CPS221 Lecture: Operating System Structure; Virtual Machines

last revised 9/9/14

Objectives

1. To discuss various ways of structuring the operating system proper
2. To discuss virtual machines

Materials:

1. Projectable of layered kernel architecture
2. Projectable of microkernel architecture
3. Projectable of place of virtual machine structure
4. Ability to show Mac Utillities, System Preferences
5. Ability to demonstrate Sheepshaver (use Letter Learner)
6. Ability to demonstrate VMWare Fusion on Mac

I. Operating System Structure.

A. We now turn from talking about what an operating system <u>does</u> to <u>how</u> it is implemented. (We will focus on the operating system proper - not the libraries and system programs.)

B. The design of an operating system is a major task.

1. Historically, the complexity of OS design was a motivating factor in the development of software engineering - specifically OS/360. (See Frederick Brooks The Mythical Man Month.)

2. The complexity arises for two reasons

a) Some of the services that the kernel must provide are inherently complex - e.g. those pertaining to managing directory structures, file permissions, etc.

b) There are a lot of dependencies between components

C. There are several key principles that are important in operating system design.

1. The distinction between MECHANISMS and POLICIES.

   a) A policy is a specification as to what is to be done.

      (1) Example: round-robin is a scheduling policy we discussed when we looked at timesharing systems.

      (2) Example: large multiuser systems may specify the hours of the day when certain users can log in to the system, as a security policy.

   b) A mechanism is a facility that actually implements a policy. Mechanisms may support multiple policies, allowing different policies to be chosen in different situations.

      Example: show options in Security and File Sharing panes of System Preferences.

2. Portability.

   a) Since an operating system represents a major software development effort, it is desirable to make it easy to port it to different hardware platform.

   b) The need for portability arises when hardware evolves

      e.g. Windows was first developed for the Intel 80386 platform, which has since evolved into 32 bit Pentium and now 64-bit platforms. Since no one makes 80386 based systems anymore, Microsoft would be out of business if they did not port Windows to these new versions of the Intel CPU architecture!

   c) Being able to port an OS to a different platform is also desirable

(1) e.g. Linux runs on a number of different platforms.

(2) e.g. MacOSX was first developed on the Motorola 68000 platform. When Apple decided to move to the IA32 and later IA64 platforms, OS X was ported, rather than being rewritten from scratch.

d) Portability is enhanced by confining the platform-specific portions of the code to a small part of the overall code, rather than spreading it out everywhere.

3. Extensibility is also important - e.g. PC operating systems have had to be extended in recent years to support 64 bit and multicore CPUs.

II. **Structuring the Core of an Operating System**

A. Broadly speaking, there are four ways to approach the structuring the core of an operating system.

B. Monolithic structure - all the functionality resides in a single large module.

1. This is the way operating systems were first structured, but is very problematic from the standpoint of code correctness, since a code error anywhere can result in a system crash.

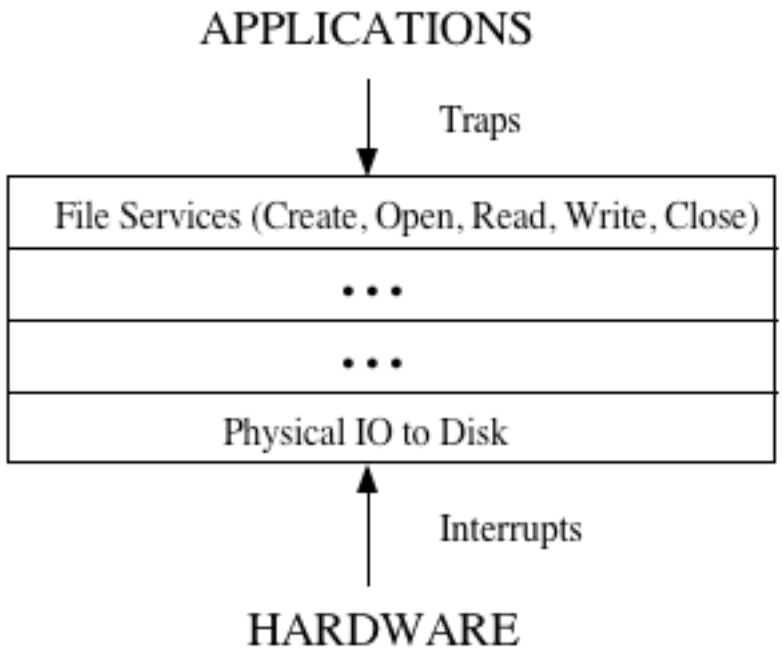
2. Monolithic operating systems were also hard to port or extend, because code that needs to be changed tends to be scattered throughout the system, rather than being localized in one place.

3. Modern operating systems for full-blown computers are not structured this way - though a simple real time system may be.

C. Layered structure.

1. A layered system consists of a series of layers, each of which depends only on the correct operation of the layer immediately beneath it. The lowest layer represents the hardware interface, and the highest layer the application interface. The operating system structure might end up looking like this:

   PROJECT



2. A key problem in a layered system is deciding on the ordering of the layers. (This is critical because of the requirement that each layer can ultimately only use services provided by layers below it - so the ordering cannot have any cycles.)

3. In a strictly-layered structure, efficiency can also become a problem because when a higher-level layer requires a lower-level operation the request must work its way down layer by layer.

D. Kernel Based Structures

1. In a kernel-based structure, the operating system proper consists of a kernel (which runs in kernel mode) and non-kernel portions (which run in user mode.)

2. For the sake of portability, platform-specific portions of the code are confined to the kernel.

3. Many architecture-independent operating system services are provided by code that runs in user mode.

4. Of course, deciding what belongs in the kernel is challenging.

   a) Operating system code that is not part of the kernel runs in user mode. Therefore, if it needs any kernel-mode operation done, it must invoke a system call, which has significant overhead.

   b) But minimizing the size of the kernel is important for the sake of correctness and security..

      (1) With good protection mechanisms in place, erroneous user code can only mess up its own process, since it cannot access memory belonging to the kernel or to other processes.

      (2) But erroneous kernel code can damage memory belonging to other processes and/or cause the whole system to crash.

      Example: Years ago, one our senior projects involved some modifications to the terminal driver software in the kernel of a Unix system we were using. Due to an error in the project code, any time a user typed a line with more than 127 characters in it, the system crashed!

E. Modular kernel

1. This is a variant of the kernel-based structure, so the issues discussed above still apply.

2. The kernel consists of a collection of independent modules, which are dynamically loaded as they are needed. When a modification must be made to the operating system, the appropriate module is changed and then is loaded the next time the system is booted.

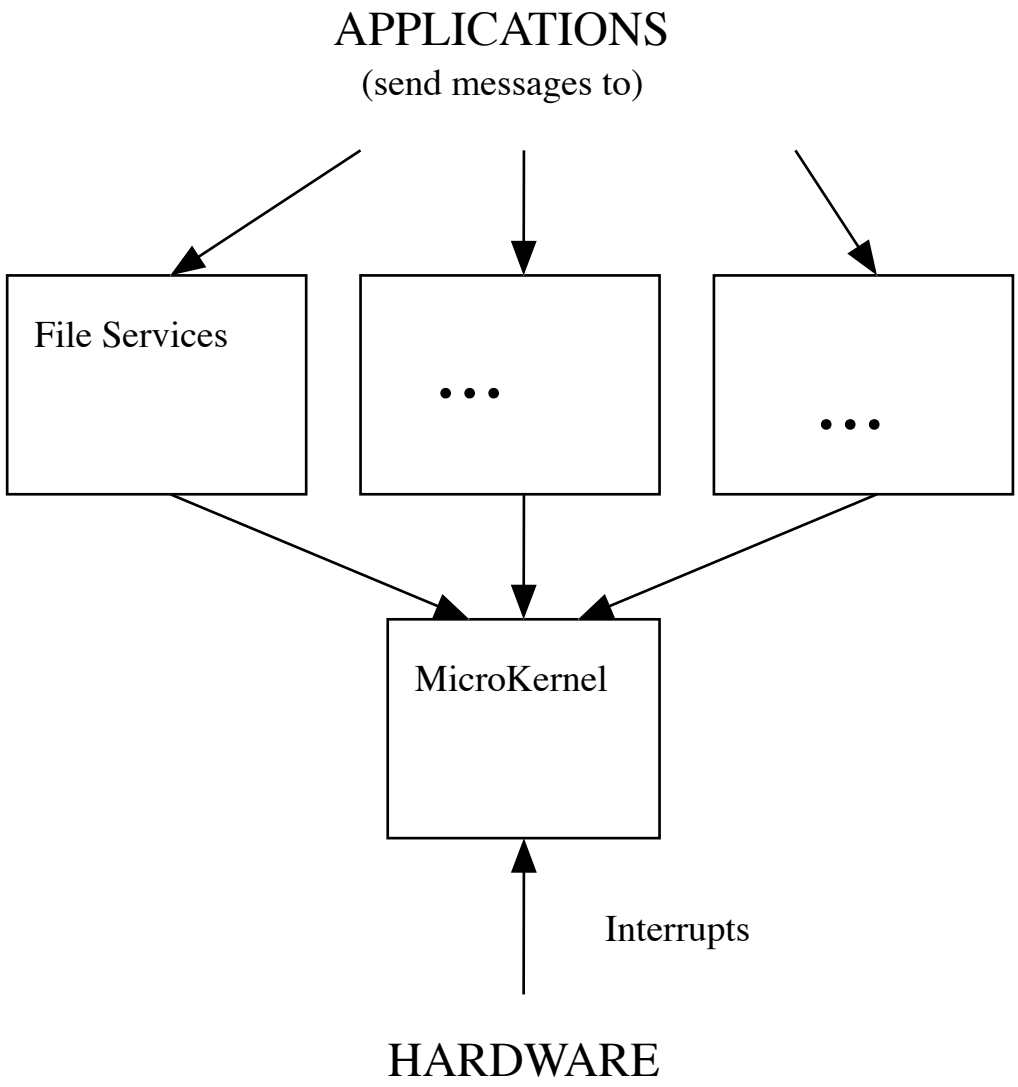
3. The operation of the resultant structure resembles a kernel-based structure, but this structure is much simpler to modify since modules can be changed independently.

4. Linux distributions are typically structured this way.

F. Micro-kernel architecture. In this approach, the kernel is made as small as possible - typically only including process management, basic memory management, IO interrupt handling, and some form of inter-process communication. All other functions are provided by code that runs in user mode in special processes - e.g. a process than manages the file system.

PROJECT diagram on next page

1. Most operating system functions are requested by means of a message from a user process to an appropriate system process, rather than being directly requested from the kernel. (Of course, this requires the use of interprocess communication facilities provided by the kernel.)
2. By minimizing the size of the kernel, this reduces risks arising from incorrect code, as well as security dangers.
3. This structure can be less efficient than the others due to the amount of overhead involved in using messages to other processes for requesting most system services.
4. MacOSX is structured this way - using a kernel called Darwin (which is actually open source). The Mac specific "look and feel"things reside in user-mode code that is not open source, of course.
5.



6.

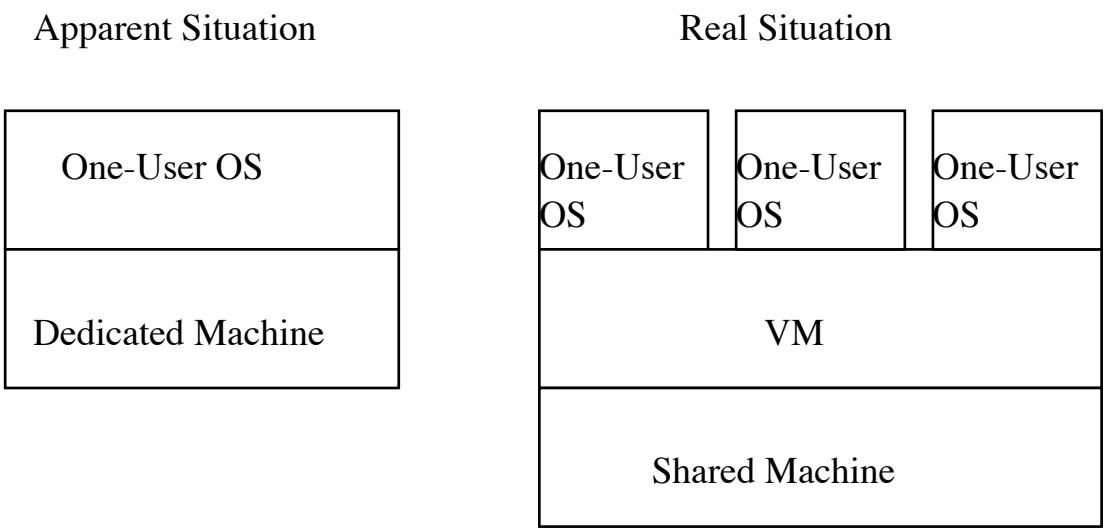
III. **A concept that has appeared repeatedly in the history of operating systems: the notion of a virtual machine.**

A. The virtual machine idea first appeared in 1972 in an IBM operating system known as VM running on mainframe computers.

1. Ordinarily, in such an environment, the individual user was quite conscious of sharing a particular machine with a number of other users. For example, the user would have to specifically request certain system resources when they were needed.

2. In VM, a high-level layer was provided that altered the users view of the underlying system. This layer made the underlying machine and software layers APPEAR to be a one-user system running a one-user operating system, whose commands and system-services might be different from those of the underlying OS - though the resultant virtual machine still appeared to have the same HARDWARE instruction set. That is, the virtual machine looked like a small member of the same family as the actual processor in use, but having its own dedicated resources such as a printer.

   PROJECT



B. This same idea has shown up in numerous places over the years, because it provides

   1. A simpler and/or more flexible interface for the user. (If a given user must use several different machines, it is nice if they can be all made to appear the same.)

   2. Greater portability of software. Software written for one OS can be run under another OS if the first OS can be run in a virtual machine on the other system.

      Example: Until recently, Macintosh OSX provided a facility known as :"Classic" which could run applications written for a

prior version of Mac OS in an OS X process that ran a copy of OS 9 in a virtual machine.

The latest releases of OS X no longer support this - and it has never been supported on Intel-based platforms, though there is an open source product known as Sheepshaver that allows essentially the same thing to be done.

Demo: Start up Sheepshaver, run Letter Learner.

3. One interesting use of this concept is in the development of new operating systems or operating system versions. A prototype system may be run as a virtual machine on an existing system, allowing the regular use of that system to continue while the new OS is being debugged.

C. The most familiar implementation of this concept at this point in time is in products such as VMWare and VirtualBox.

1. Systems like this are available for Linux, Windows, and Mac OSX, and runs on top of the appropriate operating system, referred to as the "Host" OS; and there are also stand-alone versions that run directly on the hardware.

2. Systems like this can run one or more "Guest" OS's.

   a) The Guest OS runs in user mode in a host OS process.

   b) When the guest OS executes a privileged instruction, VMWare intercepts the illegal operation trap and simulates it using the facilities of the host OS.

   c) The result is that the guest OS behaves as if it were running on its own hardware, but actually runs on simulated hardware provided by the virtual machine software.

d) The guest OS, in turn, runs its own processes that execute appropriate programs.

e) A guest OS is installed as follows:

(1) The virtual machine software creates a simulated disk of specified size - actually a very large file in the host OS's file system.

(2) The virtual machine software allows an installation copy of the guest OS (typically on a CD or DVD) to be "booted". The guest OS is then installed into the simulated disk.

(3) The guest OS can then be started up

DEMO - Linux running under VMWare Fusion on Mac

3. VMWare also sells a product known as ESXI that runs directly on a host machine, in effect serving as its own host OS. Any number of simulated machines can then be created, each running its own OS.

Example: The department has a single server machine running ESXI. Our file server "joshua", web server "david" and several other machines are actually simulated machines running on this single piece of hardware.

D. It is even possible to build a virtual machine that emulates DIFFERENT hardware, though this can be significantly slower.

Example: Sheepshaver, that I demonstrated earlier, runs on Intel 80x86 chips, but emulates the Power PC chip which Classic versions of Mac OS ran on. And Mac OS 9, in turn, included facilities to emulate the Motorola 68000 series chips used in earlier Macintoshes - which happened to be the chip Hypercard was written for - so there were actually two layers of emulation going on in the demo!

E. The term “virtual machine” is also used to describe environments such as the Java Virtual Machine or .NET. These run on top of a native operating system and provide their own API and many of the services that operating systems typically provide (e.g. file access), but don’t pretend to be full-blown operating systems in their own right.

   These differ from full-blown virtual machine operating systems in that the environment does not attempt to provide all the services of a full-blown OS.

   1. Instead, there is a provision for “native” methods that are written in a more traditional language (usually C) that run directly on top of the native OS. (Of course, such methods are platform-specific)

   2. In contrast, a full-blown VM OS totally hides the underlying platform from a program.

F. A lab that you will do soon will involve installing a copy of Linux as a guest OS running on a virtual machine on one of the workstations. In a subsequent lab, you will install various networking facilities on this virtual machine.