**Basic Set of Material to Cover in Lecture on Chapter Two of 9th Edition of Silberschatz.**

We can view the OS in terms of
* services it offers
* the interface to users it presents
* its components and their interconnections

**SECTION 2.1 - Operating System Services**

**Services that help (individual) users**

* User interface
  Possibilities are
    + command line
    + batch (command files)
    + GUI
* Program execution
  -> load, run, terminate
* I/O operation
* File-System manipulation
  + read/write files & dirs
  + create/delete/search
  + list file info
  + get/set permissions
* Process Communication
  + via shared memory
  + via message passing
* Error Detection
  + detection & (re)action

**Services that ensure efficient operation**

* Resource Allocation
* Accounting
  + what and how much each user uses
  + maybe for billing
  + maybe for research
* Protection and security
  + ensure that processes do not harm each other
  + keep system safe from outsiders

The OS performs as an intermediary between the user of the computing system and the hardware, functioning as a virtual computer that makes the use of the hardware more convenient and/or efficient.

**SECTION 2.2 - User and Operating-System Interface**

**2.2.1 - Command Interpreters**

Command interpreters are also known as shells.

+ Cycle:
  - get instruction
  - interpret instruction
  - execute instruction

+ Commands can be Internal or External

**2.2.2 - Graphical User Interfaces**

+ Desktop Metaphor
  - à la Xerox PARC, 1970s
  - later adopted by Apple, Windows, and others.
+ Mobile devices are more oriented toward touch screens and gestures

**2.2.3 - Choice of Interface**

+ Unix-like systems offer powerful command-line interfaces and scripting

capabilities.
- Most Windows users stick with the GUI functions.
- Macs offer both kinds of interfaces

Computer scientists view the user interface software as a user program - NOT part of the operating system.

**SECTION 2.3 - System Calls**

System calls provide OS services. They are operating system 'routines' written mostly in a high level programming language such as C or C++. Some low-level parts of system call code that access the hardware may have to be written in assembly language.

Most processes make 'heavy use' of system calls (thousands per second) for such things as
- keyboard & screen functions
- file system operations
- error processing
- process execution & termination

Programmers typically do not directly write code that makes system calls. Instead they use an API, which utilizes system calls indirectly through system libraries.
- Code portability is enhanced through the use of APIs.
- Also APIs make it simpler to use system calls that have complicated set of parameters.
- The APIs implement information hiding that makes system programming easier.

Methods used by system calls to pass parameters to the operating system:
- registers
- a block or table in memory (location written in a register)
- pushed onto the stack by the process that makes the system call

**SECTION 2.4 - Types of System Calls**

Categories of System Calls:

2.4.1
+ Process Control
  - end, abort (may involve a memory dump to a file)
  - Control is usually returned to a calling (parent) process - e.g. a command processor.
  - A (error) code may be returned for the information of the calling process.
  - Create/Terminate process
  - Get/Set process attributes
  - wait for a certain time
  - wait for an event
  - signal an event
  - acquire lock on resource
  - release lock

MS DOS is not a multiprogramming OS, and it does not utilize standalone processes.

Classic unix utilizes the fork() system call to duplicate a process, and then the exec() system call to overlay the new process with a new program.

2.4.2

+ File Manipulation
  - Create file or dir
  - Delete file or dir
  - open/read/write/reposition
  - close
  - get/set attributes
  - other ops like mv or cp

2.4.3

+ Device Manipulation
  - devices may be physical or virtual (abstract) e.g files.
  - request/release system calls may be used
  - read/write/reposition calls may be used
  - many systems treat files and devices similarly

2.4.4

+ Information Maintenance
  - get time, get date, list users, report on free space, dump memory, and so forth

2.4.5

+ Communication
  - **message passing** involves some kind of connection, and/or addressing technique
  - get_hostid, get_processid
  - open_connection
  - close_connection

  Typically daemons accept connections

  Typically client/server interaction is utilized in message passing.

  - read_message/write_message

  - **shared memory model**
    * one process creates shared memory region
    * a second process attaches the shared region
    * creation and attachment are system calls
    * afterwards processes read/write shared memory to communicate without further reliance on the OS required

The message passing model is good for small amounts of data and is easy to implement

The shared memory model has greater potential efficiency and convenience, but protection and synchronization are challenges.

2.4.6

+ Protection
  - get/set permissions
  - allow_user
  - deny_user

**SECTION 2.5 - System Programs**

System programs are also called system utilities. They facilitate software development and execution.

Some are only interfaces to a system call, but others can be complex.

System Program Categories are:

* File Management
  - create/delete/copy/rename /print/dump/list, and so on
* Status Information
  - date/time/disk space /current users/logging info/debugging info/ 'registry' of config info
* File Modification
  - text editors/file search or transform utilities
* Programming-language support
  - compilers/assemblers/ debuggers/interpreters
* Program loading and execution
  - loaders/relocatable loaders/linkage editors/ overly loaders/debugging systems
* Communications
  - mechanisms for making virtual connections among processes/users/ computer systems: message passing/browsing/e-mail/ remote login/file transfer
* Background services - some may be user level processes, others kernel processes
  - configuration scripts that run at boot time
  - long running daemons that provide services
  - daemons that serve incoming network connections
  - software that start up scheduled tasks
  - error monitors
  - printing servers

Application Programs may include:

- web browsers
- word processors
- text formatters
- spreadsheet/database
- and so on

Interfaces vary. Computer scientists usually think of the kernel of the OS as what goes on beneath the system call interface.

**SECTION 2.6 - Operating System Design and Implementation**

2.6.1 - Design Goals

+ Design depends on type of hardware and type of system, e.g. time-sharing, embedded ..
+ There will be user goals and system goals
+ There's no agreement on how to form design goals.
+ Principles of SW engineering are utilized

2.6.2 - Mechanism and Policy

+ Policy is WHAT it does
+ Mechanism is HOW it does it
+ Generally it's good to create mechanisms that can support a wide range of policies, so that it will not be hard to change policies

2.6.3 - Implementation

+ It's generally agreed that it is best to write as much of an OS in high level programming languages as possible (NOT assembly language)
+ However, some assembly language for things like device drivers and saving and restoring CPU registers will be required.
+ It may be necessary to write certain "bottleneck" portions of the system in assembly code.
+ Advantages of using high level language:
  * Easier to port
  * Faster to program
  * compact code
  * code easier to understand, debug, and modify
  * improvements in compilers, plus recompilation will improve OS code.

**SECTION 2.7 - Operating-System Structure**

A big program like an operating system should have a modular construction

2.7.1 - Simple Structure

* MS-DOS and Unix really don't have much modular structure.
* MS-DOS levels of functionality are not well separated.
* The original unix is separated into system programs and kernel, which is further divided into some interfaces and drivers.
* The original unix is layered to some extent, but rather "monolithic" too.
* Monolithic software is difficult to implement and maintain, but the absence of interface and communication overhead makes for performance advantages.

2.7.2 - Layered Approach

* The layered approach is one way to construct a modular OS.
  + layer zero is the hardware and the highest layer (N) is the user interface.
  + Each layer is able to utilize the functions of lower layers, but not the layers above.
  + The layers can be built from the bottom up.
  + If 'done right' any errors encountered must be in the layer currently under construction.
  + DISADVANTAGE: difficult to design layers without circular dependencies.
  + DISADVANTAGE: overhead passing information from layer to layer.
  + 'BACKLASH': newer designs with fewer layers.

2.7.3 - Microkernels
* Carnegie Mellon University's Mach OS is an example of microkernel architecture
* One removes all non-essential functions from the kernel and implements as user level software.
* Typically minimal process and memory management will reside in the microkernel.
* Importantly, the microkernel must provide message passing between user programs and system services executing outside the kernel.
* ADVANTAGE: ease of adding new services (to user space)
* ADVANTAGE: a smaller kernel that is easier to maintain, modify, and port.
* DISADVANTAGE: performance penalty due to message passing through the microkernel.

2.7.4 - Modules
* The idea of "loadable kernel modules" is
  + The kernel has "core components."
  + additional service modules are linked in at boot time or run time, as needed.
  + any module can call any other module.
  + modules do not need to communicate through the core.

2.7.5 - Hybrid Systems
* Most actual operating systems have designs that draw from more than one of these paradigms: simple structure, layered approach, microkernels, and modules.

* 2.7.5.1 - Mac OS X incorporates aspects of layering, microkernel, and modules design.

* 2.7.5.2 - The iOS mobile device OS resembles a layered design in some respects. It is closed-source.

* 2.7.5.3 - Android is an open-source OS for mobile devices. It is a layered stack of software with a flavor of Linux at level 1.

**SECTION 2.8 - Operating-System Debugging**

2.8.1 - Failure Analysis
* When a process has a failure, the OS may write information to log files or dump the process context to a file.
* If the kernel crashes, there is usually a crash dump written to disk.
* The dump can be turned into a proper file as part of the reboot sequence.

2.8.2 - Performance Tuning
* Trace listings are logs of "interesting system events" that designers use to compile statistics that can help them make design improvements.
* There are also tools like the unix "top" facility and the Windows Task Manager

that query the system about its current operating conditions to look for problems such as performance bottlenecks.

2.8.3 - DTrace

* DTrace adds software 'probes' to a running system to get information about its operation.
* Profiling periodically samples the instruction pointer to find out what code is being executed.
* DTrace probes can be backed out while not in use so that system performance is no longer affected by them in any way.
* DTrace is part of open-source OpenSolaris.
* It has been added to Mac OS X, and FreeBSD.

**SECTION 2.9 - Operating-System Generation**

* Typically a special program is run as part of the OS installation, to configure aspects of the system for its particular hardware and intended functions.
* Disk partitioning is commonly part of this generation process, and many other configurable characteristics.
* Depending on details of how it's done, configuration may require recompilation of the kernel, or not; selection of modules to be loaded at boot time, or not; and creating (or not) files and tables containing configuration information to be looked up from time to time by the booting and/or running operating system.

**SECTION 2.10 - System Boot**

* A simple bootstrap loader executing in ROM at boot time may fetch a more complex boot program from disk, which in turn loads and executes the kernel.
* The booting system also checks and initializes the hardware.