Part I - Overview

Leonardo Rodrigues

September 1, 2019

1 Introduction

Operating systems are intermediates between users and the hardware. Its purpose is to provide an enviroment where the user can execute programs in a convenient and efficient manner; it's a software that manages hardware. Hardware must provides mechanisms to avoid interferences from user programs, and ensure correct operations for the system.

1.1 What operating systems do

They control the hardware and coordinates its usage by the various application programs between various users. We can define 2 viewpoints.

User view Varies according to interface being used. Ease to use it's often privileged over performance and almost none attention is given to resource allocation. Some computers have no user view, and it's interface is basically done by keyboards and lights.

Software view It works as an resourse allocator; it's also a control program. Avoid errors and proper usage of resources, operating and controlling I/O devices

Definition There is no universally accepted definition of what a operating system is; a common definition for operating systems is that *it's the one program running at any time*. It's also called the *kernel*.

1.2 Organization

1.2.1 Operation

We have 1+ CPUS, and n device controllers, all connected by a bus to a shared memory.

We have a bootstrap software, also called the *firmware* that initialize CPU and devices values, and loads the OS into memory. OS then start it's own processes, together with any daemons, and start to listen to events.

Those events come in the shape of *interrupts*. There are 2 types of interrupts: hardware interrupts, that are naturally processed by the OS, and software interrupts, that are also called *system calls*. Those interrupts transfer control to the suited interrupt service routine.

That's a routine table kept with addresses to it's code so this control transfer can be done as fast as possible. There is also a interrupt vector that keeps an index of information about interrupting devices together with its instruction, and also the address of return after the interrupt is complete. Interrupting code must save CPU status before modifying it and restore its status before returning.

1.2.2 Storage

The CPU can load instructions only from memory, and ideally, all programs to run and data should be stored there; but the main memory is limited and volatile, and we also have different approaches to persisting data that is useful. All types of memory provide an array of bytes, each byte having its own address. Interaction between those and CPU are made by `load` or `store` instructions. This follows *von Neumann* structure. Memory unit normally only sees a stream of memory addresses.

These various storage approaches vary in:

speed the technology used to make counts toward its speed; the fastest are CPU registers, as they are made with semiconductor material, the same of the processor therefore being as fast as it can.

cost normally faster and volatile are more expensive; we have a trade-off between speed, size and volatility with cost.

size faster and volatile memories are also smaller in capacity.

volatility volatile storage loses its content when power is removed.

A complete memory system should provide a good balance between the factors above. Provide as much expensive memory as necessary together with as much inexpensive memory. We can improve performance by implementing *caching* strategy.

1.2.3 I/O

Storage is only one of many types of I/O in a computer system. A large part of OS is actually dedicated to managing I/O, because they are crucial for the reliability and performance of the system, and also because the varying nature of devices.

A computer system normally consists of a CPUs and multiple device controllers, connected through a common bus. Each device controller is responsible for a specific device, and operating systems have a *driver* for each device controller.

To perform an operation, the controller starts by loading the appropriate registers within its controller; the controller examines the contents of its registers and store in the internal buffer, and when it's complete informs the device driver via interrupt that the operation is finished. This approach is good for small operations, but larger operations use a strategy called *Direct Memory Access* so the data transfer is made in big blocks, generating only one interrupt for each block, and most of the data transfer is done without CPU intervention.

1.3 Architecture

1.3.1 Single processor systems

When we have only one general purpose CPU, with a general purpose instruction set, supported by many special purpose processors f.i. I/O processors. The usage of multiple special purpose processors doesn't turn the system into a multiprocessor system, as we still have only a single general purpose CPU.

1.3.2 Multiprocessor systems

Also known as *parallel* or *multicore* systems; have 2+ processors in close communication, providing as advantage:

1. Increased throughput
2. Economy of scale
3. Increased realibility

Multiprocessor systems can be implemented in a assymetric multiprocessing architecture, where each processor is assigned a task by a boss processor; or in symmetric multiprocessing architecture, in which each processor is a peer for the operating system operations; all processors share physical memory. This provides a good performance boost considering that n processors can process n processes, but we must implement this architecture carefully to avoid inefficiencies and to ensure I/O operations are handled by the appropriate designated processor.

We also have a modern approach in which a processor is divided in multiple cores. This approach can be more efficient because on-chip communication is faster than between-chip communication; also, a chip with multiple cores spend less energy than multiple single-core chips.

Multiprocessing can cause the system to change its memory access model from *uniform memory access* to *non-uniform memory acces*. UMA is defined by the situation in which access time to RAM is uniform from any CPU, while in NUMA sections of memory may take longer to access, creating a performance penalty. This penalty can be minimized through resource management.

1.3.3 Clustered systems

Clusters consists of several computer systems connected through a network; those systems can be either single or multiple processor. Clusters can provide *high performance computing*, using a technique called *parallelization*, which divides a program into separate components that run in parallel on individual computers. Parallel clusters can also allow multiple hosts to access the same data on shared storage.

1.4 Structure

To perform its jobs, operating systems share some concepts. As operating systems are diverse, those concepts are still present, and are consolidated among them.

multiprogramming One of the basic concepts of any operating system; multiprogramming increases CPU utilization by organizing jobs so that CPU always have something to do. The idea is to keep processes (process is a program loaded into memory with data and settings) in the main memory, and eventually one of them will need to be interrutped for I/O, which the OS switch which process has access to CPU resources. As the number of processes increases, we can have the main memory exausted of space, in which we can then use the strategy of *job pool*,

job pool Keeping processes allocated in disk until they are needed, when then they'll be allocated in memory.

time sharing Also called *multitasking*, it's a logical extension of multiprogramming. Basically, it's the capabitlity of having multiple users at once in the same system, in a transparent way. This way, the OS not only can serve multiple processes, but also multiple users.

job scheduling When multiple jobs are ready to be loaded into memory, the OS must choose among them. This is called job scheduling.

CPU scheduling When multiple processes are ready to be executed, then the OS must choose among them as well. This is called CPU scheduling.

swapping When the OS must choose which processes are to be kept in memory. Sometimes processes are not needed for a long time, and the OS then *swaps* them into disk, to be brought back to memory when relevant again.

virtual memory Another approach to handle memory exhaustion, this allows a process to be executed even when it's not fully in memory. This enables users to run programs that are larger than the *physical memory*, abstracting into a large *logical memory*.

Operating systems also must provide *file system*, that in your turn requires *disk management*. Time-sharing systems also must provide protection agains

inappropriate resource usage, mechanisms for *job sinchronization* and *job communication*, and ensure that jobs don't get stuck on *deadlocks*, forever waiting on one another.

1.5 Operations

Modern systems are *interrupt-driven*. This means that sharing resources is a task the OS must do, otherwise a process can monopolize CPU time. Without protection, the computer would have to run only one process at a time, or to suspect from each output.

1.5.1 Dual mode and multimode operation

This technique defines 2 modes of operation: *user mode* and *kernel/supervisor/privilege mode*. Then, user processes that run in user mode only have access to a set of instructions. This set of instructions is a subset of all instructions a CPU can handle, meanwhile the OS, that runs in kernel mode, have access to the entire set. When a user process wants to perform any operation, it must use the *system calls* that are handled by the kernel. A system call takes form of a trap to a specific location in the interrupt vector. Dual-mode support must be guaranteed by hardware, as it's implemented as a *mode bit* in the CPU. As such, only the kernel have the privilege to change this mode bit.

One modern concept is the *virtual machine manager*, that can have a dedicated mode that is above user mode but below kernel mode, so it can perform some CPU instructions.

We can also have *levels of privilege* in which different kernel processes run in different privilege levels, each being a subset of allowed instructions.

1.5.2 Timer

The timer is a strategy to ensure control is given back to the system. It is implemented by a fixed rate clock and a counter. After a set amount of clock ticks, a interrupt is triggered to give back control to the system. We can also set processes to have a time limit, and when this limit is expired, the control is given back to the system and the process is terminated for exceding the time. Instructions that modify the content of the timer are privileged.

1.6 Process management

A program does nothing until its instructions are executed by the CPU. A program loaded in memory, together with its context is called *process*, and multiple processes can be created from the same program. Processes are sequential, which means that every instruction is run after another until the process is completed. When the process is *multithreaded*, each thread has an unique program counter, each pointing to the next instruction to be executed for a given thread.

A process is the work unit of the system. A system is composed by a collection of processes, that can be system processes and user processes. All these

processes can be executed concurrently, and this is called *multiplexing*. Multiplexing means sharing of time(CPU usage) and space(memory).

The OS is responsible for these tasks regarding process management:

1. Process and thread scheduling
2. Creating and deletion of processes
3. Suspending and resuming processes
4. Provide mechanisms for process synchronization
5. Provide mechanisms for process communication

1.7 Memory management

Each program must have its instructions loaded to memory to be run, and then the CPU will fetch those instructions to complete the program execution. Memory is an array of bytes, each with its own unique address; so, for programs to execute, its instructions and data must be mapped into memory with those *absolute addresses*. When it is terminated, its space of memory is declared available.

The OS then is responsible for these tasks regarding memory management:

1. Track parts of memory that are being used and who is using them
2. Decide processes (and its parts) that move into and out of memory
3. Allocate and deallocate memory space as needed

1.8 Storage management

The operating system abstracts from the physical properties of its storage devices to define a logical storage unit called *file*. The OS then maps files onto physical media, and access those files via the storage devices.

1.8.1 File system management

File management is one of the most visible parts of the system. Computers can store information into a multiple type of physical media, each with its own characteristics. What the OS do is to abstract those medias with directories and files, where the information is stored. This information can be pure data or programs. When multiple users have access to files, then it's good to have control over if an user can access a file, and in which forms they can access it(read, write, append).

The OS then is responsible for these tasks regarding file system management:

1. Creating and deleting files
2. Creating and deleting directories

3. Supporting primitives for file and directory manipulation
4. Mapping files onto storage
5. Backing up files on stable storage media

1.8.2 Mass storage management

The management of storage medias can be a part of the system. Secondary memory, that means the disk, is of central importance of the system; tertiary storage devices can or can not be managed by the system, and the system can provide some functions like mounting, allocation and migration from secondary to tertiary devices.

The OS then is responsible for these tasks regarding mass storage management:

1. Free-space management
2. Storage allocation
3. Disk scheduling

1.8.3 Caching

Cache is an abstract concept in which we have multiple copies of the same information being held in multiple levels for efficiency. One of the implementations of cache is moving information that is in main memory to the CPU cache, as it is faster than main memory, so if that information is required again, we can check it in the CPU cache, and it will be accessible faster. Otherwise, the CPU would wait several instruction cycles to fetch it from the main memory.

As the concept of cache is abstract, one could say that the main memory serves as a cache for the disk, for instance. Caches have limited size, so it's crucial for the system to handle *cache management*. In a hierarchical storage structure, the same data may appear in different levels of the storage system. In the environment where only one process execute at a time is easy to do this management, but when you have a *multitasking environment*, multiple processes can be accessing the same information, the system must ensure that each access is made with the most updated value. When we have a *multiprocessing environment*, then we have multiple caches that must be *coherent* with each other as well, but *cache coherency* is normally a hardware issue. In a *distributed environment* it's when we have the most complex problem. Several copies of the same information are kept, and we have to handle an update in one of those copies in a way that the other copies are updated as soon as possible.

1.8.4 I/O systems

The I/O subsystem consists of:

1. A memory-management component that includes buffering, caching, and spooling
2. A general device-driver interface
3. Drivers for specific hardware devices

1.9 Protection and security

protection any mechanism that control the access of processes and users to the resources of the computer.

security mechanisms that protect the system from external and internal attacks that could use system resources and keep out legitimate users of the system

user identifiers unique ID for an user to distinguish they in the system

group identifiers unique ID for a set of individual users

1.10 Kernel Data Structures

1.10.1 Lists, Stacks and Queues

array collection of data of same type and size; each item can be accessed directly

list collection of data values as a sequence

singly linked list each item points to its successor

doubly linked list each item points to its successor and predecessor

circular linked list the last element in the list points to the first element

stack sequentially ordered data structure that uses the *last in, first out(LIFO)* principle of adding and removing items; the operations of adding into and removing from a stack are `push` and `pop`, respectively. An example of implementation are function calls.

queue sequentially ordered data structure that uses the *first in, first out(FIFO)* principle of adding and removing items. An example of implementation are jobs spooling.

1.10.2 Trees

Data structure to represent data hierarchically; data values are linked through parent-child relationships.

general tree a parent may have unlimited number of childen

binary tree a parent may have at most 2 children: left child and right child

binary search tree we define values for the children: $leftchild <= rightchild$

balanced binary search tree a tree containing n items must have at most $\lg n$ levels.

1.10.3 Hash Functions and Maps

hash function function that takes data as input and perform a numeric operation, returning a numeric value; this value can be used as an index into a table(array) to quickly retrieve its data.

hash map usage of a hash function for mapping pairs of `[key: value]`; once the mapping is established we can then retrieve the `value` applying the hash function to the `key`

1.10.4 Bitmaps

String of n binary bits that represent the status of n items; an implementation is in disk management, where a bitmap represents the availability of disk blocks.

1.11 Computing Environments

1.11.1 Traditional computing

1.11.2 Mobile computing

1.11.3 Distributed systems

1.11.4 Client-Server computing

1.11.5 Peer-to-Peer computing

1.11.6 Virtualization

1.11.7 Cloud computing

1.11.8 Real-time embedded systems

1.12 Open-Source Operating Systems

1.12.1 Brief history

1.12.2 Linux

1.12.3 BSD Unix

1.12.4 Solaris

1.12.5 Open-source Systems as Learning Tools