Week 12 Module 22 Intro to OS Concepts

In this Module 1.2

It's time to start talking about operating systems and in this module we're going to cover a few things about operating systems and get you prepared for the future modules which cover the rest of operating systems concerns or at least some of the basics of operating systems. What we want to take away from this is a fundamental understanding of how computer operating systems first work and what they're expected to do, and then how we can understand them and utilize them either as programmers or if you want operating systems designers. So, in this module we're going to cover what is an operating system. We're going to cover a little bit history of operating systems, so you can understand how we got to this point today. We're going to talk about preemption and how we do scheduling of different programs that are running on the system. And then I'm going to talk about a basic architecture of an operating system.

What is an OS 1.3

First, we need a definition of an operating system and what I've come up with as a definition is: a program that controls execution of application programs and acts as an interface between applications and computer hardware. And we're going to delve into that a lot more in later slides. But what I want you to understand here is that the operating system is a piece of software; it's not a component of the of the computer. So contrary to Apple's view when you buy a computer, you don't necessarily buy an operating system and that you can mix and match which operating systems you want on which hardware with some limitations. Now the operating system does interact directly with system hardware and it has to be very closely matched to that system hardware. So, it's not like we can mix and match any operating system with any hardware; there has to be some compatibility concerns there. But if we take a look at the hardware for example that Apple uses on all of their laptops and desktops. It's the exact same hardware as any PC In fact, the CPU is the same, the system bus is the same, the DMA controllers the same, the interrupt controllers the same; all of it is exactly the same as one made by HP or Dell or any other PC manufacturer.

What makes the huge difference is the operating system, and Apple has designed its operating system only to work with their hardware, but the operating system is a piece of software fundamentally. And it runs on the same processor as the user's program code. So, your programs that you're writing, your 'hello world' applications that you're writing, those are running on the processor which is the same processor as the operating system. So, we don't have more than one processor in the system; let's keep it simple and consider a system that doesn't have multi processors, uni-processor systems what we call it. The operating system has to divide the time between itself running code for the operating system to manage the hardware and manage the execution of the whole system, and actually running your program that you really care about. What the operating system doesn't include is applications. And this is going to be a big difference for you to understand, because we'll talk about it in a later slide. When you go and buy an operating system from Microsoft or something like that what you get is a lot more than just the operating system so, we're going to talk about that.

But I want you to think of the operating system as the fundamental hard software that goes along with the hardware for the computer. It doesn't have to be directly matched to the exact hardware; there can be some compatibility for multiple operating systems to multiple hardware. But fundamentally we can talk about the operating system being a piece of software that manages this system hardware and runs applications.

Layers of Interaction 1.4

It's good to have an idea, at least a visual, of what we're talking about where the OS is concerned. So, the thing that you've probably experienced as users of computers and we've all been users of computers basically our whole lives we interact with applications. So, as a user, we are only interacting with the application side of the computer. So, when we look at Visual Studio, or when we look at our Web browser, or when we watch this video we're interacting with an application. And we can control that application; we can choose which website we're viewing, we can type code into Visual Studio, or you can pause and restart this video. Don't pause and restart this video right now. The thing that applications do is they'll interact with the operating system.

So, in order for the for the application to do what it needs to do, it needs to communicate with the operating system. For example, this video is going to need to accept from the operating system a stream of data from the network, and then process that data and then display it on the screen. So, we're using the operating system both to read the data in as well as produce an image on the screen as well as produce the sound of my voice on the audio. So, the application is going to use services that the operating system provides to get that done. In your 'hello world' programs that you've written before you've opened up files and just the act of opening up a file is an operating system call; it's what we call a system call. We're asking the operating system for some help. So, as the application programmer, you've interacted with the operating system by calling a function like open and when an open opens the file it was using the service of the operating system. The operating system of course is going to need to interact with system hardware. So, in order to open that file, it's got to read from the hard drive and so the operating system directly communicates with System hardware to get the job done. These are the layers of interaction that you see. So, the user is never directly interacting with the operating system and the operating so the application is never directly interacting with system hardware; everything is insulated.

What You Buy in the Store 1.5

I want you to consider that when you go to the Comp USA, or I guess Comp USA doesn't exist anymore. If you go to Best Buy or even if you go to the Microsoft store and you buy Windows, what you're buying is not an operating system; you're buying a lot of other stuff in that operating system. In that box comes a CD that is filled with ninety nine percent stuff that has nothing to do with your operating system. The operating system as we're concerned is the kernel, and the kernel in Microsoft Windows for example is about twenty five megabytes. Now you say what comes on a CD for four gigs when the operating system itself is only twenty five or thirty megabytes, well all the other stuff.

The extra stuff that you get are web browsers, like Edge (I don't know who's using Edge yet) but let's say Edge, Internet Explorer, text editors, device drivers, really any other application that comes on that CD, calculator, minesweeper, solitaire. those will come with the CD but they have nothing to do with the operating system. Linux on the other hand, for example, you can just download the kernel. If you go to www.kernel.org; you can see that what you can download is just the kernel. It comes with no text editors, it comes with no web browsers, it comes with some device drivers for stuff that supported directly in Linux. But fundamentally the thing that we want to concern ourselves with is this kernel. And the kernel is really only the very small components that, we'll talk about this and in a later slide, but really the just the very small components that are necessary to bring the system up online and start running other programs. And really that's not very much but what you buy in the store is a lot more stuff.

The OS as a Resource Manager 1.6

If we look at the system, as a whole the computer system, what we can see is that it's a collection of resources. And the resources that we have are memory and CPU time and file handles and network connections and it goes on and on and on and on; those can all be seen as a resource. and the operating system, since it is the primary controller of the entire system the first software that will run, the operating system controls all of those resources. And it needs to allocate those resources to some programs that you as a user decide to run. Now the operating system itself is software. So, it's going to need to use some of those resources; it's going to need some memory for itself, it's certainly going to need to run code so that CPU time, it may need files to open up more connections or may need files to open up more information, whatever has to happen.

The operating system is a client of itself in a lot of cases. But the applications are really what the users care about, when you as a user start up your machine the forty five seconds or two minutes or however long it takes to boot up your computer and get to the screen where you can finally do something: the desktop. You consider that time wasted; you consider that time that you can't do anything with your computer. And we try to minimize that time as operating systems designers; we try to minimize that time, so that you don't have to go to waste that time, you can use that time for other purposes. For example, sleep mode on either a MAC or a PC, when you close the lid the computer goes to sleep rather than shut down altogether so that when you open the lid again on a laptop, it comes back to life and most immediately. The way that we can do that is actually quite simply the computer still running; it just shuts down all the non-essential services like the screen, and the wireless card and things like that and it has uses a very, very, very small amount of power to keep running to continue keeping the memory active.

But the point is the resources that we have are going to be allocated to different applications as well as to the operating system. And these resources are limited, so we have to keep track of them and we have to make sure that we're not wasting any of those resources. There's going to be a lot of situations in operating systems we concern ourselves with this in the operating systems design classes here at NYU, but one of the things we take into account is that we might have an algorithm which solves our problem perfectly but we can't use it because it takes way too much time to run that algorithm. And if we did that it would actually slow the system down as a whole. So, even though the algorithm works perfectly and doesn't exactly what we need to do, we can't use it because it just takes too much time to run. So, the operating system has to dole out these resources to various applications as well as to itself, in a studious manner so that it takes care of not wasting the resources as well as allocating them effectively so that the programs can do what they need to do.

Back in the Olden Days 1.7

So let's take a look back at the old days and see what used to happen, and that's going to help us (bring us) up to speed on what we do today. And what I want to do is dial the clock back to the era when mainframes really dominated the computing industry, and what we're talking about is the late maybe 1950's, early 1960's, maybe even into the seventy's. We generally had one computer and by that I don't mean, we as an individual had one computer, I mean the whole company, or the whole university or the whole organization would have one computer for the entire organization. So, computing time was very, very limited and we had to allocate that computing time effectively.

The computer ran only one program at a time; there's only ever one program running on the computer at any given time. We would not flip flop; we would not open Chrome to view a web page, as well as, Excel to start typing things into a worksheet or so on and so forth. We would not have two programs open at the same time; there would be one program doing some complex calculus, because it was almost always math based, doing some complex calculus to try and figure out the solution and when it finished and found that solution it was done and it would move on to another solution, another program. One program had complete access to all of the system resources; it had access to all of system memory, save for a small little area for the operating system. It had access to all of the CPU time. This one program was started, processed all of its data, ran through to completion, and when it ended another program was started.

Now there was always a mainframe operator; it was always a human being who decided which order to run programs. So, this was somebody who if you wanted your program to run, if you knew your program had to run early in the morning, you went to Dunkin Donuts and got a dozen donuts and gave it to the mainframe operator and he pushed your program up a little further up in the stack; make sure that you get that guy on your good side or else your program is going to be the last one to run in the day. When one program finished the operating system had to be ready with the next program. So, the next program had to be ready to be loaded into main memory and start executing almost immediately. The processor was really the limiting factor in these older computers, so you'd bring in everything that was needed for the one program to run and then you'd put that program on the processor and let it do whatever it wanted to do until it said it was finished.

In order to do that you require some JCL and JCL is what we call Job Control Language. Job Control Language tells the operating system what facilities: what printers, what files, what network connections, anything that the program needed. The JCL would indicate that that program needed it before even the program started. So, imagine opening up Word and when you start Word, it says list all the things that you're going to need for this run of word and you have to say I need these three files, and I need this printer, and I'm going to need this to save, and I'm going to need the spell checker, and the grammar checker and all of that. You had to say that at the beginning of the program, even in some cases you had to say that before the program was compiled; so that when you're creating a program you would have to say what resources were going to be needed. So that the operating system could make sure that all of those resources were loaded and ready to go immediately rather than having to pick and choose. So, if you didn't have JCL and your program goes to open a file, the whole system has to stop and wait until the file is actually open and ready because you're the only program that can run, and you only run… You only stop when you're when you indicate that you're completely done. So JCL was a way for the operating system to sort of pre-load some of the material.

And what we do is call this batch multiprogramming; this is called batch multi programming because we're getting things ready in main memory and batching them creating a back of programs. So, when one finished, the next program was right there in the operating system and gets it going immediately. That's the olden days; we don't do that anymore. And the reason that we can't do that anymore; there's a lot of reasons actually, but we don't think this way anymore. We don't we don't think about getting one job done and then moving on to the next job. We'd like to do a lot of jobs at the same time and the other downfall of this environment is the JCL, that when the program starts we might not know everything that we're going to need; it might not be until we do some processing that we finally know what we're going to need. So, this doesn't get done anymore and in just a minute we're going to look at what we're doing today.

Today's Environment 1.8

Today we have a lot of processing power and we have a lot of memory: eight gigs, sixteen gigs, thirty-two gigs. I've worked on machines that have one-hundred and ninety-two gigs of memory, not hard drive but memory. and we have enough to run multiple programs at the same time what we really want to do is what's called multitasking. We want to have the ability to open up a web browser, as well as open up Microsoft Word, as well as open up Microsoft Excel, as well as open up anything else that we want to open up at the given time. So, in order to do that the operating system has to become a resource manager, and what we do is allocate resources to the various programs that want to run them. And then the operating system decides which programs can run and which programs, and when they can run.

The operating system will be responsible for stopping and restarting running programs and this is what we call preemption. Now when we restart a running program, we don't start over from the beginning we start from where we left off. So, you may not have realized this in your 'hello world' programs but your hello world programs didn't start and run straight through to the end; they were stopped and restarted many, many, many times. Not from the beginning just from where we left off. So, it's sort of like the operating system lets the program do a couple of instructions, in reality it's going to be a lot of instructions but let's say a couple of instructions, and then it stops and it saves everything in the registers and saves all the information in the CPU and it goes on to doing something else entirely. And when it comes back to decide that your program is going to run again, all it has to do is restore those values to the CPU and let it run. The instruction pointer, sorry… the instruction register, the program counter, all the AX, BX: all those registers that are in the CPU are going to be saved and restored. So, your program picks up from where you left off; you don't actually have to restart the whole process over from scratch it will just stop and restart. And this is what we call a timesharing system.

Now the reason that your program stops might be because it wants to stop; it might want to do something that takes a very, very long time. So the operating system says, let's say opening up a file which takes a fairly significant amount of time on the order of let's say fifteen or twenty milliseconds. The operating system will say okay, if you're going to open up that file, there's no reason that you're going to run anymore until I have that file open. So, we'll save your settings your registers in the CPU, we'll call your state and then we'll run something else. We'll do some other code; we'll run another program, we'll do anything else. And when your file is loaded and ready to go. We'll bring you back and reload those registers into the CPU and let you run again because now we have your file available. So the operating system does this very, very quickly and it really does this hundreds or even sometimes thousands of times per second inside the system, and in doing so it creates an environment that we call a time sharing system. And it allows us to run very many applications.

So, if you're on a Windows PC right now, you can hit control shift escape and you bring up the Task Manager; you go over to performance and you can see how many programs, how many processes are actually running, and that's probably in the order of between seventy and one-hundred, maybe even more processes that are running. If you're on a Mac, you can go to the activity monitor in the Applications Utilities folder and you can see that same information. So, what we're saying is that in today's environment, going back to the batch multiprogramming days, is something that we absolutely could not do; we don't even think that way anymore. If you were restricted to only running Microsoft Word and you could never get out of Microsoft Word and go into a web browser and go into absolutely anything else or any other program, would you even be able to write a paper and I doubt it very much.

But that's the way we worked in old environments; today we don't expect to work that way. So today we're doing multitasking and we're doing time sharing.

Monitoring Running Programs 1.9

In the modern operating system, we load a lot of programs and they're load all of the same time all in main memory. And if we take that into a bigger account, you may even run chrome multiple times on the same system. So, now we've got two copies of Chrome loaded into loaded into the operating system; how do we keep track of which version of Chrome is running and really which instance, if you will, of Chrome is running? And the way we do that is called a process and we're going to talk about this extensively; in fact, we have an entire module on processes that's coming up next. But what I need to know understand is that when we load a program into main memory, we create an object (if you will) called a process to keep track of that program to see how long it's been running to record when it's ready to record when it's busy to record when a file opened for this for this process. We keep track of it by creating a process and then we have struck the idea that this is a program and instead it becomes a process. The code for the program can be loaded multiple times or not depending on how we how we implement the operating system, but we can consider that this process is a unique object all to itself and doesn't interact with any other process in the system.

OS Levels 1.10

I want you to look here at this image that I have which is just list the levels of what we consider the operating system, and this is just a rough diagram; it's not really hard and fast rule. But what we've got on the very, very bottom, the bottom four levels, are not really that concerned operating systems per se except that we have to interact with them. So things like interrupts, procedures, processor instruction sets, and electronic circuits as operating systems designers: we don't get any control over that. Those get dictated by hardware designers, computer engineers, not computer scientists, which are going to design the electronic circuits which are going to specify the processor instruction set, which are going to deal with how we can do procedures small bits of code you know the small items that are built into the processor itself, as well as interrupts that have to occur inside the system. We don't get to control much of that stuff, but we do have to interact with it.

Now in the primitive processes phase, what we're talking about are very low level pieces of code that are parts of the operating system. And in fact, if you look at those two dark lines that I've drawn, levels five six and seven are what we consider what's called a microkernel; that's the minimum that's necessary in order to have a functional operating system. Primitive processes deals mostly with scheduling and resource management. Secondary storage obviously we have to access all the other code and all the other components of the operating system, as well as all the other programs and data so that's a very low level concern. And then we have this concept of virtual memory or even we can call it memory management, which we'll talk about in a later module that has to be managed. So, we're going to need some, some fundamental control over what's in main memory.

So, those three components are absolutely necessary in order to have an operating system. Now in the old days and I'm not talking about back in the mainframe days, but really only a couple decades ago. What we can say is the rest of it would have been included in what we call a macro kernel or a large kernel, so the micro kernel has only the smallest and most essential elements. The macro kernel has everything: communication systems communication, subsystems for getting information into and out of the computer, the file systems, Windows even has some components that are built into the kernel, that are purely for the file system. But file systems concerns in an operating systems like Linux and most

Unixes, those are an upper level concern outside of the kernel, devices that we might connect to, directories that are on the file system, any user processes, and the shell, the shell being really what you're interacting with. In Windows, this is called explored dot EXE. So, the shell is a component that the user interacts with everything below that is down to Level five, is a concern of the operating system and everything below that is a concern of system hardware. But I just wanted you to get an idea of what we what we look at, what we use, because each of these levels use services of the levels below them. Virtual memory, for example, stores components of processes on the secondary storage device so the virtual memory system uses components of the secondary storage uses services of secondary storage. And because virtual memory is higher, that means we can use the services they exist already. So, what we do is we provide services to the layers above, to the levels above and then we use the services of the levels below. So, that's the way that this whole interaction works but I want to you have an idea of the core levels, if you will, of the operating system.

The Windows Model 1.11

I want to show you now, and I hope you don't get too concerned by this image that you're going to see, of what the inside of Microsoft Windows looks like or at least what Microsoft will tell us the Windows model looks like. And I really want you to concentrate on a few items that are below the line, below the solid line. What we're seeing here is the components of the kernel, and in fact you can see that indication that shows you kernel mode versus user mode; user mode being above and kernel mode being below. Now the lowest level of here that we want to consider is the HAL, the hardware abstraction layer. We're going to talk about that in just a minute. But I also want you to see that there are some other components for device drivers as well as video drivers, a few things that have to be built in. There's a virtual memory manager; those components that are the fundamental components of the operating system are down there in kernel mode. Now if you remember back from the computer architecture discussion. There is kernel mode as well as user mode, and we're going to talk about this a lot more in later modules. But kernel mode has complete access to the entire system, whereas user mode has a much more limited access.

So, what I want you to get away and take away from this is that Windows is of course a very complex model, it's been around it's been being developed now for over thirty years, so it's a very complex model. But what I want you to see is that at the very bottom there's only a few layers that directly communicate even inside the operating system with system hardware, and there's good reason for that. Inside the kernel mode, we just keep track of processes and other things, most of the work of the Windows kernel is actually done outside of kernel mode. And the reason for that is one of security as well as organization and it's much easier to update things that are outside of the kernel, than things that are inside of the kernel. So, take a look at this image just don't, don’t memorize it don't, don't study it too hard. Just take away some of the major components of it that you can at least keep track of. And understand that modular operating system design is really what we absolutely have to do today, because there are so many things that the operating system has to do just even to run the first program.

The HAL 1.12

The harder obstruction layer is a magic little layer that Microsoft built in, and to understand why Microsoft even needed the HAL in the first place you have to look back again in history. Sorry, but we're going to dial the clock back again. And go back to the mid, maybe the early 1990’s. In the early 1990’s, Microsoft was not the big player in the market that they are today. If you look today, most computers and I say most I mean almost all computers run Windows, run Microsoft, some version of Microsoft Windows. The latest indications that I have here in 2016 are that Microsoft Windows runs on between

91 and 92 percent of the hardware that's out there. So as far as computers are concerned, Apple's market share is around seven to eight percent, at least those among the numbers that I have today; seventy to eight percent, whereas Microsoft is ninety-one to ninety-two percent. And there's that one-percent/two-percent other which includes Linux and some Unix's. In any case, back in the day, back in the 1990's, early 1990's, that wasn't the case at all. The operating systems were broken down much more towards about thirty percent was windows, about thirty percent was Mac, and by Mac I mean a completely different hardware architecture, and then there was also thirty percent other things like Unix's in these sorts of hardware.

And Microsoft wanted an operating system which they could market literally to everybody, and so what they did was they designed the architecture in a modular fashion so that the lowest level, the level that actually communicates with the hardware. Not manages things like running processes and scheduling and dividing up memory and allocating it, not that thing but the code that actually communicates directly with the DMA controller, the interrupt controller, the hard drive, things like this that are very, very, very low level. Microsoft designed them into a component they called the hardware abstraction layer. And the hardware abstraction layer was a set of functions that the kernel could call on to perform tasks on different, on the various hardware. So, rather than directly programming the timer in the system to say: let this process run for twenty milliseconds. The hardware abstraction, the operating system would call a function inside the hardware abstraction layer to say program the timer to let this process run for twenty milliseconds. And the reason that that's essential is because Microsoft can change out the HAL and produce an operating system that can now run on a completely different hardware. Without making any other changes to the operating system, it can change just the functions inside the HAL and those functions will, those features will take effect on the new hardware.

And Microsoft did this in the early 90's and produced five different operating systems, now they were all Windows NT 4 0 but they were five different versions of Windows NT 4 0. One for the Intel architecture, one for the Power PC architecture, and there were various other architectures that Microsoft could support. Now over the course of the next five/ten years, all of the other architectures disappeared. Even Apple; now if you go and buy an Apple Macintosh, you're going to buy a MacBook let's call it, let's say a MacBook. You're going to buy something that runs on the Intel architecture, and that means that it could run Windows just as well as it could run MacOS. And that's to our benefit of course because we want to do that we want to do exactly that. But Windows has the support for the HAL which has support for multiple hardware, and when Microsoft, for example, released Windows RT. When it came out, Microsoft said we can now support ARM architecture and the way that they did that was in part changing the HAL. So, any changes that we make to hardware, really only require changing the HAL and that saves us from having to rewrite the whole operating system; it's a very effective tool.

Windows Device Drivers 1.13

Windows device drivers, or device drivers in general, are kernel layer software that's written by the companies that designed the hardware. Now let's take for example a video camera that's built into your laptop computer, so you might be looking up at the top of the screen right now you see a little video camera. That video camera was designed by a hardware company and we don't know who designed it; it may be some Apple company or some company that Apple contracted with or perhaps it's Logitech, which is one of the most popular web cam companies perhaps Microsoft contracted with, who knows. Some hardware developer designed a web cam and the manufacturer of your computer incorporated that hardware design into their laptop so that they could use the web cam. Now when the hardware designers designed their web cam, it works differently from one designer to the next. So, Logitech's

commands for taking a picture are not the same as Microsoft's commands for taking a picture, or Apple's commands for taking a picture. Whoever the hardware designer is, they designed the chip so that it takes a picture after it's given a certain command. Now Microsoft doesn't want to have to know all of the commands for all of the hardware manufacturers that could possibly create a web cam, so what they do is they provide a facility for incorporating a piece of software that the hardware designers will develop; software that the hardware designers will develop into the operating system and allow it access to the camera so that it can program the camera to take a picture. Now what happens is Microsoft says if you are a web cam designer you create these functions: Take a picture, take video, turn on light, whatever it is. And if Microsoft calls the take picture function then the code from that device driver does whatever task is necessary in order to take a picture and return it back to the operating system in whatever way that Microsoft is expecting it. So, they define this protocol for communications between the operating system and the device driver. and then the device driver can do whatever it needs to do in order to actually take the picture and return it to the operating system.

Now that works great, except for the fact that these device drivers have complete unfettered access to the entire operating system and what they discovered was that while hardware designers might be very good at designing hardware, They're really not good at writing software. So, in the late one 1990's and even in the early 2000's, Microsoft had a real problem with device drivers causing device driver's code, which remember it's code, crashing and causing the entire system to become unstable and eventually crashing the whole system in what we call a "Blue Screen of Death." So what Microsoft did was design or specify a lab… they created this lab that says: if you're a hardware designer and you want us to certify your drivers, send us your software, send us the source code as well as a piece of hardware, we'll test it out, make sure it's one hundred percent compatible with Microsoft Windows and it doesn't cause a crash inside the operating system. It can't crash the operating system and as long as everything's working properly and we'll certify your driver and give it back to you with the our stamp of approval. And they call this the Windows Hardware Quality Labs or WHQL.

And nowadays it's nearly impossible to install in Windows ten or any of the later versions of Windows, to install any drivers that are not WHQL certified. You have to go through an enormous process to install those drivers, just for testing purposes. You basically certify that Microsoft, that you know that Microsoft has no knowledge of these drivers, and that it may destabilize your system and cause mass floods and virus, and I'm kidding about that. But you know that you're doing something that's outside of the ordinary. So, the whole idea of the WHQL was that this device driver code had so much access to the system that if it was not written perfectly, it would crash the whole operating system and we don't want that. So, we have avoided the whole "Blue Screen of Death" now by having certified drivers that are certified with WHQL. And I think you can we can all agree anybody who lived through that era of blue screens of death, every day, if not multiple times a day, we found the this is actually a very good idea.

UNIX 1.14

Unix was designed to be a multi-user, multi-tasking operating system. And what that means is that we are designing an OS which is going to run on a centralized machine and have lots of users; not one person sitting at a keyboard a screen but in fact lots of users accessing this all the same time. And in the era of the mainframe computer, when you didn't ever access the mainframe computer; I remember people telling me stories of writing their programs on punch cards and leaving punch cards in companies for the mainframe operators to load into the mainframe. They never directly interacted with the mainframe the whole idea of Unix was actually very nice because you could directly use the mainframe or directly use the computer.

Unix was designed to allow users to manage their own tasks and this was one of the first time sharing systems that came about and that's why it was so popular because it decentralized responsibility. Your users only got a very small portion of the time allowed to run on the system and nobody could take too much time on the system or too much resources, I should say. But it was allowed to access directly the mainframe, and do your work directly on the mainframe and have immediate feedback rather than delayed feedback whenever the mainframe operators cited for your program to run. The other thing about Unix that was great was that it was released into the public domain with open source. When you got UNIX you didn't just get the operating system; you also got the source code. And it was released in the public domain with public licensing, public domain licensing, so that if you made any modifications to the Unix operating system You could re-release it as your own, except that you couldn't charge for it. So, you couldn't take the Unix operating system and make one change and sell it for hundreds of dollars; It wasn't allowed. In fact, all they allow for is what's called a media charge So a few dollars for the cost of the media on which you would distribute the source code, and you had to release the source code as well for whatever changes.

Now Unix comes in a lot of different flavors comes in AIX, Linux, Solaris, etcetera like that. And some of them are branches of the original UNIX, so they're now public domain which means if they made any if they used any of the original source code of Unix, or any subsequent source code of Unix, they have to be public domain. There are some companies like IBM who designed AIX from scratch to look like Unix, but include none of the source code of Unix so that they could license it and sell it and make money off of it. So, there are some versions like that. And that's how Apple gets away with selling, or what they used to, sell MacOS was because they used BSD, which was a completely new implementation of Unix; it didn't use any of the original Unix. So, Unix has been modified many, many, many, many times over the years, but what we have is this basic idea of a multi-user multi-tasking operating system.

In this Module, We Learned 1.15

So, we covered a real basic of operating systems; think just kind of the intro stuff. We talked about what an operating system is, we looked at some history of operating system. We did talk about preemption remember that's the stopping and restarting of a program, even if it's not doesn't need to be stopped and restarted. We talked about the architecture of an operating system. We talked about a lot of things and we're going to keep going with that. And more discussions in later modules with more of that we're going to cover on operating systems, including further discussions on processes and scheduling. We're going to talk about memory management. We're going to talk about threads and concurrency issues. So, we've got a lot to go over so keep going.