a Highly Dependable Operating System," *Proc. Sixth European Dependable Computing Conf.*, pp. 3–12, 2006.

HERDER, J.N., MOOLENBROEK, D. VAN, APPUSWAMY, R., WU, B., GRAS, B., and TANENBAUM, A.S.: "Dealing with Driver Failures in the Storage Stack ," *Proc. Fourth Latin American Symp. on Dependable Computing*, pp. 119–126, 2009.

HEWAGE, K., and VOIGT, T.: "Towards TCP Communication with the Low Power Wireless Bus," *Proc. 11th Conf. on Embedded Networked Sensor Systems*, ACM, Art. 53, 2013.

HILBRICH, T. DE SUPINSKI, R., NAGEL, W., PROTZE, J., BAIER,. C., and MULLER, M.: "Distributed Wait State Tracking for Runtime MPI Deadlock Detection," *Proc. 2013 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, ACM, New York, NY, USA, 2013.

HILDEBRAND, D.: "An Architectural Overview of QNX," *Proc. Workshop on Microkernels and Other Kernel Arch.*, ACM, pp. 113–136, 1992.

HIPSON, P.: *Mastering Windows XP Registry*, New York: Sybex, 2002.

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.

HOCKING, M: "Feature: Thin Client Security in the Cloud," *J. Network Security*, vol. 2011, pp. 17–19, June 2011.

HOHMUTH, M., PETER, M., HAERTIG, H., and SHAPIRO, J.: "Reducing TCB Size by Using Untrusted Components: Small Kernels Versus Virtual-Machine Monitors," *Proc. 11th ACM SIGOPS European Workshop*, ACM, Art. 22, 2004.

HOLMBACKA, S., AGREN, D., LAFOND, S., and LILIUS, J.: "QoS Manager for Energy Efficient Many-Core Operating Systems," *Proc. 21st Euromicro Int'l Conf. on Parallel, Distributed, and Network-based Processing*, IEEE, pp. 318–322, 2013.

HOLT, R.C.: "Some Deadlock Properties of Computer Systems," *Computing Surveys*, vol. 4, pp. 179–196, Sept. 1972.

HOQUE, M.A., SIEKKINEN, and NURMINEN, J.K.: "TCP Receive Buffer Aware Wireless Multimedia Streaming: An Energy Efficient Approach," *Proc. 23rd Workshop on Network and Operating System Support for Audio and Video*, ACM, pp. 13–18, 2013.

HOWARD, M., and LEBLANK, D.: *Writing Secure Code*, Redmond, WA: Microsoft Press, 2009.

HRUBY, T., VOGT, D., BOS, H., and TANENBAUM, A.S.: "Keep Net Working—On a Dependable and Fast Networking Stack," *Proc. 42nd Conf. on Dependable Systems and Networks*, IEEE, pp. 1–12, 2012.

HUND, R. WILLEMS, C. AND HOLZ, T.: "Practical Timing Side Channel Attacks against Kernel Space ASLR," *Proc. IEEE Symp. on Security and Privacy*, IEEE, pp. 191–205, 2013.

HRUBY, T., D., BOS, H., and TANENBAUM, A.S.: "When Slower Is Faster: On Heterogeneous Multicores for Reliable Systems," *Proc. USENIX Ann. Tech. Conf.*, USENIX, 2013.

HUA, J., LI, M., SAKURAI, K., and REN, Y.: "Efficient Intrusion Detection Based on Static Analysis and Stack Walks," *Proc. Fourth Int'l Workshop on Security*, Springer-Verlag, pp. 158–173, 2009.

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, 1990.

INTEL: "PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology," *Intel White Paper*, 2011.

ION, F.: "From Touch Displays to the Surface: A Brief History of Touchscreen Technology," *ArsTechnica, History of Tech*, April, 2013

ISLOOR, S.S., and MARSLAND, T.A.: "The Deadlock Problem: An Overview," *Computer*, vol. 13, pp. 58–78, Sept. 1980.

IVENS, K.: *Optimizing the Windows Registry*, Hoboken, NJ: John Wiley & Sons, 1998.

JANTZ, M.R., STRICKLAND, C., KUMAR, K., DIMITROV, M., and DOSHI, K.A.: "A Framework for Application Guidance in Virtual Memory Systems," *Proc. Ninth Int'l Conf. on Virtual Execution Environments*, ACM, pp. 155–166, 2013.

JEONG, J., KIM, H., HWANG, J., LEE, J., and MAENG, S.: "Rigorous Rental Memory Management for Embedded Systems," *ACM Trans. on Embedded Computing Systems*, vol. 12, Art. 43, pp. 1–21, March 2013.

JIANG, X., and XU, D.: "Profiling Self-Propagating Worms via Behavioral Footprinting," *Proc. Fourth ACM Workshop in Recurring Malcode*, ACM, pp. 17–24, 2006.

JIN, H., LING, X., IBRAHIM, S., CAO, W., WU, S., and ANTONIU, G.: "Flubber: Two-Level Disk Scheduling in Virtualized Environment," *Future Generation Computer Systems*, vol. 29, pp. 2222–2238, Oct. 2013.

JOHNSON, E.A: "Touch Display—A Novel Input/Output Device for Computers," *Electronics Letters*, vol. 1, no. 8, pp. 219–220, 1965.

JOHNSON, N.F., and JAJODIA, S.: "Exploring Steganography: Seeing the Unseen," *Computer*, vol. 31, pp. 26–34, Feb. 1998.

JOO, Y.: "F2FS: A New File System Designed for Flash Storage in Mobile Devices," *Embedded Linux Europe*, Barcelona, Spain, November 2012.

JULA, H., TOZUN, P., and CANDEA, G.: "Communix: A Framework for Collaborative Deadlock Immunity," *Proc. IEEE/IFIP 41st Int. Conf. on Dependable Systems and Networks*, IEEE, pp. 181–188, 2011.

KABRI, K., and SERET, D.: "An Evaluation of the Cost and Energy Consumption of Security Protocols in WSNs," *Proc. Third Int'l Conf. on Sensor Tech. and Applications*, IEEE, pp. 49–54, 2009.

KAMAN, S., SWETHA, K., AKRAM, S., and VARAPRASAS, G.: "Remote User Authentication Using a Voice Authentication System," *Inf. Security J.*, vol. 22, pp. 117–125, Issue 3, 2013.

KAMINSKY, D.: "Explorations in Namespace: White-Hat Hacking across the Domain Name System," *Commun. of the ACM*, vol. 49, pp. 62–69, June 2006.

KAMINSKY, M., SAVVIDES, G., MAZIERES, D., and KAASHOEK, M.F.: "Decentralized User Authentication in a Global File System," *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 60–73, 2003.

KANETKAR, Y.P.: *Writing Windows Device Drivers Course Notes*, New Delhi: BPB Publications, 2008.

KANT, K., and MOHAPATRA, P.: "Internet Data Centers," *IEEE Computer* vol. 37, pp. 35–37, Nov. 2004.

KAPRITSOS, M., WANG, Y., QUEMA, V., CLEMENT, A., ALVISI, L., and DAHLIN, M.: "All about Eve: Execute-Verify Replication for Multi-Core Servers," *Proc. 10th Symp. on Operating Systems Design and Implementation*, USENIX, pp. 237–250, 2012.

KASIKCI, B., ZAMFIR, C. and CANDEA, G.: "Data Races vs. Data Race Bugs: Telling the Difference with Portend," *Proc. 17th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, ACM, pp. 185–198, 2012.

KATO, S., ISHIKAWA, Y., and RAJKUMAR, R.: "Memory Management for Interactive Real-Time Applications," *Real-Time Systems*, vol. 47, pp. 498–517, May 2011.

KAUFMAN, C., PERLMAN, R., and SPECINER, M.: *Network Security*, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 2002.

KELEHER, P., COX, A., DWARKADAS, S., and ZWAENEPOEL, W.: "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," *Proc. USENIX Winter Conf.*, USENIX, pp. 115–132, 1994.

KERNIGHAN, B.W., and PIKE, R.: *The UNIX Programming Environment*, Upper Saddle River, NJ: Prentice Hall, 1984.

KIM, J., LEE, J., CHOI, J., LEE, D., and NOH, S.H.: "Improving SSD Reliability with RAID via Elastic Striping and Anywhere Parity," *Proc. 43rd Int'l Conf. on Dependable Systems and Networks*, IEEE, pp. 1–12, 2013.

KIRSCH, C.M., SANVIDO, M.A.A., and HENZINGER, T.A.: "A Programmable Microkernel for Real-Time Systems," *Proc. First Int'l Conf. on Virtual Execution Environments*, ACM, pp. 35–45, 2005.

KLEIMAN, S.R.: "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Proc. USENIX Summer Conf.*, USENIX, pp. 238–247, 1986.

KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., and WINWOOD, S.: "seL4: Formal Verification of an OS Kernel," *Proc. 22nd Symp. on Operating Systems Primciples*, ACM, pp. 207–220, 2009.

KNUTH, D.E.: *The Art of Computer Programming*, Vol. Boston: Addison-Wesley, 1997.

KOLLER, R., MARMOL, L., RANGASWAMI, R, SUNDARARAMAN, S., TALAGALA, N., and ZHAO, M.: "Write Policies for Host-side Flash Caches," *Proc. 11th USENIX Conf. on File and Storage Tech.*, USENIX, pp. 45–58, 2013.

KOUFATY, D., REDDY, D., and HAHN, S.: "Bias Scheduling in Heterogeneous Multi-Core Architectures," *Proc. Fifth European Conf. on Computer Systems (EUROSYS)*, ACM, pp. 125–138, 2010.

KRATZER, C., DITTMANN, J., LANG, A., and KUHNE, T.: "WLAN Steganography: A First Practical Review," *Proc. Eighth Workshop on Multimedia and Security*, ACM, pp. 17–22, 2006.

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.

KRISH, K.R., WANG, G., BHATTACHARJEE, P., BUTT, A.R., and SNIADY, C.: "On Reducing Energy Management Delays in Disks," *J. Parallel and Distributed Computing*, vol. 73, pp. 823–835, June 2013.

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, R., TULLSEN, D.M., JOUPPI, N.P., and RANGANATHAN, P.: "Heterogeneous Chip Multiprocessors," *Computer*, vol. 38, pp. 32–38, Nov. 2005.

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.

LACHAIZE, R., LEPERS, B., and QUEMA, V.: "MemProf: A Memory Profiler for NUMA Multicore Systems," *Proc. USENIX Ann. Tech. Conf.*, USENIX, 2012.

LAI, W.K., and TANG, C.-L.: "QoS-aware Downlink Packet Scheduling for LTE Networks," *Computer Networks*, vol. 57, pp. 1689–1698, May 2013.

LAMPSON, B.W.: "A Note on the Confinement Problem," *Commun. of the ACM*, vol. 10, pp. 613–615, Oct. 1973.

LAMPORT, L.: "Password Authentication with Insecure Communication," *Commun. of the ACM*, vol. 24, pp. 770–772, Nov. 1981.

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.

LANKES, S., REBLE, P., SINNEN, O., and CLAUSS, C.: "Revisiting Shared Virtual Memory Systems for Non-Coherent Memory-Coupled Cores," *Proc. 2012 Int'l Workshop on Programming Models for Applications for Multicores and Manycores*, ACM, pp. 45–54, 2012.

LEE, Y., JUNG, T., and SHIN, I.L "Demand-Based Flash Translation Layer Considering Spatial Locality," *Proc. 28th Annual Symp. on Applied Computing*, ACM, pp. 1550–1551, 2013.

LEVENTHAL, A.D.: "A File System All Its Own," *Commun. of the ACM*, vol. 56, pp. 64–67, May 2013.

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.

LEVINE, G.N.: "Defining Deadlock," *ACM SIGOPS Operating Systems Rev.*, vol. 37, pp. 54–64, Jan. 2003.

LEVINE, J.G., GRIZZARD, J.B., and OWEN, H.L.: "Detecting and Categorizing Kernel-Level Rootkits to Aid Future Detection," *IEEE Security and Privacy*, vol. 4, pp. 24–32, Jan./Feb. 2006.

LI, D., JIN, H., LIAO, X., ZHANG, Y., and ZHOU, B.: "Improving Disk I/O Performance in a Virtualized System," *J. Computer and Syst. Sci.*, vol. 79, pp. 187–200, March 2013a.

LI, D., LIAO, X., JIN, H., ZHOU, B., and ZHANG, Q.: "A New Disk I/O Model of Virtualized Cloud Environment," *IEEE Trans. on Parallel and Distributed Systems*, vol. 24, pp. 1129–1138, June 2013b.

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. USENIX Winter Conf.*, USENIX, pp. 279–291, 1994.

LI, Y., SHOTRE, S., OHARA, Y., KROEGER, T.M., MILLER, E.L., and LONG, D.D.E.: "Horus: Fine-Grained Encryption-Based Security for Large-Scale Storage," *Proc. 11th USENIX Conf. on File and Storage Tech.*, USENIX, pp. 147–160, 2013c.

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.

LING, X., JIN, H., IBRAHIM, S., CAO, W., and WU, S.: "Efficient Disk I/O Scheduling with QoS Guarantee for Xen-based Hosting Platforms," *Proc. 12th Int'l Symp. on Cluster, Cloud, and Grid Computing*, IEEE/ACM, pp. 81–89, 2012.

LIONS, J.: *Lions' Commentary on Unix 6th Edition, with Source Code*, San Jose, CA: Peer-to-Peer Communications, 1996.

LIU, T., CURTSINGER, C., and BERGER, E.D.: "Dthreads: Efficient Deterministic Multithreading," *Proc. 23rd Symp. of Operating Systems Principles*, ACM, pp. 327–336, 2011.

LIU, Y., MUPPALA, J.K., VEERARAGHAVAN, M., LIN, D., and HAMDI, M.: *Data Center Networks: Topologies, Architectures and Fault-Tolerance Characteristics*, Springer, 2013.

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., PARNO, B., MICKENS, J., RAYKOVA, M., and SCHIFFMAN, J.: "Shroud: Ensuring Private Access to Large-Scale Data in the Data Center," *Proc. 11th USENIX Conf. on File and Storage Tech.*, USENIX, pp. 199–213, 2013.

LOPEZ-ORTIZ, A., SALINGER, A.: "Paging for Multi-Core Shared Caches," *Proc. Innovations in Theoretical Computer Science*, ACM, pp. 113–127, 2012.

LORCH, J.R., and SMITH, A.J.: "Apple Macintosh's Energy Consumption," *IEEE Micro*, vol. 18, pp. 54–63, Nov./Dec. 1998.

LOVE, R.: *Linux System Programming: Talking Directly to the Kernel and C Library*, Sebastopol, CA: O'Reilly & Associates, 2013.

LU, L., ARPACI-DUSSEAU, A.C., and ARPACI-DUSSEAU, R.H.: "Fault Isolation and Quick Recovery in Isolation File Systems," *Proc. Fifth USENIX Workshop on Hot Topics in Storage and File Systems*, USENIX, 2013.

LUDWIG, M.A.: *The Little Black Book of Email Viruses*, Show Low, AZ: American Eagle Publications, 2002.

LUO, T., MA, S., LEE, R., ZHANG, X., LIU, D., and ZHOU, L.: "S-CAVE: Effective SSD Caching to Improve Virtual Machine Storage Performance," *Proc. 22nd Int'l Conf. on Parallel Arch. and Compilation Tech.*, IEEE, pp. 103–112, 2013.

MA, A., DRAGGA, C., ARPACI-DUSSEAU, A.C., and ARPACI-DUSSEAU, R.H.: "ffsck: The Fast File System Checker," *Proc. 11th USENIX Conf. on File and Storage Tech.*, USENIX, 2013.

MAO, W.: "The Role and Effectiveness of Cryptography in Network Virtualization: A Position Paper," *Proc. Eighth ACM Asian SIGACT Symp. on Information, Computer, and Commun. Security*, ACM, pp. 179–182, 2013.

MARINO, D., HAMMER, C., DOLBY, J., VAZIRI, M., TIP, F., and VITEK, J.: "Detecting Deadlock in Programs with Data-Centric Synchronization," *Proc. Int'l Conf. on Software Engineering*, IEEE, pp. 322–331, 2013.

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.

MASHTIZADEH, A.J., BITTAY, A., HUANG, Y.F., and MAZIERES, D.: "Replication, History, and Grafting in the Ori File System," *Proc. 24th Symp. on Operating System Principles*, ACM, pp. 151–166, 2013.

MATTHUR, A., and MUNDUR, P.: "Dynamic Load Balancing Across Mirrored Multimedia Servers," *Proc. 2003 Int'l Conf. on Multimedia*, IEEE, pp. 53–56, 2003.

MAXWELL, S.: *Linux Core Kernel Commentary*, Scottsdale, AZ: Coriolis Group Books, 2001.

MAZUREK, M.L., THERESKA, E., GUNAWARDENA, D., HARPER, R., and SCOTT, J.: "ZZFS: A Hybrid Device and Cloud File System for Spontaneous Users," *Proc. 10th USENIX Conf. on File and Storage Tech.*, USENIX, pp. 195–208, 2012.

McKUSICK, M.K., BOSTIC, K., KARELS, M.J., QUARTERMAN, J.S.: *The Design and Implementation of the 4.4BSD Operating System*, Boston: Addison-Wesley, 1996.

McKUSICK, M.K., and NEVILLE-NEIL, G.V.: *The Design and Implementation of the FreeBSD Operating System*, Boston: Addison-Wesley, 2004.

McKUSICK, M.K.: "Disks from the Perspective of a File System," *Commun. of the ACM*, vol. 55, pp. 53–55, Nov. 2012.

MEAD, N.R.: "Who Is Liable for Insecure Systems?" *Computer*, vol. 37, pp. 27–34, July 2004.

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.

MIKHAYLOV, K., and TERVONEN, J.: "Energy Consumption of the Mobile Wireless Sensor Network's Node with Controlled Mobility," *Proc. 27th Int'l Conf. on Advanced Networking and Applications Workshops*, IEEE, pp. 1582–1587, 2013.

MILOJICIC, D.: "Security and Privacy," *IEEE Concurrency*, vol. 8, pp. 70–79, April–June 2000.

MOODY, G.: *Rebel Code*, Cambridge. MA: Perseus Publishing, 2001.

MOON, S., and REDDY, A.L.N.: "Don't Let RAID Raid the Lifetime of Your SSD Array," *Proc. Fifth USENIX Workshop on Hot Topics in Storage and File Systems*, USENIX, 2013.

MORRIS, R., and THOMPSON, K.: "Password Security: A Case History," *Commun. of the ACM*, vol. 22, pp. 594–597, Nov. 1979.

MORUZ, G., and NEGOESCU, A.: "Outperforming LRU Via Competitive Analysis on Parametrized Inputs for Paging," *Proc. 23rd ACM-SIAM Symp. on Discrete Algorithms*, SIAM, pp. 1669–1680.

MOSHCHUK, A., BRAGIN, T., GRIBBLE, S.D., and LEVY, H.M.: "A Crawler-Based Study of Spyware on the Web," *Proc. Network and Distributed System Security Symp.*, Internet Society, pp. 1–17, 2006.

MULLENDER, S.J., and TANENBAUM, A.S.: "Immediate Files," *Software Practice and Experience*, vol. 14, pp. 365–368, 1984.

NACHENBERG, C.: "Computer Virus-Antivirus Coevolution," *Commun. of the ACM*, vol. 40, pp. 46–51, Jan. 1997.

NARAYANAN, D., N. THERESKA, E., DONNELLY, A., ELNIKETY, S. and ROWSTRON, A.: "Migrating Server Storage to SSDs: Analysis of Tradeoffs," *Proc. Fourth European Conf. on Computer Systems (EUROSYS)*, ACM, 2009.

NELSON, M., LIM, B.-H., and HUTCHINS, G.: "Fast Transparent Migration for Virtual Machines," *Proc. USENIX Ann. Tech. Conf.*, USENIX, pp. 391–394, 2005.

NEMETH, E., SNYDER, G., HEIN, T.R., and WHALEY, B.: *UNIX and Linux System Administration Handbook*, 4th ed., Upper Saddle River, NJ: Prentice Hall, 2013.

NEWTON, G.: "Deadlock Prevention, Detection, and Resolution: An Annotated Bibliography," *ACM SIGOPS Operating Systems Rev.*, vol. 13, pp. 33–44, April 1979.

NIEH, J., and LAM, M.S.: "A SMART Scheduler for Multimedia Applications," *ACM Trans. on Computer Systems*, vol. 21, pp. 117–163, May 2003.

NIGHTINGALE, E.B., ELSON, J., FAN, J., HOGMANN, O., HOWELL, J., and SUZUE, Y.: "Flat Datacenter Storage," *Proc. 10th Symp. on Operating Systems Design and Implementation*, USENIX, pp. 1–15, 2012.

NIJIM, M., QIN, X., QIU, M., and LI, K.: "An Adaptive Energy-conserving Strategy for Parallel Disk Systems," *Future Generation Computer Systems*, vol. 29, pp. 196–207, Jan. 2013.

NIST (National Institute of Standards and Technology): FIPS Pub. 180–1, 1995.

NIST (National Institute of Standards and Technology): "The NIST Definition of Cloud Computing," *Special Publication 800-145*, Recommendations of the National Institute of Standards and Technology, 2011.

NO, J.: "NAND Flash Memory-Based Hybrid File System for High I/O Performance," *J. Parallel and Distributed Computing*, vol. 72, pp. 1680–1695, Dec. 2012.

OH, Y., CHOI, J., LEE, D., and NOH, S.H.: "Caching Less for Better Performance: Balancing Cache Size and Update Cost of Flash Memory Cache in Hybrid Storage Systems," *Proc. 10th USENIX Conf. on File and Storage Tech.*, USENIX, pp. 313–326, 2012.

OHNISHI, Y., and YOSHIDA, T.: "Design and Evaluation of a Distributed Shared Memory Network for Application-Specific PC Cluster Systems," *Proc. Workshops of Int'l Conf. on Advanced Information Networking and Applications*, IEEE, pp. 63–70, 2011.

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.

ONGARO, D., RUMBLE, S.M., STUTSMAN, R., OUSTERHOUT, J., and ROSENBLUM, M.: "Fast Crash Recovery in RAMCloud," *Proc. 23rd Symp. of Operating Systems Principles*, ACM, pp. 29–41, 2011.

ORGANICK, E.I.: *The Multics System*, Cambridge, MA: M.I.T. Press, 1972.

ORTOLANI, S., and CRISPO, B.: "NoisyKey: Tolerating Keyloggers via Keystrokes Hiding," *Proc. Seventh USENIX Workshop on Hot Topics in Security*, USENIX, 2012.

ORWICK, P., and SMITH, G.: *Developing Drivers with the Windows Driver Foundation*, Redmond, WA: Microsoft Press, 2007.

OSTRAND, T.J., and WEYUKER, E.J.: "The Distribution of Faults in a Large Industrial Software System," *Proc. 2002 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*, ACM, pp. 55–64, 2002.

OSTROWICK, J.: *Locking Down Linux—An Introduction to Linux Security*, Raleigh, NC: Lulu Press, 2013.

OUSTERHOUT, J.K.: "Scheduling Techniques for Concurrent Systems," *Proc. Third Int'l Conf. on Distrib. Computing Systems*, IEEE, pp. 22–30, 1982.

OUSTERHOUT, J.L.: "Why Threads are a Bad Idea (for Most Purposes)," Presentation at *Proc. USENIX Winter Conf.*, USENIX, 1996.

PARK, S., and SHEN, K.: "FIOS: A Fair, Efficient Flash I/O Scheduler," *Proc. 10th USENIX Conf. on File and Storage Tech.*, USENIX, pp. 155–170, 2012.

PATE, S.D.: *UNIX Filesystems: Evolution, Design, and Implementation*, Hoboken, NJ: John Wiley & Sons, 2003.

PATHAK, A., HU, Y.C., and ZHANG, M.: "Where Is the Energy Spent inside My App? Fine Grained Energy Accounting on Smartphones with Eprof," *Proc. Seventh European Conf. on Computer Systems (EUROSYS)*, ACM, 2012.

PATTERSON, D., and HENNESSY, J.: *Computer Organization and Design*, 5th ed., Burlington, MA: Morgan Kaufman, 2013.

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.

PEARCE, M., ZEADALLY, S., and HUNT, R.: "Virtualization: Issues, Security Threats, and Solutions," *Computing Surveys*, ACM, vol. 45, Art. 17, Feb. 2013.

PENNEMAN, N., KUDINSKLAS, D., RAWSTHORNE, A., DE SUTTER, B., and DE BOSSCHERE, K.: "Formal Virtualization Requirements for the ARM Architecture," *J. System Architecture: the EUROMICRO J.*, vol. 59, pp. 144–154, March 2013.

PESERICO, E.: "Online Paging with Arbitrary Associativity," *Proc. 14th ACM-SIAM Symp. on Discrete Algorithms*, ACM, pp. 555–564, 2003.

PETERSON, G.L.: "Myths about the Mutual Exclusion Problem," *Information Processing Letters*, vol. 12, pp. 115–116, June 1981.

PETRUCCI, V., and LOQUES, O.: "Lucky Scheduling for Energy-Efficient Heterogeneous Multi-core Systems," *Proc. USENIX Workshop on Power-Aware Computing and Systems*, USENIX, 2012.

PETZOLD, C.: *Programming Windows*, 6th ed., Redmond, WA: Microsoft Press, 2013.

PIKE, R., PRESOTTO, D., THOMPSON, K., TRICKEY, H., and WINTERBOTTOM, P.: "The Use of Name Spaces in Plan 9," *Proc. 5th ACM SIGOPS European Workshop*, ACM, pp. 1–5, 1992.

POPEK, G.J., and GOLDBERG, R.P.: "Formal Requirements for Virtualizable Third Generation Architectures," *Commun. of the ACM*, vol. 17, pp. 412–421, July 1974.

PORTNOY, M.: "Virtualization Essentials," Hoboken, NJ: John Wiley & Sons, 2012.

PRABHAKAR, R., KANDEMIR, M., and JUNG, M: "Disk-Cache and Parallelism Aware I/O Scheduling to Improve Storage System Performance," *Proc. 27th Int'l Symp. on Parallel and Distributed Computing*, IEEE, pp. 357–368, 2013.

PRECHELT, L.: "An Empirical Comparison of Seven Programming Languages," *Computer*, vol. 33, pp. 23–29, Oct. 2000.

PYLA, H., and VARADARAJAN, S.: "Transparent Runtime Deadlock Elimination," *Proc. 21st Int'l Conf. on Parallel Architectures and Compilation Techniques*, ACM, pp. 477–478, 2012.

QUIGLEY, E.: *UNIX Shells by Example*, 4th ed., Upper Saddle River, NJ: Prentice Hall, 2004.

RAJGARHIA, A., and GEHANI, A.: "Performance and Extension of User Space File Systems," *Proc. 2010 ACM Symp. on Applied Computing*, ACM, pp. 206–213, 2010.

RASANEH, S., and BANIROSTAM, T.: "A New Structure and Routing Algorithm for Optimizing Energy Consumption in Wireless Sensor Network for Disaster Management," *Proc. Fourth Int'l Conf. on Intelligent Systems, Modelling, and Simulation*, IEEE, pp. 481–485.

RAVINDRANATH, L., PADHYE, J., AGARWAL, S., MAHAJAN, R., OBERMILLER, I., and SHAYANDEH, S.: "AppInsight: Mobile App Performance Monitoring in the Wild," *Proc. 10th Symp. on Operating Systems Design and Implementation*, USENIX, pp. 107–120, 2012.

RECTOR, B.E., and NEWCOMER, J.M.: *Win32 Programming*, Boston: Addison-Wesley, 1997.

REEVES, R.D.: *Windows 7 Device Driver*, Boston: Addison-Wesley, 2010.

RENZELMANN, M.J., KADAV, A., and SWIFT, M.M.: "SymDrive: Testing Drivers without Devices," *Proc. 10th Symp. on Operating Systems Design and Implementation*, USENIX, pp. 279–292, 2012.

RIEBACK, M.R., CRISPO, B., and TANENBAUM, A.S.: "Is Your Cat Infected with a Computer Virus?," *Proc. Fourth IEEE Int'l Conf. on Pervasive Computing and Commun.*, IEEE, pp. 169–179, 2006.

RITCHIE, D.M., and THOMPSON, K.: "The UNIX Timesharing System," *Commun. of the ACM*, vol. 17, pp. 365–375, July 1974.

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.

RIZZO, L.: "Netmap: A Novel Framework for Fast Packet I/O," *Proc. USENIX Ann. Tech. Conf.*, USENIX, 2012.

ROBBINS, A: *UNIX in a Nutshell*, Sebastopol, CA: O'Reilly & Associates, 2005.

RODRIGUES, E.R., NAVAUX, P.O., PANETTA, J., and MENDES, C.L.: "A New Technique for Data Privatization in User-Level Threads and Its Use in Parallel Applications," *Proc. 2010 Symp. on Applied Computing*, ACM, pp. 2149–2154, 2010.

RODRIGUEZ-LUJAN, I., BAILADOR, G., SANCHEZ-AVILA, C., HERRERO, A., and VIDAL-DE-MIGUEL, G.: "Analysis of Pattern Recognition and Dimensionality Reduction Techniques for Odor Biometrics," vol. 52, pp. 279–289, Nov. 2013.

ROSCOE, T., ELPHINSTONE, K., and HEISER, G.: "Hype and Virtue," *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 19–24, 2007.

ROSENBLUM, M., BUGNION, E., DEVINE, S. and HERROD, S.A.: "Using the SIMOS Machine Simulator to Study Complex Computer Systems," *ACM Trans. Model. Comput. Simul.*, vol. 7, pp. 78–103, 1997.

ROSENBLUM, M., and GARFINKEL, T.: "Virtual Machine Monitors: Current Technology and Future Trends," *Computer*, vol. 38, pp. 39–47, May 2005.

ROSENBLUM, M., and OUSTERHOUT, J.K.: "The Design and Implementation of a Log-Structured File System," *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 1–15, 1991.

ROSSBACH, C.J., CURREY, J., SILBERSTEIN, M., RAY, and B., WITCHEL, E.: "PTask: Operating System Abstractions to Manage GPUs as Compute Devices," *Proc. 23rd Symp. of Operating Systems Principles*, ACM, pp. 233–248, 2011.

ROSSOW, C., ANDRIESSE, D., WERNER, T., STONE-GROSS, B., PLOHMANN, D., DIETRICH, C.J., and BOS, H.: "SoK: P2PWNED—Modeling and Evaluating the Resilience of Peer-to-Peer Botnets," *Proc. IEEE Symp. on Security and Privacy*, IEEE, pp. 97–111, 2013.

ROZIER, M., ABROSSIMOV, 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.

RUSSINOVICH, M., and SOLOMON, D.: *Windows Internals, Part 1*, Redmond, WA: Microsoft Press, 2012.

RYZHYK, L., CHUBB, P., KUZ, I., LE SUEUR, E., and HEISER, G.: "Automatic Device Driver Synthesis with Termite," *Proc. 22nd Symp. on Operating Systems Principles*, ACM, 2009.

RYZHYK, L., KEYS, J., MIRLA, B., RAGNUNATH, A., VIJ, M., and HEISER, G.: "Improved Device Driver Reliability through Hardware Verification Reuse," *Proc. 16th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, ACM, pp. 133–134, 2011.

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.

SAITO, Y., KARAMANOLIS, C., KARLSSON, M., and MAHALINGAM, M.: "Taming Aggressive Replication in the Pangea Wide-Area File System," *Proc. Fifth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 15–30, 2002.

SALOMIE T.-I., SUBASU, I.E., GICEVA, J., and ALONSO, G.: "Database Engines on Multicores: Why Parallelize When You can Distribute?," *Proc. Sixth European Conf. on Computer Systems (EUROSYS)*, ACM, pp. 17–30, 2011.

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., and KAASHOEK, M.F.: *Principles of Computer System Design: An Introduction*, Burlington, MA: Morgan Kaufmann, 2009.

SALTZER, J.H., REED, D.P., and CLARK, D.D.: "End-to-End Arguments in System Design," *ACM Trans. on Computer Systems*, vol. 2, pp. 277–288, 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.

SASSE, M.A.: "Red-Eye Blink, Bendy Shuffle, and the Yuck Factor: A User Experience of Biometric Airport Systems," *IEEE Security and Privacy*, vol. 5, pp. 78–81, May/June 2007.

SCHEIBLE, J.P.: "A Survey of Storage Options," *Computer*, vol. 35, pp. 42–46, Dec. 2002.

SCHINDLER, J., SHETE, S., and SMITH, K.A.: "Improving Throughput for Small Disk Requests with Proximal I/O," *Proc. Ninth USENIX Conf. on File and Storage Tech.*, USENIX, pp. 133–148, 2011.

SCHWARTZ, C., PRIES, R., and TRAN-GIA, P.: "A Queuing Analysis of an Energy-Saving Mechanism in Data Centers," *Proc. 2012 Int'l Conf. on Inf. Networking*, IEEE, pp. 70–75, 2012.

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 J.*, vol. 18, pp. 4–17, 1979.

SEREBRYANY, K., BRUENING, D., POTAPENKO, A., and VYUKOV, D.: "AddressSanitizer: A Fast Address Sanity Checker," *Proc. USENIX Ann. Tech. Conf.*, USENIX, pp. 28–28, 2013.

SEVERINI, M., SQUARTINI, S., and PIAZZA, F.: "An Energy Aware Approach for Task Scheduling in Energy-Harvesting Sensor Nodes," *Proc. Ninth Int'l Conf. on Advances in Neural Networks*, Springer-Verlag, pp. 601–610, 2012.

SHEN, K., SHRIRAMAN, A., DWARKADAS, S., ZHANG, X., and CHEN, Z.: "Power Containers: An OS Facility for Fine-Grained Power and Energy Management on Multicore Servers," *Proc. 18th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, ACM, pp. 65–76, 2013.

SILBERSCHATZ, A., GALVIN, P.B., and GAGNE, G.: *Operating System Concepts*, 9th ed., Hoboken, NJ: John Wiley & Sons, 2012.

SIMON, R.J.: *Windows NT Win32 API SuperBible*, Corte Madera, CA: Sams Publishing, 1997.

SITARAM, D., and DAN, A.: *Multimedia Servers*, Burlington, MA: Morgan Kaufman, 2000.

SLOWINSKA, A., STANESCU, T., and BOS, H.: "Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation," *Proc. USENIX Ann. Tech. Conf.*, USENIX, 2012.

SMALDONE, S., WALLACE, G., and HSU, W.: "Efficiently Storing Virtual Machine Backups," *Proc. Fifth USENIX Conf. on Hot Topics in Storage and File Systems*, USENIX, 2013.

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.

SNOW, K., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., and SADEGHI, A.-R.: "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," *Proc. IEEE Symp. on Security and Privacy*, IEEE, pp. 574–588, 2013.

SOBELL, M.: *A Practical Guide to Fedora and Red Hat Enterprise Linux*, 7th ed., Upper Saddle River, NJ: Prentice-Hall, 2014.

SOORTY, B.: "Evaluating IPv6 in Peer-to-peer Gigabit Ethernet for UDP Using Modern Operating Systems," *Proc. 2012 Symp. on Computers and Commun.*, IEEE, pp. 534–536, 2012.

SPAFFORD, E., HEAPHY, K., and FERBRACHE, D.: *Computer Viruses*, Arlington, VA: ADAPSO, 1989.

STALLINGS, W.: *Operating Systems*, 7th ed., Upper Saddle River, NJ: Prentice Hall, 2011.

STAN, M.R., and SKADRON, K: "Power-Aware Computing," *Computer*, vol. 36, pp. 35–38, Dec. 2003.

STEINMETZ, R., and NAHRSTEDT, K.: *Multimedia: Computing, Communications and Applications*, Upper Saddle River, NJ: Prentice Hall, 1995.

STEVENS, R.W., and RAGO, S.A.: "Advanced Programming in the UNIX Environment," Boston: Addison-Wesley, 2013.

STOICA, R., and AILAMAKI, A.: "Enabling Efficient OS Paging for Main-Memory OLTP Databases," *Proc. Ninth Int'l Workshop on Data Management on New Hardware*, ACM, Art. 7. 2013.

STONE, H.S., and BOKHARI, S.H.: "Control of Distributed Processes," *Computer*, vol. 11, pp. 97–106, July 1978.

STORER, M.W., GREENAN, K.M., MILLER, E.L., and VORUGANTI, K.: "POTSHARDS: Secure Long-Term Storage without Encryption," *Proc. USENIX Ann. Tech. Conf.*, USENIX, pp. 143–156, 2007.

STRATTON, J.A., RODRIGUES, C., SUNG, I.-J., CHANG, L.-W., ANSSARI, N., LIU, G., HWU, W.-M., and OBEID, N.: "Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems," *Computer*, vol. 45, pp. 26–32, Aug. 2012.

SUGERMAN, J., VENKITACHALAM , G., and LIM, B.-H: "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," *Proc. USENIX Ann. Tech. Conf.*, USENIX, pp. 1–14, 2001.

SULTANA, S., and BERTINO, E.: "A File Provenance System," *Proc. Third Conf. on Data and Appl. Security and Privacy*, ACM, pp. 153–156, 2013.

SUN, Y., CHEN, M., LIU, B., and MAO, S.: "FAR: A Fault-Avoidance Routing Method for Data Center Networks with Regular Topology," *Proc. Ninth ACM/IEEE Symp. for Arch. for Networking and Commun. Systems*, ACM, pp. 181–190, 2013.

SWANSON, S., and CAULFIELD, A.M.: "Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage," *Computer*, vol. 46, pp. 52–59, Aug. 2013.

TAIABUL HAQUE, S.M., WRIGHT, M., and SCIELZO, S.: "A Study of User Password Strategy for Multiple Accounts," *Proc. Third Conf. on Data and Appl. Security and Privacy*, ACM, pp. 173–176, 2013.

TALLURI, M., HILL, M.D., and KHALIDI, Y.A.: "A New Page Table for 64-Bit Address Spaces," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 184–200, 1995.

TAM, D., AZIMI, R., and STUMM, M.: "Thread Clustering: Sharing-Aware Scheduling," *Proc. Second European Conf. on Computer Systems (EUROSYS)*, ACM, pp. 47–58, 2007.

TANENBAUM, A.S., and AUSTIN, T.: *Structured Computer Organization*, 6th ed., Upper Saddle River, NJ: Prentice Hall, 2012.

TANENBAUM, A.S., HERDER, J.N., and BOS, H.: "File Size Distribution on UNIX Systems: Then and Now," *ACM SIGOPS Operating Systems Rev.*, vol. 40, pp. 100–104, Jan. 2006.

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*, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 2007.

TANENBAUM, A.S., and WETHERALL, D.J: *Computer Networks*, 5th ed., Upper Saddle River, NJ: Prentice Hall, 2010.

TANENBAUM, A.S., and WOODHULL, A.S.: *Operating Systems: Design and Implementation*, 3rd ed., Upper Saddle River, NJ: Prentice Hall, 2006.

TARASOV, V., HILDEBRAND, D., KUENNING, G., and ZADOK, E.: "Virtual Machine Workloads: The Case for New NAS Benchmarks," *Proc. 11th Conf. on File and Storage Technologies*, USENIX, 2013.

TEORY, T.J.: "Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 1–11, 1972.

THEODOROU, D., MAK, R.H., KEIJSER, J.J., and SUERINK, R.: "NRS: A System for Automated Network Virtualization in IAAS Cloud Infrastructures," *Proc. Seventh Int'l Workshop on Virtualization Tech. in Distributed Computing*, ACM, pp. 25–32, 2013.

THIBADEAU, R.: "Trusted Computing for Disk Drives and Other Peripherals," *IEEE Security and Privacy*, vol. 4, pp. 26–33, Sept./Oct. 2006.

THOMPSON, K.: "Reflections on Trusting Trust," *Commun. of the ACM*, vol. 27, pp. 761–763, Aug. 1984.

TIMCENKO, V., and DJORDJEVIC, B.: "The Comprehensive Performance Analysis of Striped Disk Array Organizations—RAID-0," *Proc. 2013 Int'l Conf. on Inf. Systems and Design of Commun.*, ACM, pp. 113–116, 2013.

TRESADERN, P., COOTES, T., POH, N., METEJKA, P., HADID, A., LEVY, C., McCOOL, C., and MARCEL, S.: "Mobile Biometrics: Combined Face and Voice Verification for a Mobile Platform," *IEEE Pervasive Computing*, vol. 12, pp. 79–87, Jan. 2013.

TSAFRIR, D., ETSION, Y., FEITELSON, D.G., and KIRKPATRICK, S.: "System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications," *Proc. 19th Ann. Int'l Conf. on Supercomputing*, ACM, pp. 303–312, 2005.

TUAN-ANH, B., HUNG, P.P., and HUH, E.-N.: "A Solution of Thin-Thick Client Collaboration for Data Distribution and Resource Allocation in Cloud Computing," *Proc. 2013 Int'l Conf. on Inf. Networking*, IEEE, pp. 238–243, 2103.

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.

UHLIG, R. NEIGER, G., RODGERS, D., SANTONI, A.L., MARTINS, F.C.M., ANDERSON, A.V., BENNET, S.M., KAGI, A., LEUNG, F.H., and SMITH, L.: "Intel Virtualization Technology," *Computer*, vol. 38, pp. 48–56, 2005.

UR, B., KELLEY, P.G., KOMANDURI, S., LEE, J., MAASS, M., MAZUREK, M.L., PASSARO, T., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., and CRANOR, L.F.: "How Does Your Password Measure Up? The Effect of Strength Meters on Password Creation," *Proc. 21st USENIX Security Symp.*, USENIX, 2012.

VAGHANI, S.B.: "Virtual Machine File System," *ACM SIGOPS Operating Systems Rev.*, vol. 44, pp. 57–70, 2010.

VAHALIA, U.: *UNIX Internals—The New Frontiers*, Upper Saddle River, NJ: Prentice Hall, 2007.

VAN DOORN, L.: *The Design and Application of an Extensible Operating System*, Capelle a/d Ijssel: Labyrint Publications, 2001.

VAN MOOLENBROEK, D.C., APPUSWAMY, R., and TANENBAUM, A.S.: "Integrated System and Process Crash Recovery in the Loris Storage Stack," *Proc. Seventh Int'l Conf. on Networking, Architecture, and Storage*, IEEE, pp. 1–10, 2012.

VAN 'T NOORDENDE, G., BALOGH, A., HOFMAN, R., BRAZIER, F.M.T., and TANENBAUM, A.S.: "A Secure Jailing System for Confining Untrusted Applications," *Proc. Second Int'l Conf. on Security and Cryptography*, INSTICC, pp. 414–423, 2007.

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.

VAN DER VEEN, V., DDUTT-SHARMA, N., CAVALLARO, L., and BOS, H.: "Memory Errors: The Past, the Present, and the Future," *Proc. 15th Int'l Conf. on Research in Attacks, Intrusions, and Defenses*, Berlin: Springer-Verlag, pp. 86–106, 2012.

VENKATACHALAM, V., and FRANZ, M.: "Power Reduction Techniques for Microprocessor Systems," *Computing Surveys*, vol. 37, pp. 195–237, Sept. 2005.

VIENNOT, N., NAIR, S., and NIEH, J.: "Transparent Mutable Replay for Multicore Debugging and Patch Validation," *Proc. 18th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, ACM, 2013.

VINOSKI, S.: "CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 35, pp. 46–56, Feb. 1997.

VISCAROLA, P.G, MASON, T., CARIDDI, M., RYAN, B., and NOONE, S.: *Introduction to the Windows Driver Foundation Kernel-Mode Framework*, Amherst, NH: OSR Press, 2007.

VMWARE, Inc.: "Achieving a Million I/O Operations per Second from a Single VMware vSphere 5.0 Host," *http://www.vmware.com/files/pdf/1M-iops-perf-vsphere5.pdf*, 2011.

VOGELS, W.: "File System Usage in Windows NT 4.0," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 93–109, 1999.

VON BEHREN, R., CONDIT, J., and BREWER, E.: "Why Events Are A Bad Idea (for High-Concurrency Servers)," *Proc. Ninth Workshop on Hot Topics in Operating Systems*, USENIX, pp. 19–24, 2003.

VON EICKEN, T., CULLER, D., GOLDSTEIN, S.C., and SCHAUSER, K.E.: "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Int'l Symp. on Computer Arch.*, ACM, pp. 256–266, 1992.

VOSTOKOV, D.: *Windows Device Drivers: Practical Foundations*, Opentask, 2009.

VRABLE, M., SAVAGE, S., and VOELKER, G.M.: "BlueSky: A Cloud-Backed File System for the Enterprise," *Proc. 10th USENIX Conf. on File and Storage Tech.*, USENIX, pp. 124–250, 2012.

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.

WALDSPURGER, C.A.: "Memory Resource Management in VMware ESX Server," *ACM SIGOPS Operating System Rev.*, vol 36, pp. 181–194, Jan. 2002.

WALDSPURGER, C.A., and ROSENBLUM, M.: "I/O Virtualization," *Commun. of the ACM*, vol. 55, pp. 66–73, 2012.

WALDSPURGER, C.A., and WEIHL, W.E.: "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. First Symp. on Operating Systems 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.

WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., and HSU., W.: "Characteristics of Backup Workloads in Production Systems," *Proc. 10th USENIX Conf. on File and Storage Tech.*, USENIX, pp. 33–48, 2012.

WANG, L., KHAN, S.U., CHEN, D., KOLODZIEJ, J., RANJAN, R., XU, C.-Z., and ZOMAYA, A.: "Energy-Aware Parallel Task Scheduling in a Cluster," *Future Generation Computer Systems*, vol. 29, pp. 1661–1670, Sept. 2013b.

WANG, X., TIPPER, D., and KRISHNAMURTHY, P.: "Wireless Network Virtualization," *Proc. 2013 Int'l Conf. on Computing, Networking, and Commun.*, IEEE, pp. 818–822, 2013a.

WANG, Y. and LU, P.: "DDS: A Deadlock Detection-Based Scheduling Algorithm for Workflow Computations in HPC Systems with Storage Constraints," *Parallel Comput.*, vol. 39, pp. 291–305, August 2013.

WATSON, R., ANDERSON, J., LAURIE, B., and KENNAWAY, K.: "A Taste of Capsicum: Practical Capabilities for UNIX," *Commun. of the ACM*, vol. 55, pp. 97–104, March 2013.

WEI, M., GRUPP, L., SPADA, F.E., and SWANSON, S.: "Reliably Erasing Data from Flash-Based Solid State Drives," *Proc. Ninth USENIX Conf. on File and Storage Tech.*, USENIX, pp. 105–118, 2011.

WEI, Y.-H., YANG, C.-Y., KUO, T.-W., HUNG, S.-H., and CHU, Y.-H.: "Energy-Efficient Real-Time Scheduling of Multimedia Tasks on Multi-core Processors," *Proc. 2010 Symp. on Applied Computing*, ACM, pp. 258–262, 2010.

WEISER, M., WELCH, B., DEMERS, A., and SHENKER, S.: "Scheduling for Reduced CPU Energy," *Proc. First Symp. on Operating Systems Design and Implementation*, USENIX, pp. 13–23, 1994.

WEISSEL, A.: *Operating System Services for Task-Specific Power Management: Novel Approaches to Energy-Aware Embedded Linux*, AV Akademikerverlag, 2012.

WENTZLAFF, D., GRUENWALD III, C., BECKMANN, N., MODZELEWSKI, K., BELAY, A., YOUSEFF, L., MILLER, J., and AGARWAL, A.: "An Operating System for Multicore and Clouds: Mechanisms and Implementation," *Proc. Cloud Computing*, ACM, June 2010.

WENTZLAFF, D., JACKSON, C.J., GRIFFIN, P., and AGARWAL, A.: "Configurable Fine-grain Protection for Multicore Processor Virtualization," *Proc. 39th Int'l Symp. on Computer Arch.*, ACM, pp. 464–475, 2012.

WHITAKER, A., COX, R.S., SHAW, M, and GRIBBLE, S.D.: "Rethinking the Design of Virtual Machine Monitors," *Computer*, vol. 38, pp. 57–62, May 2005.

WHITAKER, A., SHAW, M, and GRIBBLE, S.D.: "Scale and Performance in the Denali Isolation Kernel," *ACM SIGOPS Operating Systems Rev.*, vol. 36, pp. 195–209, Jan. 2002.

WILLIAMS, D., JAMJOOM, H., and WEATHERSPOON, H.: "The Xen-Blanket: Virtualize Once, Run Everywhere," *Proc. Seventh European Conf. on Computer Systems (EUROSYS)*, ACM, 2012.

WIRTH, N.: "A Plea for Lean Software," *Computer*, vol. 28, pp. 64–68, Feb. 1995.

WU, N., ZHOU, M., and HU, U.: "One-Step Look-Ahead Maximally Permissive Deadlock Control of AMS by Using Petri Nets," *ACM Trans. Embed. Comput. Syst. ,#*, vol. 12, Art. 10, pp. 10:1–10:23, Jan. 2013.

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.

YANG, J., TWOHEY, P., ENGLER, D., and MUSUVATHI, M.: "Using Model Checking to Find Serious File System Errors," *ACM Trans. on Computer Systems*, vol. 24, pp. 393–423, 2006.

YEH, T., and CHENG, W.: "Improving Fault Tolerance through Crash Recovery," *Proc. 2012 Int'l Symp. on Biometrics and Security Tech.*, IEEE, pp. 15–22, 2012.

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.

YUAN, D., LEWANDOWSKI, C., and CROSS, B.: "Building a Green Unified Computing IT Laboratory through Virtualization," *J. Computing Sciences in Colleges*, vol. 28, pp. 76–83, June 2013.

YUAN, J., JIANG, X., ZHONG, L., and YU, H.: "Energy Aware Resource Scheduling Algorithm for Data Center Using Reinforcement Learning," *Proc. Fifth Int'l Conf. on Intelligent Computation Tech. and Automation*, IEEE, pp. 435–438, 2012.

YUAN, W., and NAHRSTEDT, K.: "Energy-Efficient CPU Scheduling for Multimedia Systems," *ACM Trans. on Computer Systems*, ACM, vol. 24, pp. 292–331, Aug. 2006.

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 Systems Design and Implementation*, USENIX, pp. 87–100, 1994.

ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., and ZOU, W.: "Practical Control Flow Integrity and Randomization for Binary Executables," *Proc. IEEE Symp. on Security and Privacy*, IEEE, pp. 559–573, 2013b.

ZHANG, F., CHEN, J., CHEN, H., and ZANG, B.: "CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization," *Proc. 23rd Symp. on Operating Systems Principles*, ACM, 2011.

ZHANG, M., and SEKAR, R.: "Control Flow Integrity for COTS Binaries," *Proc. 22nd USENIX Security Symp.*, USENIX, pp. 337–352, 2013.

ZHANG, X., DAVIS, K., and JIANG, S.: "iTransformer: Using SSD to Improve Disk Scheduling for High-Performance I/O," *Proc. 26th Int'l Parallel and Distributed Processing Symp.*, IEEE, pp. 715-726, 2012b.

ZHANG, Y., LIU, J., and KANDEMIR, M.: "Software-Directed Data Access Scheduling for Reducing Disk Energy Consumption," *Proc. 32nd Int'l Conf. on Distributed Computer Systems*, IEEE, pp. 596–605, 2012a.

ZHANG, Y., SOUNDARARAJAN, G., STORER, M.W., BAIRAVASUNDARAM, L., SUBBIAH, S., ARPACI-DUSSEAU, A.C., and ARPACI-DUSSEAU, R.H.: "Warming Up Storage-Level Caches with Bonfire," *Proc. 11th Conf. on File and Storage Technologies*, USENIX, 2013a.

ZHENG, H., ZHANG, X., WANG, E., WU, N., and DONG, X.: "Achieving High Reliability on Linux for K2 System," *Proc. 11th Int'l Conf. on Computer and Information Science*, IEEE, pp. 107–112, 2012.

ZHOU, B., KULKARNI, M., and BAGCHI, S.: "ABHRANTA: Locating Bugs that Manifest at Large System Scales," *Proc. Eighth USENIX Workshop on Hot Topics in System Dependability*, USENIX, 2012.

ZHURAVLEV, S., SAEZ, J.C., BLAGODUROV, S., FEDOROVA, A., and PRIETO, M.: "Survey of scheduling techniques for addressing shared resources in multicore processors," *Computing Surveys*, ACM , vol 45, Number 1, Art. 4, 2012.

ZOBEL, D.: "The Deadlock Problem: A Classifying Bibliography," *ACM SIGOPS Operating Systems Rev.*, 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

This page intentionally left blank

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

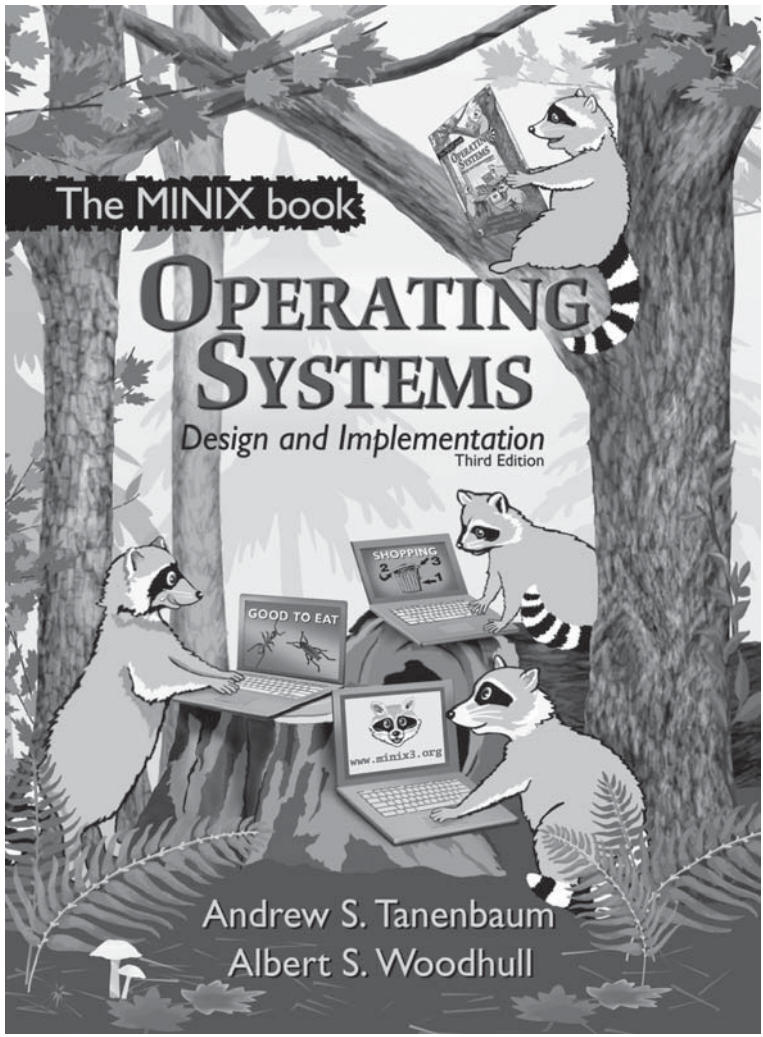
Z

This page intentionally left blank

Also by Andrew S. Tanenbaum and Albert S. Woodhull

Operating Systems: Design and Implementation, 3rd ed.

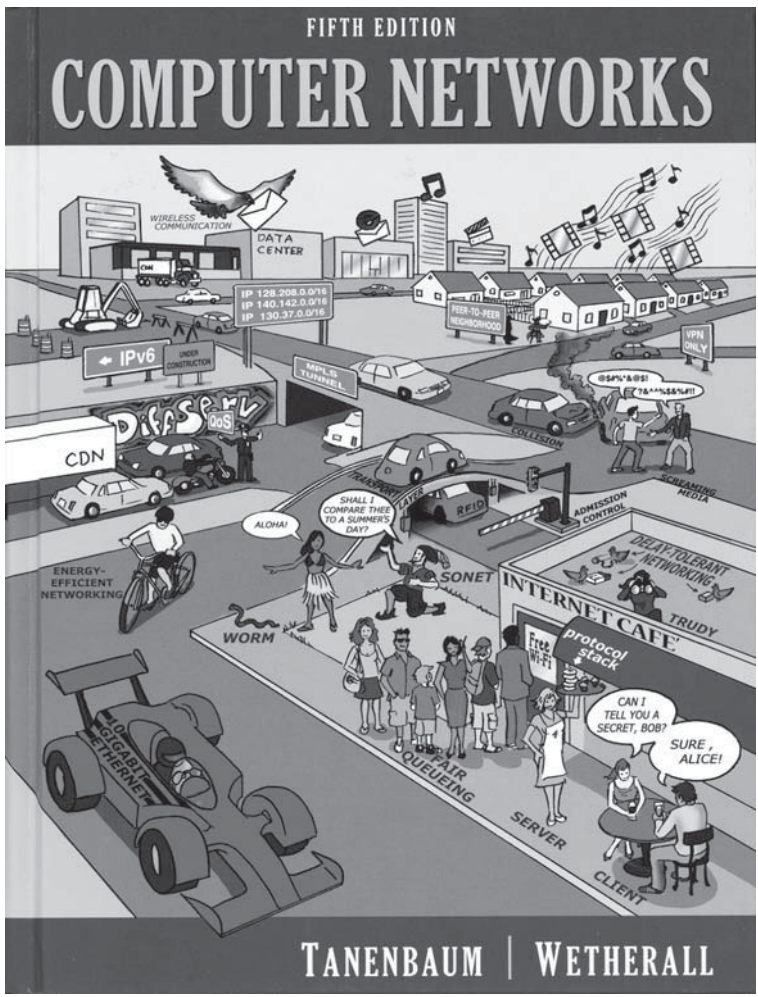
All other textbooks on operating systems are long on theory and short on practice. This one is different. In addition to the usual material on processes, memory management, file systems, I/O, and so on, it contains a CD-ROM with the source code (in C) of a small, but complete, POSIX-conformant operating system called MINIX 3 (see *www.minix3.org*). All the principles are illustrated by showing how they apply to MINIX 3. The reader can also compile, test, and experiment with MINIX 3, leading to in-depth knowledge of how an operating system really works.



Also by Andrew S. Tanenbaum and David J. Wetherall

Computer Networks, 5th ed.

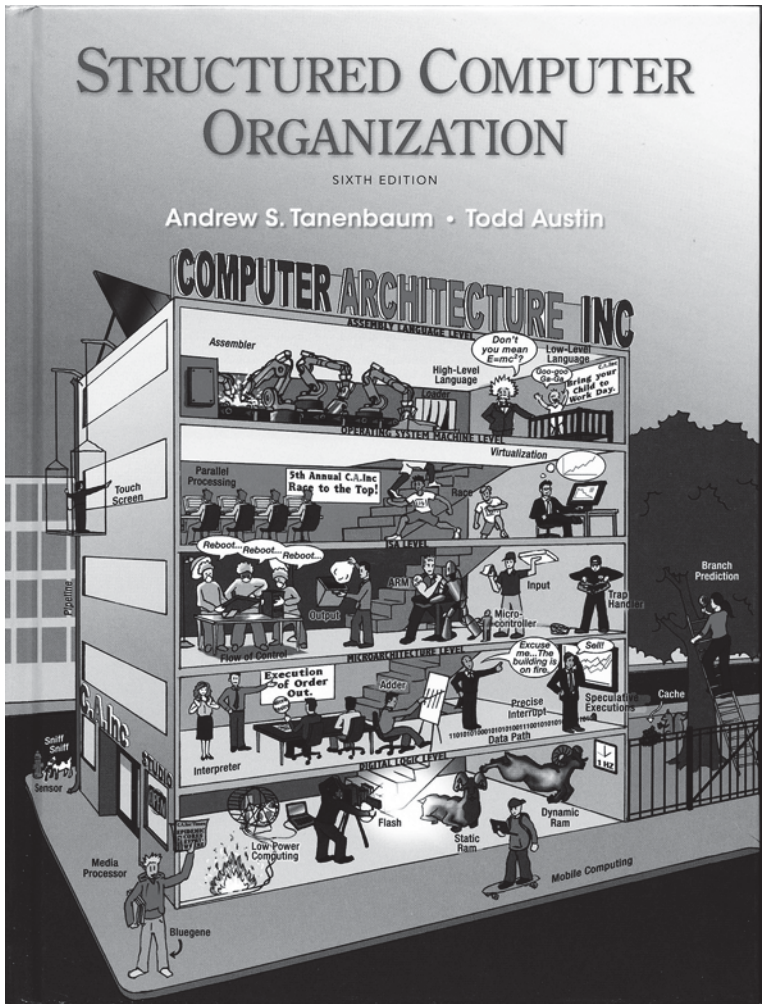
This widely read classic, with a fifth edition co-authored with David Wetherall, provides the ideal introduction to today's and tomorrow's networks. It explains in detail how modern networks are structured. Starting with the physical layer and working up to the application layer, the book covers a vast number of important topics, including wireless communication, fiber optics, data link protocols, Ethernet, routing algorithms, network performance, security, DNS, electronic mail, the World Wide Web, and multimedia. The book has especially thorough coverage of TCP/IP and the Internet.



Also by Andrew S. Tanenbaum and Todd Austin

Structured Computer Organization, 6th ed.

Computers are getting more complicated every year but this best-selling book makes computer architecture and organization easy to understand. It starts at the very beginning explaining how a transistor works and from there explains the basic circuits from which computers are built. Then it moves up the design stack to cover the microarchitecture, and the assembly language level. The final chapter is about parallel computer architectures. No hardware background is needed to understand any part of this book.



Also by Andrew S. Tanenbaum and Maarten van Steen

Distributed Systems: Principles and Paradigms, 2nd ed.

Distributed systems are becoming ever-more important in the world and this book explains their principles and illustrates them with numerous examples. Among the topics covered are architectures, processes, communication, naming, synchronization, consistency, fault tolerance, and security. Examples are taken from distributed object-based, file, Web-based, and coordination-based systems.