19CS2106S
Lecture Notes
Operating Systems Design Approaches
Session – 2

List of Topics	Learning Outcomes:	Questions answered in this Lecture notes:
1. Operating-System Structure: Monolithic, Layered Approach, Microkernels, Modules, Hybrid Systems. 2. Kernel Architecture for Traditional UNIX Systems, SVR4, BSD, Solaris 11, Linux	1. Understand operating systems design approaches. **2.** Understand UNIX kernel Architectures of Traditional UNIX Systems, SVR4, BSD, Solaris 11, Linux	1. Explain various Operating-System Structures 2. Differentiate monolithic and micro kernel? 3. Understand UNIX Family Tree 4. Explain Linux: Modular Structure and Kernel Components 5. Illustrate kernel design for SVR4, BSD, Solaris 11, Linux

References:

1. Silberschatz, A., Galvin, P.B. and Gagne, G., Operating system concepts essentials. 10th edition. Chapter: 2.8, pages: 81 – 91
2. Stallings, W. and Manna, M.M. Operating systems: internals and design principles. 9th Edition. Chapter: 2.8, 2.9, 2.10. pages: 108 – 118

2.1. Operating-System Design and Implementation

2.1.1 Design Goals

- **Requirements** define properties which the finished system must have, and are a necessary first step in designing any large complex system.
  - **User requirements** are features that users care about and understand, and are written in commonly understood vernacular. They generally do not include any implementation details, and are written similar to the product description one might find on a sales brochure or the outside of a shrink-wrapped box.
  - **System requirements** are written for the developers, and include more details about implementation specifics, performance requirements, compatibility constraints, standards compliance, etc. These requirements serve as a "contract" between the customer and the developers, (and between developers and subcontractors), and can get quite detailed.
- Requirements for operating systems can vary greatly depending on the planned scope and usage of the system. (Single user / multi-user, specialized system / general purpose, high/low security, performance needs, operating environment, etc.)

2.1.2 Mechanisms and Policies

- Policies determine *what* is to be done. Mechanisms determine *how* it is to be implemented.
- If properly separated and implemented, policy changes can be easily adjusted without re-writing the code, just by adjusting parameters or possibly loading new data / configuration files. For example the relative priority of background versus foreground tasks.

2.1.3 Implementation

- Traditionally OSes were written in assembly language. This provided direct control over hardware-related issues, but inextricably tied a particular OS to a particular HW platform.
- Recent advances in compiler efficiencies mean that most modern OSes are written in C, or more recently, C++. Critical sections of code are still written in assembly language, (

or written in C, compiled to assembly, and then fine-tuned and optimized by hand from there.)

- Operating systems may be developed using **emulators** of the target hardware, particularly if the real hardware is unavailable (e.g. not built yet), or not a suitable platform for development, (e.g. smart phones, game consoles, or other similar devices.)

2.2. Operating-System Structure

For efficient performance and implementation an OS should be partitioned into separate subsystems, each with carefully defined tasks, inputs, outputs, and performance characteristics. These subsystems can then be arranged in various architectural configurations:

2.2.1 Simple Structure

When DOS was originally written its developers had no idea how big and important it would eventually become. It was written by a few programmers in a relatively short amount of time, without the benefit of modern software engineering techniques, and then gradually grew over time to exceed its original expectations. It does not break the system into subsystems, and has no distinction between user and kernel modes, allowing all programs direct access to the underlying hardware. (Note that user versus kernel mode was not supported by the 8088 chip set anyway, so that really wasn't an option back then.)

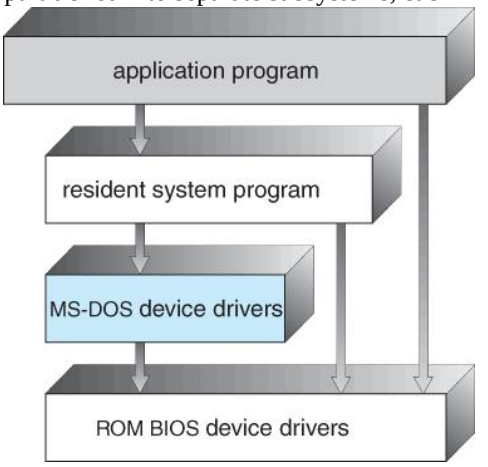

Figure - MS-DOS layer structure

The original UNIX OS used a simple layered approach, but almost all the OS was in one big layer, not really breaking the OS down into layered subsystems:

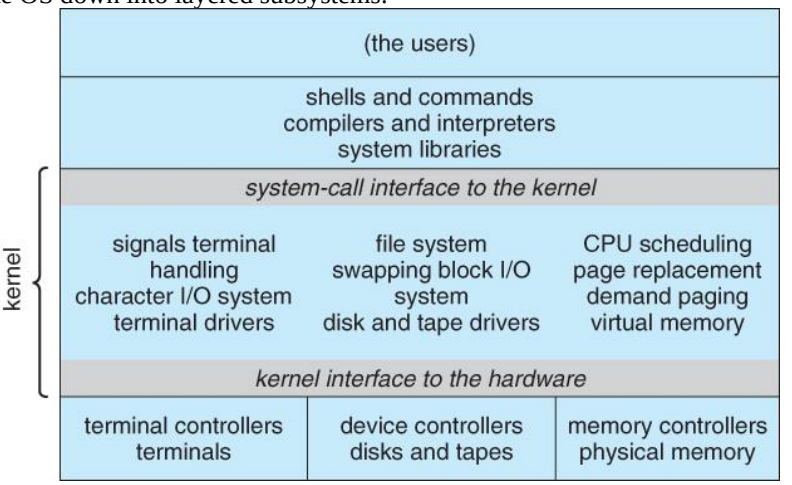

Figure - Traditional UNIX system structure

2.2.2 Layered Approach

- Another approach is to break the OS into a number of smaller layers, each of which rests on the layer below it, and relies solely on the services provided by the next lower layer.
- This approach allows each layer to be developed and debugged independently, with the assumption that all lower layers have already been debugged and are trusted to deliver proper services.
- The problem is deciding what order in which to place the layers, as no layer can call upon the services of any higher layer, and so many chicken-and-egg situations may arise.
- Layered approaches can also be less efficient, as a request for service from a higher layer has to filter through all lower layers before it reaches the HW, possibly with significant processing at each step.

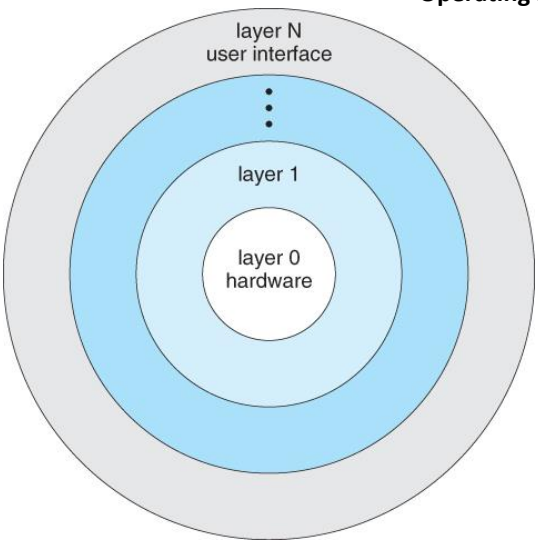

Figure - A layered operating system

2.2.3 Microkernels

- The basic idea behind micro kernels is to remove all non-essential services from the kernel, and implement them as system applications instead, thereby making the kernel as small and efficient as possible.
- Most microkernels provide basic process and memory management, and message passing between other services, and not much more.
- Security and protection can be enhanced, as most services are performed in user mode, not kernel mode.
- System expansion can also be easier, because it only involves adding more system applications, not rebuilding a new kernel.
- Mach was the first and most widely known microkernel, and now forms a major component of Mac OSX.
- Windows NT was originally microkernel, but suffered from performance problems relative to Windows 95. NT 4.0 improved performance by moving more services into the kernel, and now XP is back to being more monolithic.
- Another microkernel example is QNX, a real-time OS for embedded systems.

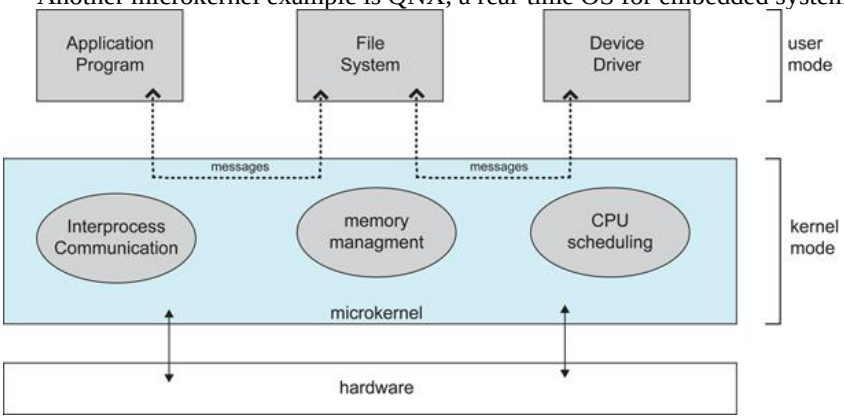

Figure - Architecture of a typical microkernel

2.2.4 Modules

- Modern OS development is object-oriented, with a relatively small core kernel and a set of ***modules*** which can be linked in dynamically. See for example the Solaris structure, as shown in Figure below.
- Modules are similar to layers in that each subsystem has clearly defined tasks and interfaces, but any module is free to contact any other module, eliminating the problems of going through multiple intermediary layers, as well as the chicken-and-egg problems.

- The kernel is relatively small in this architecture, similar to microkernels, but the kernel does not have to implement message passing since modules are free to contact each other directly.

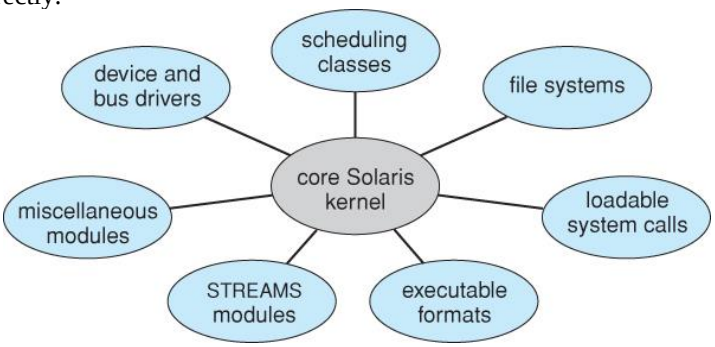

Figure - Solaris loadable modules

2.2.5 Hybrid Systems

- Most OSes today do not strictly adhere to one architecture, but are hybrids of several.

2.2.5.1 Mac OS X

- The Max OSX architecture relies on the Mach microkernel for basic system management services, and the BSD kernel for additional services. Application services and dynamically loadable modules (kernel extensions) provide the rest of the OS functionality:

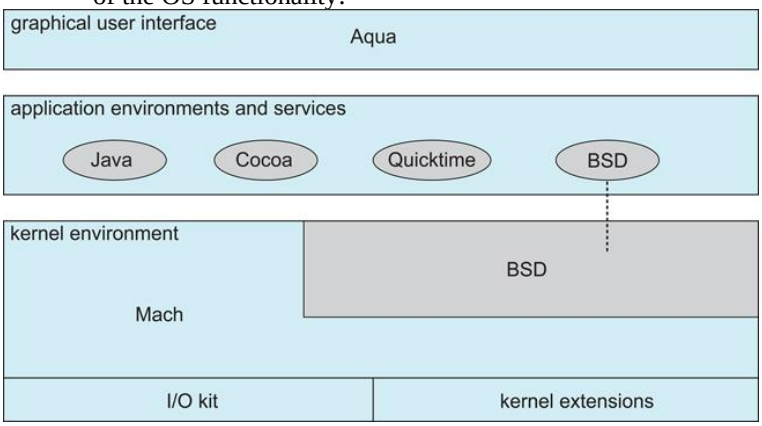

Figure - The Mac OS X structure

2.2.5.2 iOS

- The **iOS** operating system was developed by Apple for iPhones and iPads. It runs with less memory and computing power needs than Max OS X, and supports touchscreen interface and graphics for small screens:

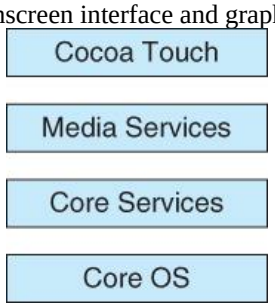

Figure - Architecture of Apple's iOS.

2.2.5.3 Android

- The Android OS was developed for Android smartphones and tablets by the Open Handset Alliance, primarily Google.
- Android is an open-source OS, as opposed to iOS, which has lead to its popularity.

- Android includes versions of Linux and a Java virtual machine both optimized for small platforms.
- Android apps are developed using a special Java-for-Android development environment.

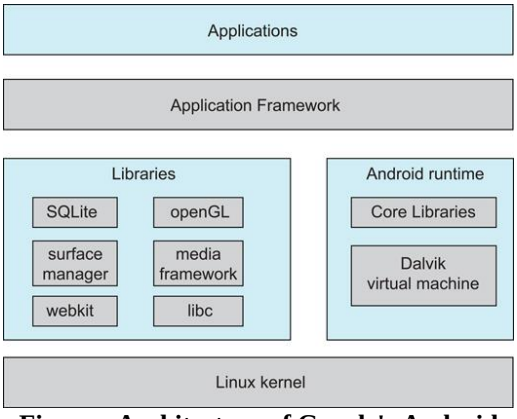

Figure - Architecture of Google's Android

2.3 TRADITIONAL UNIX SYSTEMS

History

UNIX was initially developed at Bell Labs and became operational on a PDP-7 in 1970. Work on UNIX at Bell Labs, and later elsewhere, produced a series of versions of UNIX. The first notable milestone was porting the UNIX system from the PDP-7 to the PDP-11. This was the first hint that UNIX would be an OS for all computers. The next important milestone was the rewriting of UNIX in the programming language C. This was an unheard-of strategy at the time. It was generally felt that something as complex as an OS, which must deal with time-critical events, had to be written exclusively in assembly language. Reasons for this attitude include the following:

- Memory (both RAM and secondary store) was small and expensive by today's standards, so effective use was important. This included various techniques for overlaying memory with different code and data segments, and self-modifying code.
- Even though compilers had been available since the 1950s, the computer industry was generally skeptical of the quality of automatically generated code. With resource capacity small, efficient code, both in terms of time and space, was essential.
- Processor and bus speeds were relatively slow, so saving clock cycles could make a substantial difference in execution time.

The C implementation demonstrated the advantages of using a high-level language for most if not all of the system code. Today, virtually all UNIX implementations are written in C.

These early versions of UNIX were popular within Bell Labs. In 1974, the UNIX system was described in a technical journal for the first time [RITC74]. This spurred great interest in the system. Licenses for UNIX were provided to commercial institutions as well as universities. The first widely available version outside Bell Labs was Version 6, in 1976. The follow-on Version 7, released in 1978, is the ancestor of most modern UNIX systems. The most important of the non-AT&T systems to be developed was done at the University of California at Berkeley, called UNIX BSD (Berkeley Software Distribution), running first on PDP and then on VAX computers. AT&T continued to develop and refine the system. By 1982, Bell Labs had combined several AT&T variants of UNIX into a single system, marketed commercially as UNIX System III. A number of features was later added to the OS to produce UNIX System V.

Description

The classic UNIX architecture can be pictured as in three levels: hardware, kernel, and user. The OS is often called the system kernel, or simply the kernel, to emphasize its isolation from the user and applications. It interacts directly with the hardware. It is the UNIX kernel that we will be concerned with in our use of UNIX as an example in this book. UNIX also comes equipped with a number of user services and interfaces that are considered part of the system. These can be grouped into the shell, which supports system calls from applications, other interface software, and the components of the C compiler (compiler, assembler, loader). The level above this consists of user applications and the user interface to the C compiler.

A look at the kernel is provided in Figure 2.15. User programs can invoke OS services either directly, or through library programs. The system call interface is the boundary with the user and allows higher-level software to gain access to specific kernel functions. At the other end, the OS contains primitive routines that interact directly with the hardware. Between these two interfaces, the system is divided into two main parts: one concerned with process control, and the other concerned with file management and I/O. The process control subsystem is responsible for memory management, the scheduling and dispatching of processes, and the synchronization and interprocess communication of processes. The file system exchanges data between memory and external devices either as a stream of characters or in blocks. To achieve this, a variety of device drivers are used. For block-oriented transfers, a disk cache approach is used: A system buffer in main memory is interposed between the user address space and the external device.

The description in this subsection has dealt with what might be termed *traditional UNIX systems*; [VAHA96] uses this term to refer to System V Release 3 (SVR3), 4.3BSD, and earlier versions. The following general statements may be made about a traditional UNIX system. It is designed to run on a single processor, and lacks the ability to protect its data structures from concurrent access by multiple processors. Its kernel is not very versatile, supporting a single type of file system, process scheduling policy, and executable file format. The traditional UNIX kernel is not designed to be extensible and has few facilities for code reuse. The result is that, as new features were added to the various UNIX versions, much new code had to be added, yielding a bloated and unmodular kernel.

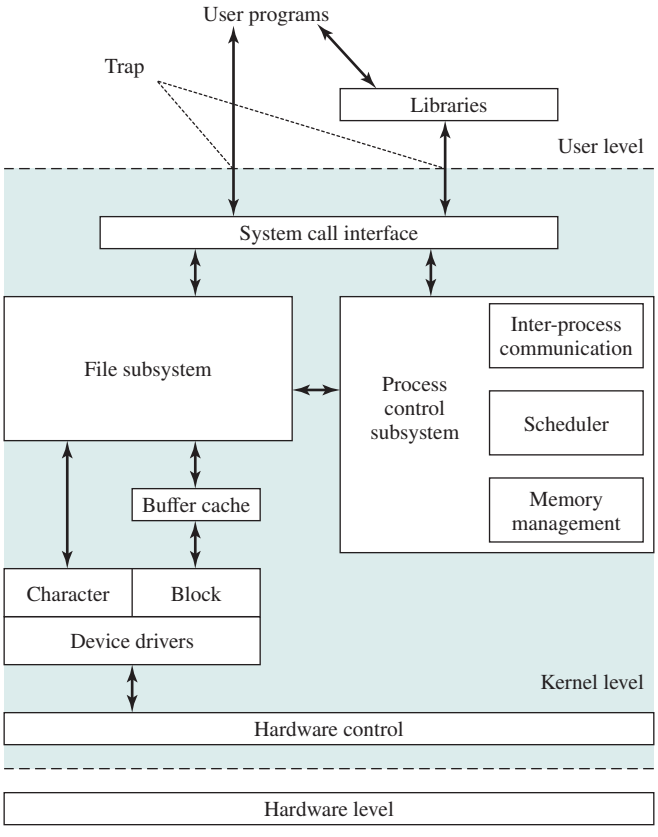

Figure 2.15 **Traditional UNIX Architecture**

2.4 MODERN UNIX SYSTEMS

As UNIX evolved, the number of different implementations proliferated, each providing some useful features. There was a need to produce a new implementation that unified many of the important innovations, added other modern OS design features, and produced a more modular architecture. Typical of the modern UNIX kernel is the architecture depicted in Figure 2.16. There is a small core of facilities, written in a modular fashion, that provide functions and services needed by a number of OS processes. Each of the outer circles represents functions and an interface that may be implemented in a variety of ways.

We now turn to some examples of modern UNIX systems (see Figure 2.17).

System V Release 4 (SVR4)

SVR4, developed jointly by AT&T and Sun Microsystems, combines features from SVR3, 4.3BSD, Microsoft Xenix System V, and SunOS. It was almost a total rewrite of the System V kernel and produced a clean, if complex, implementation. New features in the release include real-time processing support, process scheduling classes,

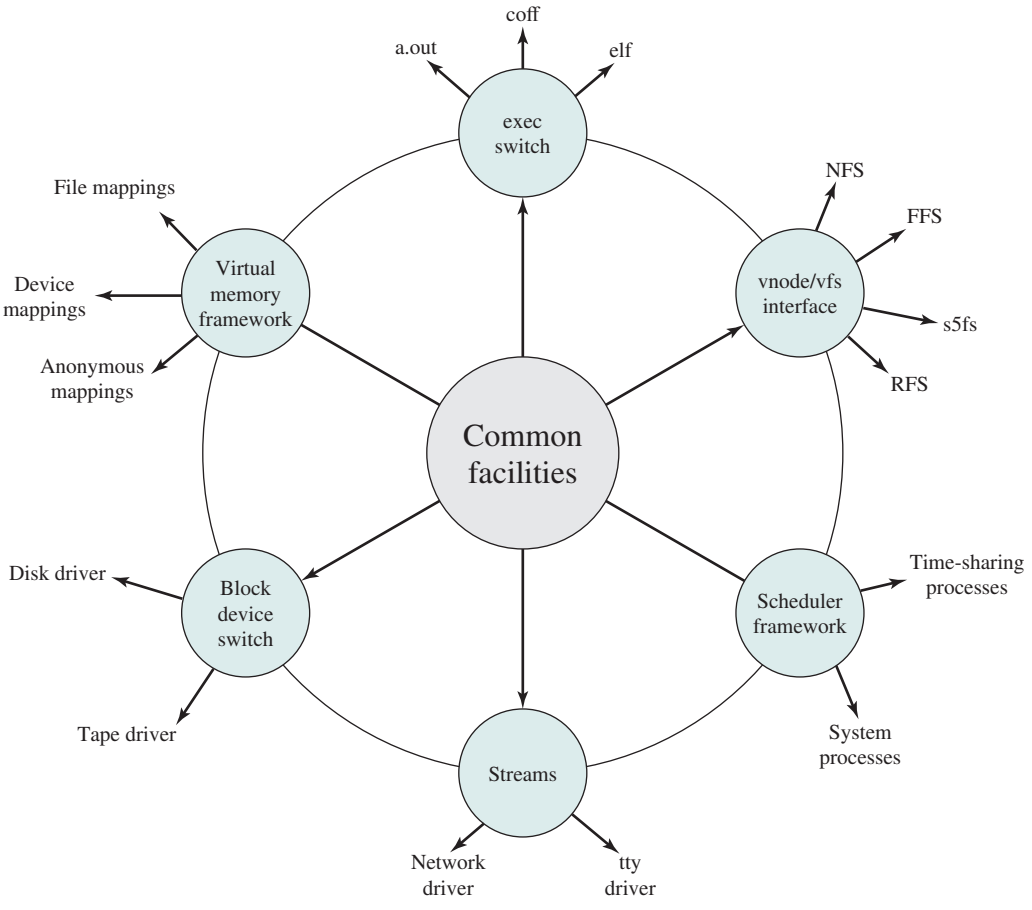

Figure 2.16 **Modern UNIX Kernel**

dynamically allocated data structures, virtual memory management, virtual file system, and a preemptive kernel.

SVR4 draws on the efforts of both commercial and academic designers, and was developed to provide a uniform platform for commercial UNIX deployment. It has succeeded in this objective and is perhaps the most important UNIX variant. It incorporates most of the important features ever developed on any UNIX system and does so in an integrated, commercially viable fashion. SVR4 runs on processors ranging from 32-bit microprocessors up to supercomputers.

BSD

The Berkeley Software Distribution (BSD) series of UNIX releases have played a key role in the development of OS design theory. 4.xBSD is widely used in academic installations and has served as the basis of a number of commercial UNIX products. It is probably safe to say that BSD is responsible for much of the popularity of UNIX, and that most enhancements to UNIX first appeared in BSD versions.

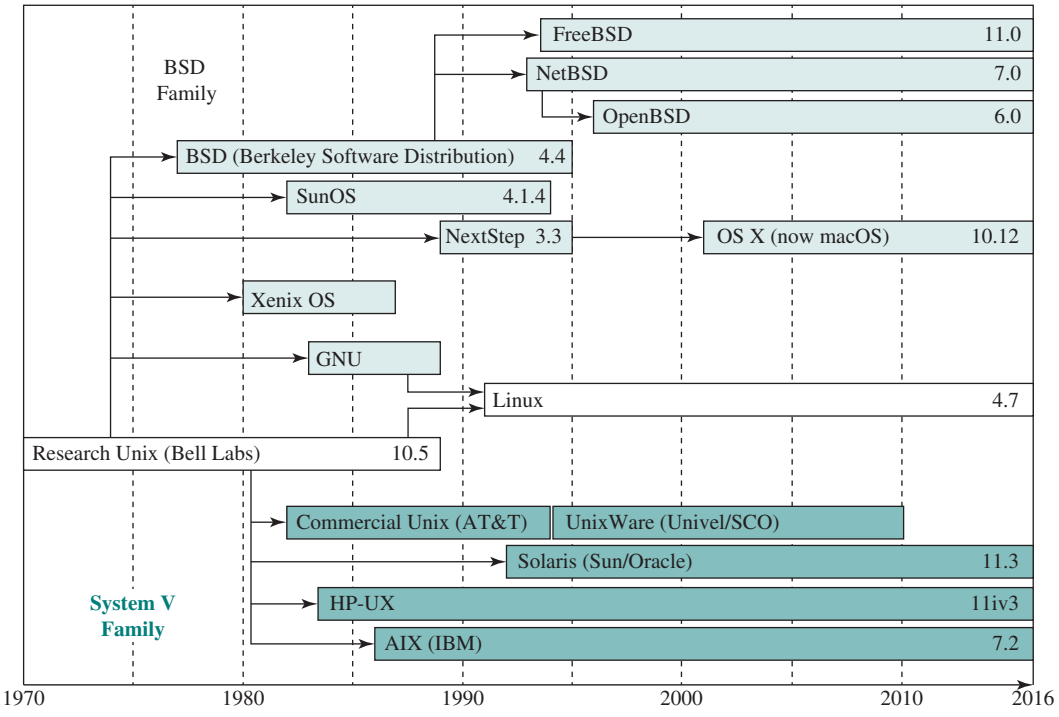

Figure 2.17 **UNIX Family Tree**

4.4BSD was the final version of BSD to be released by Berkeley, with the design and implementation organization subsequently dissolved. It is a major upgrade to 4.3BSD and includes a new virtual memory system, changes in the kernel structure, and a long list of other feature enhancements.

There are several widely used, open-source versions of BSD. FreeBSD is popular for Internet-based servers and firewalls and is used in a number of embedded systems. NetBSD is available for many platforms, including large-scale server systems, desktop systems, and handheld devices, and is often used in embedded systems. OpenBSD is an open-source OS that places special emphasis on security.

The latest version of the Macintosh OS, originally known as OS X and now called MacOS, is based on FreeBSD 5.0 and the Mach 3.0 microkernel.

Solaris 11

Solaris is Oracle's SVR4-based UNIX release, with the latest version being 11. Solaris provides all of the features of SVR4 plus a number of more advanced features, such as a fully preemptable, multithreaded kernel, full support for SMP, and an object-oriented interface to file systems. Solaris is one the most widely used and most successful commercial UNIX implementations.

2.5 LINUX

History

Linux started out as a UNIX variant for the IBM PC (Intel 80386) architecture. Linus Torvalds, a Finnish student of computer science, wrote the initial version. Torvalds posted an early version of Linux on the Internet in 1991. Since then, a number of people, collaborating over the Internet, have contributed to the development of Linux, all under the control of Torvalds. Because Linux is free and the source code is available, it became an early alternative to other UNIX workstations, such as those offered by Sun Microsystems and IBM. Today, Linux is a full-featured UNIX system that runs on virtually all platforms.

Key to the success of Linux has been the availability of free software packages under the auspices of the Free Software Foundation (FSF). FSF's goal is stable, platform-independent software that is free, high quality, and embraced by the user community. FSF's GNU project[2] provides tools for software developers, and the GNU Public License (GPL) is the FSF seal of approval. Torvalds used GNU tools in developing his kernel, which he then released under the GPL. Thus, the Linux distributions that you see today are the product of FSF's GNU project, Torvald's individual effort, and the efforts of many collaborators all over the world.

In addition to its use by many individual developers, Linux has now made significant penetration into the corporate world. This is not only because of the free software, but also because of the quality of the Linux kernel. Many talented developers have contributed to the current version, resulting in a technically impressive product. Moreover, Linux is highly modular and easily configured. This makes it easy to squeeze optimal performance from a variety of hardware platforms. Plus, with the source code available, vendors can tweak applications and utilities to meet specific requirements. There are also commercial companies such as Red Hat and Canonical, which provide highly professional and reliable support for their Linux-based distributions for long periods of time. Throughout this book, we will provide details of Linux kernel internals based on Linux kernel 4.7, released in 2016.

A large part of the success of the Linux Operating System is due to its development model. Code contributions are handled by one main mailing list, called LKML (Linux Kernel Mailing List). Apart from it, there are many other mailing lists, each dedicated to a Linux kernel subsystem (like the netdev mailing list for networking, the linux-pci for the PCI subsystem, the linux-acpi for the ACPI subsystem, and a great many more). The patches which are sent to these mailing lists should adhere to strict rules (primarily the Linux Kernel coding style conventions), and are reviewed by developers all over the world who are subscribed to these mailing lists. Anyone can send patches to these mailing lists; statistics (for example, those published in the lwn.net site from time to time) show that many patches are sent by developers from famous commercial companies like Intel, Red Hat, Google, Samsung, and others. Also, many maintainers are employees of commercial companies (like David

[2] GNU is a recursive acronym for *GNU's Not Unix*. The GNU project is a free software set of packages and tools for developing a UNIX-like operating system; it is often used with the Linux kernel.

Miller, the network maintainer, who works for Red Hat). Many times such patches are fixed according to feedback and discussions over the mailing list, and are resent and reviewed again. Eventually, the maintainer decides whether to accept or reject patches; and each subsystem maintainer from time to time sends a pull request of his tree to the main kernel tree, which is handled by Linus Torvalds. Linus himself releases a new kernel version in about every 7–10 weeks, and each such release has about 5–8 release candidates (RC) versions.

We should mention that it is interesting to try to understand why other open-source operating systems, such as various flavors of BSD or OpenSolaris, did not have the success and popularity which Linux has; there can be many reasons for that, and for sure, the openness of the development model of Linux contributed to its popularity and success. But this topic is out of the scope of this book.

Modular Structure

Most UNIX kernels are monolithic. Recall from earlier in this chapter, a monolithic kernel is one that includes virtually all of the OS functionality in one large block of code that runs as a single process with a single address space. All the functional components of the kernel have access to all of its internal data structures and routines. If changes are made to any portion of a typical monolithic OS, all the modules and routines must be relinked and reinstalled, and the system rebooted, before the changes can take effect. As a result, any modification, such as adding a new device driver or file system function, is difficult. This problem is especially acute for Linux, for which development is global and done by a loosely associated group of independent developers.

Although Linux does not use a microkernel approach, it achieves many of the potential advantages of this approach by means of its particular modular architecture. Linux is structured as a collection of modules, a number of which can be automatically loaded and unloaded on demand. These relatively independent blocks are referred to as **loadable modules** [GOYE99]. In essence, a module is an object file whose code can be linked to and unlinked from the kernel at runtime. Typically, a module implements some specific function, such as a file system, a device driver, or some other feature of the kernel's upper layer. A module does not execute as its own process or thread, although it can create kernel threads for various purposes as necessary. Rather, a module is executed in kernel mode on behalf of the current process.

Thus, although Linux may be considered monolithic, its modular structure overcomes some of the difficulties in developing and evolving the kernel. The Linux loadable modules have two important characteristics:

1. **Dynamic linking:** A kernel module can be loaded and linked into the kernel while the kernel is already in memory and executing. A module can also be unlinked and removed from memory at any time.
2. **Stackable modules:** The modules are arranged in a hierarchy. Individual modules serve as libraries when they are referenced by client modules higher up in the hierarchy, and as clients when they reference modules further down.

Dynamic linking facilitates configuration and saves kernel memory [FRAN97]. In Linux, a user program or user can explicitly load and unload kernel modules using the `insmod` or `modprobe` and `rmmod` commands. The kernel itself monitors the need

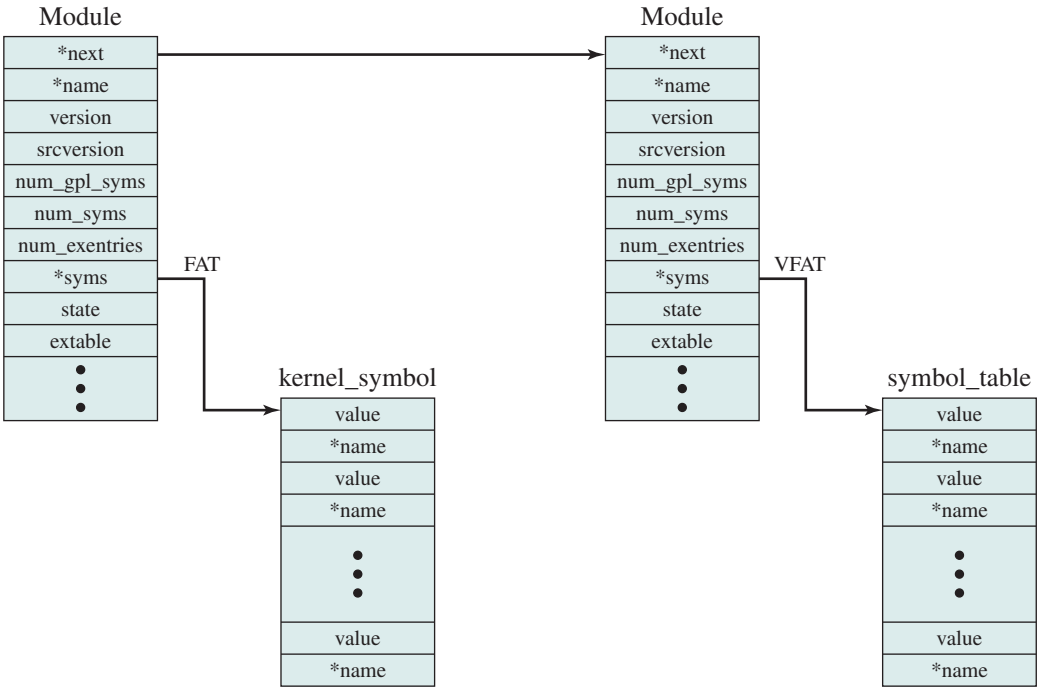

Figure 2.18 **Example List of Linux Kernel Modules**

for particular functions, and can load and unload modules as needed. With stackable modules, dependencies between modules can be defined. This has two benefits:

1. Code common to a set of similar modules (e.g., drivers for similar hardware) can be moved into a single module, reducing replication.
2. The kernel can make sure that needed modules are present, refraining from unloading a module on which other running modules depend, and loading any additional required modules when a new module is loaded.

Figure 2.18 is an example that illustrates the structures used by Linux to manage modules. The figure shows the list of kernel modules after only two modules have been loaded: FAT and VFAT. Each module is defined by two tables: the module table and the symbol table (kernel_symbol). The module table includes the following elements:

- ***name:** The module name
- **refcnt:** Module counter. The counter is incremented when an operation involving the module's functions is started and decremented when the operation terminates.
- **num_syms:** Number of exported symbols.
- ***syms:** Pointer to this module's symbol table.

The symbol table lists symbols that are defined in this module and used elsewhere.

Kernel Components

Figure 2.19, taken from [MOSB02], shows the main components of a typical Linux kernel implementation. The figure shows several processes running on top of the kernel. Each box indicates a separate process, while each squiggly line with an arrowhead represents a thread of execution. The kernel itself consists of an interacting collection of components, with arrows indicating the main interactions. The underlying hardware is also depicted as a set of components with arrows indicating which kernel components use or control which hardware components. All of the kernel components, of course, execute on the processor. For simplicity, these relationships are not shown.

Briefly, the principal kernel components are the following:

- **Signals:** The kernel uses signals to call into a process. For example, signals are used to notify a process of certain faults, such as division by zero. Table 2.6 gives a few examples of signals.
- **System calls:** The system call is the means by which a process requests a specific kernel service. There are several hundred system calls, which can be roughly grouped into six categories: file system, process, scheduling, interprocess communication, socket (networking), and miscellaneous. Table 2.7 defines a few examples in each category.
- **Processes and scheduler:** Creates, manages, and schedules processes.
- **Virtual memory:** Allocates and manages virtual memory for processes.

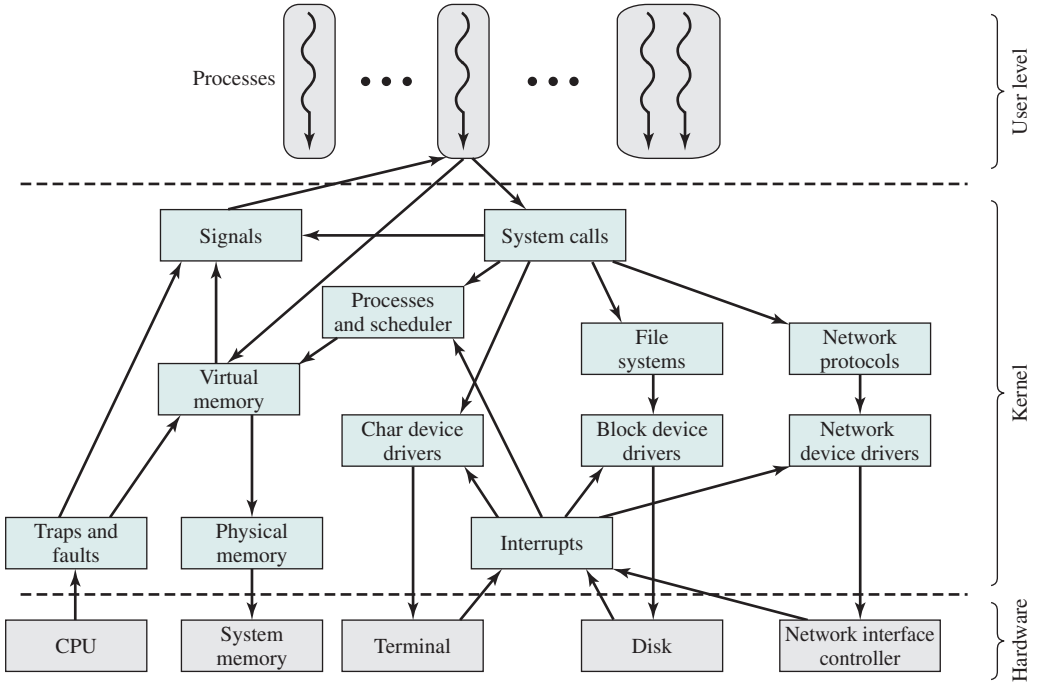

Figure 2.19 Linux Kernel Components

Table 2.6 Some Linux Signals

SIGHUP	Terminal hangup	SIGCONT	Continue
SIGQUIT	Keyboard quit	SIGTSTP	Keyboard stop
SIGTRAP	Trace trap	SIGTTOU	Terminal write
SIGBUS	Bus error	SIGXCPU	CPU limit exceeded
SIGKILL	Kill signal	SIGVTALRM	Virtual alarm clock
SIGSEGV	Segmentation violation	SIGWINCH	Window size unchanged
SIGPIPT	Broken pipe	SIGPWR	Power failure
SIGTERM	Termination	SIGRTMIN	First real-time signal
SIGCHLD	Child status unchanged	SIGRTMAX	Last real-time signal

- **File systems:** Provide a global, hierarchical namespace for files, directories, and other file-related objects and provide file system functions.
- **Network protocols:** Support the Sockets interface to users for the TCP/IP protocol suite.

Table 2.7 Some Linux System Calls

File System Related	
close	Close a file descriptor.
link	Make a new name for a file.
open	Open and possibly create a file or device.
read	Read from file descriptor.
write	Write to file descriptor.
Process Related	
execve	Execute program.
exit	Terminate the calling process.
getpid	Get process identification.
setuid	Set user identity of the current process.
ptrace	Provide a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers.
Scheduling Related	
sched_getparam	Set the scheduling parameters associated with the scheduling policy for the process identified by `pid`.
sched_get_priority_max	Return the maximum priority value that can be used with the scheduling algorithm identified by `policy`.
sched_setscheduler	Set both the scheduling policy (e.g., FIFO) and the associated parameters for the process `pid`.
sched_rr_get_interval	Write into the timespec structure pointed to by the parameter to the round-robin time quantum for the process `pid`.
sched_yield	A process can relinquish the processor voluntarily without blocking via this system call. The process will then be moved to the end of the queue for its static priority and a new process gets to run.

Table 2.7 (*Continued*)

Interprocess Communication (IPC) Related	
msgrcv	A message buffer structure is allocated to receive a message. The system call then reads a message from the message queue specified by `msqid` into the newly created message buffer.
semctl	Perform the control operation specified by `cmd` on the semaphore set `semid`.
semop	Perform operations on selected members of the semaphore set `semid`.
shmat	Attach the shared memory segment identified by `semid` to the data segment of the calling process.
shmctl	Allow the user to receive information on a shared memory segment; set the owner, group, and permissions of a shared memory segment; or destroy a segment.
Socket (networking) Related	
bind	Assign the local IP address and port for a socket. Return 0 for success or −1 for error.
connect	Establish a connection between the given socket and the remote socket associated with sockaddr.
gethostname	Return local host name.
send	Send the bytes contained in buffer pointed to by *msg over the given socket.
setsockopt	Set the options on a socket.
Miscellaneous	
fsync	Copy all in-core parts of a file to disk, and wait until the device reports that all parts are on stable storage.
time	Return the time in seconds since January 1, 1970.
vhangup	Simulate a hangup on the current terminal. This call arranges for other users to have a "clean" tty at login time.

- **Character device drivers:** Manage devices that require the kernel to send or receive data one byte at a time, such as terminals, modems, and printers.
- **Block device drivers:** Manage devices that read and write data in blocks, such as various forms of secondary memory (magnetic disks, CD-ROMs, etc.).
- **Network device drivers:** Manage network interface cards and communications ports that connect to network devices, such as bridges and routers.
- **Traps and faults:** Handle traps and faults generated by the processor, such as a memory fault.
- **Physical memory:** Manages the pool of page frames in real memory and allocates pages for virtual memory.
- **Interrupts** Handle interrupts from peripheral devices.