Lecture #7: Operating Systems

CS106E Spring 2018, Young

In this lecture we take a look at the Operating System (OS). The OS is a program which acts as a layer between application programs and the computer hardware.

We study how the Operating System allows us to run multiple programs simultaneously on both single and multiple CPUs. Programs can generally be written as if they are the only program running on a computer. The OS is responsible for sharing the CPU between different processes. Programs are also written as if they are the only program using computer memory. The OS handles the relationship between the program's simplified view of memory and the actual physical memory in the computer.

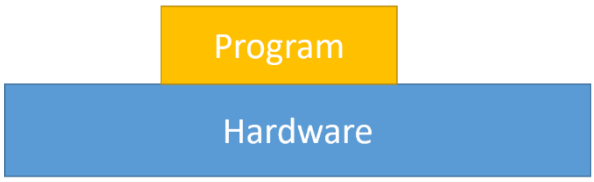
We also take a look at how multi-threaded programs allow us to take full advantage of multi-core CPUs, but also introduce some real perils for the programmer. Interaction between threads is non-deterministic, giving different results for the same input, depending on outside factors. This makes it difficult to find and debug errors.

The Need for Operating Systems

The Operating System is system software which acts as a layer between application software and the actual computer hardware. Let's start out by taking a look at why we need an operating system.

No Operating System

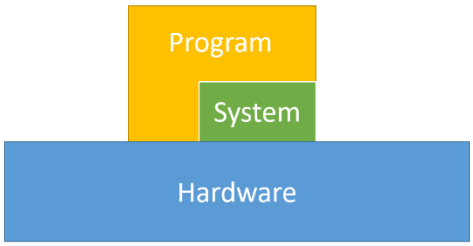
In the simplest case, there is no system software. The program runs directly on the computer hardware.



While this approach certainly works, as we saw previously in our CPU lecture, working directly with hardware can be very difficult. Just as we probably don't want to write machine language code to access the CPU directly, accessing Input/Ouput devices directly can also be difficult.

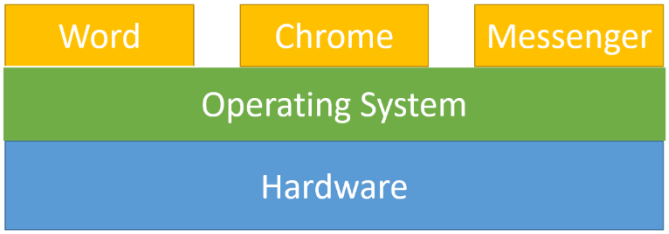
Hardware Abstraction

System software called device drivers provides application programmers with a higher, more abstract view of the Input/Ouput devices. We'll take a closer look at these later in the lecture.



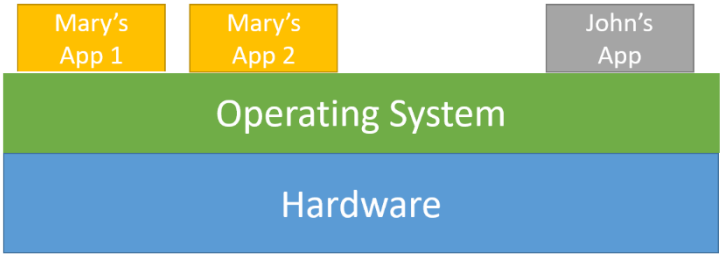
Supporting Multiple Applications

On desktop, laptop, and mobile devices, we all run multiple programs simultaneously. How is this feat done? The Operating System coordinates and controls access to the hardware for different applications.



Supporting Multiple Users

Computers not only support multiple applications, they also sometimes support multiple users. Whether allowing different users to have accounts on the same machine and controlling who can read which files, or going further and allowing different users to run applications simultaneously, the Operating System protects and controls who has access to which files and resources.



Operating System Summary

The Operating System or OS acts as an intermediary between end-user applications and the hardware. Its tasks include:

- Simplifying access to computer hardware and in particular input/output devices.
- Coordinating access to the CPU and hardware for programs which are running simultaneously.
- Providing protection to ensure that multiple applications do not accidently or deliberately interfere with each other.
- Providing user-level (and, in some cases, group-level) security to ensure that other users can only see files they have permission to access.

Also, while this doesn't necessarily follow from our discussion above, another important task of the OS is:

- It allows programs to be written as if they are alone on the computer, rather than sharing the CPU and memory with other programs.

Device Drivers
Device Drivers provide a higher-level abstraction of the actual underlying hardware. This concept of abstraction is something we've seen before and will see again. We can think of High-Level Languages as providing a higher-level of abstraction for the underlying CPU and Machine Language. We'll see a similar situation when we look at Computer Networks.

Processes and Threads
As we saw earlier, one important job of the Operating System is to allow multiple applications to run simultaneously. At a more technical level, we don't think about running multiple applications, instead we think of running multiple processes and multiple threads.

Processes correspond to programs. Each process has its own independent area of memory and has the all the sections of memory we studied in the previous lecture – with space set aside for instructions, global variables, and a heap for objects. In addition, it has one call stack per thread.

Threads are subordinate to Processes. In the simplest case, a process has a single thread of execution. However, as we discussed when we talked about multicore CPUs, to take full advantage of multiple cores, we need different parts of the program to run simultaneously. This is where threads come into play. Each thread executes a different set of instructions, and on a multicore CPU, two or more threads associated with the same process can run simultaneously.

While each Process has its own memory with its own global variables and objects, Threads are associated with processes and share the global variables and objects with other threads of the same process. Each thread does have its own Call Stack, since each thread is executing a different set of functions. The ability for threads to access shared variables and objects makes coordinating between threads quick, but introduces some very sticky problems.

So is a process the same as an application? No. First, there are many different processes running on the computer which do not correspond to user applications (many of these are part of the Operating System). Second, an application may actually require multiple processes to carry out its task.

Executing Processes

- At an informal level, we think of threads and processes executing simultaneously. However, we typically have many more processes and threads running than we have cores in our CPU.
- To simplify things, we'll start by considering just processes and start by considering the single CPU case. We'll take a look at issues with threads later in this handout.

    - If we have a single CPU and many different processes are running simultaneously, how does this work?

    - The CPU executes the instructions for one of the processes. It continues executing those processes for some limited amount of time.
    - At some point, the CPU switches to another process by performing what is called a ***Context Switch*** or ***Process Switch***.
        - In a Context Switch all the settings needed to run the process are saved. The CPU registers, and in particular the Instruction Register and Instruction Counter[1] storing information about the current process's place in the program instructions are stored.
        - The settings needed by another process are swapped in in their place, and the CPU begins executing the new process.
        - At some later point in time, our original process will get switched back into the CPU by another Context Switch, and all the CPU settings and the Instruction Register and Instruction Counter will be loaded back into the CPU. Our original process will continue executing as if nothing has happened.

    - Context Switching can occur for several reasons.
        - In general, the Operating System handles ***Scheduling*** of processes. Different Scheduling schemes can be used, depending on the objectives of the system.
            - For example, an Operating System concerned with accurate real-time monitoring of laboratory equipment might use a different scheduling scheme than an OS for general consumers.
        - A context switch may also occur in response to an ***Interrupt***, which indicates that a condition has occurred that needs handling.
            - One common cause of an Interrupt would be in response to Input/Output.

    - As long as the CPU changes rapidly between processes, from a user's perspective it appears that al the processes are running simultaneously.

    - Multicore CPUs work similarly, except the scheduling task now has more CPU resources to allocate.

Interaction between Processes

- Generally running processes will not interact with each other.
- From the standpoint of any given process, it appears to be the only thing running on the computer.
    - The fact that it is being context-switched in and out of the CPU is transparent to the process.
    - The existence of other processes is not something that the programmer generally needs to take into account. (We'll see that the situation is quite different with Threads).
- Operating Systems do provide methods for Processes to explicitly interact with one another.

[1] You'll recall that the Instruction Register stores the current binary machine instruction that is being executed and that the Instruction Counter keeps track of the memory address for the next binary machine instruction to execute.

- This is called *Inter-Process Communication (IPC).*
- A variety of different methods may be provided depending on the operating system. For example, processes might pass information using files, by passing messages, or by explicitly sharing a section of memory.

Virtual and Physical Address Spaces

- Just as the process thinks it's the only process running on the CPU, it also views memory as if it's the only process currently running. From the point of view of the process, it has full access to memory, starting at address 0x00000000 on up.
- In fact, memory is being shared between all the active processes.
- We distinguish between the two using the concepts of virtual address space and physical address space.
  - The process's view of memory is the virtual address space, where it is the only program active on the computer.
  - As its name implies, the physical address space is how memory is actually allocated in the computer.
  - When a process tries to access a particular location in memory, the virtual address it is trying to access needs to be converted to the actual physical address where the data is stored.
  - Translation from virtual addresses to physical addresses is the job of the Operating System, and on modern computers is often aided by a special hardware Memory Management Unit (MMU) that is part of the CPU.
  - This system of virtual and physical address spaces is directly tied to the Virtual Memory we talked about in the last lecture, where portions of Secondary Storage are used to simulate extra Primary Storage.

  - A common approach to implementing this is through the use of ***Pages***. Virtual and physical memory is divided up into fixed sized units called Pages. A data structure called the *Page Table* keeps track of the relationship between Virtual Pages and their corresponding actual Physical Pages. When a process accesses a particular virtual address, the OS determines which Virtual Page it is accessing and uses the Page Table to translate to the corresponding Physical Page. If the Physical Page is actually in Secondary Storage, the OS copies the contents of that Physical Page back into real Main Memory, swaps a different Page into Secondary Memory to take its place.

Protection

- We need to have protections in place to prevent either deliberate, malicious processes from making trouble or just poorly written programs from causing problems. For example we need to ensure that
  - one process cannot trash another process's memory
  - a process cannot go rogue and take over the CPU preventing other processes from being swapped in

- The Operating System prevents normal processes from directly accessing some of the mechanisms that the OS itself uses to handle tasks such as memory management and scheduling.
  - In general, we distinguish between ***User Mode***, and what is called ***Supervisor Mode*** or ***Kernel Mode***.
  - You may also hear this referred to as **Privileged Mode** and **Unprivileged Mode**.
  - Some instructions can only be accessed in Supervisor Mode.
  - Modern CPUs provide special hardware support for these special modes.

Threads

- Our previous discussions have focused on Processes, rather than Threads.
- As we've previously mentioned a Process will have one or more Threads of Control associated with it.
- If we have multiple Threads executing for the same process, this means the Threads share the same memory space.
    - Because they share the same memory space, global variables and objects are also both shared.
- Programming with threads is both important and difficult.
    - As we saw last week, modern CPUs provide multiple cores. In order to run faster, programmers needs to take advantage of these cores. This means that they need to divide their program into multiple threads, so that as many parts of their program run simultaneously as possible.
    - At the same time, programming with Threads is very error prone.

- Let's take a look at some of the issues that come up when programming with Threads.
    - Actions that appear to be ***Atomic*** often aren't.
        - Suppose I have two threads both try to increment the global variable x:

```
x++;
```

          If x started out with the value of 0 and both threads have completed executing x++, I might reasonably expect that x will be 2.

          However, it turns out that in fact x might be 1. Why?

          We think of x++ as a single atomic action, that cannot be interrupted, but if we think about what happens at the machine language level, actually x++ consists of:

          1) retrieve the value of x from main memory and store it in a CPU register
          2) add one to the value in the CPU register
          3) store the value back into main memory.

          Now, what happens when I'm working in a multi-threaded environment? Thread A might retrieve the value of x from main memory and is now storing a 0 in its CPU register. Before it can add 1 to the register and store the result back into memory, Thread B starts executing x++. It also retrieves a 0 from memory. Both Thread A and Thread B increment 0 to 1 and both store the value 1 into main memory.

    - Results are ***Non-Deterministic***. This means that even if the inputs are exactly the same, the results may change from one run of a program to another.
        - Consider our x++ example. Our problem resulting in x of 1 instead of x of 2 only occurs if Thread A's and Thread B's attempts to increment x overlap with each other.

          What determines whether or not these overlap? That's determined by the scheduling algorithm and might be affected by other programs running on the computer, user interactions, or other things that are both completely out of our control and which may change from one run of the program to another.

- When we have a situation where sometimes the program will work and sometimes it won't depending on the exact sequence in which two different threads operate we say that we have a ***Race Condition***. Errors caused by Race Conditions are very hard to find and remove because they are not easily reproduced.

- The Operating System (often in conjunction with the CPU) will provide mechanisms allowing Threads to coordinate and prevent the type of errors described above.
- However, these mechanisms do require programmers to maintain a disciplined approach, and multi-threaded programs remain difficult to write correctly and difficult to test.

Files and Users

- The Operating System is responsible for managing the file system.
- The exact form of the files and the directory structure is determined by the Operating System.
- In addition the OS handles user accounts and if available user groups.
- It also manages file permissions – who can execute, read, write, or modify different files.