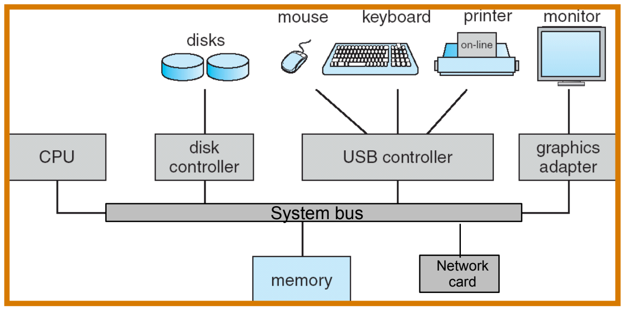
Last Class: OS and Computer Architecture



- 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

- More on System Calls
- Introduce the organization and components in an OS.
- **Four example OS organizations**
  - Monolithic kernel
  - Layered architecture
  - Microkernel
  - Modular

Class Exercise

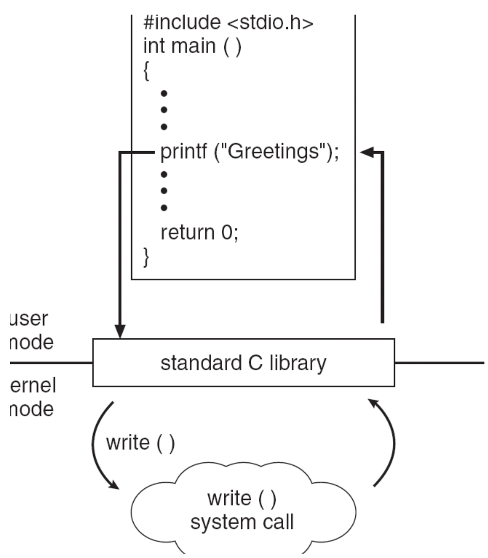
- ios 7 and iphone 5S
  - "iphone 5S first 64-bit smartphone, ios7 64-bit OS"
- iphone has M7 co-processor in addition to main A7 processor
  - Offloads work (primarily sensor data processing) from main CPU to co-processor
- Critique these design decisions. Benefits?

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?

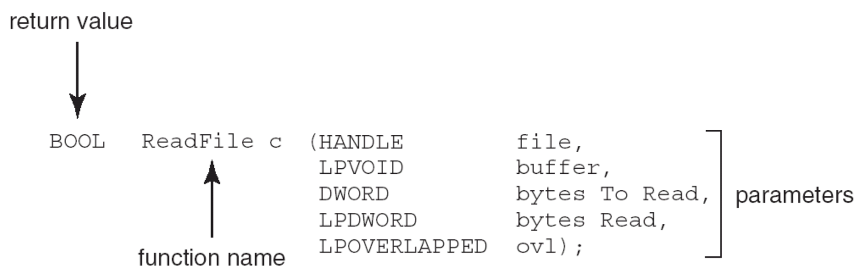
Standard C Library Example

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



Example of Standard API

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

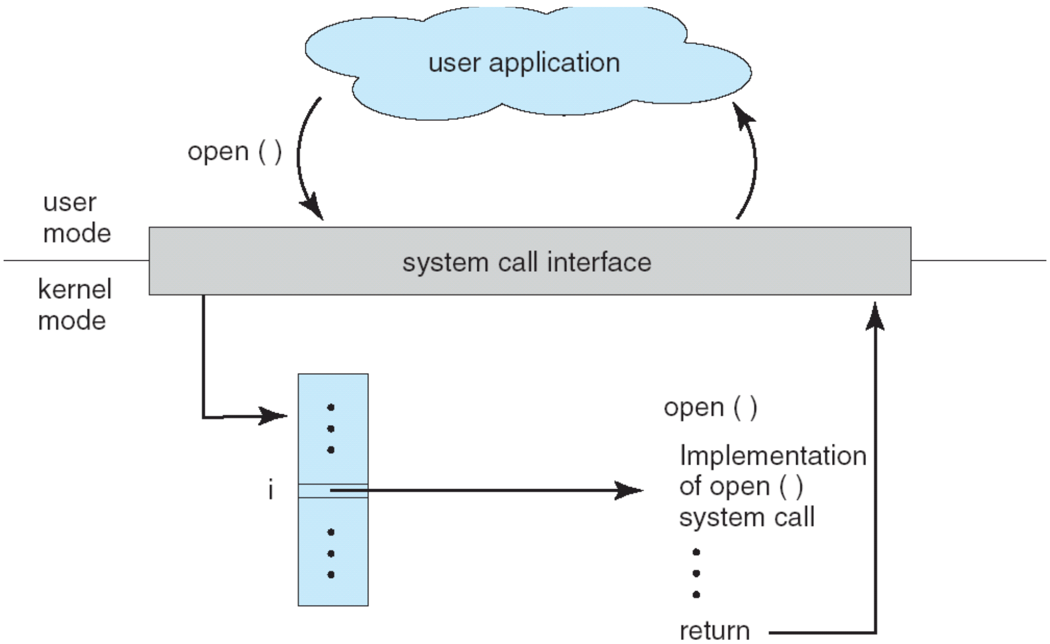


- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - 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



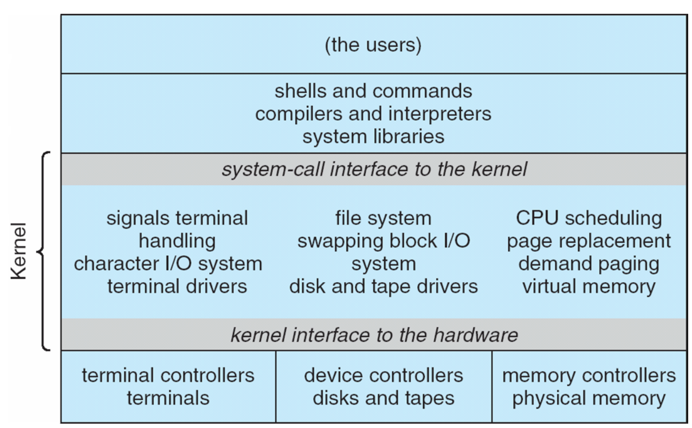
System Call Parameter Passing

- 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

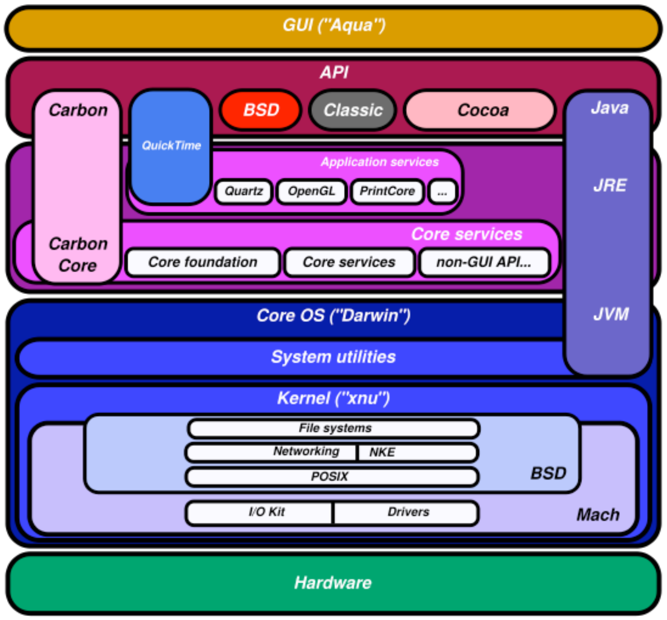
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information 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

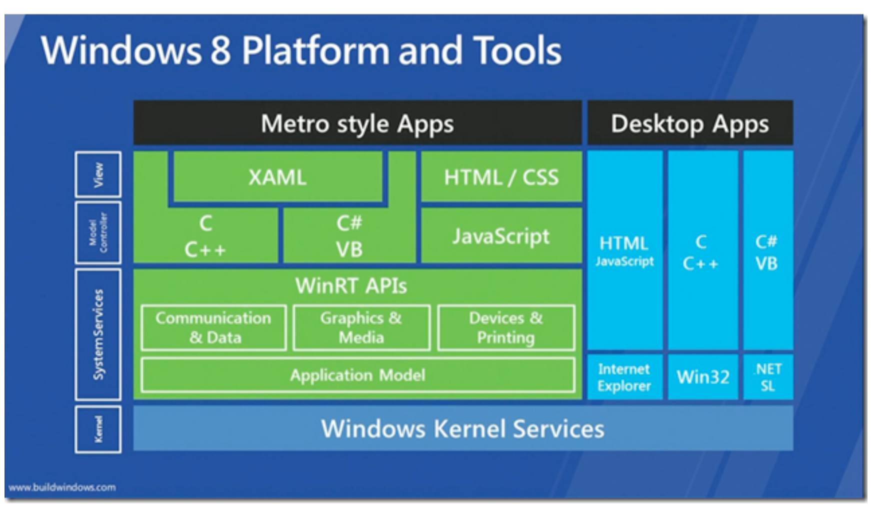


- The *kernel* is the protected part of the OS that runs in kernel mode, protecting the critical OS data structures and device registers from user programs.
- Debate about what functionality goes into the kernel (above figure: UNIX) - "monolithic kernels"

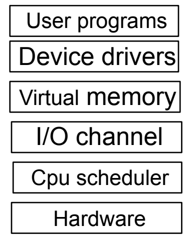
Mac OS X Architecture



Windows 8 Architecture



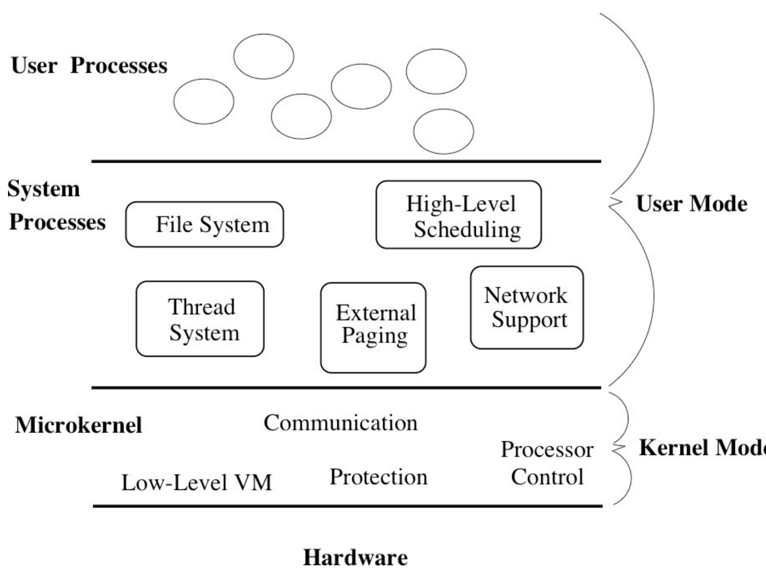
Layered OS design



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



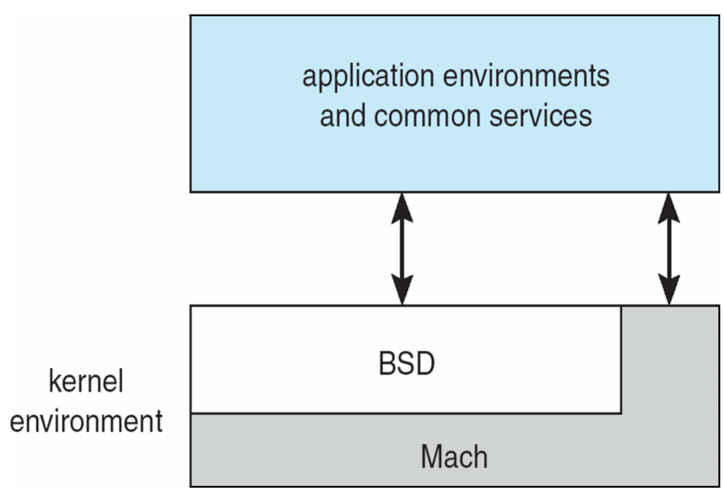
- Small kernel that provides communication (message passing) and other basic functionality
  - other OS functionality implemented as user-space processes

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

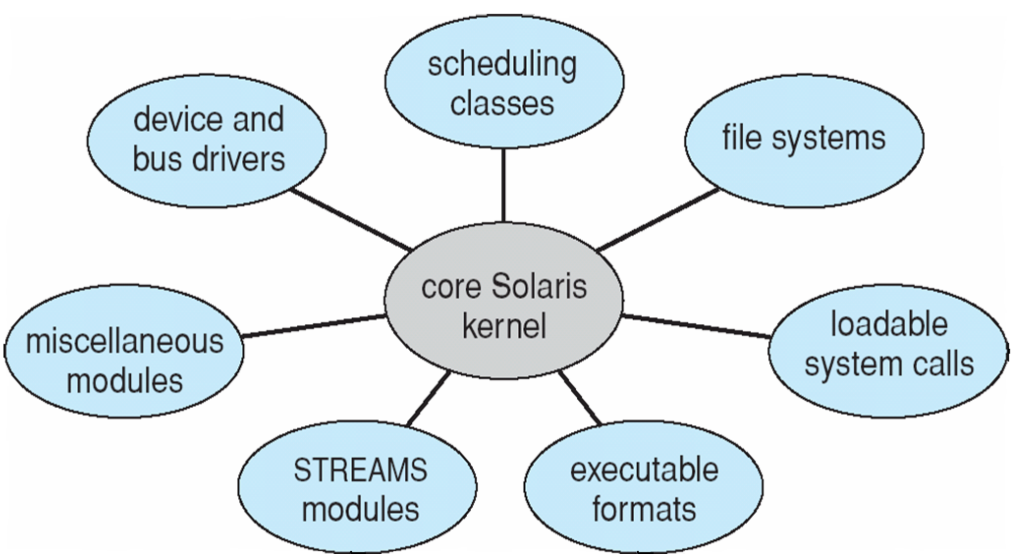


- Layered system: Mach microkernel (mem, RPC, IPC) + BSD (threads, CLI, networking, filesystem) + user-level services (GUI)

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)

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.

What's in a Process?

- **Process**: dynamic execution context of an executing program
- Several processes may run the same program, but each is a distinct process with its own state (e.g., MS Word).
- A process executes sequentially, one instruction at a time
- Process state consists of at least:
  - the code for the running program,
  - the static data for the running program,
  - space for dynamic data (the heap), the heap pointer (HP),
  - the Program Counter (PC), indicating the next instruction,
  - an execution stack with the program's call chain (the stack), the stack pointer (SP)
  - values of CPU registers
  - a set of OS resources in use (e.g., open files)
  - process execution state (ready, running, etc.).

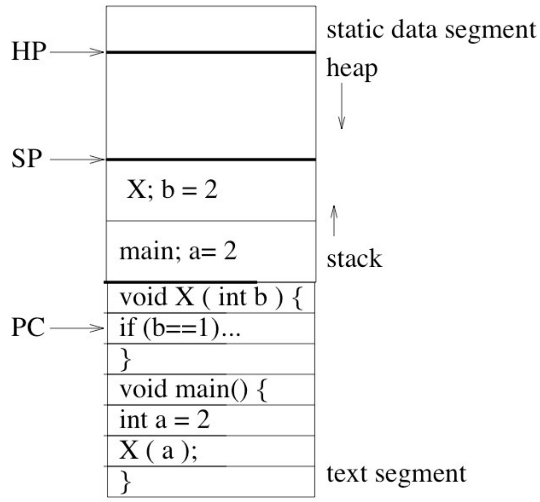
Example Process State in Memory

What you wrote:

```
   void X (int b){
PC ->   if ( b == 1 ) …
   }

   main(){
      int a = 2;
      X ( a );
   }
```

What's in memory



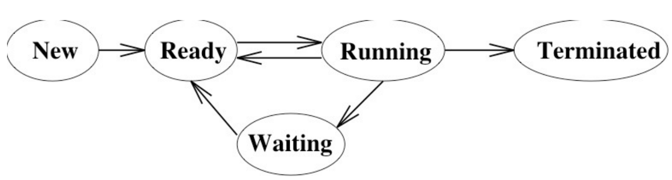
Process State

Process Execution State

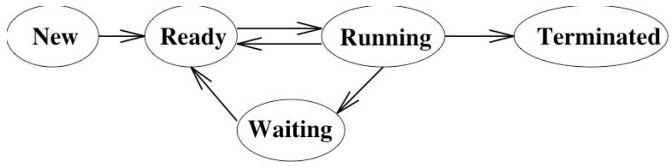
- Execution state of a process indicates what it is doing

new:	the OS is setting up the process state
running:	executing instructions on the CPU
ready:	ready to run, but waiting for the CPU
waiting:	waiting for an event to complete
terminated:	the OS is destroying this process

- As the program executes, it moves from state to state, as a result of the program actions (e.g., system calls), OS actions (scheduling), and external actions (interrupts).



Process Execution State



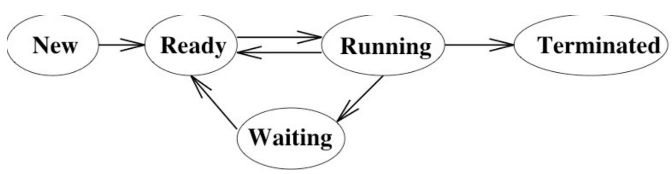
state sequence

Example:

```
void main() {
  printf('Hello World');
}
```

- The OS manages multiple active process using *state queues* (More on this in a minute…)

Process Execution State



state sequence

new

ready

running

waiting for I/O

ready

running

terminated

Example:

```
void main() {
  printf('Hello World');
}
```

- The OS manages multiple active process using *state queues* (More on this in a minute…)