Module 25 Week 13 Memory Management

In This Module 1.2

We're going to look at what the operating system has to do to take care of all of main memory, all of the RAM in the system, and there's a lot that has to be done. So, in this module we're going to cover what the needs are for memory management: some requirements that we have to meet for memory management. We're going to talk about logical addresses and physical addresses and how to convert between them. We're going to talk about a few partitioning strategies for dividing up main memory. We're going to talk about what is virtual memory, when it's in use, how it works, and that sort. We're going to talk about working set, and resident set strategies related to virtual memory, and what the those definitions are. We're going to talk about load control and we're going to talk about how pages and memory can be shared between processes. So, we’ve got a lot to cover in this module; let’s get to it.

Reasons for Memory Management 1.3

When we look at the operating system, we're looking at a multiprogramming system. And in a multiprogramming system, obviously, we're going to have a lot of processes running all at the same time. And since we're going to have a lot of processes running all the same time, there's never going to be enough main memory. And I've been in this industry now for over twenty years and I can tell you that the problems never change. The sizes certainly do change; when we started out, when I started out in this industry we were talking about main memory sizes of probably around one-hundred twenty-eight megabytes or two-hundred fifty-six megabytes. So, your average system might have two-hundred fifty-six megabytes of main memory and if you bought a system like that today it would be absolutely useless. And we look at more common systems today of eight gigabytes, sixteen gigabytes, probably thirty-two gigabytes is not in the very far distant future, and we think that the problem is solved by adding more memory and it isn't. There will never be enough main memory. Wherever we add memory, programmers tend to use that memory.

So, it's always a little bit of a battle between hardware and software with the hardware trying to keep up, usually, and the software trying to use more and more memory to do more and more work. The operating system is going to be responsible for allocating memory to lots of programs running in the system all at the same time, and the operating system is going to need to move parts between main memory and secondary memory. So, there's a lot of work that the operating system has to do even though we have main memory and main memory just looks like a very large array. We're going to have to divide that up and share it among a lot of different programs that are all running at the same time; that all want access to that main memory, and that all want to use more and more and more.

Memory Management Requirements 1.4

The operating system has to meet a number of requirements, in terms of taking care of main memory. The first one is relocation. In years past, the operating system, or rather the program, would be the only thing running on the system. If we look back at the old era of the IBM mainframes, or even older than that when we look at really big computers, there was only ever one program running on the system at any given time, and since there was only one program that program had access to the entirety of main memory; it could do whatever it wanted with it. So, in reality the programmer had access to all of main memory inside his program and the programmer could put portions of the program, portions of data, portions of code, anywhere he wanted inside main memory. Unfortunately, that's not true anymore. We're looking at a system where we have a lot of programs running, and we don't know what the order of the programs is that are going to start; we don't know what's free when a program starts, we don't know what's occupied when a program starts. And the operating system is going to have to be responsible for putting a running program into a particular space in main memory and during its lifetime even while the program is running, not actively running on the CPU, but maybe in the ready state or in the block state or even in one of the suspended states. The operating system should be able to relocate that process to a completely different section of main memory, I mean actually take it and move it to a different place in main memory. And so, that's relocation and the operating system has to be able to do that, so that's one of our requirements.

Another requirement is protection. The operating system will be the only entity in the computer system, which is allowed to access all of main memory. A program cannot be allowed to access another program's memory space, and that's a prime tenet of memory management. So, that we need to make sure that one program cannot interfere with another running program in the system. In fact, one program shouldn't even know that another program is necessarily running without the intervention of the operating system. Now obviously, two programs may need to communicate and we have facilities for doing that; we have the inter-process communication facility. But overall, we should say that protection should be enforced, such that one program cannot access another program's memory space. And we saw this in when we've talking about pointers in earlier modules, because if we let the pointer go outside the memory space of the program then the program would be shut down and that was protection. If you remember or if you ever saw that message that came up in Windows it says, “Microsoft is sorry but your program has performed an illegal operation,” that's a protection mechanism. Your program, probably via a pointer, was trying to access memory outside of its memory space, and so Windows intervened and said that's not allowed and it shut your program down immediately. One of the problems with this is that the operating system can't prescreen. We can't go through and look at all the memory accesses that your program is going to do before you actually do them, because that would take just too much time. It has to be done dynamically. So as you make a memory request, the CPU, the hardware actually has to look at the memory request and decide whether it's in your memory space, your program's memory space, or outside your program's memory space and should be terminated. So that's protection.

We also have some situations where sharing might make sense, and we'll talk about these later on. Where the process is actually going to share code with another process. In fact we're going to talk about one situation where we execute a process, and then we fork and create an exact duplicate of this process as a new process. So, we'll talk about that in a little bit.

We can talk about logical organization where we create modules. So, a lot of programs are written these days using shared objects or dynamically linked libraries and what this is is rather than creating every possible function. So, for example when you were writing your “hello world” program, your simple “hello world “programs, you didn't overload the output operator for the IO stream class. You had a library that did that for you and all you had to do was pound include IO stream. Well what that did was tell the operating system that, through a lot of steps I'm oversimplifying it, but that tells the operating system that you're going to use a library and one of the libraries is the Microsoft Visual C++ library MSVCRTDLL. Inside that library is the code for the function on how to output to C-out to the O-stream class. So you don't write that code, it's already in Windows and every program that uses C++ has use for that code. Well we don’t want to load that code with every program. We want to load that code once and have it shared between every program that was written in C++. So that might be seventy programs and we only have one copy of that MSVCRTDLL, it's not a large DLL maybe it takes up only one megabyte or two megabytes. But if we're talking about seventy different programs using it. That means we're

saving between seventy and one hundred forty megabytes of memory and that all of a sudden becomes significant.
And then we'll talk about the memory management requirement of physical organization, which is really just how we map a logical memory addresses to physical memory addresses. So, these are the requirements that we have to meet. So, we've got relocation, protection, sharing, logical organization, and physical organization. And those are the requirements that the operating system has to meet in order to complete its memory management tasks.

Logical vs Physical Addresses 1.5

The operating system understands physical addresses, and the computer system, of course, understands, as you saw back when we were talking about instruction sets, the computer system understands physical addresses. main memory is broken down in physical addresses. so if you have four gigabytes of main memory, for example, you have adverse zero through Agis four point two billion. So, the computer system understands a physical address and expects to be communicated with physical addresses. Unfortunately, the programs will only be able to use logical addresses. And that's because the program has no idea where it will be loaded physically. So, we can do this quite simply, we can think of this quite simply, as a relocation problem. If we can use the offset from the beginning of the program, since the program should have no access outside of its own memory space. If the addresses that the program is going to use for all the pointers, and all the code, and all the jumps, and everything that it's going to do, if those addresses could be logical addresses relative to the offset from the beginning of the program, then the program doesn't really need to know where it's physically loaded in main memory.

So, if we have that then we're going to recognize that those logical addresses need to be converted dynamically at run time into physical addresses. And so what we can do is use what's called the hardware memory management unit, CPUs have now created they were evolved in the 1980’s to have this hardware memory management unit, and that would be responsible for converting during run time the logical address to the physical address. And it's not a difficult calculation, in a very simple example, we can say that the hardware memory management unit can know the starting address of the program. and then when it sees a memory reference, when it sees a pointer, when it sees it jump to code, or anything like that, when we see that logical memory reference the hardware memory management unit will convert it by adding the base address of the program into the logical address that the program is trying to reference to produce a physical address. And then the CPU will actually access that physical address, rather than accessing a logical address that the program is asking for. So, at run time without the operating systems intervention because remember the operating system is asleep while the program is running, the program will make a reference to a logical address and then the OP, the hardware memory management unit, will convert that logical address into a physical address and make the appropriate reference.

So, the program doesn't have to know anything more about where things are physically located, and it meets our relocation and protection requirements because we can now physically move the entire process to a completely different location, while it's not running and then update the hardware memory management unit the next time it runs to tell it where the new, where the process is now. And then the hardware memory management and of course will do that calculation at run time. So it doesn't require a lot of work it meets our protection and relocation requirements; this is a perfect solution.

Partitioning Strategies 1.6

When the system boots up, we’re looking at what is effectively a desert. There is nothing in main memory, short of a couple of locations where we can communicate with hardware, but the majority of main memory is simply completely empty. And as the operating system, it becomes our responsibility to divide main memory into sections where we're going to allow programs to use. And we're going to have to talk about a number of different partitioning strategies for how to get that done. So, the core of the idea is that we're going to look at a way to break down main memory into different sections so that the programs can all be in their own individual sections, so that's again part of our part of our relocation of protection requirements, more protection than anything else that we can set guidelines we can set borders for a process. But we have to recognize that when the system starts memory is just empty and it's just available for us to use. So, some of the strategies we're going to look at our fix partitioning and dynamic partitioning. We’re going to take a look at a buddy system which is sort of fixed and dynamic both. Then, we're going to look at paging and segmentation and paging segmentation is really what's done today. But we have to understand fixed in the name of partitioning before we can get into the more advanced stuff.

Fixed Partitioning 1.7

What we do in fixed partitioning is when the system boots up, we can divide main memory into some static number of parts. Let's imagine a situation where we have four megabytes, four gigabytes of main memory and we're going to divide it up into let's say sixteen different partitions, now those could be either equal sized partitions or unequal sized partitions. And if they are equal sized partitions, then let's imagine that those sixteen different partitions, each run two-hundred and fifty megabytes in size, so our four gigabytes of main memory is broken down into sixteen equal partitions of two-hundred and fifty megabytes each. Now that works; it’s fine. It means we have a maximum limit of running sixteen different processes; we can't run any more than sixteen different processes. and one other problem is if we start up notepad, and notepad is going to just edit a small text file and it only needs one megabyte of main memory, well the smallest we can put it in is two hundred fifty megabytes, because we have equal sized partitions. So, what that means is that we're wasting two-hundred and forty megabytes of main memory and that's pretty significant. That's what we call internal fragmentation. We have this division allocated, for notepad, we give it two-hundred and fifty megabytes for notepad and notepad only really wants one megabyte. And the other two hundred forty megabytes now are wasted in internal fragmentation.

One solution that we could use is to maybe use unequal sized partition, so partitions which are differing in size. And if we do that we might say there are sixteen different partitions, one partition is very very big it's one gigabyte, and other partitions are very very small there might be ten meg, ten megabyte partitions or just slightly larger. The point is that we're always going to have some internal fragmentation with these solutions, but we're going to maybe minimize the amount of internal fragmentation.

Now if we use unequal sized partitions, we’re going to have to worry about which partition we decide to place this process in. Let’s say we open up notepad and we open up a small text file and so the operating system decides, okay we'll put it into one of the ten megabyte partitions and then somebody chooses to open up an enormous file of one gigabyte file. Let's say a video file but they're opening in Notepad. Well now the operating system has to move that process out of its memory space; we can do that because of relocation and move it into the one gigabyte partition that we have, the only one gigabyte partition that we have. And what happens if that one gigabyte partition is taken by something else, maybe chrome, and now notepad cannot run at the same time as Chrome, because they both need that one gigabyte partition. So, there are some real disadvantages to having fixed partition. But one of the big advantages of it is it's really simple, really requires very little effort on the part of the operating system. But the downside is that we have very limiting factors on the number of processes that we can run, and we also waste a lot of memory in internal fragmentation.

Dynamic Partitioning 1.8

Alright, fixed partitioning didn't work out great. What if we took an opposite end of the spectrum and said, okay let's make dynamic partitions. Let's ask the process exactly how much memory it needs and let's give it exactly that much memory. So, if a process says it needs ten megabytes; we'll give it ten megabytes. Now that causes a problem with the process decides to grow to later on we don't have an issue, we don't have a way of dealing with that. But if a process is in a particular memory space, it can grow as much as it wants inside that memory space. So, once we allocate a memory space for that process, it really can't grow but it can use up everything that that's inside that partition. But what this causes is actually external fragmentation, extremal fragmentation is caused by the dynamic partitioning and what this is is wasted memory between allocations.

Now what's happening here as you can see in the diagram, is that we created three different processes. We created three processes and the middle process sort of disappears, it ends. Now we've got that little chunk of memory free, but what we would like to do is run a program that's just slightly larger than the space that was allocated there. And unfortunately because it was larger, I can't allocate that space; because the new process is larger than the space that was allocated, I can't put that new process into that memory space. So, what I have to do is put it at the end and that means we're we're using up more and more and more at the end. And then of course what happens if all we have are these little slivers of memory left over and I want to run a big process, which is not so large that it would that we don't have enough memory free memory for it, we have enough free memory for it but we have as these little segments of free memory. And we have to compact all the running processes now into the beginning of main memory to make a big space at the end, so that we can run the new process and that of course requires a lot of copying of memory, a lot of CPU time and that's a big waste of time. We have an issue with dynamic partitioning in that the data structures for the operating system get rather complex because we have to record both the start and the ending location of the process.

And then the last problem is where do we choose to put a process if we have enough space available, assuming that we have enough space available, where to be choose to put a new process into its into what partition. So, we could have the best fit which is the area size closest, and of course larger, than what the process is asking for. If the process is asking for ten megabytes and we have an eleven megabyte sliver of available then, we can put this ten megabyte process into the eleven megabyte sliver of available space. Well of course unfortunately that means that the last one megabyte is not inside of partition, it's still available space we didn't know allocate it but it's never going to be used because its so small; we're never going to a process that so small it'll fit into a one megabyte little sliver of space, and that's called external fragmentation because it's outside of the area of an allocation. We could use first fit, which just says the first spot that we becomes available when we start looking, so once we start looking from the beginning of main memory. If a spot is available that's larger than what the process is asking for we'll just put it in there. And the next bits as we begin looking from where we last left off. Of all the options it actually turns out that next fit has the minimum CPU time with the best average utilization of memory, but the point is moot because we don't do this anymore. Dynamic partitioning has so much overhead, that we just can't do it in today's environment.

Buddy System 1.9

The buddy system is actually a compromise between the fixed and the dynamic partitionings. And again there's not a lot of operating systems that do this today, in fact I don't know any of them, but for example, Linux uses this for what it calls its kernel slab allocator. So, when the kernel itself needs to create some memory for the kernel, it will use a chunk of memory and divide that chunk of memory as a buddy system, just to make smaller partitions of a larger chunk of main memory. So, while we don't do this on a large scale for all of main memory, there are certainly situations where even Linux today will do this for parts of its slab allocator for creation of, or for allocation of smaller parts of main memory for kernel purposes. Anyhow, what it does is we take main memory and divide it in multiples of two. And the reason of course that we're doing multiples of two is because it makes it easier on the calculations; we don't have to necessarily record the starting in the ending point, we can record what multiple of two this is and so on and so forth.

Let's take for example, a two megabyte memory allocation, so let's say the the Linux kernel has a two megabyte memory allocation for it and wants to divide it up for purposes for storage of an IO table, or storage of a main memory management table, or who knows. It’s got to store some portion of information but two megabytes would be a huge waste, it needs something much smaller. The kernel means is let's say one hundred kilobytes, for whatever it's going to do. The process then is that we take that two megabytes and start breaking it down into divisions, so you can see that what we've created initially, we take the two megabytes and divide it in half so that we end up with the a one thousand and twenty four kilobyte allocation and another one thousand twenty four kilobyte allocation. But that's way over what we need of hundred kilobytes, so we break down one of the one thousand twenty-four kilobytes into two five hundred twelve kilobyte allocations. Then we'll take one of the five hundred twelve kilobyte allocations and break that down into two two hundred fifty six kilobyte allocations and that's still too much. So, we take that and divide it down again into two one hundred twenty eight kilobyte allocations. If we were to divide the one twenty eight KP allocation again, it would be a sixty-four kilobyte allocation and that's too small to fit the hundred kilobytes need that we have. So, we would have to we would have to stop there and use one hundred twenty eight killed by an allocation.

Now this results of course in internal fragmentation, because it's going to end up where… we’re going to end up allocating one hundred twenty-eight kilobyte, when only one hundred kilobytes was needed. But we're talking about twenty-eight kilobytes so it's not an overly significant factor. We might end up with some extra fragmentation if we have a lot of these small slivers but one of the nice features of the buddy system is what's called coalescing. We can take those two hundred twenty-eight kilobit allocations, and recombine them up into a two hundred fifty-six kilobyte allocation. and we can keep doing that ultimately getting back to the original two megabyte allocation, if we ever need to. So, coalescing allows us to bring these back into a larger hole.

If we have a large number of hundred kilobytes processes, we might take that hundred one hundred twenty eight kilobyte allocations that we already have but we'll also divide the two fifty-six into two one twenty-eight. We divide the five twelve into two two fifty-sixes in each of those into one twenty-eight. So, we can we can create a lot of one hundred twenty-eight kilobyte partitions out of this original two megabytes. So, we end up decreasing the amount of internal fragmentation, overall, because if we had only two megabyte allocations there would be one point nine megabytes of completely wasted space, because we'd have to allocate two megabytes for one hundred kilobyte allocation. It makes the operating system structures easier, because they're beginning and ending on a two to the N boundary it.

It actually does work quite well, the only problem is coalescing back does take some effort. And so Linux actually delays that, it's called delayed coalescing. It waits until much later on, when the operating system is not very busy and it says: “you’re probably not going to use these anymore” and then it recombines them back up. But ultimately this does work; it doesn't work on a grand scale, it works on a micro scale. But it is a solution that involves both fixed and dynamic partitioning and it is a compromise between the two.

Paging 1.10

So, let's look at something that actually happens in today's environment: it’s called paging. What we do is take main memory and we break it down into a lot of equal size frames, and each of the frames are the same size, every frame is the same size. And it's something small like four kilobytes; it's not something significant. Then what we do is we take down a process, take a process and we break that down into the exact same size pages. So, we’ll take the process and break it down into four kilobyte pages. Now I'm sure you can see where this fits, and that is that one page fits exactly into one frame. This is RAM okay, and what RAM means is that we have random access memory, that means that every single frame, in fact every single byte, can be accessed all at the same amount of time. So, there's no benefit to keeping the process as one coherent entity. There's no reason we should keep the process in sequential memory; that wasn't a failed assumption that we had for fixed and dynamic partitioning. That we had to keep the process all together; we don’t.

If we have bits of the process at the beginning of main memory and bits of the process the end of main memory, it doesn't matter as long as we can keep track of them in order. We can access the front of main memory and the back of them in memory all at the same time. So, there's no reason that the process needs to be continuous. Now the operating system is going to have to keep track of where each page is located in main memory; what frame number each page is loaded into. And it can do that, it does that, in what's known as a page map table or some books call it or just simply a page table. And the page map table is simply in an array of frame numbers and if we look at index three, for example into the page map table.
That's going to tell us where the fourth page, zero one two three, where the fourth page of the process is located; what frame number the fourth page of the process is loaded into. So, we can look now into the page map table and find out where each of the pages in the system is in physical memory, and that's exactly what the hardware memory management needs to do.

Whenever any logical memory request is made, to a logical address, basically to the offset from the beginning of the program, the hardware memory management unit needs to calculate how to convert that logical memory address into a physical memory address. And it doesn't take a lot of effort to do so it can simply do a number of bit shift left and bit shift right to get the the physical address from the logical address. and one of the features that it's going to have to do is a look up into the page map table. So, the format of the page map table now is going to be defined by the hardware manufacturer and not by the operating system designer, because the hardware needs to know first where the page map table is stored and each process is going to need its own page map table; each process has its own page map table. And the hardware memory management unit is programmed, during a switch, during a context switch, it's programmed for where that page or app table is. And the hardware memory management unit will look up in the page map table where the appropriate page is loaded and do the conversion between page number and frame number and then look at the offset. So, this conversion process is a little bit involved, and we’ll see it a little bit later on. But the benefit here is that we can use paging, and we'll see the benefits of paging the next slide.

Benefits of Paging 1.11

Well let's look at the benefits of paging. There’s no external fragmentation; every frame of main memory is equally good, and we can put any page in any frame and there's no detriment. So, there's no reason to keep pages together, which means any pages, any frame is available for use as long as of course it's not occupied. So, there's no external fragmentation. Some argument is that there's a minimum of internal fragmentation. Okay, yes there is internal fragmentation, but how much? We're talking the maximum amount of internal fragmentation we could have is four kilobytes minus one byte. So, if the process is going to use one extra byte beyond the end of a four kilobyte page, then it's wasting four thousand and ninety five bytes of main memory. A process wasting four kilobytes of main memory is not terribly significant, even if we had one hundred processes running on the system that means a maximum of four hundred kilobytes of main memory wasted and that's insignificant in today's environment. So, we don't care about it; so we say yes it's going to have some internal fragmentation possibilities, but it's so small we just don't care.

Protection is really easy because if the process is making a reference beyond the end of its page map table, the hardware memory management unit won’t be able to convert that into a physical memory address. We know there's a problem and we can interrupt and call the operating system immediately to say that something's gone wrong. So, protection becomes really easy the process literally cannot access anything outside its own memory space. relocation becomes easy and, in fact, relocation now is talking about relocating a page.

We don't necessarily have to relocate the entire process; we could if we want but if all we need to do is relocate one page, we can just update the page map table to indicate where the new the pages now located what the new frame number is and we're done. It really doesn't take a lot of effort, of course we have to copy those four killer bytes, but copying four kilobytes is not terribly significant anyhow. Besides why would we do relocation? There's very little reason in an simple paging system, there’s very little reason to move a page from one to the other. And then easy sharing which we're going to see a little bit later on. If we just have two processes that want to share a page and it's allowed to do so, doesn't violate protection, if they're allowed to share a page then we can simply enter the same frame number into their page map tables at the same location and we don't have any problems with that. So, there's a huge number of benefits to paging and that's why we'll never go back to the old fixed and dynamic partitioning systems.

Converting Logical to Physical 1.12

I wanted to just look a little bit more at how we convert a logical address to a physical address in a paging system, and I know it sounds a little bit complex but we're going to boil it down and make it really easy. We have to ultimately remember that what we're trying to do…
We're given a logical address that is the offset from the beginning of the program, and the program doesn't know that it's not, that it's memory is not completely sequential. The program doesn't know that; it sees logical addresses that are all completely sequential. So, what we're going to do is use a very simple calculation, which is going to be that the physical address is equal to a look up in the page map table for the logical address divided by the page size multiplied by the page size plus the logical address mod the page size. And you see that that's a divide and a mod operation and you say, well wow that's got to be a lot of work; it would be if the page size were something other than a multiple of two a factor of two. And that's what makes this really easy because the divisions and the mod operation are now simply going to become bit shifts. So, because we have the bit shift, and bits shift really doesn't take

much effort inside the CPU certainly not in comparison to a divide operation. To calculate the page number: we'll do bit shifts to find the page number, we'll look up the page number in the page map table, we'll find out what the frame number is, do the bit shift again as is the multiple of the page size, and then we'll add the offset. And to calculate the offset we simply looking for the latter half of the physical, excuse me, of the logical address. So, we can do an XOR to wipe out the early parts of the logical address and end up with just the offset, and that tells us where the page is located (what frame number it's in) and how far into that page the offset we're looking for for that particular byte of memory that we need. And that's how we convert a logical to a physical address in a paging system.

Segmentation 1.13

The idea of segmentation makes sense and there's a lot of systems that do it today; it's very popular. Fortunately, it's not substantially different than paging. The only key difference here is that we're now allowed to have unequal sized partitions. So, in paging system, we're assuming that somebody came down and said that it's going to be a four kilobyte page size and you might argue well what if I don't want a four kilobyte page size, for some reason I want something larger or something smaller. Segmentation says, in a simple segmentation system,you could use any segment size and we don't call it a page anymore, we call it a segment; you can choose any segment size. But now the logical addresses really have to change because we need to talk about not just the the logical address, we need to break the logical address into two parts: a segment number and an offset into that segment. And then the operating system can keep different segments in different locations, of course, we need space for different segments.

So, this is kind of leading to a more dynamic partitioning strategy, but the segments don't necessarily have to be contiguous, so we can now divide them up. This isn't as as universal as you might see a paging system. What usually happens in a segmentation system is the CPU designer will give you a few options of segment sizes to choose from and so you might not have just four kilobyte pages, You might have four kilobyte pages, sixteen kilobyte pages and sixty-four kilobyte pages. The reason for this is that the operating system if it knows it's going to need a lot of main memory for this process, we can use all sixty-four pages and then we have less of them. So, the page map tables are smaller and simpler to maintain. So, we don't really have the exact dynamic partitioning or the design of a simple segmentation system, where we have an infinite number of segments sizes. Instead we just have a few limiting segments sizes and the CPU designers decide what segment sizes we have as options.

Virtual Memory 1.14

If we're using a paging or segmentation system, we have to recognize that the process is only going to be making one memory reference at any given time. Now I know that's an overly simplistic view of things, it's actually going to be making a lot of memory references. But one of the things that you can see is that the memory references are all going to be pretty much clustered together. If we, for example, start a process and the process throws a splash screen up on the screen. You know starting Word or PowerPoint, that splash screen will never appear again during that run of the program. But that splash screen takes a memory, there's code that tells it how to display that splash screen, and there's the splash screen itself probably an image. So, that memory is loaded into main memory and it's never going to be used again.
So the question then becomes: why should still be loaded in main memory, which is a finite quantity very much in demand, why can't we take it out of main memory and put it on the hard drive? Somewhere that we can bring it back if we ever need to, if the process never needs it again then it can

just be deleted off the hard drive and never recovered. But what if the the process was never going to use that splash screen for that run of the process again.
And what if we just took it out of main memory and left it on a hard drive. And if we did that we would free up some main memory and we would allow that main memory to be used by other processes.

So, this is the concept known as virtual memory; what we're doing is we're using a separate portion of the hard drive, it’s not where the process is normally stored. It's a separate portion of the hard drive; Windows calls a virtual memory, I think Mac OS calls it the swap space, most Linux and Unix clones call it a swap space. But we have this area of a Hard drive dedicated for storing pages of main memory, which are no longer in main memory, which are no longer going to be used. Now the operating system can decide, dynamically, which pages to save in main memory, which pages stay in memory and which pages come out. And there has to be some operations that has to go back and forth between main memory and the secondary storage device, but it's a mechanism for actually saving a lot of main memory for more processes than would normally be allowed to fit. So, now if we say have only four gigabytes of main memory, and we want to run one hundred processes, each of which needs a gigabyte of main memory, under normal circumstances there's no way that we could do it. But with virtual memory, we would be allowed to do that; each of the hundred processes would have a portion of the four gigabytes, a portion of itself loaded in the four gigabytes, the rest of it would be on secondary storage and available. So, we have to move these pages in and out into main memory, and a lot of discussion to talk about how that's done, but we have that available now. We can use more memory than is actually in the system.

What Stays? 1.15

So, what do we keep in main memory and what do we throw out? What is the process going to use and what is the process not going to use? How do we is the operating system, a group of people who had never seen this program before, decide what the process should be able, to be allowed to keep and what it should throw out?

Well the first issue that comes up is that we have a resident set. The resident said is that portion that's in main memory, and we have a working set and that's the portion of the process wants to use. Now in order to run, the working set has to be in the residence set. So, the working set must contain the residence. So, we have to have the stuff that we want loaded into main memory. We're going to extend on that page map table that we talked about just a few slides ago and we're going to say that it now has an a present bit, and of course that means we have to update the hardware memory management unit and that's a much more involved process. The memory management unit has to know about all the stuff that's going on with virtual memory and it does, these days we know about it. The page map table is going to contain what's called a present bit, a P bit and the present bit tells the memory management unit whether or not that page is actually in main memory. If the page is not in main memory, then the present bit will be unset and then we know that we have to look for that page on a secondary storage device if the process wants it.

We also have something known as a modified bit, and the modified bit is useful because we're going to make copies of pages in secondary storage before we realize that the the process doesn't need them anymore. So, what we would like to do is make duplications of the main memory into the secondary storage device, so that when it's time to decide to remove a page from main memory we can look for a page that has the modified bit unset. And what that means is that the copy on the secondary storage device Is the exact same as the copy in main memory, which means we don't have to go to IO to actually do the write operation we can simply erase the page out of main memory and trust that the copy on the secondary storage device is valid.

As I mentioned the hardware memory management unit has to know what happens when it finds a present bit set to zero, and what it does is whenever it looks at the page map table and finds the present bits at zero, it's what's called a page fault. and the CPU stops running the process and instead switches to the operating system, very much in the way that an interrupt would occur, it switches to the operating system and begins running code to handle the page fault. And then the operating system can make the decision on whether the page fault was caused by a page which is no longer present, or perhaps was caused by a page which the process should not have any access to to begin with. And that's a page fault that's going to result in an exception error; that's going to shut down the process. That hardware memory management unit also has to recognize that the modified bit has to be changed any time we make a change to that page in main memory, so any write operations to a page in main memory would cause the modify bit to get set. But now the hardware memory management unit knows all about that and as the operating system we're just responsible for writing the code that brings in and out pages, and updates and maintains the present bits, and works with the modified bits.

Benefits of VM 1.16

Of course the benefits of a virtual memory are huge; the programmer can look at this as a much larger memory space. So, now each program, or each process even, can view main memory as completely available purely to itself. In a thirty-two bit address space, each process is going to think that it has four gigabytes of main memory available; it doesn’t. There may not even be four gigabytes of main memory available in the system. But that process is going to think that it has four gigabytes because it's using virtual memory. In a sixty-four bit operating system, we’re going… The programmers are going to perceive a eighteen exabyte memory space which is just enormous, but that's what we can view this as. Even if the system doesn't have that much memory the program can be written as if it were available. The system the operating system can remove unused pages, so pages like that splash screen I talked about a few slides ago, those can be eliminated out of main memory; they're not going to ever be used again. So, let's take them out of main memory, store them in a second or storage device, just in case because we have no guarantee that they're not going to be used again, store them in the swap space and when they're never used again, we just don't put them back in main memory; so freeze up that main memory. More processes can run in the system and that means we have better performance, that means more processes generally means that there's a higher probability that a process will find itself in the ready state, so that will find a process that's in the ready state. So, ultimately we can result in better system performance just by having more main memory available. Huge benefits available for virtual armoring, which is why it's the standard today.

Lookup Problems 1.17

Well one of the problems that we get now is that memory management is always going to require two look ups. So when we're talking about accessing any memory space, any logical address, it Now requires to look ups into main memory. So the M.M.U. is going to have to first look up in the page map table where this is located, and then translate that into a frame number, and then access that frame number. So every main memory access now, by a process, is going to result in two main memory accesses. Now main memory is fast; There’s no question it's that. But when we're doubling the price of everything, It really starts to add up and it slows down the C.P.U. So with the C.P.U. implemented with C.P.U. designers implemented that we just have to take into account, There’s not much as the operating system we need to do about it. but what the C.P.U. designers implemented was what's called a

translation look aside buffer. and the translation look aside buffer is a type of content addressable memory, which stores a cache. and I've got a picture here for you So that you can see it. it stores a cache of those entries in the P.M.T. which have been retrieved recently, and what it basically is is a table where we're looking up the virtual page number, which on that image that you're seeing is the the V.P N in that table; the virtual page number with a physical page number. so that virtual page number to physical page number mapping is only temporary, It's only there for a short period of time. in fact once we have a context switch, we basically have to dump the translation look aside buffer. But that virtual page number to physical page number mapping can be looked up because this is content addressable memory, It can be looked up in a time a big O of one, constant time search through the entire V.P.N. table.
So the V.P.N. table Can be looked at and if the the virtual page number entered is there, then we'll get back the physical page number and we can use that immediately inside the C.P.U. without accessing main memory. So it's just for speeding things up, so that we can avoid one of those memory references that we can avoid going to main memory and asking for that value. if we don't have the value from the page map table already loaded in the T.L. B, then we have to go look at the page map table and there's no way of to avoid that.
But the the the page map table is not going to change while the process is running, of course because we can't do relocation while the process is running. So the translation look aside buffer is a way to avoid that double memory access for multiple accesses to the same virtual page number. and it does work, It's not a huge savings but over the course of a run of the program, It does cut the memory references not in half but certainly significantly less.

Replacement Policy 1.18

What happens when we run out of main memory? I mean it can happen ultimately, since we're giving each process as much memory as it wants affectively there's going to be a point in time when main memory runs out. And of course we're still going to be using virtual memory, so the swap space is going to start to be used, but what happens when we physically run out of main memory. Well now we're going to have to make a decision on which pages to pull out of the main memory, which pages no longer needed in main memory, and which we should load. This is called stealing; the choice of pulling out a page, and pulling a page out of main memory, it's called stealing. And unfortunately if we choose poorly we can really cause a lot of performance issues because if we remove a page which is going to be used, that's going to require an IO operation, it's going to require us possibly writing that page to the secondary storage device if the modified bit is set, we're going to have to write it to the secondary storage device then remarked the present bit to zero. And when the process runs again and wants to use that page, We're going to have to have a page fault and bring that back in. So, one one way of looking at the performance of virtual memory is to count the total number of page faults that are happening perception and if that number is is too high then one of the problems might be, the first obvious problem is that we have an insufficient amount of main memory. But the second obvious problem might be that we're choosing the wrong pages to remove; we're choosing pages that the process will immediately will want.

We have a number of different algorithms that we can use for choosing which page the process might want. One of them that we call… One of them we call the optimal algorithm or this might be called the crystal ball algorithm. Because what it is is we look into our crystal ball, not literally of course, but we would look into our crystal ball and decide which page is not needed for the longest period of time. Now of course without looking at the process and what it's going to do in the next execution cycle, we can't know how long it's not going to need a page for. So, this is more a comparison algorithm, we can look at the memory accesses that a process did in its entire history and use that as a judge for deciding what would have been the optimal choices for page replacement, but ultimately we can't do this in real life.

There's also the LRU algorithm, the least recently used, and in order to do this we would need a timestamp on each page to tell us when the last time that page was accessed; not modified, accessed. And unfortunately, we don't have time stamps on every page and it's too much overhead inside the CPU to timestamp every page. So the LRU is not really feasible either.

The FIFO algorithm is really simple; we just throughout the oldest page that we've brought in. But that doesn't say anything as to whether or not we're actually still using that page, and if we're still using that page and we throw it out it's going to immediately cause a page fault. In other words, it might be the oldest because the one that's needed the most.

The last algorithm is relatively easy to implement it does require use bits. And thankfully, the hardware memory management units did recognize that, and what we have is the clock page replacement algorithm. So, we're using use bits to indicate when a page was used and what we do is… The hardware memory management unit updates those used bits whenever a page is used. In fact, it doesn't even have a problem with the translation look aside buffer, because what the hardware memory management unit does is when it loads the physical page number, sorry the frame number if you will, into the TLB it's going to update the use bit to be a one. So, the use bits just indicate whether or not the page was used recently and that's going to allow us to do the clock paid for placement algorithm which we're going to see in the next slide. So, we have to choose a replacement algorithm in some way to choose which slide is going to be, sorry to choose which page is going to be stolen, but if we make the decision with some reasonable piece of information then we can make it an effective decision making process.

Clock Page Replacement Algorithm 1.19

So this is a very simple view of what the clock page replacement algorithm here is. We have just end pages, end frames available, and what we're going to do is keep a pointer that goes around and that's why we call it the clock page replacement algorithm. Now when we load up a page, we're going to set it's use bit equal to one and that's fine. So, we're going to have N frames and let's just say it's ten frames, just for easy math. We have ten frames and as the frames, as the pages are loaded up into those frames, we said all their use bits equal to zero and the pointer is pointing at let's say a page, frame number two, as you see it in the the left side image. When it's pointing at page number two, you can see that the use bit for frame number two page forty-five is set to one, the use bit for frame number three page one-ninety-one is set equal to one but the use bit for frame number four page five-hundred fifty-six to set equal to zero. So, what happens now when we have a page fault, we're going to have to remove a page and we're going to have to load up the new page. So, to do that what we do is real look for advancing the pointer; we're looking for use bits of zero. But if we find a use bit of one, we're going to reset its value back to zero. We're not going to steal it quite yet. We just set the use bit back zero. So, frame two and frame three as you can see they their page numbers didn't change but their use bits didn't change to zero. Now we need to load page seven-hundred twenty-seven but page five-hundred fifty-six, which is loaded in frame four is unused. So, when the pointer gets around to that point, we're going to remove that page stored in a second or a storage device if the modify bit or whatever, and load up page seven twenty-seven and set its use bit equal to one. The reason that this is effective is because if page forty-five is no longer in use, then next time the clock pointer comes around to frame two, the use bit will still be set to zero because it's not in use and page forty-five will be stolen now. So, the clock page replacement algorithm does have a very high efficacy rate; it's very effective in not throwing out pages which are actively in use, and it does have a pretty good rate of throwing out pages which aren't in use. So, we can save a lot of memory using this and it's really very simple to implement doesn't require a lot of effort to implement.

Resident Set Management 1.20

Resident set management actually asks two different questions or two different problems. One of the problems is now how much memory do we give to each process? How much main memory does each process deserve? So, notepad for example, might only ask for one megabyte of main memory and Chrome might be asking for one gigabyte of main memory; should they get the same amount? And the answer probably is no, but how do we make the decision of how much to give each process. Now if we use smaller allocations, if we minimize the allocations then that means we have more processes but that also means we have more page faults because each one has, each process, has a smaller allocation. If we choose larger allocations, well that means we might have less page faults but less processes as well. So, it's sort of a give and take that we're going to see in the next slide, where allocations that are larger tend to mean less page faults and allocations that are smaller tend to mean more page faults. But how do we make the decision of what, how much notepad gets and how much chrome gets? So, that's one of the issues.

And then the next issue is when a process page faults, we're going to have to steal a page where, we're going to have to make room for the page. And are we going to choose this process, that it's page faulting to steal out of in order to make room for the new page or are we going to say that the entire system is open for stealing, and we can choose from any process? And obviously it means that that has a a role in whether or not the allocation is a variable allocation, meaning can change over time, or is a fixed allocation. Meaning that it's always going to be exactly the same. So, resident set management tries to deal with these two problems.

Controlling Page Faults by Resident Set Size 1.21

So, I'm looking at this chart and what the chart is showing you is the page fault rate as compared to the number of frames allocated to the process. Now if we take a look, obviously if we only have one frame for the process it's going to create an almost infinite number of page faults; every memory access, almost, will need a page fault. And on the other side if we have an infinite number of frames, enough that it fits the entire process into main memory, then we have no page fault. But where it's interesting is that there's a non-linear progression between those two points. If we look at the process that has all the memory that it could possibly want and we remove just one frame, we only cause a very small number of page faults. and on the flip side if we look at a process which only has one Frame and we double the number of frames, that's going to result in a massive decrease in the number of page faults. So, there's a non linear progression between the two. And one of the things that we can extrapolate is that there is a sort of optimal location, where the process is not faulting too much but also is not is faulting a little bit. In other words, we're saying that page faults are not necessarily a bad thing; that since main memory is a quantity which is not infinite that we would like to have some number of page faults to indicate that the process hasn't been given too many frames, in other words that were starving other processes. And we can use a couple of algorithms to try and cause the process to try, and manage the resident set, to cause the process to fall in between the upper and lower bounds. If it's above the upper bounds, we can increase the number of frames that are available to that process thereby, effectively decreasing its page fault rate. And if it's below the lower bounds, then we can decrease the number of frames in the process give them to somebody else to hopefully lower its fault rate. And we can leave it in between this upper and lower bounds so that it has enough frames to do the work that it needs to do, but not so many friends that it's starving other processes.

PFF Algorithm 1.22

One of the algorithms for resident set management that we're going to talk about is the page fault frequency algorithm or the PFF algorithm. The PFF algorithm tries to look at the frequency of the page faults by looking at the time between the current page fault and the last page fault. And we have some value that we'd like to hit, we have some value we call F. And the F says if we're page faulting more frequently than F, in other words, if the time between page faults is less than F, then the solution is to add a frame. And what that's going to do is if we're page faulting too often it should decrease the page fault rate because now we've given it more memory. But if the page fault rate is greater than F, then we can use those use bits look at all the used bits of zero discard those pages they're not use anymore, reset all the used bits of the remaining pages to zero and load up the the frame, the page, that we need into an available frame.

So one of the points of the PFF algorithm is that we can manage the resident set, how many pages are loaded into main memory, by I looking at how often the algorithm is or how often the process is page faulting. One of course the difficult values is how to figure out a value for F. One of the difficult things is how to figure out a value for F. How do we set a value for F? How do we know what a reasonable F is? And in fact there isn't a reasonable value for F. As you saw in the previous slide it would be better to have an upper bounds and a lower bounds, so that we're not looking for a moving target, we're not looking for F and deciding whether it's too frequent or too infrequent. We can use an upper bounds and lower bounds to say we might add a frame, or we might steal from the current process, or we might steal from the global set. So, there's a number of different variations on the PFF algorithm that we can use to to make it more effective.

One problems of PFF though is that when the process moves from one locality to a completely different locality; so, it's starting in one area of its code and it jumps very far to do some other work. And it's going to be in the new locality for a long period of time, during a locality shift those pages in the new locality probably are not loaded. So, we'll have to load all those pages from the new locality and then when we're in the new locality, the page fault rate will decrease significantly because we have all the pages that we want. And so now we would, the next time try to unload those old pages from the old locality but unfortunately if we never fault again, for example, unlikely. But if we never fault again we're never going to clear out those old pages. So, the page fault frequency during a locality shift can result in both the old locality and the new locality using memory and that means double the memory and that's a huge waste. So, there are some downsides to using the PFF algorithm.

VSWS Algorithm 1.23

The variable interval sampled working set or VSWS algorithm tries to solve PFF's problems by setting a number of different parameters. The first one is an M value, a minimum value for clearing out the unused pages. If we're ever below M, we're always going to add on a new page. If the page fault ever happens below M, we always add on a new page.

Then we set a Q value, and we say if we're above M and we've reached Q page faults. We've had Q page faults since the last time we reset all the used bits and threw out the unused pages, then let's go ahead and do that algorithm again. So between M, above M and above Q, we would throw out all the unused pages and reset the use bits. Even if we're above M, but we've only had a very small number of page faults then it's too much work to go through the whole algorithm of resetting the used bits and throwing out the unused pages; so, we'll just add a page.

But once we get to L, L is the maximum limit. Once we get to L time units then it's time. It doesn't matter how many page faults we've had; this process had its chance. It's now time to throw out the unused pages, reset the use bits ,and let the process run again. So, even if we don't get a page fault by the time we get to L, we now say it's time to run this algorithm and clear out the old waste. And that's the point that VSWS solves the PFF shortcomings by through setting the L value to say that even if we haven't had a large number of page faults by the time we reach L, there's got to be some stale information there and we'll throw out that that old locality’s information. So, it does actually solve the problems of PFF and it actually works fairly effectively.

Load Control 1.24

Another issue that we can use page faults to look at is what we call load control, or we can use the number of page faults that are happening in a system to make a decision of whether or not to swap a process altogether out. Now the benefit, of course, of swapping a process is that we get all of its memory, not just a portion of its memory. But that whole amount of main memory is going to be removed and reallocated to the other running processes and of course that process can be brought back at a later time. We can do that we've always been able to do that but now we can do it on a much grander scale.
So, now we can use the number of page faults to tell us when it's time to choose a process to swap. But the question then becomes which process do we swap? And we have a couple of options, of course, if we have priorities in our processes, we could use the lowest priority process. We could also choose the faulting process as it's the greatest probability that it doesn't have its working set in the resident.

So, the faulting process of course, if we swap that out, that means we don't have to recover a page; we don't have to deal with the page fault, we effectively deal with the page fault by swapping the entire process. The last process activated is certainly the least likely to have its working set resident. We could also choose the process with the smallest local resident set because it's the one that's going to be easiest to offload and easiest to onload later on, when we decide to run this process and finish it up. We could also choose the largest process, the exact the opposite, because we get the most amount of main memory and it's most likely that we're going to be able finish those other processes If we get back a large chunk of main memory. If we know how long we expect this process to run for, we can look at the process with the largest remaining execution window. And because the if we aren't going to finish this process for another five hours, who cares if it's another five hours and five minutes more. So, load control can be done by looking just at the number of page faults and it is a reasonable solution for the medium term scheduling algorithm.

Shared Pages 1.25

I recognize that if we're using virtual memory and/or paging, then we can understand that we can share structures; we can share portions of a process. And the way that we can do that is quite simple: we can just have duplicate entries, duplicate frame number entries in a page map table. So, since the code doesn't change, for example, if we opened up five copies of Chrome would we need five different copies of the code for Chrome. The answer is no, not necessarily. The operating system may be smart enough, may be capable, of recognizing that multiple copies of the same program are running and it will share the code pages, not the data pages, not the context, just the code pages of that process because the code is not going to change from one process to the next process as long as they are the same program.

So, if we're running Chrome twice, we should probably only load it's code once; it will, each copy will have a different data section but who cares because at least we're saving the memory for the code.

Now there's also the possibility that a process will want to share data pages, but unfortunately if we're going to share data pages any changes to those data pages have to result in a duplications of those pages of data; not necessarily the entire portion of data but just those pages that are going to change. So, even if you're just changing one byte inside a page, we have to make an entire copy of the page. Now this is what's called copy on write, or cow, and it allows the sharing of the pages. The page table is now extended to the to have a read only attribute and if the page is marked as read only any attempt to change that page is going to result in a call to the operating system. And the operating system can then say this process is trying to change a page which has COW enabled. And when we change that page the operating system will make a duplications of just that page, update the appropriate page map tables, unsetting the read only bit and then re running the instruction. So, that the the operating system can intervene only of course when necessary, make the copy of the page, and then the processes can run again not realizing that the pages are now separate and different.

Unix FORK Function 1.26

The greatest example of this is the Unix FORK function; it's been around for a while but it really does work quite well. The Unix FORK function when you call it, it actually returns two values. The reason or returns two different values is because it's going to create an exact copy of the entire current process. Now this is useful if, for example I'm just using this as an example. It's not actually how it works, but if for example in Chrome you click the button to create a new tab, we wanted different version of, we want to a different process for example, but it might be the exact same code. You might have not just a brand new copy but a duplicate an exact duplicate of the existing page.

And so what we can do is call the Unix FORK function. FORK will return I believe it's a zero if the child, if the process is the child process what's known as the child process which is the new process, or it return the process identifier of the child process if this is the parent process. So, we're creating two processes it's sort of like, the creation of a new amoeba out of an existing amoeba, right. That's an exact copy of the existing process; but how does it do this? Well, with the operating system will do is create a new PCB a new process control block, it creates all the new page tables, but the page tables are an exact copy of the original page table and everything in both page tables is marked as COW, Copy-On-Write. if any of the process in either process changes the page then that page alone is duplicated, but it saves the operating system having to actually go into main memory and duplicate all of main memory for this process into a new process. All it does is duplicate the page when it changes.

In this Module We Learned 1.27

We covered a lot of material in this module, and memory management is not a simple task at all, we know that. The operating system does a lot of work for memory management. But we did cover the need for memory management and we covered the memory management requirements; we understand those requirements of protection, and relocation, logical organization, sharing, physical organization. We talked about partitioning strategies. We talked about logical versus physical addresses, and how to convert them, specifically in the paging scenario we did a lot of calculations on how it works. We talked about virtual memory. We talked about the working set strategies and the resident set strategies and we used resident set strategies to determine how we can control the load of the system, how many processes we really have running. And then we talked about shared pages and copy on write and then the magic of the Unix FORK function.

So, we certainly covered a lot on memory management there, of course in a current and current system today's environment, there’s a lot more that has to go into memory management and these are only the basics. But at least with the basics we can get started with understanding some of the more complex features, and the operating systems courses are going to cover a lot of memory management techniques that we just kind of glazed over. So, there's a lot more out there; some of it that Microsoft will even tell us about we have to find out manually. But there's at least a good basics in this module.