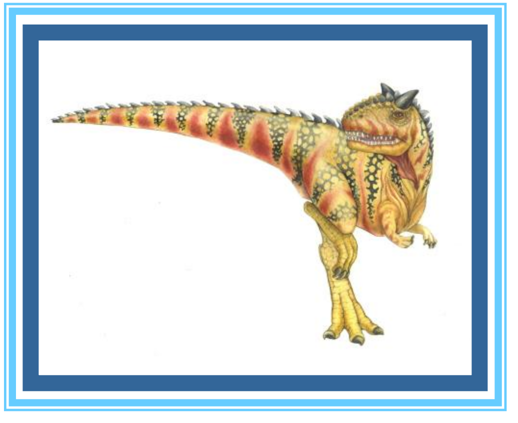
Chapter 2: Operating-System Structures



Chapter 2: Operating-System Structures

1. Operating System Services
2. User Operating System Interface
3. System Calls
4. Types of System Calls
5. System Programs
6. **Operating System Design and Implementation**
7. **Operating System Structure**
8. **Operating System Debugging**
9. **Operating System Generation**
10. **System Boot**

2.6 OPERATING-SYSTEM DESIGN AND IMPLEMENTATION

2.6.1 Design Goals

- Design and Implementation of OS not "solvable", but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely

- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- User goals and System goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
    - **no** general agreement on how to achieve them.
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
    - **vague** and may be interpreted in various ways
  - no unique solution
- Specifying and designing an OS is highly creative task of software engineering

2.6.2 Mechanisms and Policies

- Important principle to separate
- Policy: What will be done?
  Mechanism: How to do it?
- Mechanisms determine how to do something, policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum **flexibility** if policy decisions are to be changed later (example – timer)
- **Microkernel**-based operating systems (Section 2.7.3) take the separation of mechanism and policy to one extreme by implementing a basic set of primitive building blocks.
  - almost policy free
- UNIX
  - At first, a time-sharing scheduler
  - Solaris - scheduling is controlled by loadable tables.
- Policy decisions are important

2.6.3 Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware (MS-DOS vs. LINUX)
- disadvantages
  - reduced speed and increased storage requirements
- **Emulation** can allow an OS to run on non-native hardware
- modern compiler can perform complex analysis and apply sophisticated optimizations that produce excellent code
- major performance improvements are more likely to be the result of better data structures and algorithms than of excellent assembly-language code

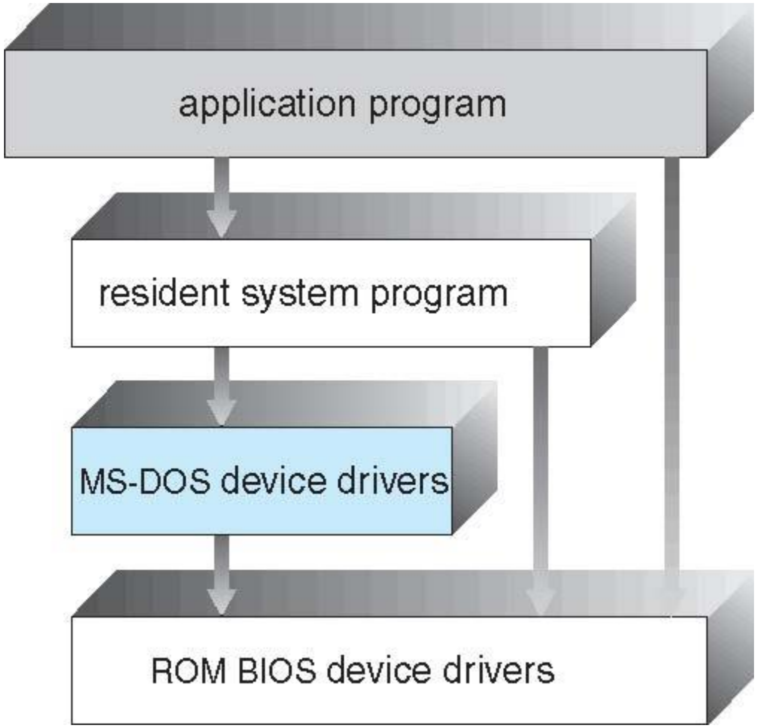
2.7 OPERATING-SYSTEM STRUCTURE

Operating System Structure

- General-purpose OS is very large program
- A system as large and complex must be engineered carefully if it is to function properly and be modified easily.
  - A common approach is to partition the task into small **components**, or **modules**, rather than have one **monolithic** system
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex -- UNIX
  - Layered – an abstrcation
  - Microkernel -Mach

2.7.1 Simple Structure

- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated



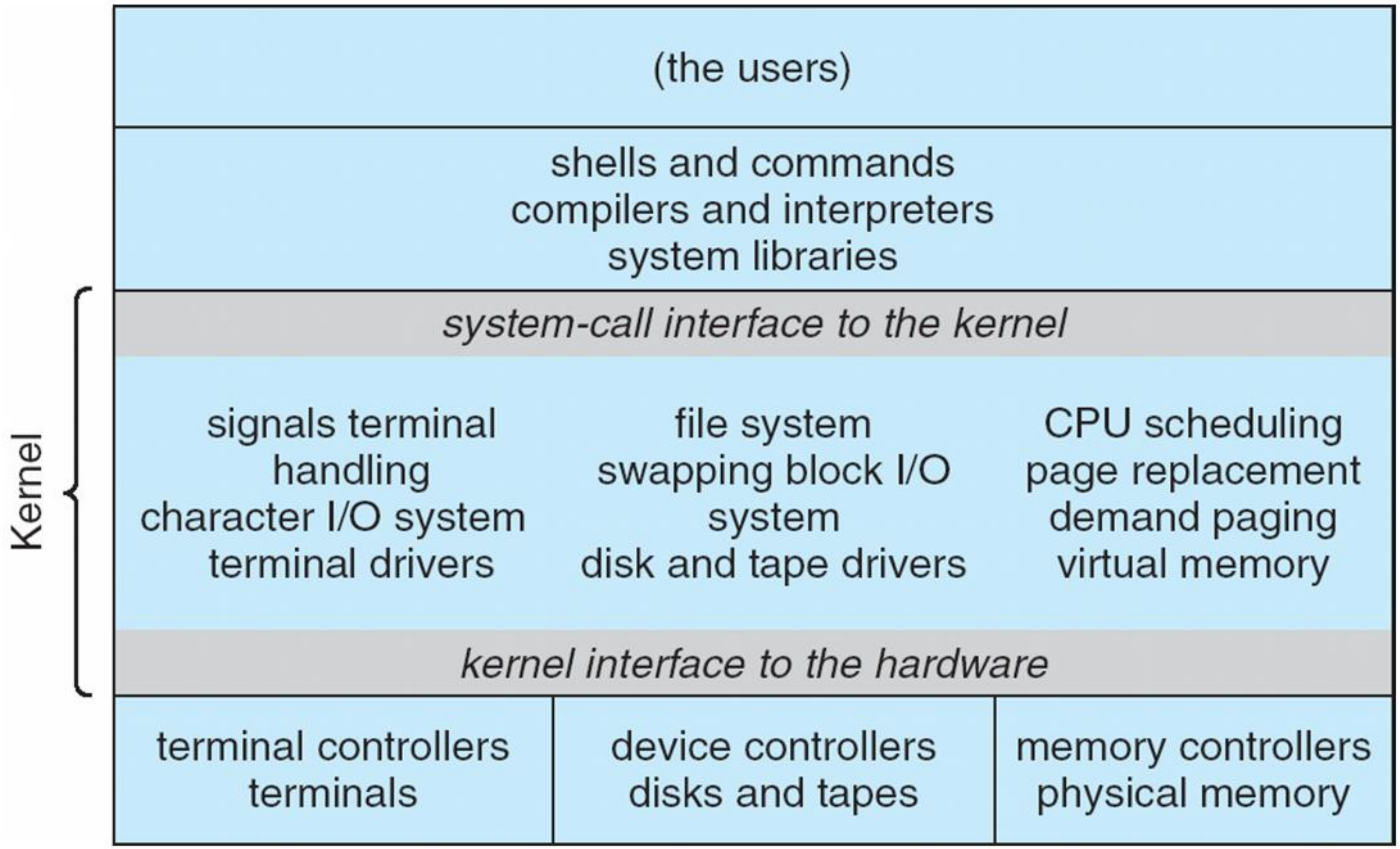
Simple Structure - the original UNIX

- UNIX – limited by hardware functionality, **the original UNIX** operating system had limited structuring.

- The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

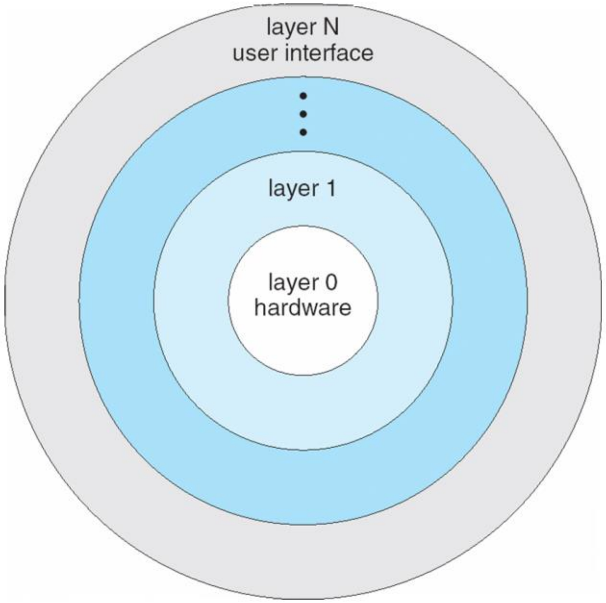
Traditional UNIX System Structure

Beyond simple but not fully layered



2.7.2 Layered Approach

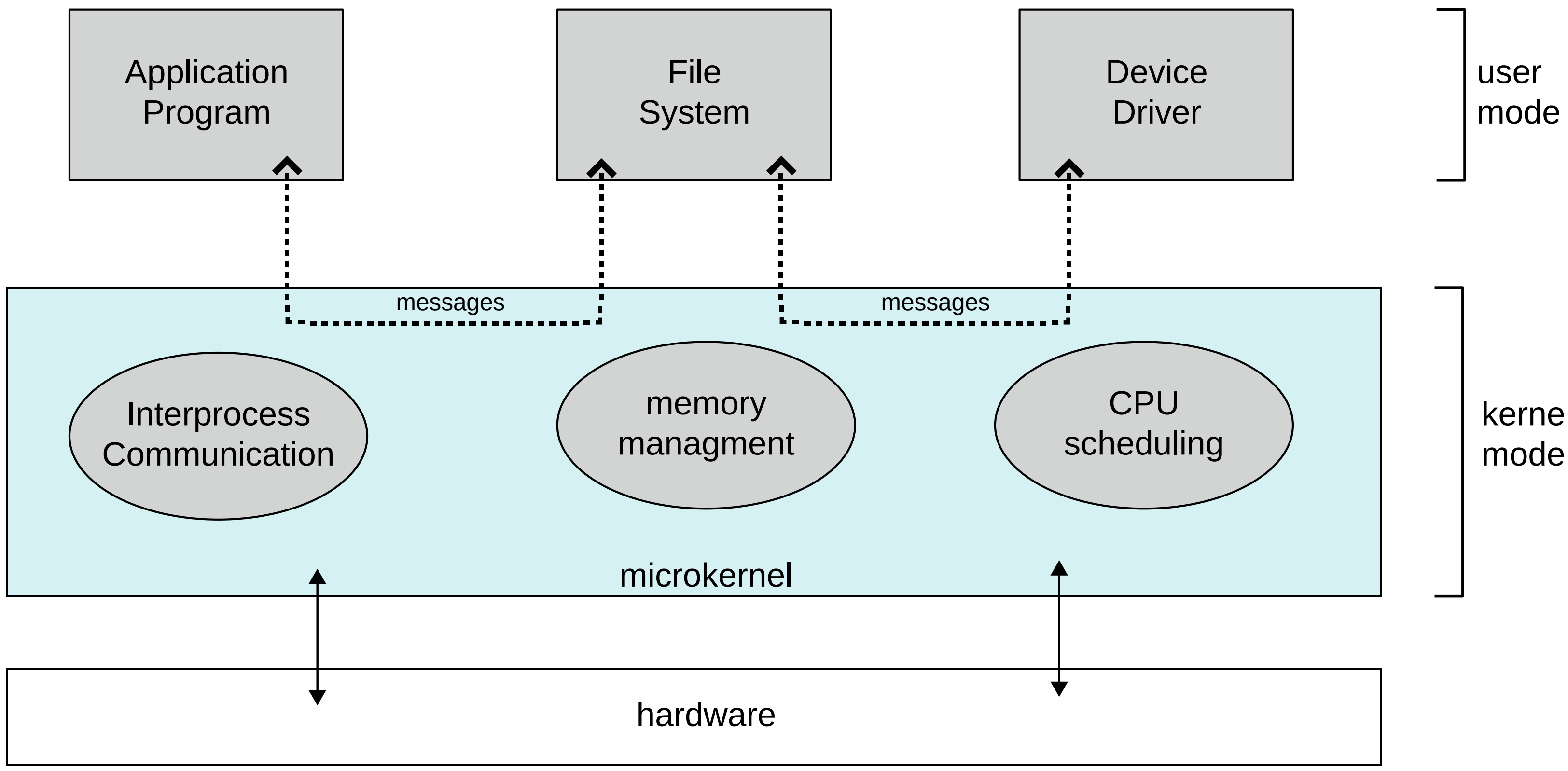
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers
- Main advantage
  - simplicity of construction and debugging
- Major difficulty
  - appropriately defining the various layers
- Problem
  - be less efficient than other types
    - passing parameters
- a small backlash
  - fewer layers with more functionality



2.7.3 Microkernels

- Moves as much from the kernel into user space
- Mach example of microkernel
  - Mac OS X kernel (Darwin) partly based on Mach
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

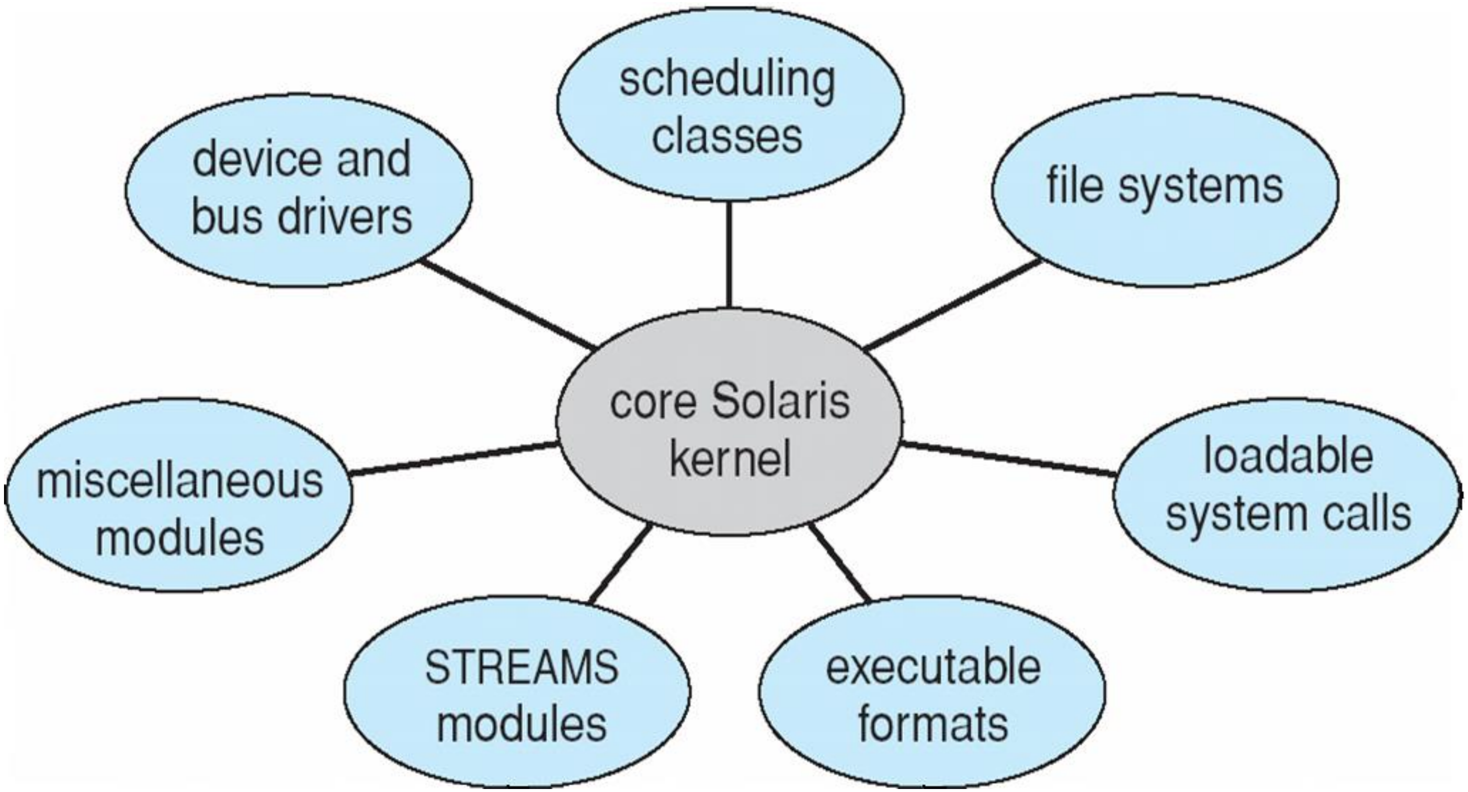
Microkernel System Structure



2.7.4 Modules

- Many modern operating systems implement **loadable 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
  - because any module can call any other module
- similar to the microkernel approach but more efficient
  - because modules do not need to invoke message passing to communicate
- Linux, Solaris, etc

Solaris Modular Approach

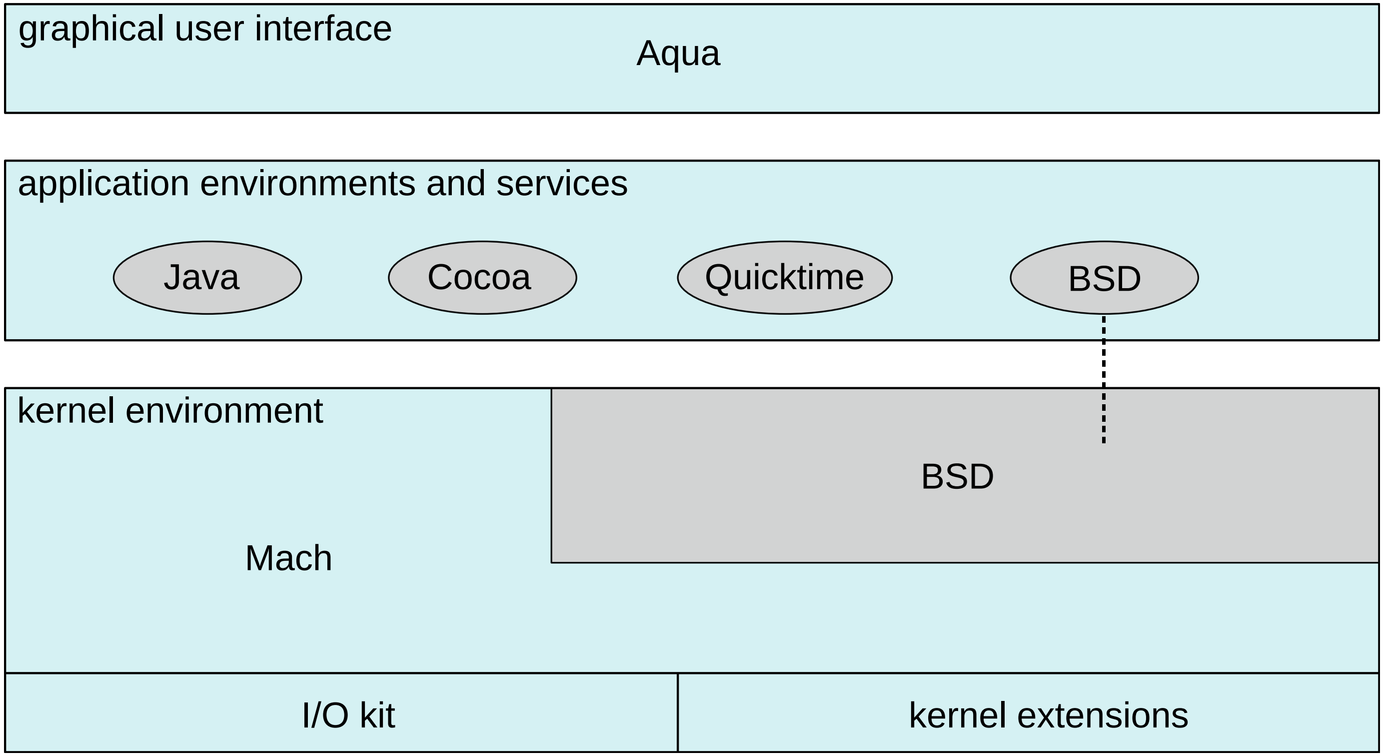


2.7.5 Hybrid Systems

- Most modern operating systems are actually **not** one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so **monolithic**, plus **modular** for dynamic loading of functionality
  - Windows mostly **monolithic**, plus **microkernel** for different subsystem personalities

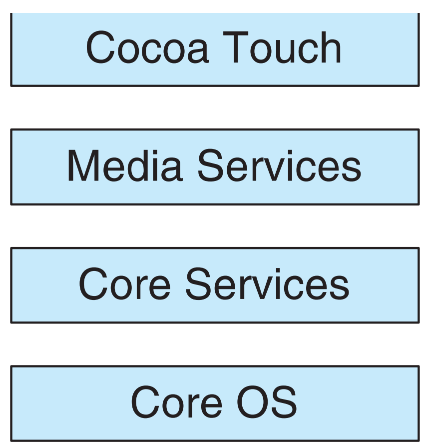
2.7.5.1 Mac OS X

- Apple Mac OS X hybrid, layered, Aqua UI plus Cocoa programming environment
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called kernel extensions)



2.7.5.2 iOS

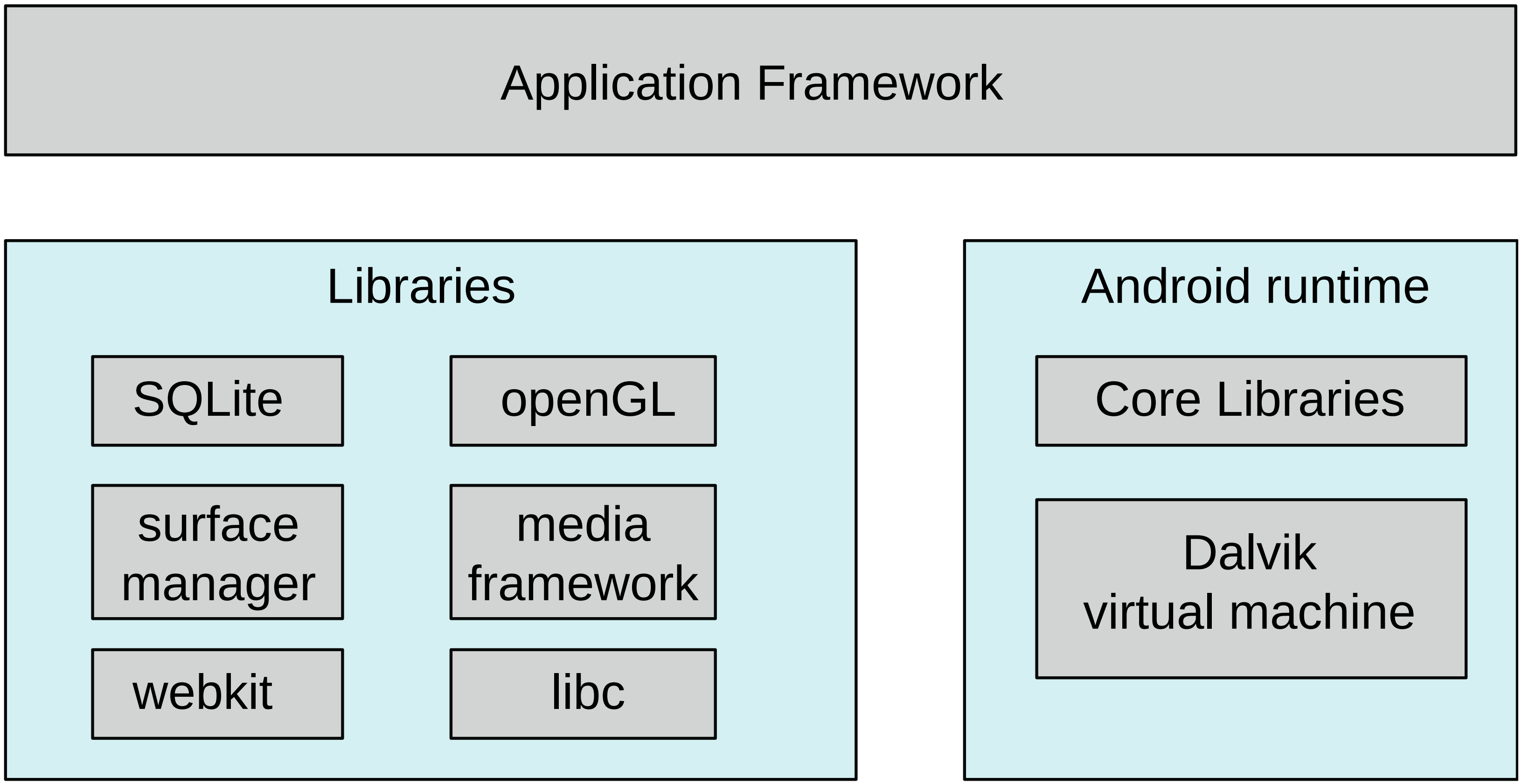
- Apple mobile OS for iPhone, iPad
  - Structured on Mac OS X, added functionality
  - Does not run OS X applications natively
    - Also runs on different CPU architecture (ARM vs. Intel)
  - Cocoa Touch Objective-C API for developing apps
  - Media services layer for graphics, audio, video
  - Core services provides cloud computing, databases
  - Core operating system, based on Mac OS X kernel



2.7.5.3 Android

- Developed by Open Handset Alliance (mostly Google)
  - Open Source
- Similar stack to IOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
    - Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

Android Architecture



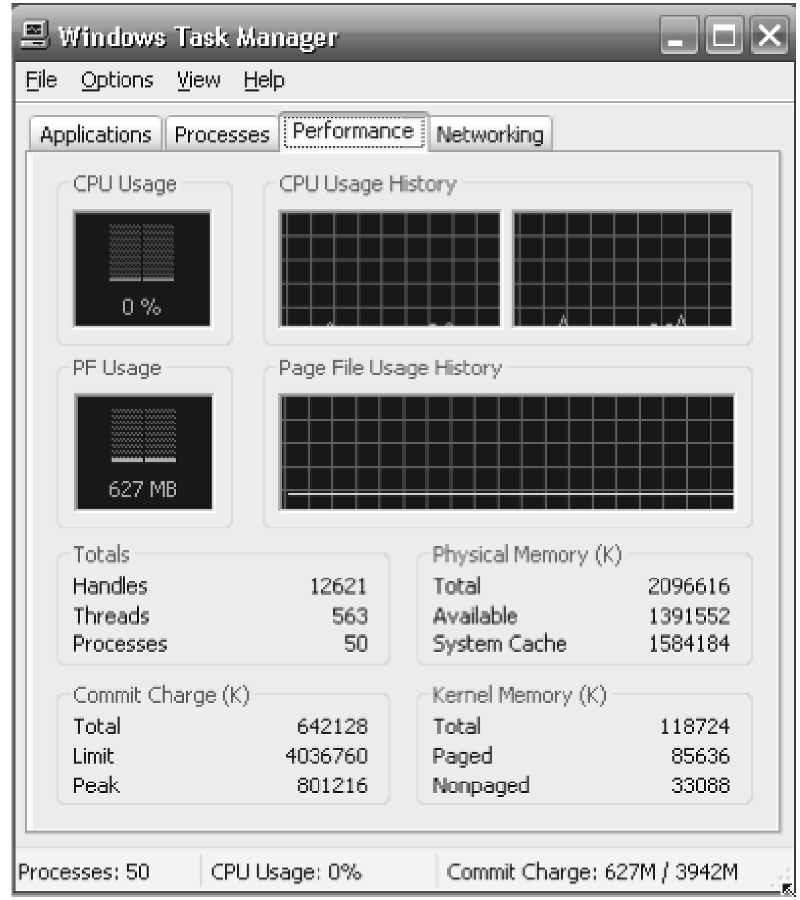
2.8 OPERATING-SYSTEM DEBUGGING

2.8.1 Failure Analysis

- Debugging is finding and fixing errors, or bugs
- OS generate **log** files containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating-system kernel debugging is even more complex
  - because of the size and complexity of the kernel, its control of the hardware, and the lack of user-level debugging tools.
- Operating system failure can generate **crash dump file** containing kernel memory
- Kernighan’s Law: “Debugging is twice as hard as writing the code in th e first place. Therefore, if you write the code as cleverly as possible, yo u are, by definition, not smart enough to debug it.”

2.8.2 Performance Tuning

- Beyond crashes, performance tuning can optimize system performance
  - Sometimes using trace listings of activities, recorded for analysis
  - Profiling is periodic sampling of instruction pointer to look for statistical trends
- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- For example, "top" program or Windows Task Manager



2.8.3 DTrace

- DTrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems
- **Probes** fire when code is executed within a **provider**, capturing state data and sending it to **consumers** of those probes
- Example of following XEventsQueued system call move from libc library to kernel and back

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                         U
  0   -> _XEventsQueued                      U
  0     -> _X11TransBytesReadable            U
  0     <- _X11TransBytesReadable            U
  0     -> _X11TransSocketBytesReadable U
  0     <- _X11TransSocketBytesreadable U
  0     -> ioctl                             U
  0       -> ioctl                           K
  0         -> getf                          K
  0           -> set_active_fd               K
  0           <- set_active_fd               K
  0         <- getf                          K
  0         -> get_udatamodel                K
  0         <- get_udatamodel                K
...
  0         -> releasef                      K
  0           -> clear_active_fd             K
  0           <- clear_active_fd             K
  0           -> cv_broadcast                K
  0           <- cv_broadcast                K
  0         <- releasef                      K
  0       <- ioctl                           K
  0     <- ioctl                             U
  0   <- _XEventsQueued                      U
  0 <- XEventsQueued                         U
```

Dtrace (Cont.)

- DTrace code to record amount of time each process with UserID 101 is in running mode (on CPU) in nanoseconds

```
sched:::on-cpu
uid == 101
{
   self->ts = timestamp;
}

sched:::off-cpu
self->ts
{
   @time[execname] = sum(timestamp - self->ts);
   self->ts = 0;
}
```

```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
     gnome-settings-d          142354
     gnome-vfs-daemon          158243
     dsdm                      189804
     wnck-applet               200030
     gnome-panel               277864
     clock-applet              374916
     mapping-daemon            385475
     xscreensaver              514177
     metacity                  539281
     Xorg                     2579646
     gnome-terminal           5007269
     mixer_applet2            7388447
     java                    10769137
```

Figure 2.21 Output of the D code.

2.9 OPERATING-SYSTEM GENERATION

Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- SYSGEN program obtains information concerning the specific configuration of the hardware system
  - Used to build system-specific compiled kernel or system-tuned
  - Can general more efficient code than one general kernel

2.10 SYSTEM BOOT

System Boot

- When power initialized on system, execution starts at a fixed memory location
  - Firmware ROM used to hold initial boot code
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – bootstrap loader, stored in ROM or EEPROM locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where boot block at fixed location loaded by ROM code, which loads bootstrap loader from disk
- Common bootstrap loader, GRUB, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then running

End of Chapter 2