CS 646: Homework-4

Prithul Sarker

Essay on operating system design choices:

An operating system is a framework that enables user application programs to communicate with system hardware. The operating system does not perform any functions on its own, but rather creates an environment in which various apps and programs can do valuable tasks.

Many issues might arise during the design and implementation of an operating system. It is quite complicated to define all the goals and specifications of the operating system while designing it. The design changes depending on the type of the operating system i.e., if it is batch system, time shared system, single user system, multi-user system, distributed system etc.

There are basically two types of goals while designing an operating system.

User Goals: According to the users, the operating system should be convenient, easy to use, dependable, safe, and fast. These standards, on the other hand, are useless because there is no established procedure for achieving these goals.

System Goals: The operating system should be simple to create, deploy, and maintain. These are the requirements for individuals who design, maintain, and use the operating system. However, there is no single strategy for achieving these objectives.

In the following sections, we'll look at several methods for boosting performance in situations where it's needed.

Optimization: Optimizations should be made based on these figures where they will help the most. It is usually not worth the work and complexity to squeeze out the remaining few percent after the performance has reached an acceptable level. Jumping through hoops to achieve ideal performance is usually not worth it if the page rate is low enough that it is not a bottleneck. It is considerably more vital to avoid disaster than to achieve peak performance. Another issue to consider is premature optimization or deciding what to optimize and when. The problem is that after optimization, the system may be less clean, making it harder to maintain and debug.

Space-Time Tradeoffs: The trade-off of time vs. space is one general strategy to enhancing performance. In computer science, it is common to have to choose between an algorithm that uses minimal memory but is slow and an approach that requires a lot of memory but is faster. When making a significant optimization, seek for algorithms that increase speed by consuming more memory or, conversely, conserve valuable memory by performing more work.
Small routines can be replaced using macros, which is an approach that can be useful at times. The overhead associated with a procedure call is eliminated when using a macro. If the call is made inside a loop, the advantage is even greater. For example, PostScript is such a trade-off which is a programming language that can be used to describe images. Actually, any programming language can describe images, but PostScript is tuned for this purpose. Many printers have a PostScript interpreter built into them to be able to run PostScript programs sent to them. Data structures are frequently involved in other trade-offs. Although doubly linked lists consume more memory than singly linked lists, they frequently provide faster access to entries. Hash tables waste even more space but are even faster. In brief, while optimizing a piece of code, one of the most important things to examine is if adopting other data structures will result in the optimum time-space trade-off.

Project Management: Even before coding begins on a major project, a significant amount of time is spent considering how to divide the work into modules, carefully specifying the modules and their interfaces, and attempting to envisage how the modules will interact. The modules must then be coded and debugged separately. Finally, the modules must be connected, and the entire system must be tested. When each module is tested separately, it usually works OK, but when all the pieces are placed together, the system crashes.

Two things are to be carefully chosen: 1. Team Structure: Any large project needs to be organized as a hierarchy. In practice, large companies, which have had long experience producing software and know what happens if it is produced haphazardly, tend to at least try to do it right. 2. The Role of Experience: Having experienced designers is absolutely critical to any software project. Since most of the errors are not in the code, but in the design.

The details of design choices based on the performance are described below:

User interface Paradigm

Many GUIs for desktop machines use the WIMP (Windows Icons Menus and Pointers). To offer architectural coherence to the entire interface, this paradigm employs point-and-click, point-and-double-click, dragging, and other idioms throughout. Having a menu bar with FILE, EDIT, and other entries, each of which includes certain well-known menu options, is a common need for programs. Users who are familiar with one software can quickly learn another.

The WIMP user interface isn't the only one that may be used. Touch screens are used in tablets, smartphones, and some computers to allow users to interact with the device more directly and intuitively. A stylized handwriting interface is used on several palmtop computers. A VCR-like interface may be used on dedicated multimedia devices. Voice input, on the other hand, follows an entirely different paradigm. The fact that there is a single overarching paradigm that unifies the entire user interface is more significant than the paradigm chosen.

It is critical that whatever paradigm is chosen, all application applications use it. As a result, system designers must supply libraries and toolkits to application developers that allow them to access methods that provide a consistent look and feel. Application developers will all do something different if they don't have access to tools.

Execution Paradigm

Algorithmic and Event Driven Paradigms are the two types of execution paradigms. The algorithmic paradigm is based on a previously known program or a program that is run using parameters. When an event occurs, event driven paradigms often run.

The algorithmic paradigm is founded on the assumption that a program is begun to execute a function that it knows ahead of time or that it can infer from its parameters. It could be compiling a program, processing payroll, or flying an airplane to San Francisco. The software makes system calls from time to time to acquire user input, obtain operating system services, and so on, with the essential logic hardwired into the code.

The event-driven paradigm is the alternative execution paradigm. The program begins by performing some kind of initialization, such as displaying a specific screen, and then waits for the operating system to notify it of the first event. A keystroke or a mouse movement is frequently the trigger. This layout is ideal for apps that require a lot of interaction. Each of these models produces

its own programming style. Algorithms are at the heart of the algorithmic paradigm, and the operating system is viewed as a service provider. The operating system also provides services under the event-driven paradigm, but this role is overshadowed by its position as a coordinator of user activities and a creator of events that processes consume.

System structure

It is better that operating systems have a modular structure, unlike MS-DOS. That would lead to greater control over the computer system and its various applications. The modular structure would also allow the programmers to hide information as required and implement internal routines as they see fit without changing the outer specifications.

One way to achieve modularity in the operating system is the layered approach. In this, the bottom layer is the hardware, and the topmost layer is the user interface. Each upper layer is built on the bottom layer. All the layers hide some structures, operations etc. from their upper layers.

The main difference between monolithic and layered operating systems is that, in monolithic operating systems, the entire operating system work in the kernel space while layered operating systems have a number of layers, each performing different tasks. Layering makes it easier to enhance the operating system as implementation of a layer can be changed easily without affecting the other layers. It is very easy to perform debugging and system verification. However, in this structure the application performance is degraded as compared to simple structure and it requires careful planning for designing the layers as higher layers use the functionalities of only the lower layers.

Binding time

Programming languages often support multiple binding times for variables. Global variables are bound to a particular virtual address by the compiler. This exemplifies early binding. Variables local to a procedure are assigned a virtual address (on the stack) at the time the procedure is invoked. This is intermediate binding. Variables stored on the heap (those allocated by malloc in C or new in Java) are assigned virtual addresses only at the time they are actually used. Here we have late binding.

Operating systems often use early binding for most data structures, but occasionally use late binding for flexibility. Memory allocation is a case in point. Early multiprogramming systems on machines lacking address-relocation hardware had to load a program at some memory address and relocate it to run there. If it was ever swapped out, it had to be brought back at the same memory address or it would fail. In contrast, paged virtual memory is a form of late binding. The actual physical address corresponding to a given virtual address is not known until the page is touched and actually brought into memory.

Another example of late binding is window placement in a GUI. In contrast to the early graphical systems, in which the programmer had to specify the absolute screen coordinates for all images on the screen, in modern GUIs the software uses coordinates relative to the window's origin, but that is not determined until the window is put on the screen, and it may even be changed later.

Data Structure type

The kernel stores and organizes a lot of information. Operating system designers are constantly forced to choose between static and dynamic data structures. Static ones are always simpler to understand, easier to program, and faster in use; dynamic ones are more flexible. Early systems simply allocated a fixed array of per-process structures. So, it has data about which processes are running in the system, their memory requirements, files in use etc. To handle all this, three important structures are used. These are process table, file table and v node/ i node information.

Process Table: The process table stores information about all the processes running in the system. These include the storage information, execution status, file information etc. When a process forks a child, its entry in the process table is duplicated including the file information and file pointers. So the parent and the child process share a file.

File Table: The file table contains entries about all the files in the system. If two or more processes use the same file, then they contain the same file information and the file descriptor number. Each file table entry contains information about the file such as file status (file read or file write), file offset etc. The file offset specifies the position for next read or write into the file. The file table also contains v-node and i-node pointers which point to the virtual node and index node respectively. These nodes contain information on how to read a file.

V-Node and I-Node Tables: Both the v-node and i-node are references to the storage system of the file and the storage mechanisms. They connect the hardware to the software. The v-node is an abstract concept that defines the method to access file data without worrying about the actual structure of the system. The i-node specifies file access information like file storage device, read/write procedures etc.

Implementation

While it is preferable to design the system from the top down, it can theoretically be implemented from the bottom up. The implementers of a top-down approach start with the system-call handlers and see what mechanisms and data structures are required to support them. Until the hardware is achieved, these methods are written, and so on. The difficulty with this technique is that using only top-level procedures makes it difficult to test anything. As a result, many developers believe it is more realistic to construct the system from the ground up. The first step in this method is to write code that hides the low-level hardware. Early on, interrupt handling and the clock driver are also required.

Then multiprogramming can be tackled, along with a simple scheduler (e.g., round-robin scheduling). At this point it should be possible to test the system to see if it can run multiple processes correctly. If it works, it's time to start carefully defining the various tables and data structures required throughout the system, particularly those for process and thread management, as well as memory management later on. Except for a crude way to read the keyboard and write to the screen for testing and debugging, I/O and the file system can wait at first. In some circumstances, essential low-level data structures should be secured by limiting access to specific access procedures—in other words, object-oriented programming, regardless of programming language. Lower layers can be thoroughly checked as they are completed. In this approach, the system grows from the ground up, similar to how contractors construct towering office buildings.

Communication

Some systems, such as Amoeba, embrace synchronous design and use blocking client-server calls to communicate between processes. The concept of fully synchronous communication is quite basic. A process sends out a request and then waits for a response. When there are a lot of clients clamoring for the server's attention, things get a little more challenging. Each request may become stuck for a lengthy period while waiting for other requests to finish. Making the server multi-threaded such that each thread can handle one client solves this problem. The paradigm has been tried and proven in a variety of real-world applications, including operating systems and user applications.

If the threads often access and write shared data structures, things get even more difficult. Locking is unavoidable in that situation. Regrettably, getting the locks just right isn't easy. The most straightforward option is to apply a single large lock to all shared data structures (similar to the big kernel lock). When a thread wishes to access shared data structures, it must first acquire the lock. A single large lock is terrible for performance since threads wind up waiting for each other all the time, even if they don't clash. The other extreme, a large number of micro locks for (parts) of individual data structures, is significantly faster, but it goes against our first goal– “simplicity”.

Finally, if we can make the optimal choices as we made in each section and make them coordinate properly with each other, a better performing operating system can be built.