Chapter 2 - OS Structures

1. **Chapter 2: Operating System Structures:** In this chapter we will try to answer the following questions.

   - What are the services provided by an OS ?
   - What are system calls ?
   - What are some common categories of system calls ?
   - What are the principles behind OS design & implementation ?
   - What are common ways of structuring an OS ?
   - How are VMs and OS related ?

   Detailed discussion on many of these issues will continue in the remaining portion of the semester.

2. **Operating System Services:**

   - Operating system services may change from one OS to the other. For example, some embedded OS may not allow user interaction. However, many services remain the same.
   - CLI uses text commands, as those issued from a shell in linux. Batch interface executes files containing commands, like commands written and execuetd from makefiles or shell scripts. GUI is well-known using windowing system, mouse pointer, menus, etc.
   - Trivia: Who invented the first OS graphical interface?
   - I/O devices include disks, CD, DVD, USB devices, printer, etc. How are I/O devices protected? For ease of operation, efficiency, and protection, I/O cannot be directly controlled in user-mode.
   - Several varieties of file systems are available, like the disk file system (FAT, NTFS), flash file system (flash devices), database file systems (in addition to hierarchical management, files are identified by other characteristics, like type of file, topic, author), network file system providing access to files on a remote server (FTP client).

3. **Operating System Services (2):**

   - The processes may be on the same computer, or on distinct computers separated by some network. Shared memory can be be simultaneously accesses by multiple processes, across distinct address spaces. We will study microkernel OS that rely on messages for communication between user processes.
   - The OS detects and reports errors to the user. In Unix, a return value of zero means correct execution, higher values report increasing severness of errors.

4. **Operating System Services (3):** Various scheduling algorithms may be employed to determine access to different devices. CPU cycles uses time-shared allocation, printer uses spooling algorithms, etc.

   If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

5. **A View of Operating System Services:** Figure shows a view of operating system services and how they inter-relate.

6. **System Calls:**

   - Programs in user mode do not have *direct* access to system resources, such as I/O devices, which are shared.
   - The API interface handles the low-level details of passing information to the kernel and switching to supervisor mode, as well as any data processing and preparation which does not need to be done in privileged mode.
   - Programs using an API (and not system calls) will work on any system that supports the same API.
   - System call implementation requires some architecture-specific mechanism. RISC architectures generally only provide an interrupt mechanism. CISC architectures may provide a direct *syscall* instruction to quickly transfer control to the kernel for a system call without the overhead of an interrupt.
   - Example: Linux 2.5 began using this on the x86, where available; formerly it used the INT instruction, where the system call number was placed in the EAX register before interrupt 0x80 was executed.
   - Can you explain the difference between a system call Vs. Library call?

7. **Example of System Calls:** Even simple programs make heavy use of system calls, typically thousands of such calls executed per second.

8. **API – System Call – OS Relationship:** A number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers.

   The caller does not worry about how the system call is implemented. The caller obeys the API, which hides the details of the OS interface. The API call massages the arguments as desired by the system call. Linux has between 300-350 system calls.

9. **Standard C Library Example:** `printf()` is part of the libc API library. It calls the `write` system call to write the passed message to the specified I/O device. These are passed as arguments.

10. **System Call Parameter Passing:** Register method is not general enough to handle all cases. The argument passing specifications as given as part of the ABI to maintain system portability.

11. **Types of System Calls:** Some system calls may be present in multiple categories. Some important system calls:
    create process – fork()
    terminate process – exit()
    allocate and free memory – brk()
    etc.

12. **System Programs:** System programs shipped with the OS may vary with the OS, and may provide different functionalities. These programs are outside the kernel, and are not traditionally a part of the OS. System programs can be user interfaces to system calls, or may be considerably more complex.

    Note: Make sure you can differentiate between system calls, system programs, and application programs.

    Note: How many of these (Unix) system programs do you recognize?

13. **Role of Linker and Loader:** This figure shows the various systems tools to transform a program written in a high-level language to a binary executable.

    - **Compiler:** Transform a high-level language program to assembly program. (Given, test1.c; `gcc -S test1.c` will generate assembly program, `test1.S`.
    - **Assembler:** Assembly code to object code. (`gcc -c test1.c` will generate `test1.o`, a binary file. Use "objdump" to disassemble object files.)
    - **Linker:** Resolve external references, and generate executable file. (`gcc -o test1.exe test1.c` will generate the executable file, `test1.exe`)
    - **Loader:** is a part of the OS, that creates a *process* by loading the program into memory, allocating resources, etc.

14. **OS Design and Implementation:**

    - Lets see problems in designing and implementing an OS. No unique solution is presented, just some engineering principles.
    - At the highest design level, is the type of the OS. Note the requirement for each type of system. For example,
      Batch OS – does not does very fast switching between jobs, can reduce switch overhead.
      Single-user OS – may need to worry less about about security, etc.
    - Very important to separate policy from mechanism. 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. Best to devise mechanisms so that several different user or system policies can be configured just by changing arguments.
    - With good compilers higher-level language programs can deliver performance very close to assembly programs. Compiler technology is constantly improving, which means that recompiled systems should perform better. Profiling can be used to only improve the critical sections of an OS.

15. **Operating System Structure:**

    - OS is large and complex piece of software. Proper engineering is essential. An OS is modularized into distinct components with well-defined interfaces. The OS should fulfil the essential software engineering design principles. We will see how these components are combined to work together.
    - Many OS have a simple monolithic structure that makes it hard to maintain and update. DOS started off as a simple OS then became very popular and had to be extended to other environments. Similarly, UNIX also encompasses a massive amount of functionality in the kernel. A better or poor OS structure plays little bearing on the popularity of the OS !!

16. **Operating System Structure (2):** Application programs in DOS had the power to change anything.

    While many modern OSes do not use the BIOS once started, DOS used BIOS interrupts extensively. Modern OS installs its own interrupt handlers. BIOS operates in 16-bit real mode, making it diffuclt to work with 32 and 64-bit OS.

17. **Operating System Structure (3):** Unix consists of 2 separable parts: the kernel and system programs. The kernel consists of everything below the system-call interface and above

the physical hardware. It provides the file system, CPU scheduling, memory management, and other operating-system functions. Thus, the structure is highly monolithic with a large number of functions for one level.

18. **Operating System Structure (4):**

    - The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface. Each layer defines data structures and functions to be invoked by upper-level layers.
    - Achieves information hiding since lower level routines can be implemented as seen fit as long as the interface they provide to the upper levels remains unchanged.
    - Each layer can be debugged individually, and when verifying layer n, all lower layers can be assumed correct, since they have already been debugged and verified.
    - Interdependece between layers is an issue. For example, device drivers for the backing store must be at a lower level then memory management routines, because memory management requires those drivers. However, backing store drivers need to be above CPU scheduler, since scheduler can schedule around the driver if it is waiting for I/O, however, if scheduling tables are large then they themselves might require swapping.
    - Inter-layer calls may require argument massaging at every level, and additional calls.

19. **Operating System Structure (5):** Shows layered OS with N layers. Layer 0 is hardware, layer N is the user interface. Note that most designs try to adopt a layered approach, here we are just stressing more conspicuous layers, rather than just 2 used in Unix.

20. **Operating System Structre (6):** Microkernels only provide limited process, memory management and communication facilities. The user-level programs never interact directly, but exchange messages with the kernel.

    New services are added to the user space and do not need modifications to the kernel. Its small size results in ease of porting, reliability, security.

    Tru64 Unix implements a Mach kernel, but provides a Unix user interface. Windows NT started as a microkernel but performance reasons drove it to include more within the kernel.

21. **Operating System Structure (7):**

    - Most modern operating systems implement kernel modules. Each kernel section has defined, protected interfaces like in a layered approach, but any module can talk to any other module. Thus, no restriction of only talking to the lower layer.
    - Modules can talk directly without passing messages to the kernel, hence more efficient.
    - Loadable kernel modules in Linux are loaded (and unloaded) by the modprobe command. They are located in /lib/modules and have had the extension .ko ("kernel object") since version 2.6. The lsmod command lists the loaded kernel modules.
    - The modules allow easy extension of the operating systems's capabilities as required. So, this is more flexible approach to handle the OS changes at run-time.
    - A related concept is statically Vs. dynamically linked libraries.

22. **Operating System Structure (8):** Solaris design: Organized around a core kernel, with seven types of loadable kernel modules.

23. **Operating System Structure (9):** Most modern operating systems are actually not one pure model. Hybrid combines multiple approaches to address performance, security, usability needs May contain 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.

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) Mac OS X is a hybrid design, with a layered approach and microkernel design. The lowest layer uses the Mach microkernel, with the next layer based on BSD Unix. The top layer include application programs, GUIs, etc. and can talk directly with the microkernel or the kernel.

Android VM – Based on Linux kernel but modified to provide process, memory, device-driver management. Also adds power management.

24. **Virtual Machines:**

    - Why are we discussing VMs in this class? Because in future systems, the OS may not be in control of the computer resources, even though it may not realize it. The hypervisor will do the job of resource management, the OS will provide the user interface.
    - A multi-user OS provides the illusion that each user in the sole person operating the machine. This is not real, therefore most general-purpose OSes are actually running a virtual machine.
    - The VM running on an x86 machine can expose an interface for a PowerPC machine. Then VM will need to simulate the instructions of the PowerPC binary on the x86 machine.
    - The hypervisor provides scheduling, timer services, multiplexes hardware resources, memory management, etc.
    - In this course we will look at multiprogramming OS, and virtual memory, which are examples of virtualization. The same concept of virtualization extended to cover all aspects of a machine is called a virtual machine.

25. **Virtual Machines (2):** The virtual machine implementation is also called the hypervisor. Each process running on the hypervisor, can run its own OS. This provides complete separation between the applications running on different VMs.

26. **Virtual Machine History and Benefits:** Some VM bebefits include:

    - A virus in one OS can affect application in the VM, but should not affect the host or other VMs.
    - Applications running on different VMs communicate as if on different machines by sharing a file-system volume, or by using a virtual network.
    - Bugs in the OS can be damaging to data. So, OSes can be tested in an isolated VM environment. At the same time other users can keep using the system on the older, original OS.
    - Applications can be tested on multiple platforms without the need to maintain multiple hardware configurations.
    - Currently, a major application of VMs is server consolidation. Another big application is application portability by writing programs for a virtual machine, such as Java or MS-.Net.

27. **VMware Architecture:** VMware runs on traditional OSes such as Linux and Windows as a user application. Allows running several different guest OSes as independent virtual machines.

28. **Java Virtual Machine:** Java is a popular OO language that compiles its code to a machine-independent bytecode format in the .class files. The JVM is a specification for an abstract computer, with its own ISA. Loader loads the bytecodes and verifies their sanity, such as out-of-bound jumps, pointer arithmetic, etc. Since the ISA is different that the native ISA, we need interpretation of the bytecodes. The JVM is more than a pure interpreter, since it provides verification, memory management (garbage collection), JIT compilation, etc.