Justin Ciocoi

Oct. 27, 2023

CSCI 375 Textbook Notes

Chapter 2: Operating-System Structures

2.1 Operating System Services

- Operating services can vary in multiple ways across different systems and architectures, but we can identify a few common classes of services that will normally be implemented in a general purpose computer
- One set of operating system services provides functions that are helpful to the user
  - **User Interface**
    - Most common modern computers use a *graphical user interface (GUI)*
    - Another option is a *command-line interface (CLI)*
  - **Program Execution**, which allows users to load programs into memory and execute them
  - **I/O Operations**, such that the user has a reasonable and effective way to conduct I/O
  - **File System Manipulation**, such that a user is able to organize files into folders and create directories
  - **Communications**, which refer to communications between processes, either on the same computer, or on different computers which are connected by a network
    - Communications can be implement utilizing *shared memory* pr *message passing*, in which packets of predefined formats are moved between processes
  - **Error Detection**, for the many different types of error that might occur in a computer system
- There is another set of operating system services which exist not for the user directly, but rather for ensuring the efficient operation of the system itself
  - **Resource Allocation**
    - When multiple processes are executing at the same time, resources have to be appropriately allocated to each of them
    - The operating system manages many different types of resources (CPU cycles, main memory, file storage, etc.)
  - **Logging**, such that the computer will keep track of resource usage, which can be used either for financial accounting purposes, or simply for data retention purposes

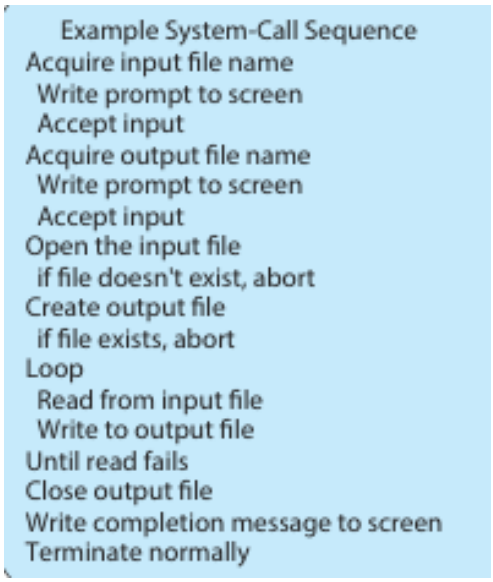
- **Protection and Security**
  - This refers to the ability to segment permissions across a multi-user system whether it is a single computer system or a file sharing system over a network
  - Such multi-user security implementation generally begins with the introduction of user password at computer startup and certain privileged actions

2.2: User and Operating-System Interface

- **2.2.1: Command Interpreters**
  - In most modern operating systems (Windows, MacOS, Linux)the command interpreter is treated as a special program that is running when a process is initiated or when a user first logs on
  - The command line will interpret text-based commands from the user, but there is a steep learning curve associated with using a command interpreter efficiently
- **2.2.2: Graphical User Interface**
  - A second, more popular strategy for user interface is through a user friendly graphical user interface
  - This is the interface with which most people are most accustomed to, featuring windows and menus, a moving mouse for selection, and icons representing different programs and files

2.3: System Calls

- *System calls* provide an interface to these operating-system services, and are usually functions written in C or C++, although certain low-level tasks may have to be implemented using assembly language
- **2.3.1: Example**
  - Let us consider the UNIX `cp` command in the context:
    - `cp in.txt out.txt`
  - This command will copy the input file, `in.txt` to the output file `out.txt`
  - So, once the two file names have been obtained, the program will open the input file, and create and open the output file
    - Each of these operations requires another system call

- The full list of system calls involved in such an operation is shown below
- 

- **2.3.2: Application Programming Interface**
  - On many occasions, systems will execute thousands of system calls per second
  - However, programmers never see this level of detail since typically developers design programs according to an *application programming interface (API)*
  - The API specifies a set of functions that are available to an application programmer, including 5the parameters that are passed to each function and the return values the programmer can expect
  - The system call names used throughout this text are generic ones, and each operating system has its own name for each system call
  - So now we ask the question of why a a programmer would prefer to program according to an API rather than invoking specific system calls
  - One reason is portability, since a developer can expect their program to compile and run on any system that supports the same API
  - Another reason is that actual system calls can often be more detailed and difficult to work with than APIs
- **2.3.3: Types of System Calls**

- Process control
  - create process, terminate process
  - load, execute
  - get process attributes, set process attributes
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices
- Protection
  - get file permissions
  - set file permissions

2.5: Linkers and Loaders

- Usually, a program will reside on a disk as a binary executable file
- In order to run on a CPU, the process must be brought into memory and then placed in the context of a process
- Source files of program scripts must first be compiled into object files which are designed to be loaded into any physical memory location
- Next, the *linker* will combine these object files into a single binary executable file
- During this linking phase, other object files or libraries, such as the standard C or math libraries
- A *loader* is then used to load the binary executable file into memory, where it is eligible to run on a CPU core

2.6: Why Applications are Operating-System Specific

- On a fundamental level, applications that are compiled on one operating system are not executable on other operating systems
- There are ways, however to make an application executable on multiple operating systems
    - The application can be written in an *interpreted language* such as Python, which has an interpreter available for multiple operating systems
    - The application can be written in a language that includes a virtual machine containing the running application, such as Java and the Java Virtual Machine
    - The application can be *ported* or translated by a developer to be able to run on multiple operating systems using a standard API
- The specificity of programs as far as operating systems comes down to three main factors
    - Each operating system has a binary format for applications that dictates the layout of thew header, instructions, and variables, which must all be in the proper location to ensure proper execution
    - Different CPUs have varying instruction sets, and only applications containing the appropriate instructions can execute correctly
    - System calls that are implemented directly in a program, or through the use of a standard API, vary from operating system to operating system

2.7: Operating-System Design and Implementation

- **2.7.1: Design Goals**
    - Design goals for an operating system can be fundamentally divided into *user goals* and *system goals*
    - There is no unique definition for what these goals should be since computers are used in such a wide variety of ways and such a wide variety of operating systems exists
- **2.7.2: Mechanisms and Policies**
    - One important principle in operating system design is the separation of *policy* from *mechanism*
    - *Mechanisms* determine how something will be done, whereas *policies* determine what will be done

    - For example, the timer concept is a mechanism, but the duration to which the timer is set for each user is a policy decision
    - Separating these two is an important concept when it comes to flexibility
    - Since policies are likely to change across time or place, it would be ideal for mechanisms to be able to adapt
    - The worst case scenario would be each policy change requiring an underlying mechanism change
    - A general mechanism, which is flexible enough to work across a range of policies is preferable
- **2.7.3: Implementation**
    - Once an operating system has been designed, it must be implemented
    - Because of the nature of operating systems and the different ways in which they are designed, it is difficult to make general statements about operating system implementation
    - Early operating systems were written entirely in assembly language, but nowadays most are written in higher level languages like C or C++, with small amounts of the system implemented using assembly language
    - Using higher level languages in operating system development provides many of the same advantages as it does in application development
        - Code is easier to write, understand, and debug
        - Code is far more compact
        - Code is easier to rework in order to run on various computer hardware
    - The only potential disadvantages that come with implementing an operating system using higher level languages are reduced speed and increased storage requirements due to the need to 'translate' or compile high-level languages to machine code

2.8: Operating System Structure

- This section will discuss how the common components of an operating system are interconnected and melded into a kernel
-

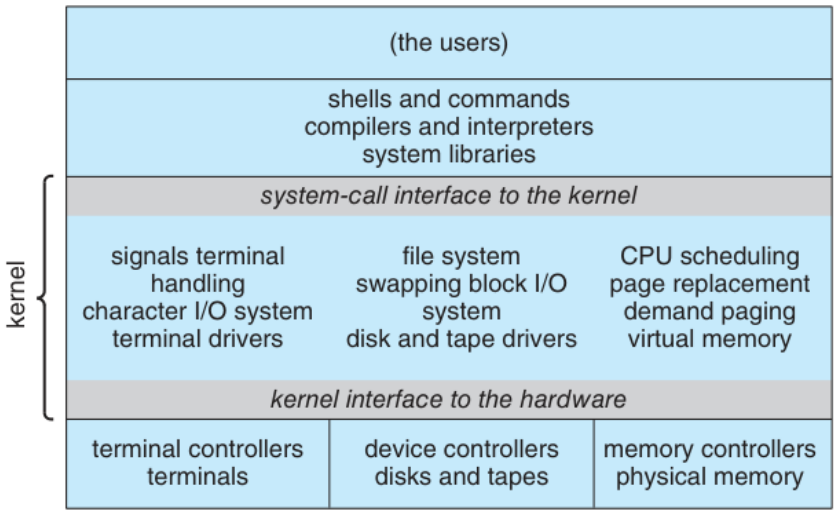


Figure 2.12 Traditional UNIX system structure.

- **2.8.1: Monolithic Structure**
  - A monolithic structure is the simplest structure an operating system can take, owing to the fact that a monolithic system has no structure at all
  - The entire functionality of the kernel is placed into a single, static binary file that runs in a single address space
  - This is a common technique for designing operating systems
  - Monolithic kernels are difficult to implement and extend, but they do have a distinct performance advantage since there is very little performance overhead in the system-call interface and communication with the kernel is very fast \
  - The monolithic approach is often known as a *tightly coupled* system since changes to one part of the system can have wide-ranging effects on other parts of the system
- **2.8.2: Layered Approach**
  - Alternatively, we could design a *loosely coupled system* such that the system is divided into separate, smaller components that have specific and limited functionality
  - One method for designing a loosely coupled system is the layered approach
  - The operating system is broken down into a number of layers, the lowest of which (layer 0) is the computer hardware, and the highest of which (layer n) is the user interface
  - The main advantage of the layered approach is the simplicity of construction and debugging

        - The layers are selected such that each uses functions and services of only lower-level layers

- **2.8.3: Microkernels**
    - As computing's history went on, the kernels that were being used became large and unwieldy as more and more functionality was added to operating systems
    - Thus, microkernels have evolved, or kernels from which all nonessential components are separated and implemented as user level programs that reside in separate address spaces
    - One large benefit of the microkernel structure is that it makes extending the operating system easier since new services are added to the user space and consequently do not require modifications to be made of the kernel
- **2.8.4: Modules**
    - Currently, the best methodology for operating-system design involves using *loadable kernel modules*, where the kernel has a set of core components and can also link in additional services using modules, either at boot time, or at run time
    - This is similar to a layered approach, but it is more flexible in that any module can call any other modules rather than layers only being able to call lower-level layers
- **2.8.5: Hybrid Systems**
    - In practice, there are very few operating systems which strictly adopt one of the previously examined structures
    - Instead, most modern operating systems combine different structures, which results in a variety of *hybrid systems*

2.9: Building and Booting an Operating System

- **2.9.1: Operating-System Generation**
    - Usually, when a computer is purchased, it has an operating system already installed
    - However, if you want to change operating systems, one option you have is *generating* an operating system, which will consist of the following steps
        - Write the operating system source code, or obtain previously written source code
        - Configure the operating system for the hardware on which it will run

        - Compile the operating system
        - Install the operating system
        - Boot the computer and its new operating system
    - You can also download an .iso file which contains a compiled version of general purpose operating systems, which can be flashed onto a USB drive or optical disc and installed onto a computer

- **2.9.2: System Boot**
    - The process of starting a computer by loading the kernel is known as *booting* the system
    - On most systems, the boot procedure is as follows
        - A small piece of code known as the *bootstrap program* or *boot loader* locates the kernel
        - The kernel is loaded into memory and started
        - The kernel initializes hardware
        - The root file system is mounted
    - Many modern computers use a *multistage* boot process
        - When the computer is first powered on, a small boot loader located in nonvolatile firmware known as *BIOS* is run
        - This initial boot loader does nothing more than load a second boot loader, which is located at a fixed disk location called the *boot block*
    - Recently, many computer systems have replaced the BIOS-based boot process with *UEFI* or *Unified Extensible Firmware Interface*, which provides more modern support, as well as the advantage of being a single, complete boot manager and therefore is faster than the multistage BIOS boot process