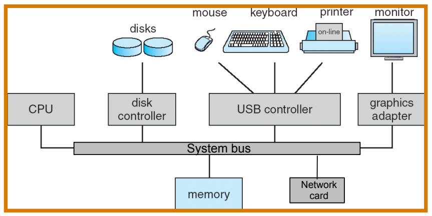
Last Class: OS and Computer Architecture



- CPU, memory, I/O devices, network card, system bus

Last Class: OS and Computer Architecture

OS Service	Hardware Support
Protection	Kernel/user mode, protected instructions, base/limit registers
Interrupts	Interrupt vectors
System calls	Trap instructions and trap vectors
I/O	Interrupts and memory mapping
Scheduling, error recovery, accounting	Timer
Syncronization	Atomic instructions
Virtual memory	Translation look-aside buffers

Today: OS Structures & Services

- Introduce the organization and components in an OS.
- OS Components
  - Processes
  - Synchronization
  - Memory & Secondary Storage Management
  - File Systems
  - I/O Systems
  - Distributed Systems
- **Four example OS organizations**
  - Monolithic kernel
  - Layered architecture
  - Microkernel
  - Modular

From the Architecture to the OS to the User

From the Architecture to the OS to the User: Architectural resources, OS management, and User Abstractions.

Hardware abstraction	Example OS Services	User abstraction
Processor	Process management, Scheduling, Traps, protection, accounting, synchronization	Process
Memory	Management, Protection, virtual memory	Address spaces
I/O devices	Concurrency with CPU, Interrupt handling	Terminal, mouse, printer, system calls
File System	File management, Persistence	Files
Distributed systems	Networking, security, distributed file system	Remote procedure calls, network file system

Processes

- The OS manages a variety of activities:
  - User programs
  - Batch jobs and command scripts
  - System programs: printers, spoolers, name servers, file servers, network listeners, etc.
- Each of these activities is encapsulated in a **process.**
- A process includes the execution context (PC, registers, VM, resources, etc.) and all the other information the activity needs to run.
- *A process is not a program*. A process is one instance of a program in execution. Many processes can be running the same program. Processes are independent entities.

OS and Processes

- The OS creates, deletes, suspends, and resumes processes.
- The OS schedules and manages processes.
- The OS manages inter-process communication and **synchronization**.
- The OS allocates resources to processes.

Synchronization Example:

Banking transactions

- Cooperating processes on a single account: ATM machine transaction, balance computation, Monthly interest computation and addition.
- All of the processes are trying to access the same account simultaneously. What can happen?

Memory & Secondary Storage Management

Main memory

- is the direct access storage for the CPU.
- Processes must be stored in main memory to execute.
- The OS must
  - allocate memory space for processes,
  - deallocate memory space,
  - maintain the mappings from virtual to physical memory (page tables),
  - decide how much memory to allocate to each process, and when a process should be removed from memory (policies).

File System

Secondary storage devices (disks) are too crude to use directly for long term storage.

- The file system provides logical objects and operations on these objects (files).
- A file is the long-term storage entity: a named collection of persistent information that can be read or written.
- File systems support directories which contain the names of files and other directories along with additional information about the files and directories (e.g., when they were created and last modified).

File System Management

- The File System provides *file management*, a standard interface to
  - create and delete files and directories
  - manipulate (read, write, extend, rename, copy, protect) files and directories
  - map files onto secondary storage
- The File System also provides general services such as backups, maintaining mapping information, accounting, and quotas.

Secondary Storage (disk)

- Secondary Storage = persistent memory (endures system failures)
- Low-level OS routines: responsible for low-level disk functions, such as scheduling of disk operations, head movement, and error handling.
  - These routines may also be responsible for managing the disk space (for example, keeping track of the free space).
  - The line between managing the disk space and the file system is very fuzzy, these routines are sometimes in the file system.
- **Example**: A program executable is stored in a file on disk. To execute a program, the OS must load the program from disk into memory.

I/O Systems

The I/O system supports communication with external devices: terminal, keyboard, printer, mouse, network card

The I/O System:

- Supports buffering and spooling of I/O
- Provides a general device driver interface, hiding the differences among devices, often mimicking the file system interface
- Provides device driver implementations specific to individual devices.

Distributed Systems

- A distributed system is a collection of processors that do not share memory or a clock.
  - To use non-local resources in a distributed system, processes must communicate over a network,
  - The OS must provide additional mechanisms for dealing with
  - failures and deadlock that are not encountered in a centralized system.
- The OS can support a distributed file system on a distributed system.
  - Users, servers, and storage devices are all dispersed among the various sites.
  - The OS must carry out its file services across the network and manage multiple, independent storage devices.

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?

Example of Standard API

- Consider the ReadFile() function in the
- Win32 API—a function for reading from a file

return value

```
BOOL   ReadFile c  (HANDLE        file,
                    LPVOID        buffer,
                    DWORD         bytes To Read,
                    LPDWORD       bytes Read,
                    LPOVERLAPPED  ovl);
```

parameters

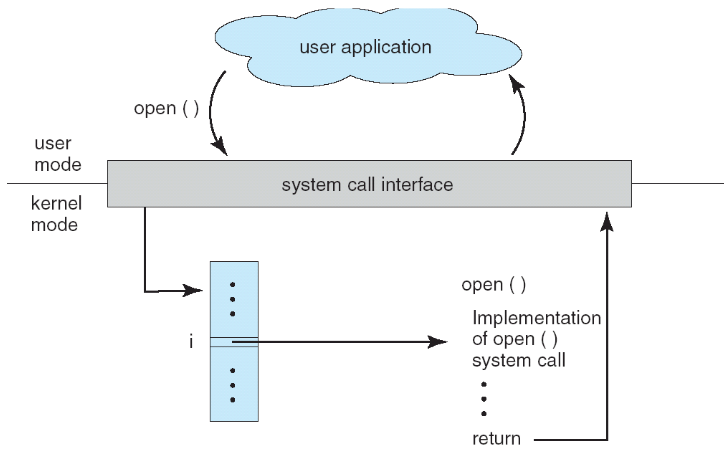
function name

- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

System Call Implementation

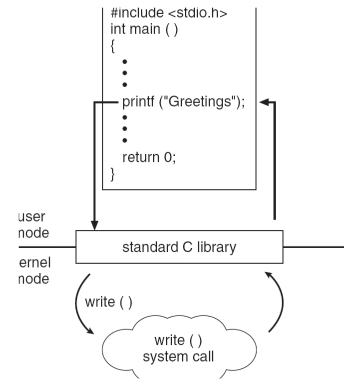
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



Standard C Library Example

- C program invoking printf() library call, which calls write() system call



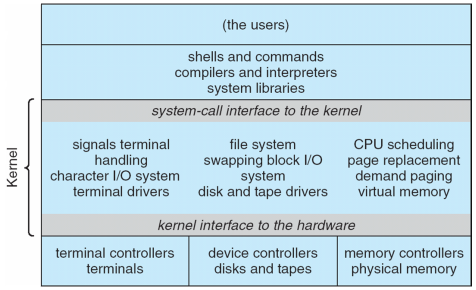
System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  – Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  – Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers
  – Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  – Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system
  – Block and stack methods do not limit the number or length of parameters being passed

Examples of Windows and Unix System Calls

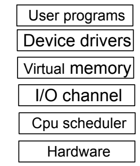
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

One Basic OS Structure



- The *kernel* is the protected part of the OS that runs in kernel mode, protecting the critical OS data structures and device registers from user programs.
- Debate about what functionality goes into the kernel (above figure: UNIX)

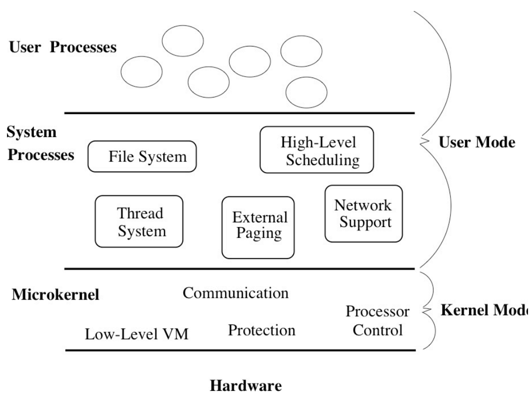
Layered OS design



Layer N: uses layer N-1 and provides new functionality to N+1

- Advantages: modularity, simplicity, portability, ease of design/debugging
- Disadvantage - communication overhead between layers, extra copying, book-keeping

Microkernel

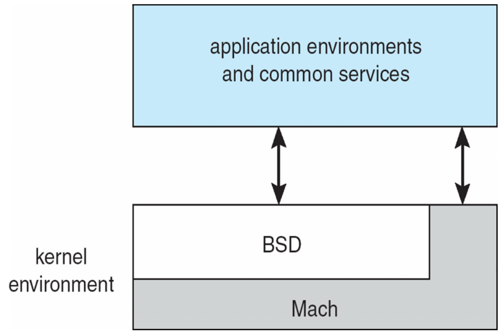


- Small kernel that provides communication (message passing) and other basic functionality
  - other OS functionality implemented as user-space processes

Microkernel Features

- **Goal**: to minimize what goes in the kernel (mechanism, no policy), implementing as much of the OS in User-Level processes as possible.
- Advantages
  - better reliability, easier extension and customization
  - mediocre performance (unfortunately)
- First Microkernel was Hydra (CMU '70). Current systems include Chorus (France) and Mach (CMU).

Mac OS X - hybrid approach

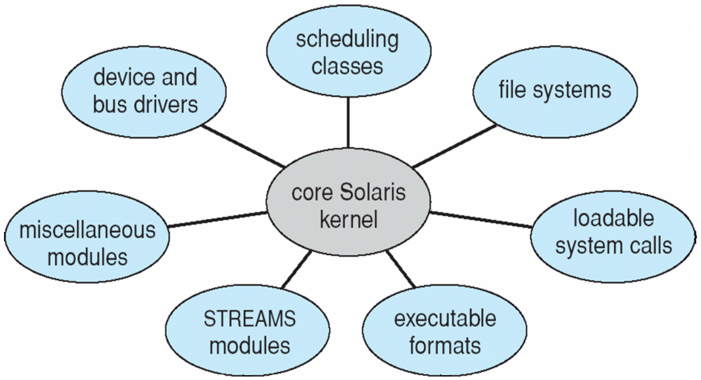


- Layered system: Mach microkernel (mem, RPC, IPC) + BSD (threads, CLI, networking, filesystem) + user-level services (GUI)

Modules

- Most modern operating systems implement kernel modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible

Solaris Modular Approach



Summary

- **Big Design Issue**: How do we make the OS efficient, reliable, and extensible?

- **General OS Philosophy**: The design and implementation of an OS involves a constant tradeoff between *simplicity* and *performance*. As a general rule, strive for simplicity except when you have a strong reason to believe that you need to make a particular component complicated to achieve acceptable performance (strong reason = simulation or evaluation study)