Interaction between the Computer Architecture and the Operating System

Smruti R. Sarangi

October 15, 2020

Up till now we have seen how to build processors that essentially execute a stream of instructions. Furthermore, we have proposed a variety of optimizations for improving performance, such as forwarding, renaming, branch prediction, etc. However, these are inadequate for running modern computer systems. A modern computer system has a host of software that include text editors, web browsers, video players, and windowing systems – all running at the same time. It is possible that when we are playing a video, the web browser is fetching the contents of a web page. it is not possible to do all of these things. on a bare processor unless it is augmented with additional functionalities. Furthermore, we have a host of I/O devices such as the keyboard, mouse, and the printer. These devices. also need to be controlled and moreover need to be accessed in a secure manner. If I am entering my credit card number on one web page, I do not want another web page to somehow see the number and steal it.

To solve all of these problems, it is necessary to have one master program that has more privileges. That can regulate the access to the entire system. This is known as an *operating system*. It is a regular program, albeit with some special privileges. We can think of it as a class monitor, who is a regular student, but has some additional privileges responsibilities which other students do not have.

An operating system specifically plays the role of managing the following entities.

1. It is a process manager, where a process is defined as the running instance of a program. If you type Ctrl+Alt+Del on Windows or ps on Linux, you get to see a list of processes that are currently running on your system. They are all alive and running concurrently.

2. It is a memory manager: it ensures that one process cannot inadvertently or maliciously access memory locations used by another process.

3. It is a device manager. It regulates accesses to external devices such as mice, keyboards, speakers, and the hard disk. It is often necessary to issue dedicated I/O instructions to control the behavior of these devices and to read or write data. Another way is to share a part of the physical

memory space With the I/O device and communicate with it via writing to dedicated memory locations. This is known as memory-mapped at I/O. Regardless of the method, it is not a wise idea to encumber regular programs with so much of additional code for performing a simple operation, such as writing to the terminal or printing a page on the printer. A dedicated piece of code within the operating system called the device driver provides these services. Furthermore, it ensures that the I/O device is operated correctly and there are no security loopholes. One important example in this space of the hard disk. The hard disk presents itself as a very large 1-dimensional array of bytes to the processor. However, we typically do not access the hard disk in this fashion. Instead, we assume that there is a file system on the hard disk or an USB drive and the process only accesses particular files that are organized in tree-structured file system. The operating system implements the file system.

To summarize, the basic processor-memory-storage system is too raw and unless there are layers of software over it for regulating accesses, enforcing security, and above all, providing a very easy and seamless method to perform a wide variety of operations, it will be very hard to use such a system in practice. This is why an operating system is used to provide all of these services. As far as we are concerned, an operating system can be thought of as a collection of programs such as a process scheduler, memory manager, file system implementer, and device manager.

Now, from the point of view of computer architecture, let us discuss some of the most important artifacts that determine the interaction between the architecture and the operating system. These are some basic hardware mechanisms that need to be there for any operating system to successfully function.

1 Timer Interrupts

At one point of time, several processes execute. How is this possible? Let us for the sake of simplicity assume that we have a single *core* (single pipeline + instruction cache + data cache). If we have a single pipeline, then we can only execute a single process at a time. We cannot execute two processes at a time. This means that to provide the illusion that multiple processors are executing concurrently, it is necessary for the processor to quickly switch between the processes hundreds of times a second such that human beings will never be able to perceive that we actually have a single core and secondly such fast-switching is actually happening. To type a key on a keyboard, or even move the mouse pointer, we need the intervention of the operating system. This means that the program that we are currently working on needs to pause, the operating system needs to compute the new coordinate of the mouse pointer, erase the image of the mouse pointer at the previous coordinates and draw the image of the mouse pointer at the new coordinates. Given that human response times are of the order of hundreds of milliseconds, we never perceive the fact that so much of

pausing and resuming is actually going on. We instead perceive an extremely seamless experience while using a computing system.

When we move an USB mouse, the hardware attached with the USB port generates an interrupt and sends it to the processor. Along with the interrupt, it sends an interrupt vector, which indicates the type of the interrupt, and also sends some data. In this case, it would be the displacement of the mouse. Subsequently, the processor will pause the current process, invoke the interrupt service routine that is a part of the operating system, set the state of the paused process in a data structure called the *process control block*, and then move on to service the interrupt. This would involve erasing the image of the mouse pointer at the previous position and redrawing the image of the mouse pointer at the new position. Subsequently, the interrupt service routine will hand over control to the scheduler process. The scheduler process will find a *ready process* to execute based on predetermined priorities. This is a simple mechanism and has been dealt with in the class.

The more important question is when there is no external interrupt, what happens to a process? It will continue to run because there is nobody to stop it or interrupt it. This means that if it needs to run for 10 hours, it will continue to run for 10 hours. Many a time, students wrongly suggest that the scheduler will schedule some other process. However, this is not correct because the scheduler is not running. Hence, we need an external mechanism to send an interrupt to the processor. This is precisely what is achieved by an external timer chip that sends an interrupt to the processor typically once every millisecond (also known as a *jiffy*). The timer interrupt is processed in the same manner as a regular interrupt. Once the processor receives an interrupt, it looks up the interrupt table, which is populated when the operating system is booted for the first time. This interrupt table has two columns: type of the interrupt and program counter of the interrupt service routine. The interrupt service routine in this case will invoke the scheduler process. This process might end up scheduling the original process that was interrupted by the timer interrupt or it can schedule some other process. It has the discretion to schedule any process it wants, depending upon its priorities.

2 Memory Manager

The key mechanism that allows the operating system to act as a memory manager is known as *virtual memory*. This will be covered later when we discuss Chapter 7.

3 System Calls

Let us now focus on the fact that the operating system is a service provider. It essentially abstracts complex. I/O devices and provides a simple interface that programmers can use. For example, the programmer may not be aware of the

complexities of the printer and what exact commands and instructions need to be sent to print a page correctly. The device driver within an operating system implements all of these functionalities and simply exports a simple function to the programmer. All that the programmer needs to do is provide the data that needs to be printed. Now, the key question here is how does the programmer invoke the services provided by the operating system?

Ideally, it would have been the best if the operating system would have exported a function and the programmer could have simply called the function the same way that regular functions are called. However, note that an operating system is a different process and the regular function call mechanism will not work in this case. It can also be argued that maybe the program can write something in a shared place, such as a temporary file, then the operating system can pick the function call arguments from there. This is an inefficient process when it comes to performance and furthermore this is not a very secure mechanism; lastly, we need to wait for the operating system to run.

Note that the key element of security within the processor is the current privilege level (CPL) bit and associated logic. If it is set to 0, it means that the operating system is running and a host of privileged instructions can be issued. It also means that the current process can control the I/O devices. However, when the processor runs regular processes (user processes), the CPL bit is set to 1. In this mode, a lot of instructions cannot be issued and a lot of registers cannot be accessed. The user process sees a very reduced view of the entire system, primarily because its privileges are limited.

Now coming back to the main issue, we need to devise a mechanism for invoking the attention of the operating system and for providing some data to it. The only way of attracting the attention of the operating system is to somehow raise an interrupt. This is a known, tried, and tested mechanism when it comes to hardware devices. We can implement something very similar, at least conceptually, at the software level as well. Almost all instruction sets provide an interrupt instruction, known as INT (0x80 in x86), which allows the software program to simply interrupt itself. The software program loads a predefined set of registers with inputs, and then issues the INT instruction. The rest of the processing is the same as an hardware interrupt. The processor pauses the current process, looks up the interrupt table, and depending upon the type of the interrupt, loads the appropriate interrupt handler. The interrupt handler subsequently saves the state of the current process, and begins to process the interrupt. This mechanism is known as a *system call*, which is rather heavy in terms of performance, but is the only method to attract the attention of the operating system. After receiving such a system call, the corresponding module of the operating system takes control, completes the requested action if the request is valid, and then invokes the scheduler process. It is important to note that whenever the operating system takes control and is done with its work, it always invokes the scheduler process. The scheduler process has the freedom to execute any process that it wishes to. This can be the process that invoked the system call or some other process that has a higher priority.

4 The Chipset

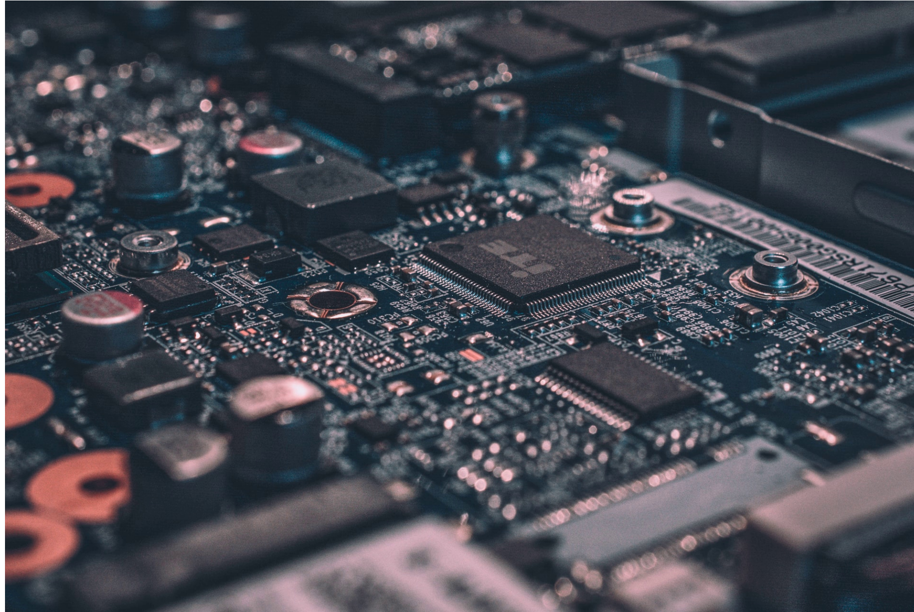

Figure 1: Image of a motherboard, source: Photo by Alexandre Debieve on Unsplash

Look at the image of a motherboard in Figure 1. The large chip at the center is the processor. However, kindly turn your attention to all the smaller chips that surround it. They serve a very important function. Most of them are I/O processing chips that interface with the I/O devices and act as an interface between them and the CPU. For example, some of them might be processing inputs and outputs from USB devices and some might be connected to the speakers. All of them comprise the *chip set*.

Some of these chips are *DMA controllers*. DMA stands for *direct memory access*. These chips can directly transfer data between the main memory (located off-chip) and I/O devices in both directions. The CPU basically outsources the job of transferring data between I/O devices and memory to the DMA controllers and continues to execute other processes while the data keeps getting transferred in the background.