Lecture.2&3.OS.Structures.txt

- Operating System Services
    - Provide functions that are helpful to the user. For example:
        - User interface
            - i.e. CLI, GUI, Touch-screen, Batch, Voice-controlled, etc.
        - Program execution
            - Load program into memory, run program, end execution normally or abnormally
        - I/O operation
            - May involve a file or an I/O device
        - File system (manipulation)
            - Read, write, create, delete, search files/directories
            - List information and manage permission for a file or directory
        - Communications
            - Via shared memory or through message passing
                - Packets moved by the OS
        - Error detection
            - Identify errors on CPU, memory, hardware, I/O devices, user programs, etc.
            - Identify each type of error and be able to take the appropriate action
                - The action is done by the operating system
        - Resource allocation
            - Resources must be allocated to each user or job that is running concurrently
                - i.e. In DOS, only one application was allowed to run at a time, but now, modern OS's can run multiple programs at once
        - Accounting/Logging
            - Keep track of how much computer resource each user uses
                - A record of how much CPU time and storage each user uses can be used to bill them in the future
                    - i.e. Pay per use data, Dropbox, etc.
        - Protection & Security
            - Ensures all access to system resources is controlled, and requires user authentication
                - i.e. Permission to use webcam, mic, etc.
                - i.e. Permission to read or write a file/directory

- A View Of Operating System Services
    - An accounting service is a record that keeps information about the resources used by a user
        - This can be used for billing purposes
    - The kernel is part of the operating system
        - Drivers are also part of the operating system
            - Although some drivers may be part of the kernel
    - System calls provide an interface to the services made available

by an operating system
- Operating Systems are written in a mix of Assembly and C
  - Time critical structures are written in Assembly

- System Calls
  - The programming interface to the services provided by the OS
  - System calls are mostly accessed by programs via a high-level Application
    Programming Interface (API) rather than direct system call use
    - The three most common APIs are:
      - Win32 API for Windows
      - POSIX API for POSIX-based systems
        - Includes all versions of UNIX, Linux, and Mac OS X
      - Java API for the Java virtual machine (JVM)
    - i.e.
      The Unix command: `cp in.txt out.txt`
    - On Linux and other Unix-like operating systems, `man` is the interface used to view the system's reference manuals

- System Call Implementation
  - Typically, a number is associated with each system call
    - The system call interface maintains a table indexed according to these numbers
      - It invokes the intended system call in the OS kernel and returns the status of the system call and any return values
  - The caller does not need to know anything about how the system call is implemented
    - All it needs to do is obey the API and understand what the OS will do as a result of making the system call
  - The details of the OS interface are hidden from programmers by an API
    - This is abstraction; a core principle of (sofware) engineering
      - Abstraction is purposeful suppression, or hiding, of some details of a process or artifact, in order to bring out more clearly other aspects, details, or structure
    - The API is managed by run-time support libraries
      - This is a set of functions built into libraries and included with the compiler

- System Call Parameter Passing
  - There are three general methods to pass parameters to the OS
    1. Register
      - Passes the parameters in the registers
      - Usually the fastest way to pass parameters
      - Number of parameters that can be passed is limited to number of registers
    2. Block
      - Parameters are stored in a block of memory, and the

address of the block is passed as a parameter in a register
3. Stack
   - Parameters are pushed onto the stack by a program, and then popped off the stack by the operating system
- Note: Block and Stack methods do not limit the number or length of parameters being passed

- Types Of System Calls (0)
  1. Process control
  2. File management
  3. Device management
  4. Information maintenance
  5. Communications
  6. Protection

- Types Of System Calls (1)
  - Process Control
    - Create process, terminate process
    - End, abort, load, execute
    - Get process attributes, set process attributes
    - Wait for time, wait event, signal event
    - Allocate and free memory, dump memory if error
    - Debugger for determining bugs, single step execution
    - Locks for managing access to shared data between processes
    => Example: Unix system calls: fork(), exit(), wait()

- Types Of System Calls (2, 3)
  - File Management
    - Create, delete, open, close file
    - Read, write, reposition
    - Get and set file attributes
    => Example: Unix system calls: open(), read(), write(), close()
  - Device Management
    - Request device, release device
    - Read, write, reposition
    - Get device attributes, set device attributes, logically attach or detach devices
    => Example: Unix system calls: ioctl(), read(), write()

- Types Of System Calls (4, 5)
  - Information Maintenance
    - Get time or date, set time or date, get system data, set system data
    - Get and set process, file, or device attributes
    => Example: Unix system calls: getpid(), alarm(), sleep()
  - Communications
    - Create, delete communication connection, transfer status information, attach and detach remote devices

- Send, receive messages if message passing model; create and gain access to memory regions in shared-memory model
  => Example: Unix system calls: pipe(), shm_open(), mmap()

- Types Of System Calls (6)
    - Protection
        - Control access to resources
        - Get and set permissions
        - Allow and deny user access
        => Example: Unix system calls: chmod(), umask(), chown()

- System Call Interface
    - The standard C library provides a portion of the system-call interface
        - i.e. A C program invoking the 'printf()' function, located in the "stdio" library, which invokes write(), a system call
        - i.e.

```
#include <stdio.h>
int main () {
    // code ...
    printf("Hello World\n");
    // code ...
    return 0;
}
/*
 * The 'printf()' function in 'stdio' calls the standard C
 * library and tells it to print. The standard C library
 * invokes the 'write()' system call and performs the
 * task. Then it returns tothe standard C library, which
 * returns back to the 'printf()' function at the point it
 * was called in the C program.
 *
 * The 'printf()' function is in user mode.
 * The 'write()' function is in kernel mode.
 */
```

- Example: Arduino
    - Single tasking, single memory space
    - No operating system, boot loader loads program (sketch) loaded via USB into flash memory
        - 1KB = 1024 bytes

- Example: FreeBSD
    - FreeBSD is a Unix-like OS via Berkeley Software Distribution
        - Commonly abbreviated as BSD
    - The Shell executes the 'fork()' command to create process(es)
        - Then, it executes 'exec()' to load programs into process(es)
        - Finally, the Shell waits for process(es) to terminate or continues to carry out user commands

- A process can exit with:
  1. An exit code of 0
     - Process finished with no errors
  2. An exit code greater than 0
     - Represents an error code
  3. An exit code less than 0
     - The process was not created
- Running processes in parallel is/was not possible in DOS operating system

- Why Applications Are OS Specific?
  - Each operating system provides its own unique system calls
    - i.e. Own file formats, etc.
    - This is why an application built for Windows will not work on a Mac
      - The system calls for Windows are different from a Mac
        - i.e. Both have different ways of creating processes, threads, etc.
  - Apps can run on different operating systems, If:
    - Written in a high level language like Python, Ruby, etc., and if there is an interpreter available for that language on multiple operating systems
    - The app is written in a language that includes a virtual machine (VM), that the app can run on
      - i.e. Java's motto is "Write Once, Run Anywhere", and this is due to the Java Virtual Machine, that runs on multiple operating systems
    - A standard language, like C, needs to be compiled separately on each operating system in order to run
  - The Application Binary Interface (ABI) is the architecture equivalent of the API
    - The ABI defines how different components of binary code can interface for a given OS on a given architecture, CPU, etc.

- Operating System Design Challenges
  - Even though the internal structure of different operating systems can vary widely, their goals are pretty much the same
    - These goals can be divided into:
      - User goals
        - The OS should be convenient to use, easy to learn, reliable, safe, and fast
          - There may be a trade-off between these goals
            - i.e. More reliability at the cost of speed
      - System goals
        - The OS should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
          - There can be a trade-off between these goals
            - i.e. More flexibility at the cost of making the system more prone to errors; less

error-free
- These goals are affected by choice of hardware, type of system, customer base, etc.

- Implementation
  - Operating systems are made using a mix of programming languages
    - i.e.
      - The lowest levels are in Assembly
      - The main body is in C
      - System programs are in C, C++, scripting languages like PERL, Python, shell scripts, and other high-level languages
        - Even though high-level languages are easier to code and port to other hardware, they come at the cost of slow(er) performance
    - C and Assembly are still used to write major components of operating systems
  - Emulation can allow an OS to run on non-native hardware

- Operating System Monolithic Structure
  - The original Unix OS consists of two separable parts:
    1. System Programs
    2. Kernel
      - Consists of everything below the system call interface, and above the physical hardware
      - Provides the file system, CPU scheduling, memory management, virtual memory, etc., and other operating system functions
        - This is a large number of functions for one level
    - The advantage to this is:
      - Little overhead in the system call interface
      - Communcation within the kernel is fast
    - The disadvantage is that:
      - It is difficult to implement and extend
        - Changes in one component can affect changes to another component
          - i.e. Changing how the file-system works can introduce bugs into the memory-management component

- Traditional Unix System Structure
  - The original Unix structure was monolithic
    - This makes it difficult to implement and extend
      - This is very bad
    - There is little overhead in the system call interface
      - This is good
    - Communication within the kernel is fast
      - This is good

- Linux System Structure

- The Linux OS relies on a Monolithic plus modular design
    - In a Monolithic plus modular design, each component is placed into its own block of code and these blocks comprise the system call interface
        - The benefit is that changes in one component/block, only affect that block/component
            - i.e. Changing the CPU scheduler won't alter the file system or memory management blocks/components
    - Remember: the Linux system structure is modular based

- Layered Approach
    - The operating system is divided into a number of layers/levels
        - Each subsequent layer is built on top of lower layers
            - The bottom layer, layer 0, is the hardware
            - The highest layer, layer N, is the user interface
        - Each layer uses functions/operations and services of only lower level layers
            - i.e. Layer 3 uses Layer 2, 1 and 0, but Layer 1 does not use Layer 2 or 3
    - Advantages are
        - System verification and debugging are simplified
        - Each layer hides the existence of certain data structures, operations, and hardware from higher level layers
    - Disadvantage(s)
        - Performance of this system is poor
    - The Layered Approach is borrowed from telecommunications

- Microkernels
    - A microkernel design moves as much code from the kernel into user space as possible
        - Mach is an example of microkernel
            - Mac OS X and Darwin, their kernels, are partly based on Mach
                - OS X is based on Darwin
        - Communication takes place between user modules using message passing
            - In other words, when a function makes a call, that message gets passed from user-space into kernel-space, then back into user-space, where the correct module is called and executes the required function, and then once it finishes, it passes a message from user-space into kernel-space, then back into user-space where the original calling function is given an exit code
    - Benefits are
        - Kernel is more reliable and faster
        - Easier to extend a microkernel
        - Easier to port operating systems to new architectures
        - Less code is running in kernel mode, and more in user-space
            - This makes the kernel more secure
                - See below

    - Microkernels are more secure than other structures
      - This is because only the core structures are in the kernel, and everything else is in user-space
        - This makes it easier to debug, test all possible combinations, and decreases the attack vector because there are less possible holes in the shrunken (kernel) code base
      - Remember: The kernel controls everything
        - A malfunctioning kernel is a malfunctioning system
  - Detriments are
    - The performance overhead of user-space to kernel-space communication
      - Running back and forth between user-space and kernel-space is costly
        - Passing messages in the kernel-space is quicker

- Microkernel System Structure
  - Assigns only a few essential functions to the kernel
  - Non-essential components from the kernel are implemented as user level programs

- Modules
  - Many modern operating systems implement loadable kernel modules; referred to as LKM
    - Each LKM...
      - Uses an object-oriented approach
      - Each core component is separate
      - Talks to other modules over known interfaces
      - Is loadable as needed within the kernel
        - You use only what you need to use
    - This is similar to layers, but it is much more flexible
  - This type of design is common in modern implementations of UNIX, such as Linux, macOS, and Solaris, as well as Windows, which is not part of the UNIX family

- Hybrid Systems
  - Most modern operating systems do not follow one model; instead, they are a hybrid
    - The hybrid model combines multiple approaches to address performance, security, usability needs, etc.
  - The major operating systems have adopted a hybrid model
    - i.e.
      - Linux and Solaris use a monolithic kernel model, but they incorporate modular aspects for dynamic loading of functionality
      - Windows is mostly monolithic, but incorporates microkernels for different subsystem personalities
        - This model was chosen for speed
      - Apple's Mac OS X is hybrid, and layered
        - The kernel consists of Mach-based Darwin microkernel

                    and BSD Unix parts, plus I/O kit and dynamically loaded modules
                        - The dynamically loaded modules are called kernel extensions
                            - This is where the term "kexts" comes from
                        - Programming environment is Aqua UI plus Cocoa
                            - The application framework "Cocoa" provides an API for the Objective-C and Swift programming langugaes
                        - Core frameworks support graphics and media
                            - i.e. Quicktime, OpenGL, etc.
                        - In iOS, the environment (home-screen) is called SpringBoard

- Darwin
    - Darwin consists primarily of the Mach microkernel and the UNIX kernel
        - There are two system call interfaces:
            1. Mach system calls (aka Mach traps)
                - Note: Do not confuse Mach traps with trap exceptions
            2. BSD Unix system calls
                - POSIX
    - IPC = Inter-process communication

- Android (1)
    - Developed by Open Handset Alliance (mostly Google)
    - Android is open source
        - Anybody can take the code and develop their own version
            - i.e. GrapheneOS, LineageOS, Pixel ROM, AOKP, etc.
        - Linux is also open source
            - There are more than 600 variants of Linux
                - i.e. Manjaro, Arch, Ubuntu, Kali, Etc.
    - The stack is similar to Apple's iOS; which was built for iPhone and iPad
        - Note: iPads now run iPadOS, which is a spin-off of iOS
    - Android is based on the Linux kernel
        - This provides it with process, memory, and device-driver management
            - And other features/services
        - Adds power management
            - This was added later because Android is built to primarily run on mobile devices, which have a limited capacity battery

- Android (2)
    - Android was originally running on a virtual machine called Dalvik
        - Since 2015, Dalvik runtime is no longer maintained or available to the public
            - Dalvik has been replaced by the Android RunTime (ART) VM

        - In addition to the VM, Android ships with essential core set of libraries
    - Google switched Android from Dalvik to ART to fully control everything
        - ART was built to be better, faster, and more flexible than Dalvik
    - Apps are developed in Java plus the Android API
        - Java class files are compiled to Java bytecode, and then translated to executable that runs in ART VM
    - Android libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc, openGL, etc.

- Android (3)
    - Hardware Abstraction Layer (HAL)
        - The HAL is a map between generic hardware commands and responses and those unique to a specific platform
            - By abstracting all hardware, the HAL provides applications with a consistent view independent of specific hardware
                - Example of hardware: GPS chip, camera, microphone, etc.
    - Bionic is a C library developed by Google to avoid using GNU's C
        - Bionic was developed because of:
            - Issues with the general public license (GPL)
            - Google wanted more control and the ability to fine tune things

- Android Runtime
    - JNI stands for Java Native Interface
        - JNI allows us to access hardware functions without going through the virtual machine
    - Virtual machines allow developers to run their code on different platforms with little to no additional effort
    - Every Android application runs in its own process with its own instance of the ART VM
        - The ART VM executes files in the Dalvik Executable (.dex) format
    - Android includes a set of core libraries that provides most of the functionality available in Java's core libraries
        - In other words, existing Java programmers can easily write apps for Android devices
    - To execute an operation, the ART VM calls on the corresponding C/C++ library using the Java Native Interface (JNI)

- Power Management
    - Power management is important because Android runs on mobile devices, which run on batteries
        - Being able to control how much power is used is essential in conserving battery power and capacity
    - Android adds two features to the Linux kernel to enhance the

ability to perform power management

1. Alarms
   - Implemented in the Linux kernel and is visible to the app developer through the AlarmManager in the Runtime core libraries
     - The implementation is in the kernel, so an alarm can ring even if the system is in sleep mode
2. Wakelocks
   - Prevents an Android system from entering sleep mode
     - i.e. Keep phone on while user watches video or plays game
   - Wakelocks are requested through the API whenever an application requires an I/O to remain powered on

- End Of OS Structures
  - Operating Systems are among the most complex pieces of software ever developed
    - When you change something in the OS, don't change too many things or else you will break something