Introduzione ai Sistemi Operativi

Alessio Vecchio
alessio.vecchio@unipi.it
Dip.di ingegneria dell’Informazione
Università di Pisa

Based on original slides by Silberschatz, Galvin, and Gagne
Operating System Concepts, IX edition
What is an Operating System?

- A program that acts as an intermediary between a user of a computer and the computer hardware
- Operating system goals:
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner
Abstract View

user 1 → compiler
user 2 → assembler
user 3 → text editor
... → ... system and application programs
... → database system

operating system

computer hardware
What Operating Systems Do

- Depends on the point of view
- Users want convenience, **ease of use** and **good performance**
  - Don’t care about **resource utilization**
- But shared computer such as **mainframe** or **minicomputer** must keep all users happy
- Users of dedicate systems such as **workstations** have dedicated resources but frequently use shared resources from **servers**
- Handheld computers are resource poor, optimized for usability and battery life
- Some computers have little or no user interface, such as embedded computers in devices and automobiles
Operating System Definition

■ OS is a resource allocator
  ● Manages all resources
  ● Decides between conflicting requests for efficient and fair resource use

■ OS is a control program
  ● Controls execution of programs to prevent errors and improper use of the computer
- No universally accepted definition
- “Everything a vendor ships when you order an operating system” is a good approximation
  - But varies wildly
- “The one program running at all times on the computer” is the **kernel**.
- Everything else is either
  - a system program (ships with the operating system), or
  - an application program.
Computer-System Architecture

- Most systems use a single general-purpose processor
  - Most systems have special-purpose processors as well
- **Multiprocessors** systems growing in use and importance
  - Also known as *parallel systems*, *tightly-coupled systems*
  - Advantages include:
    1. Increased throughput
    2. Economy of scale
    3. Increased reliability – graceful degradation or fault tolerance
- Two types:
  1. **Asymmetric Multiprocessing** – each processor is assigned a specific task.
  2. **Symmetric Multiprocessing** – each processor performs all tasks
Symmetric Multiprocessing Architecture
A Dual-Core Design

- Multi-chip and **multicore**
- Systems containing all chips
  - Chassis containing multiple separate systems
Clustered Systems

- Like multiprocessor systems, but multiple systems working together
  - Usually sharing storage via a **storage-area network (SAN)**
  - Provides a **high-availability** service which survives failures
    - Asymmetric clustering has one machine in hot-standby mode
    - Symmetric clustering has multiple nodes running applications, monitoring each other
  - Some clusters are for **high-performance computing (HPC)**
    - Applications must be written to use **parallelization**
Clustered Systems
Operating System Structure

- **Multiprogramming (Batch system)** needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job

- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - **Response time** should be small
  - Each user has at least one program executing in memory ⇒ **process**
  - If several jobs ready to run at the same time ⇒ **CPU scheduling**
  - If processes don’t fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes not completely in memory
Memory Layout for Multiprogrammed System

```
0  operating system

512M

job 1

job 2

job 3

job 4
```
Dual-mode operation allows OS to protect itself and other system components

- **User mode** and **kernel mode**
- **Mode bit** provided by hardware
  - Provides ability to distinguish when system is running user code or kernel code
  - Some instructions designated as **privileged**, only executable in kernel mode
  - System call changes mode to kernel, return from call resets it to user

Increasingly CPUs support multi-mode operations

- i.e. **virtual machine manager (VMM)** mode for guest VMs
Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
  - Timer is set to interrupt the computer after some time period
  - Keep a counter that is decremented by the physical clock.
  - Operating system set the counter (privileged instruction)
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time
A process is a program in execution. It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.

Process needs resources to accomplish its task
- CPU, memory, I/O, files
- Initialization data

Process termination requires reclaim of any reusable resources

Single-threaded process has one *program counter* specifying location of next instruction to execute
- Process executes instructions sequentially, one at a time, until completion

Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
- Concurrency by multiplexing the CPUs among the processes
The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling
To execute a program all (or part) of the instructions must be in memory.

All (or part) of the data that is needed by the program must be in memory.

Memory management determines what is in memory and when
  ● Optimizing CPU utilization and computer response to users

Memory management activities
  ● Keeping track of which parts of memory are currently being used and by whom
  ● Deciding which processes (or parts thereof) and data to move into and out of memory
  ● Allocating and deallocating memory space as needed
Storage Management

- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit - file
  - Each medium is controlled by device (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)

- File-System management
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
  - OS activities include
    - Creating and deleting files and directories
    - Primitives to manipulate files and directories
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media
Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- OS activities
  - Free-space management
  - Storage allocation
  - Disk scheduling
- Some storage need not be fast
  - Tertiary storage includes optical storage, magnetic tape
  - Still must be managed – by OS or applications
  - Varies between WORM (write-once, read-many-times) and RW (read-write)
I/O Subsystem

- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
  - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
  - General device-driver interface
  - Drivers for specific hardware devices
Protection and Security

- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – defense of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
  - User identities (**user IDs**, security IDs) include name and associated number, one per user
  - User ID then associated with all files, processes of that user to determine access control
  - Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
  - **Privilege escalation** allows user to change to effective ID with more rights
Operating-System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- System Boot
Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users.

- One set of operating-system services provides functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (UI).
    - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch.
  - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error).
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device.
One set of operating-system services provides functions that are helpful to the user (Cont.):

- **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.

- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS)

- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user’s and programmer’s abilities to efficiently use the system
Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing:

- **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them.
  - Many types of resources - CPU cycles, main memory, file storage, I/O devices.

- **Accounting** - To keep track of which users use how much and what kinds of computer resources.

- **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other.
  - **Protection** involves ensuring that all access to system resources is controlled.
  - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts.
A View of Operating System Services

user and other system programs

<table>
<thead>
<tr>
<th>GUI</th>
<th>batch</th>
<th>command line</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>user interfaces</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

system calls

<table>
<thead>
<tr>
<th>program execution</th>
<th>I/O operations</th>
<th>file systems</th>
<th>communication</th>
<th>resource allocation</th>
<th>accounting</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>error detection</th>
<th>protection and security</th>
</tr>
</thead>
</table>

operating system

hardware
CLI or **command interpreter** allows direct command entry

- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – **shells**
- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs
  - If the latter, adding new features doesn’t require shell modification
Bourne Shell Command Interpreter

```
PBG-Mac-Pro:~ pbg$ w
15:24 up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER    TTY     FROM     LOGIN@     IDLE     WHAT
pbg      console  -          -        14:34 50 -
pbg      s000    -          -        15:05 - w
PBG-Mac-Pro:~ pbg$ iostat 5
            disk0       disk1       disk10     cpu   load average
    KB/t  tps  MB/s  KB/t  tps  MB/s  KB/t  tps  MB/s  us  sy  id  1m  5m  15m
-----   ------  ------  ------  ------  ------  ------  ------  ------  ---  ---  ---  ---  ---  ---
33.75  343    11.30  64.31    14    0.88  39.67    0    0.02  11  5  84  1.51  1.53  1.65
  5.27  320    1.65  0.00  0.00    0.00    0.00  0.00    0.00  4  2  94  1.39  1.51  1.65
  4.28  329    1.37  0.00  0.00    0.00    0.00  0.00    0.00  5  3  92  1.44  1.51  1.65
^C
PBG-Mac-Pro:~ pbg$ ls
Applications  Music     WebEx
Applications (Parallels)  Pando Packages  config.log
Desktop        Pictures  getsmartdata.txt
Documents      Public     imp
Downloads      Sites      log
Dropbox        Thumbs.db  panda-dist
Library        Virtual Machines  prob.txt
Movies         Volumes    scripts
PBG-Mac-Pro:~ pbg$ pwd
/Users/pbg
PBG-Mac-Pro:~ pbg$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBG-Mac-Pro:~ pbg$ 
```
User Operating System Interface - GUI

- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC

- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)
Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
  - Voice commands.
The Mac OS X GUI
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 Programming 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)
Example of System Calls

- System call sequence to copy the contents of one file to another file

Example System Call Sequence
- Acquire input file name
- Write prompt to screen
- Accept input
- Acquire output file name
- Write prompt to screen
- Accept input
- Open the input file
  - if file doesn't exist, abort
- Create output file
  - if file exists, abort
- Loop
  - Read from input file
  - Write to output file
  - Until read fails
- Close output file
- Write completion message to screen
- Terminate normally
Example of Standard API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```
on the command line. A description of this API appears below:

```c
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count)
```

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- **int fd** — the file descriptor to be read
- **void *buf** — a buffer where the data will be read into
- **size_t count** — the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.
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 the 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

user application

open ()

system call interface

user mode

kernel mode

i

Implementation of open ()
system call
... return
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
Parameter Passing via Table

X: parameters for call
load address X
system call 13

X
register

use parameters from table X

code for system call 13

user program

operating system
Types of System Calls

- Process control
  - create process, terminate process, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - Locks for managing access to shared data between processes
Types of System Calls

- **File management**
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

- **Device management**
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
Types of System Calls (Cont.)

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data

- Communications
  - create, delete communication connection
  - send, receive messages if message passing model to host name or process name
  - Shared-memory model create and gain access to memory regions
  - attach and detach remote devices
Types of System Calls (Cont.)

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access
## Examples of Windows and Unix System Calls

<table>
<thead>
<tr>
<th>Category</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process/Control</td>
<td>CreateProcess()</td>
<td>fork()</td>
</tr>
<tr>
<td></td>
<td>ExitProcess()</td>
<td>exit()</td>
</tr>
<tr>
<td></td>
<td>WaitForSingleObject()</td>
<td>wait()</td>
</tr>
<tr>
<td>File/Manipulation</td>
<td>CreateFile()</td>
<td>open()</td>
</tr>
<tr>
<td></td>
<td>readFile()</td>
<td>read()</td>
</tr>
<tr>
<td></td>
<td>writeFile()</td>
<td>write()</td>
</tr>
<tr>
<td></td>
<td>CloseHandle()</td>
<td>close()</td>
</tr>
<tr>
<td>Device/Maintenance</td>
<td>SetConsoleMode()</td>
<td>ioctl()</td>
</tr>
<tr>
<td></td>
<td>ReadConsole()</td>
<td>read()</td>
</tr>
<tr>
<td></td>
<td>WriteConsole()</td>
<td>write()</td>
</tr>
<tr>
<td>Information/Maintenance</td>
<td>GetCurrentProcessID()</td>
<td>getpid()</td>
</tr>
<tr>
<td></td>
<td>SetTimer()</td>
<td>alarm()</td>
</tr>
<tr>
<td></td>
<td>Sleep()</td>
<td>sleep()</td>
</tr>
<tr>
<td>Communication</td>
<td>CreatePipe()</td>
<td>pipe()</td>
</tr>
<tr>
<td></td>
<td>CreateFileMapping()</td>
<td>shmget()</td>
</tr>
<tr>
<td></td>
<td>MapViewOfFile()</td>
<td>mmap()</td>
</tr>
<tr>
<td>Protection</td>
<td>SetFileSecurity()</td>
<td>chmod()</td>
</tr>
<tr>
<td></td>
<td>InitializeSecurityDescriptor()</td>
<td>umask()</td>
</tr>
<tr>
<td></td>
<td>SetSecurityDescriptorGroup()</td>
<td>chown()</td>
</tr>
</tbody>
</table>
Standard C Library Example

- C program invoking printf() library call, which calls write() system call
Example: MS-DOS

- Single-tasking
- Shell invoked when system booted
- Simple method to run program
  - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded

At system startup

running a program
Example: FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user’s choice of shell
- Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0 – no error
  - code > 0 – error code

| process D |
| free memory |
| process C |
| interpreter |
| process B |
| kernel |
System Programs

System programs provide a convenient environment for program development and execution. They can be divided into:

- File manipulation
- Status information
- Programming language support
- Program loading and execution
- Communications
- Background services
- Application programs

Most users’ view of the operation system is defined by system programs, not the actual system calls.
System Programs

- Provide a convenient environment for program development and execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex

- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

- **Status information**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
System Programs (Cont.)

- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text

- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided

- **Program loading and execution** - Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language

- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another’s screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another
System Programs (Cont.)

**Background Services**
- Launch at boot time
  - Some for system startup, then terminate
  - Some from system boot to shutdown
- Provide facilities like disk checking, process scheduling, error logging, printing
- Run in user context not kernel context
- Known as *services, subsystems, daemons*

**Application programs**
- Don’t pertain to system
- Run by users
- Not typically considered part of OS
- Launched by command line, mouse click, finger poke
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
  - But slower
- **Emulation** can allow an OS to run on non-native hardware
Operating System Structure

- General-purpose OS is very large program
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex -- UNIX
  - Layered – an abstraction
  - Microkernel -Mach
Simple Structure  -- MS-DOS

- 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
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

<table>
<thead>
<tr>
<th>(the users)</th>
</tr>
</thead>
<tbody>
<tr>
<td>shells and commands</td>
</tr>
<tr>
<td>compilers and interpreters</td>
</tr>
<tr>
<td>system libraries</td>
</tr>
</tbody>
</table>

**system-call interface to the kernel**

- signals terminal handling
- character I/O system
- terminal drivers
- file system
- swapping block I/O system
- disk and tape drivers
- CPU scheduling
- page replacement
- demand paging
- virtual memory

**kernel interface to the hardware**

- terminal controllers
- terminals
- device controllers
- disks and tapes
- memory controllers
- physical memory
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.
Microkernel System Structure

- 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

Application Program

File System

Device Driver

Interprocess Communication

memory management

CPU scheduling

messages

messages

user mode

kernel mode

microkernel

hardware
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
- Linux, Solaris, etc
Solaris Modular Approach

- device and bus drivers
- scheduling classes
- file systems
- miscellaneous modules
- loadable system calls
- STREAMS modules
- executable formats
Mac OS X Structure

graphical user interface

Aqua

application environments and services

Java
Cocoa
Quicktime
BSD

kernel environment

Mach

BSD

I/O kit

kernel extensions
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
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

Application Framework

Libraries
- SQLite
- OpenGL
- surface manager
- media framework
- webkit
- libc

Android runtime
- Core Libraries
- Dalvik virtual machine
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
Il disco può essere suddiviso in partizioni ognuna contenente un proprio file system

Il partizionamento del disco avviene mediante la formattazione di alto livello

**Diagramma**

- **Tabella delle Partizioni**
  - **MBR**
  - **Partizioni**
  - **Blocco di avvio**
  - **Super blocco**
  - **Gestione blocchi liberi**
  - **I-node**
  - **Directory Radice**
  - **File e Directory**
Disk Organization

■ MBR (Master Boot Record)
  ● Contiene programma di avvio
  ● La fine del MBR contiene la tabella delle partizioni

■ Tabella delle partizioni
  ● Contiene punto di inizio e fine di ogni partizione
  ● Una sola partizione è marcata come attiva
  ● È la partizione da cui verrà caricato il SO
Disk Organization

- **Blocco di avvio**
  - Contiene semplice codice eseguito in fase di bootstrap e serve a caricare il kernel
  - Ogni partizione contiene il Blocco di Avvio anche se non contiene il SO (potrebbe contenerne uno)

- **Superblocco**
  - Contiene informazioni sul file system
    - Numero magico che identifica il FS
    - Numero di blocchi del FS
    - ...

- **Gestione per lo spazio libero**
  - Strutture dati per la gestione dei blocchi liberi

- **I-node**
  - Nei SO che utilizzano gli i-node questi sono raggruppati in una parte del disco

- **Directory radice**

- **File e directory**
Two-Step Bootstrap

- Esecuzione del programma di avvio in ROM
  - Diagnosi
  - Caricamento del MBR

- Esecuzione del codice di avvio contenuto nel MBR
  - Localizza la partizione attiva dalla tabella delle partizioni
  - Legge il primo blocco (blocco di avvio) e lo esegue

- Esecuzione del codice nel Blocco di Avvio
  - Localizza il kernel nella partizione attiva
  - Carica in memoria il kernel
  - Cede il controllo al kernel