Lecture Notes - Introduction

Debadatta Mishra
Indian Institute of Technology Kanpur

Operating System: Motivation

Modern day computers can execute many applications concurrently sharing underlying physical resources like CPU, Memory and I/O devices. Moreover, new applications can be developed, deployed and used with ease. An intriguing question is, how a complex software system that we use today comes into being on physical computing devices like computers and mobiles? In this course we will try to find answers to the above question.

Let us visualize a physical machine and try to connect it to the way we use computers today. A physical computer, in a broad sense, can be thought as collection of processing units, storage units, other logic circuits and Interconnects. How can this complex interconnected hardware be programmed? Does every programmer need to know the intricate details of an ALU? These questions are addressed using a very well known principle in computer science known as abstraction. The hardware intricacies in terms of logic, communication etc. are hidden by an abstraction, called the Instruction Set Architecture (ISA). ISA is an exposure of operations and machine state to the software to operate the hardware. For example, the complexities of instruction execution (fetch, decode etc.) is hidden behind a simple architectural interface called the program counter (PC) register. The CPU uses the address stored in PC to fetch and execute the instruction. As it is easy to develop applications in high level languages, compilers perform the translation of high level language to instructions supported by the ISA.

At this point it may seem that we have everything we need to develop, deploy and use applications on a physical computer---we write a program, the compiler translates it to equivalent machine code and the hardware executes it. However, if we carefully analyse a simple application (e.g., finding the sum of all elements in an array), the following points may highlight some gaps.

- The PC register must be set to the address of the first instruction to be executed.
- The program is required to store and access the array elements but the programmer does not know whether a given memory area is free or used by any other application.
- The program reads array elements from the user through keyboard and displays the output on the monitor after successful completion of the program.
- If you observe our computers today, we do not download applications from some other machine before using it, because they are stored in a persistent storage like hard disk. By implication, the program's location in hard disk must be identified and it must be loaded into memory so that the processor can execute it.

To answer some of the points raised above, it may be useful to understand the boot sequence of a typical computer system. A simplified system boot sequence is as follows,

- On system reset, the PC points to the first instruction in the BIOS code (which resides in ROM).
- The BIOS loader loads the content of boot sector from the hard disk to RAM.
- Finally, the BIOS sets the PC to the first instruction.

With this information in the back of our minds, let us try to answer some of the questions raised earlier.

Application load and execution

- Let the BIOS load applications from disk. Several issues with this approach--- applications can be executed one after the other, a machine restart is required, applications must be placed in boot sector etc.
- Let BIOS load a *special application* (say **K**) which in turn loads other applications. This special application must have access to all the resources and should have the capability to locate and load other applications.

Input/Output

- All applications independently implement their own I/O operations. That means each application developer should know the exact low-level I/O interfacing mechanisms.
- Alternatively, a standard library for I/O operations is developed which can be used by all applications which means the library code become part of the executable during execution. But, if some applications bypass these libraries and access the resources directly, they may impact other applications. For example, if one application bypasses the library implementing access to a network device, it may overwrite network packets of another application (accidentally or with malicious intentions).
- Another alternative is to allow the special application **K** to provide interfaces such that, all applications mandatorily access the I/O device through K. Advantages of such an approach are,
    - Applications can use the same resource without impacting each other
    - Resources can be managed efficiently

  However, the challenge is to devise a technique such that the special application **K** can disallow direct access of I/O devices from applications.

Resource sharing (multiplexing) across multiple applications

In the array sum program (mentioned before), one of the difficulty was to find free memory for storing the array elements without encroaching on memory used by other applications. The problem of resource sharing is generic across resources like memory, I/O and CPU. Approaches of resource sharing can be,

- No multiplexing, exclusive use: wasteful, resource underutilization.
- Cooperating multiplexing can be another approach where applications communicate with each other regarding their resource usage and share the resource. The problem with this approach are two fold---(i) all applications must cooperate, (ii) all applications should communicate and know about each other, which can result in large overheads.
- The application **K** can multiplex a resource across multiple applications, providing guarantees of non-overlapping resource usage along with maximum possible resource utilization. However, not allowing direct access of resources from the

applications is required to realize such a solution. Furthermore, **K** must provide simple and efficient APIs to applications so that the resources can be efficiently used by the applications.

Efficient resource management

- If applications are allowed to allocate resources to themselves, they may not make resource management decisions to achieve the *global resource efficiency*. Further, *fairness* can be another problem, where some applications dominate the resource usage while others starve for resources.
- The application **K** can manage the resources in accordance to policies mandated by system administrators or according to quality of service guarantees or any other suitable objective. By implication, K should control all the resources in order to take efficient management decisions at different points of time.

What is OS?

By now it should be clear that **K** actually is the operating system, a system software with the following properties,

- Operating system is a middleware between applications and hardware which provides simple and easy to use API by leveraging the hardware features in an efficient manner.
- Operating system acts as a resource multiplexer by enabling resource sharing across multiple applications.
- Operating system also manages resources in accordance to high level policy objectives like resource efficiency and fairness while providing support for flexible policy design.

Resource multiplexing

Orchestrating resource partitioning across multiple applications is one of the primary responsibilities of an operating system. Below listed are the commonly used resource partitioning schemes in operating systems.

- *Time sharing*: Different applications use the resource at different points of time. Many applications may own and relinquish the resources multiple times during their respective life times. Therefore, for applications to progress, state of the resource should be saved every time the application relinquishes the resource and restored when it starts using the resource again. CPU is an example of such a resource.
- *Space sharing:* Different applications use different parts/chunks of resource simultaneously without encroaching on each other's resource chunk. This requires partitioning support from the resource e.g., volatile memory (RAM).
- *Software (OS) enabled sharing:* Many resources are suitable for neither time sharing nor space sharing because the resources do not support the required features discussed above. In such a scenario, operating system provides interfaces to applications to access to the resource. For example, a disk storage device is used by applications through OS enabled file system interfaces.

While partitioning resources across multiple applications, following requirements should be considered.

- *Isolation:* Resources when used by one application (say A) should not be accessible from other applications, if not specifically allowed by A. The question is how to achieve this? One solution is to monitor every access of the resource by application at the OS level which will result in a lot of overheads. Other approach is partition the resource using techniques discussed above (if possible) and let the program use it directly. With this scheme, applications should not be allowed to modify partitioning information, otherwise the isolation property is violated.
- *Resource control*: The OS can take control of any resource allocated to any application at any point of time. This property is essential for efficient management of resources. For example, If an application is not using all of its allocated memory, the OS should have a way to take control of the unused memory (may be allocate to some other application). If accesses are through the OS, resource control is obvious. However, with a partitioned scheme, OS requires intervention points to gain control of the resources. For example, if an application is executing on a CPU for a long time while other applications are waiting, the OS must have some mechanism to gain control of CPU.
- *Efficiency:* As much as possible, applications should access the resources directly, without OS intervention. The challenging question is how to allow direct access, while at the same time achieve control and isolation?

Limited direct execution is a phenomenon where an application is allowed to execute and access resources without OS intervention most of the times. However, there are certain limits imposed to direct access behavior. Enforcing these limits using software techniques is in direct contradiction of efficiency. *Therefore there is a requirement for architectural support to impose the limits*.

Nature of architectural support

- Applications may not be allowed to execute certain instructions which allow to bypass the limited direct access mechanisms (directly or indirectly). For example, applications should not be allowed to execute the HLT instruction.
- Some instructions are allowed depending on the type of operands. For example, MOV instruction, used to copy the content of one register to another register/memory location is allowed in general, but not when MOV uses CR3 register as one of its operands. In some cases, these restrictions apply depending on the position of the operand, e.g., MOV from code segment register is allowed while MOV to code segment register is not allowed.
- Restrictions on application's access to resource is necessary, but OS should provide legitimate methods for applications to---relinquish allocated resources or request for additional resource allocation, as per the application requirements . For example, if an application wants to SLEEP, there it should be allowed to execute the HALT instruction by requesting the OS.

Architectural support for limited direct execution (X86)

- Intel X86 architecture supports four levels (rings) of privilege, i.e., 0,1,2,3 where ring-0 is the highest privilege level and ring-3 is the lowest. Most of the OSs use two privilege levels i.e., 0 and 3.
- At any point of time, the processor executes with one of the privilege levels,referred to as current privilege level (CPL).
- Switching between privilege levels is possible through architectural mechanisms. However, applications should be allowed to switch with OS enabled *gateways*, not in an arbitrarily manner.
- Limited direct execution discussed above is implemented using the privilege checks in hardware. For example, if an application (CPL=3) executes HLT, the underlying architecture raise a general protection fault.
- How does an application sleep? OS must provide gateways to perform privileged actions on behalf of the applications. These gateways are commonly known as system call API. For the specific case of sleep, the OS either executes halt (if no other application is ready) or schedules another application.

Entry into OS:

Apart from the system call, entry into highest privilege level (0) becomes necessary in the following situations,

- *External interrupts:* OS should handle external interrupts generated by I/O devices shared across multiple applications. This is primarily due to two reasons,
    - External events are asynchronous and can occur when the target application is not executing. In such a scenario, OS must handle the event and deliver the I/O event to the intended application. For example, if music player application is executing when a network packet for browser application is received, the packet should not be delivered to music player.
    - There are certain external events which provide execution control to the OS. For example, OS should handle periodic timer interrupt to gain control of CPU and decide on scheduling of applications.
- *Software caused faults:* OS should handle faults caused due to applications to,
    - Isolate the fault, so that other applications are not impacted.
    - Recover from error conditions by restoring to a pristine state.
    - Enhance resource utilization by performing lazy allocation e.g., page faults

Execution stack

- Execution stack is required to support function call and return, and provide allocation space for local variables.
- Operating system software also require stacks to implement system calls, handle exceptions and external interrupts. The OS should not use stacks of low privileged application due to isolation and security reasons.
- How many stacks are required to handle system calls and software caused exceptions?
    - One solution is to maintain a unique stack in the OS to handle system calls from all applications. The problem with this approach is that no two applications can enter OS at the same time.

    - Ideally, there should be one OS level stack for each execution context in the user level.
- How many stacks are required to handle external interrupts?
    - Using one OS stack for all interrupts disallows handling multiple interrupts
    - Using one OS stack for each interrupt is not flexible; consider a case when new devices are added to a computer.
    - Using one OS stack for each CPU works if each CPU handles one interrupt at a time. Even nested interrupts (interrupt during handling of an interrupt) can be handled using the same stack as the CPL does not change.