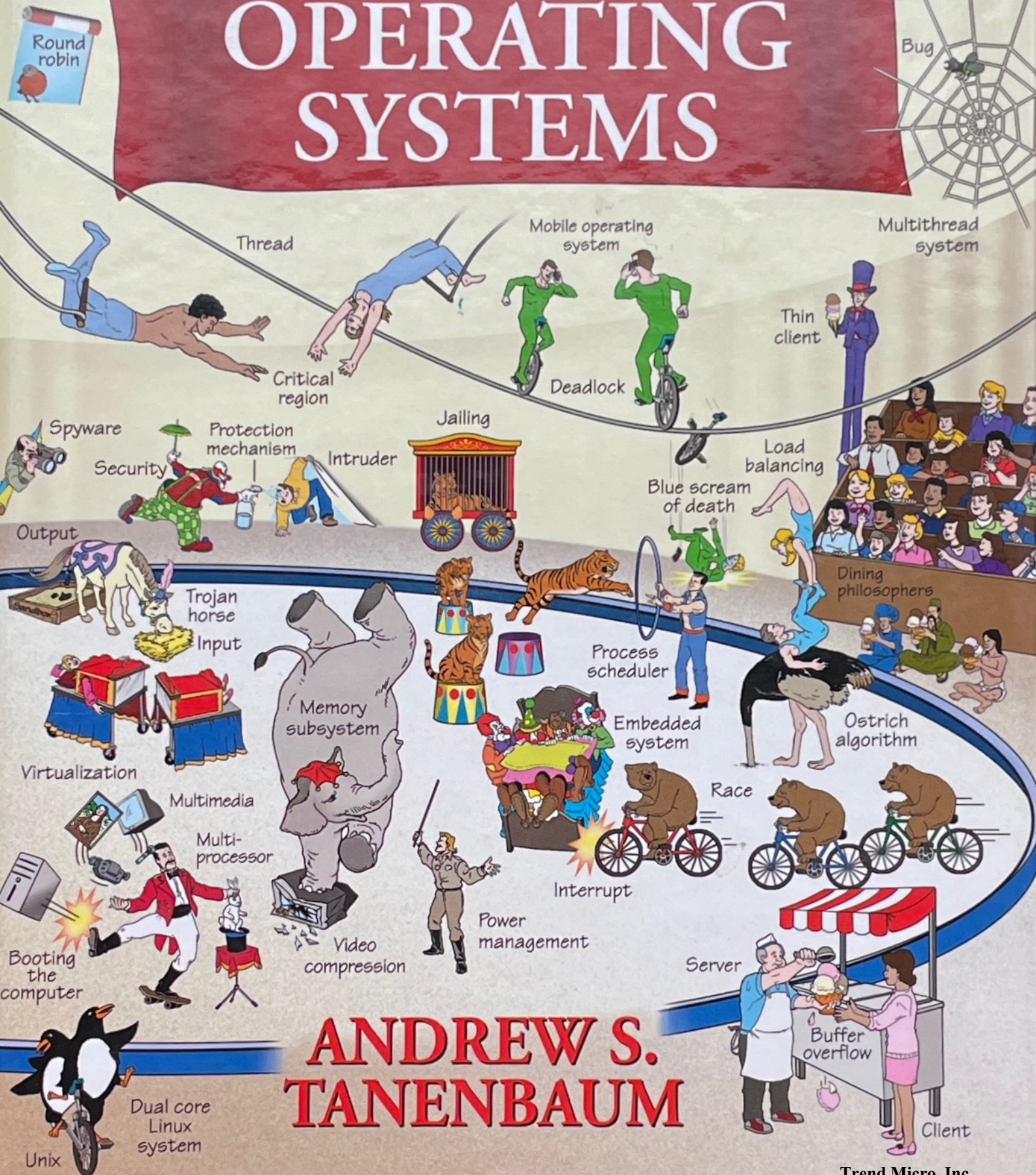


MODERN^{3e} OPERATING SYSTEMS



**ANDREW S.
TANENBAUM**

Library of Congress Cataloging-in-Publication Data on File.

Editorial Director, Computer Science, Engineering, and Advanced Mathematics: *Marcia J. Horton*
Executive Editor: *Tracy Dunkelberger*
Editorial Assistant: *Melinda Haggerty*
Associate Editor: *ReeAnne Davies*
Senior Managing Editor: *Scott Disanno*
Production Editor: *Irwin Zucker*
Cover Concept: *Andrew S. Tanenbaum and Tracy Dunkelberger*
Cover Design: *Tamara Newnam*
Cover Illustrator: *Steve Lefkowitz*
Interior design: *Andrew S. Tanenbaum*
Typesetting: *Andrew S. Tanenbaum*
Art Director: *Kenny Beck*
Art Editor: *Gregory Dulles*
Media Editor: *David Alick*
Manufacturing Manager: *Alan Fischer*
Manufacturing Buyer: *Lisa McDowell*
Marketing Manager: *Mack Patterson*



© 2008 Pearson Education, Inc.
Pearson Prentice Hall
Pearson Education, Inc.
Upper Saddle River, NJ 07458

All rights reserved. No part of this book may be reproduced in any form or by any means, without permission in writing from the publisher.

Pearson Prentice Hall™ is a trademark of Pearson Education, Inc.

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-600663-9
978-0-13-600663-2

Pearson Education Ltd., London
Pearson Education Australia Pty. Ltd., Sydney
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd., Hong Kong
Pearson Education Canada, Inc., Toronto
Pearson Educación de México, S.A. de C.V.
Pearson Education—Japan, Tokyo
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., Upper Saddle River, New Jersey

Trend Micro, Inc.
EX1007-2

With Windows, the situation is radically different. To start with, the library calls and the actual system calls are highly decoupled. Microsoft has defined a set of procedures called the **Win32 API (Application Program Interface)** that programmers are expected to use to get operating system services. This interface is (partially) supported on all versions of Windows since Windows 95. By decoupling the interface from the actual system calls, Microsoft retains the ability to change the actual system calls in time (even from release to release) without invalidating existing programs. What actually constitutes Win32 is also slightly ambiguous because Windows 2000, Windows XP, and Windows Vista have many new calls that were not previously available. In this section, Win32 means the interface supported by all versions of Windows.

The number of Win32 API calls is extremely large, numbering in the thousands. Furthermore, while many of them do invoke system calls, a substantial number are carried out entirely in user space. As a consequence, with Windows it is impossible to see what is a system call (i.e., performed by the kernel) and what is simply a user-space library call. In fact, what is a system call in one version of Windows may be done in user space in a different version, and vice versa. When we discuss the Windows system calls in this book, we will use the Win32 procedures (where appropriate) since Microsoft guarantees that these will be stable over time. But it is worth remembering that not all of them are true system calls (i.e., traps to the kernel).

The Win32 API has a huge number of calls for managing windows, geometric figures, text, fonts, scrollbars, dialog boxes, menus, and other features of the GUI. To the extent that the graphics subsystem runs in the kernel (true on some versions of Windows but not on all), these are system calls; otherwise they are just library calls. Should we discuss these calls in this book or not? Since they are not really related to the function of an operating system, we have decided not to, even though they may be carried out by the kernel. Readers interested in the Win32 API should consult one of the many books on the subject (e.g., Hart, 1997; Rector and Newcomer, 1997; and Simon, 1997).

Even introducing all the Win32 API calls here is out of the question, so we will restrict ourselves to those calls that roughly correspond to the functionality of the UNIX calls listed in Fig. 1-18. These are listed in Fig. 1-23.

Let us now briefly go through the list of Fig. 1-23. `CreateProcess` creates a new process. It does the combined work of `fork` and `execve` in UNIX. It has many parameters specifying the properties of the newly created process. Windows does not have a process hierarchy as UNIX does so there is no concept of a parent process and a child process. After a process is created, the creator and createe are equals. `WaitForSingleObject` is used to wait for an event. Many possible events can be waited for. If the parameter specifies a process, then the caller waits for the specified process to exit, which is done using `ExitProcess`.

The next six calls operate on files and are functionally similar to their UNIX counterparts although they differ in the parameters and details. Still, files can be

I look for patterns in files. Please enter your pattern.” After getting the pattern, *grep* prompts for a file name. Then it asks if there are any more file names. Finally, it summarizes what it is going to do and asks if that is correct. While this kind of user interface may be suitable for rank novices, it drives skilled programmers up the wall. What they want is a servant, not a nanny.

10.2.2 Interfaces to Linux

A Linux system can be regarded as a kind of pyramid, as illustrated in Fig. 10-1. At the bottom is the hardware, consisting of the CPU, memory, disks, a monitor and keyboard, and other devices. Running on the bare hardware is the operating system. Its function is to control the hardware and provide a system call interface to all the programs. These system calls allow user programs to create and manage processes, files, and other resources.

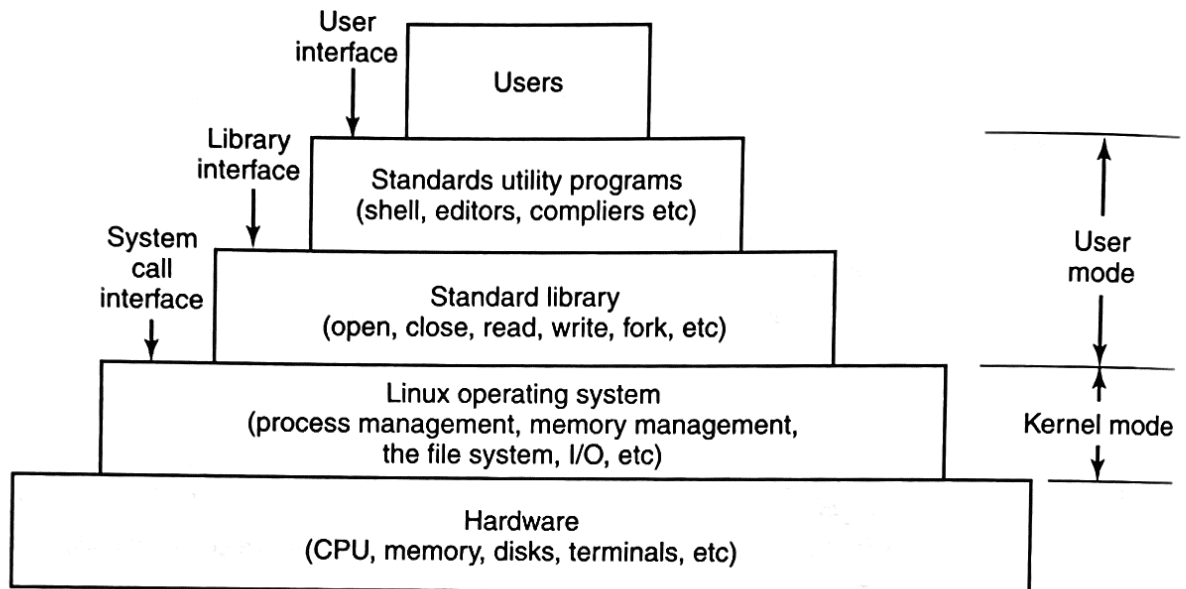


Figure 10-1. The layers in a Linux system.

Programs make system calls by putting the arguments in registers (or sometimes, on the stack), and issuing trap instructions to switch from user mode to kernel mode. Since there is no way to write a trap instruction in C, a library is provided, with one procedure per system call. These procedures are written in assembly language, but can be called from C. Each one first puts its arguments in the proper place, then executes the trap instruction. Thus to execute the read system call, a C program can call the *read* library procedure. As an aside, it is the library interface, and not the system call interface, that is specified by POSIX. In other words, POSIX tells which library procedures a conformant system must supply, what their parameters are, what they must do, and what results they must return. It does not even mention the actual system calls.

or modify the affinity requirements of a select thread. Finally, the scheduler performs periodic load balancing across runqueues of different CPUs to ensure that the system load is well balanced, while still meeting certain performance or affinity requirements.

The scheduler considers only runnable tasks, which are placed on the appropriate runqueue. Tasks which are not runnable and are waiting on various I/O operations or other kernel events are placed on another data structure, **waitqueue**. A waitqueue is associated with each event that tasks may wait on. The head of the waitqueue includes a pointer to a linked list of tasks and a spinlock. The spinlock is necessary so as to ensure that the waitqueue can be concurrently manipulated through both the main kernel code and interrupt handlers or other asynchronous invocations.

In fact, the kernel code contains synchronization variables in numerous locations. Earlier Linux kernels had just one **big kernel lock (BLK)**. This proved highly inefficient, particularly on multiprocessor platforms, since it prevented processes on different CPUs from executing kernel code concurrently. Hence, many new synchronization points were introduced at much finer granularity.

10.3.5 Booting Linux

Details vary from platform to platform, but in general the following steps represent the boot process. When the computer starts, the BIOS performs Power-On-Self-Test (POST) and initial device discovery and initialization, since the OS' boot process may rely on access to disks, screens, keyboards, and so on. Next, the first sector of the boot disk, the **MBR (Master Boot Record)**, is read into a fixed memory location and executed. This sector contains a small (512-byte) program that loads a standalone program called **boot** from the boot device, usually an IDE or SCSI disk. The *boot* program first copies itself to a fixed high-memory address to free up low memory for the operating system.

Once moved, *boot* reads the root directory of the boot device. To do this, it must understand the file system and directory format, which is the case with some bootloaders such as **GRUB (GRand Unified Bootloader)**. Other popular bootloaders, such as Intel's LILO, do not rely on any specific filesystem. Instead, they need a block map and low-level addresses, which describe physical sectors, heads, and cylinders, to find the relevant sectors to be loaded.

Then *boot* reads in the operating system kernel and jumps to it. At this point, it has finished its job and the kernel is running.

The kernel start-up code is written in assembly language and is highly machine dependent. Typical work includes setting up the kernel stack, identifying the CPU type, calculating the amount of RAM present, disabling interrupts, enabling the MMU, and finally calling the C-language *main* procedure to start the main part of the operating system.

The C code also has considerable initialization to do, but this is more logical than physical. It starts out by allocating a message buffer to help debug boot problems. As initialization proceeds, messages are written here about what is happening, so that they can be fished out after a boot failure by a special diagnostic program. Think of this as the operating system's cockpit flight recorder (the black box investigators look for after a plane crash).

Next the kernel data structures are allocated. Most are fixed size, but a few, such as the page cache and certain page table structures, depend on the amount of RAM available.

At this point the system begins autoconfiguration. Using configuration files telling what kinds of I/O devices might be present, it begins probing the devices to see which ones actually are present. If a probed device responds to the probe, it is added to a table of attached devices. If it fails to respond, it is assumed to be absent and ignored henceforth. Unlike traditional UNIX versions, Linux device drivers do not need to be statically linked and may be loaded dynamically (as can all versions of MS-DOS and Windows, incidentally).

The arguments for and against dynamically loading drivers are interesting and worth stating briefly. The main argument for dynamic loading is that a single binary can be shipped to customers with divergent configurations and have it automatically load the drivers it needs, possibly even over a network. The main argument against dynamic loading is security. If you are running a secure site, such as a bank's database or a corporate Web server, you probably want to make it impossible for anyone to insert random code into the kernel. The system administrator may keep the operating system sources and object files on a secure machine, do all system builds there, and ship the kernel binary to other machines over a local area network. If drivers cannot be loaded dynamically, this scenario prevents machine operators and others who know the superuser password from injecting malicious or buggy code into the kernel. Furthermore, at large sites, the hardware configuration is known exactly at the time the system is compiled and linked. Changes are sufficiently rare that having to relink the system when a new hardware device is added is not an issue.

Once all the hardware has been configured, the next thing to do is to carefully handcraft process 0, set up its stack, and run it. Process 0 continues initialization, doing things like programming the real-time clock, mounting the root file system, and creating *init* (process 1) and the page daemon (process 2).

Init checks its flags to see if it is supposed to come up single user or multiuser. In the former case, it forks off a process that executess the shell and waits for this process to exit. In the latter case, it forks off a process that executes the system initialization shell script, */etc/rc*, which can do file system consistency checks, mount additional file systems, start daemon processes, and so on. Then it reads */etc/ttys*, which lists the terminals and some of their properties. For each enabled terminal, it forks off a copy of itself, which does some housekeeping and then executess a program called *getty*.

Getty sets the line speed and other properties for each line (some of which may be modems, for example), and then types

login:

on the terminal's screen and tries to read the user's name from the keyboard. When someone sits down at the terminal and provides a login name, *getty* terminates by executing */bin/login*, the login program. *Login* then asks for a password, encrypts it, and verifies it against the encrypted password stored in the password file, */etc/passwd*. If it is correct, *login* replaces itself with the user's shell, which then waits for the first command. If it is incorrect, *login* just asks for another user name. This mechanism is shown in Fig. 10-11 for a system with three terminals.

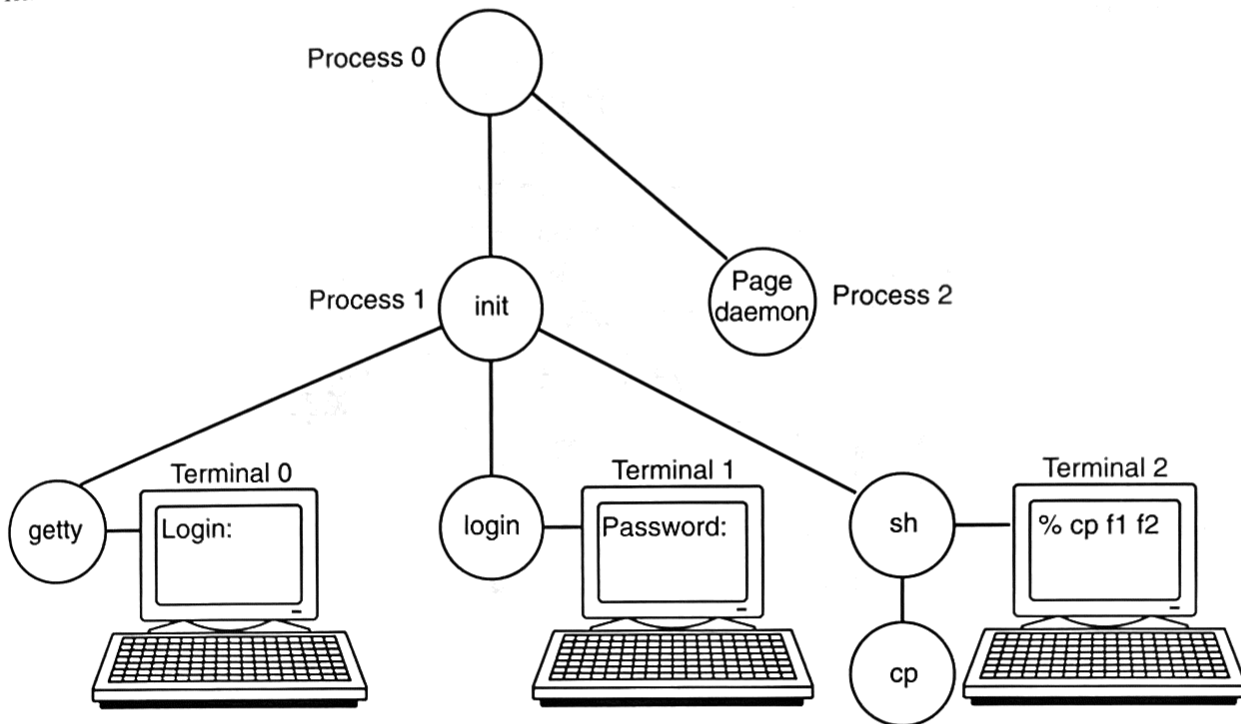


Figure 10-11. The sequence of processes used to boot some Linux systems.

In the figure, the *getty* process running for terminal 0 is still waiting for input. On terminal 1, a user has typed a login name, so *getty* has overwritten itself with *login*, which is asking for the password. A successful login has already occurred on terminal 2, causing the shell to type the prompt (%). The user then typed

`cp f1 f2`

which has caused the shell to fork off a child process and have that process execute the *cp* program. The shell is blocked, waiting for the child to terminate, at which time the shell will type another prompt and read from the keyboard. If the user at terminal 2 had typed *cc* instead of *cp*, the main program of the C compiler would have been started, which in turn would have forked off more processes to run the various compiler passes.

special prefix strings, like "\\"). In Win32, files are accessed relative to *drive letters*. The NT directory `\DosDevices` contains a set of symbolic links from drive letters to the actual device objects. For example `\DosDevices\C:` might be a link to `\Device\HarddiskVolume1`. This directory also contains links for other Win32 devices, such as `COM1:`, `LPT1:`, and `NUL:` (for the serial and printer ports, and the all-important null device). `\DosDevices` is really a symbolic link to `\??` which was chosen for efficiency. Another NT directory, `\BaseNamedObjects` is used to store miscellaneous named kernel-mode objects accessible through the Win32 API. These include synchronization objects like semaphores, shared memory, timers, and communication ports. MS-DOS and device names.

In addition to low-level system interfaces we have described, the Win32 API also supports many functions for GUI operations, including all the calls for managing the graphical interface of the system. There are calls for creating, destroying, managing and using windows, menus, tool bars, status bars, scroll bars, dialog boxes, icons, and many more items that appear on the screen. There are calls for drawing geometric figures, filling them in, managing the color palettes they use, dealing with fonts, and placing icons on the screen. Finally, there are calls for dealing with the keyboard, mouse and other human input devices as well as audio, printing, and other output devices.

The GUI operations work directly with the `win32k.sys` driver using special interfaces to access these functions in kernel mode from user-mode libraries. Since these calls do not involve the core system calls in the NTOS executive, we will not say more about them.

11.2.3 The Windows Registry

The root of the NT namespace is maintained in the kernel. Storage, such as file system volumes, is attached to the NT namespace. Since the NT namespace is constructed afresh every time the system boots, how does the system know about any specific details of the system configuration? The answer is that Windows attaches a special kind of file system (optimized for small files) to the NT namespace. This file system is called the **registry**. The registry is organized into separate volumes called **hives**. Each hive is kept in a separate file (in the directory `C:\Windows\system32\config\` of the boot volume). When a Windows system boots, one particular hive named *SYSTEM* is loaded into memory by the same boot program that loads the kernel and other boot files, such as boot drivers, from the boot volume.

Windows keeps a great deal of crucial information in the *SYSTEM* hive, including information about what drivers to use with what devices, what software to run initially, and many parameters governing the operation of the system. This information is used even by the boot program itself to determine which drivers are boot drivers, being needed immediately upon boot. Such drivers include those

that understand the file system and disk drivers for the volume containing the operating system itself.

Other configuration hives are used after the system boots to describe information about the software installed on the system, particular users, and the classes of user-mode **COM (Component Object-Model)** objects that are installed on the system. Login information for local users is kept in the SAM (Security Access Manager) hive. Information for network users is maintained by the *lsass* service in the SECURITY hive, and coordinated with the network directory servers so that users can have a common account name and password across an entire network. A list of the hives used in Windows Vista is shown in Fig. 11-11.

Hive file	Mounted name	Use
SYSTEM	HKLM TEM	OS configuration information, used by kernel
HARDWARE	HKLM DWARE	In-memory hive recording hardware detected
BCD	HKLM BCD*	Boot Configuration Database
SAM	HKLM	Local user account information
SECURITY	HKLM URITY	Isass' account and other security information
DEFAULT	HKEY_USERS.DEFAULT	Default hive for new users
NTUSER.DAT	HKEY_USERS <user id>	User-specific hive, kept in home directory
SOFTWARE	HKLM TWARE	Application classes registered by COM
COMPONENTS	HKLM NENTS	Manifests and dependencies for sys. componer

Figure 11-11. The registry hives in Windows Vista. HKLM is a short-hand for *HKEY_LOCAL_MACHINE*.

Prior to the introduction of the registry, configuration information in Windows was kept in hundreds of *.ini* (initialization) files spread across the disk. The registry gathers these files into a central store, which is available early in the process of booting the system. This is important for implementing Windows plug-and-play functionality. But the registry has become very disorganized as Windows has evolved. There are poorly defined conventions about how the configuration information should be arranged, and many applications take an ad hoc approach. Most users, applications, and all drivers run with full privileges, and frequently modify system parameters in the registry directly—sometimes interfering with each other and destabilizing the system.

The registry is a strange cross between a file system and a database, and yet really unlike either. Entire books have been written describing the registry (Born, 1998; Hipson, 2000; and Ivens, 1998), and many companies have sprung up to sell special software just to manage the complexity of the registry.

To explore the registry Windows has a GUI program called **regedit** that allows you to open and explore the directories (called *keys*) and data items (called *values*). Microsoft's new **PowerShell** scripting language can also be useful for walking through the keys and values of the registry as if they were directories and

files. A more interesting tool to use is *procmon*, which is available from Microsoft's tools' Website: www.microsoft.com/technet/sysinternals.

Procmon watches all the registry accesses that take place in the system and is very illuminating. Some programs will access the same key over and over tens of thousands of times.

As the name implies, *regedit* allows users to edit the registry—but be very careful if you ever do. It is very easy to render your system unable to boot, or damage the installation of applications so that you cannot fix them without a lot of wizardry. Microsoft promised to clean up the registry in future releases, but for now it is a huge mess—far more complicated than the configuration information maintained in UNIX.

Beginning with Windows Vista Microsoft has introduced a kernel-based transaction manager with support for coordinated transactions that span both file system and registry operations. Microsoft plans to use this facility in the future to avoid some of the metadata corruption problems that occur when software installation does not complete correctly and leaves around partial state in the system directories and registry hives.

The registry is accessible to the Win32 programmer. There are calls to create and delete keys, look up values within keys, and more. Some of the more useful ones are listed in Fig. 11-12.

Win32 API function	Description
RegCreateKeyEx	Create a new registry key
RegDeleteKey	Delete a registry key
RegOpenKeyEx	Open a key to get a handle to it
RegEnumKeyEx	Enumerate the subkeys subordinate to the key of the handle
RegQueryValueEx	Look up the data for a value within a key

Figure 11-12. Some of the Win32 API calls for using the registry

When the system is turned off, most of the registry information is stored on the disk in the hives. Because their integrity is so critical to correct system functioning, backups are made automatically and metadata writes are flushed to disk to prevent corruption in the event of a system crash. Loss of the registry requires reinstalling *all* software on the system.

11.3 SYSTEM STRUCTURE

In the previous sections we examined Windows Vista as seen by the programmer writing code for user mode. Now we are going to look under the hood to see how the system is organized internally, what the various components do,

There are other drivers that also implement such arrangements, which Windows calls **mini-ports**. The shared functionality is in a **class driver**. For example, common functionality for SCSI or IDE disks or USB devices is supplied by a class driver, which mini-port drivers for each particular type of such devices link to as a library.

We will not discuss any particular device driver in this chapter, but will provide more detail about how the I/O manager interacts with device drivers in Sec. 11.7.

11.3.2 Booting Windows Vista

Getting an operating system to run requires several steps. When a computer is turned on, the CPU is initialized by the hardware, and then set to start executing a program in memory. But the only available code is in some form of nonvolatile CMOS memory that is initialized by the computer manufacturer (and sometimes updated by the user, in a process called **flashing**). On most PC's this initial program is the BIOS (Basic Input/Output System) which knows how to talk to the standard types of devices found on a PC. The BIOS brings up Windows Vista by first loading small bootstrap programs found at the beginning of the disk drive partitions.

The bootstrap programs know how to read enough information off a file system volume to find the standalone Windows *BootMgr* program in the root directory. *BootMgr* determines if the system had previously been hibernated or was in stand-by mode (special power-saving modes that allow the system to turn back on without booting). If so, *BootMgr* loads and executes *WinResume.exe*. Otherwise it loads and executes *WinLoad.exe* to perform a fresh boot. *WinLoad* loads the boot components of the system into memory: the kernel/executive (normally *ntoskrnl.exe*), the HAL (*hal.dll*), the file containing the SYSTEM hive, the *Win32k.sys* driver containing the kernel-mode parts of the Win32 subsystem, as well as images of any other drivers that are listed in the SYSTEM hive as **boot drivers**—meaning they are needed when the system first boots.

Once the Windows boot components are loaded into memory, control is given to low-level code in NTOS which proceeds to initialize the HAL, kernel and executive layers, link in the driver images, and access/update configuration data in the SYSTEM hive. After all the kernel-mode components are initialized, the first user-mode process is created using for running the *smss.exe* program (which is like */etc/init* in UNIX systems).

The Windows boot programs have logic to deal with common problems users encounter when booting the system fails. Sometimes installation of a bad device driver, or running a program like *regedit* (which can corrupt the SYSTEM hive), will prevent the system from booting normally. There is support for ignoring recent changes and booting to the *last known good* configuration of the system. Other boot options include **safe-boot** which turns off many optional drivers and

Windows Vista makes significant use of user-mode service processes to extend the functionality of the system. Some of these services are strongly tied to the operation of kernel-mode components, such as *lsass.exe* which is the local security authentication service which manages the token objects that represent user-identity, as well as managing encryption keys used by the file system. The user-mode plug-and-play manager is responsible for determining the correct driver to use when a new hardware device is encountered, installing it, and telling the kernel to load it. Many facilities provided by third parties, such as antivirus and digital rights management, are implemented as a combination of kernel-mode drivers and user-mode services.

In Windows Vista *taskmgr.exe* has a tab which identifies the services running on the system. (Earlier versions of Windows will show a list of services with the *net start* command). Multiple services can be seen to be running in the same process (*svchost.exe*). Windows does this for many of its own boot-time services to reduce the time needed to start up the system. Services can be combined into the same process as long as they can safely operate with the same security credentials.

Within each of the shared service processes, individual services are loaded as DLLs. They normally share a pool of threads using the Win32 threadpool facility, so that only the minimal number of threads needs to be running across all the resident services.

Services are common sources of security vulnerabilities in the system because they are often accessible remotely (depending on the TCP/IP firewall and IP Security settings), and not all programmers who write services are as careful as they should be to validate the parameters and buffers that are passed in via RPC.

The number of services running constantly in Windows is staggering. Yet few of those services ever receive a single request, though if they do it is likely to be from an attacker attempting to exploit a vulnerability. As a result more and more services in Windows are turned off by default, particularly on versions of Windows Server.

11.4 PROCESSES AND THREADS IN WINDOWS VISTA

Windows has a number of concepts for managing the CPU and grouping resources together. In the following sections we will examine these, discussing some of the relevant Win32 API calls, and show how they are implemented.

11.4.1 Fundamental Concepts

In Windows Vista processes are containers for programs. They hold the virtual address space, the handles that refer to kernel-mode objects, and threads. In their role as a container for threads they hold common resources used for thread

11.7.3 Implementation of I/O

The Windows I/O system consists of the plug-and-play services, the power manager, the I/O manager, and the device driver model. Plug-and-play detects changes in hardware configuration and builds or tears down the device stacks for each device, as well as causing the loading and unloading of device drivers. The power manager adjusts the power state of the I/O devices to reduce system power consumption when devices are not in use. The I/O manager provides support for manipulating I/O kernel objects, and IRP-based operations like `IoCallDrivers` and `IoCompleteRequest`. But most of the work required to support Windows I/O is implemented by the device drivers themselves.

Device Drivers

To make sure that device drivers work well with the rest of Windows Vista, Microsoft has defined the **WDM (Windows Driver Model)** that device drivers are expected to conform with. The WDM was designed to work with both Windows 98 and NT-based Windows beginning with Windows 2000, allowing carefully written drivers to be compatible with both systems. There is a development kit (the Windows Driver Kit) that is designed to help driver writers produce conformant drivers. Most Windows drivers start out by copying an appropriate sample driver and modifying it.

Microsoft also provides a **driver verifier** which validates many of the actions of drivers to be sure that they conform to the WDM requirements for the structure and protocols for I/O requests, memory management, and so on. The verifier ships with the system, and administrators can control it by running *verifier.exe*, which allows them to configure which drivers are to be checked and how extensive (i.e., expensive) the checks should be.

Even with all the support for driver development and verification, it is still very difficult to write even simple drivers in Windows, so Microsoft has built a system of wrappers called the **WDF (Windows Driver Foundation)** that runs on top of WDM and simplifies many of the more common requirements, mostly related to correct interaction with power management and plug-and-play operations.

To further simplify driver writing, as well as increase the robustness of the system, WDF includes the **UMDF (User-Mode Driver Framework)** for writing drivers as services that execute in processes. And there is the **KMDF (Kernel-Mode Driver Framework)** for writing drivers as services that execute in the kernel, but with many of the details of WDM made automagical. Since underneath it is the WDM that provides the driver model, that is what we will focus on in this section.

Devices in Windows are represented by device objects. Device objects are also used to represent hardware, such as buses, as well as software abstractions like file systems, network protocol engines, and kernel extensions, like antivirus

filter drivers. All these are organized by producing what Windows calls a *device stack*, as was previously shown in Fig. 11-16.

I/O operations are initiated by the I/O manager calling an executive API `IoCallDriver` with pointers to the top device object and to the IRP representing the I/O request. This routine finds the driver object associated with the device object. The operation types that are specified in the IRP generally correspond to the I/O manager system calls described above, such as `CREATE`, `READ`, and `CLOSE`.

Fig. 11-38 shows the relationships for a single level of the device stack. For each of these operations a driver must specify an entry point. `IoCallDriver` takes the operation type out of the IRP, uses the device object at the current level of the device stack to find the driver object, and indexes into the driver dispatch table with the operation type to find the corresponding entry point into the driver. The driver is then called and passed the device object and the IRP.

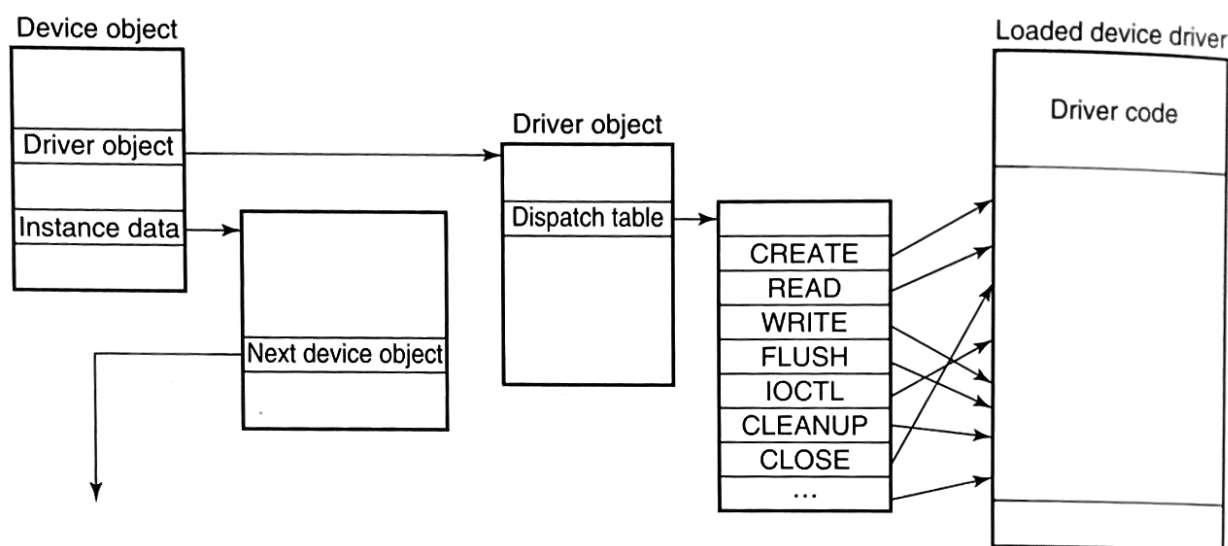


Figure 11-38. A single level in a device stack.

Once a driver has finished processing the request represented by the IRP, it has three options. It can call `IoCallDriver` again, passing the IRP and the next device object in the device stack. It can declare the I/O request to be completed and return to its caller. Or it can queue the IRP internally and return to its caller, having declared that the I/O request is still pending. This latter case results in an asynchronous I/O operation, at least if all the drivers above in the stack agree and also return to their callers.

I/O Request Packets

Fig. 11-39 shows the major fields in the IRP. The bottom of the IRP is a dynamically sized array containing fields that can be used by each driver for the device stack handling the request. These *stack* fields also allow a driver to specify