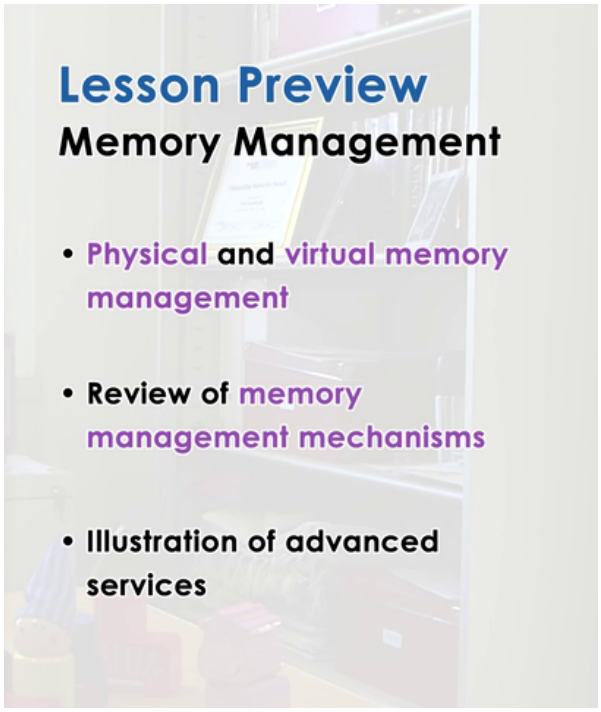
P3L2 - Memory Management

01 - Lesson Preview

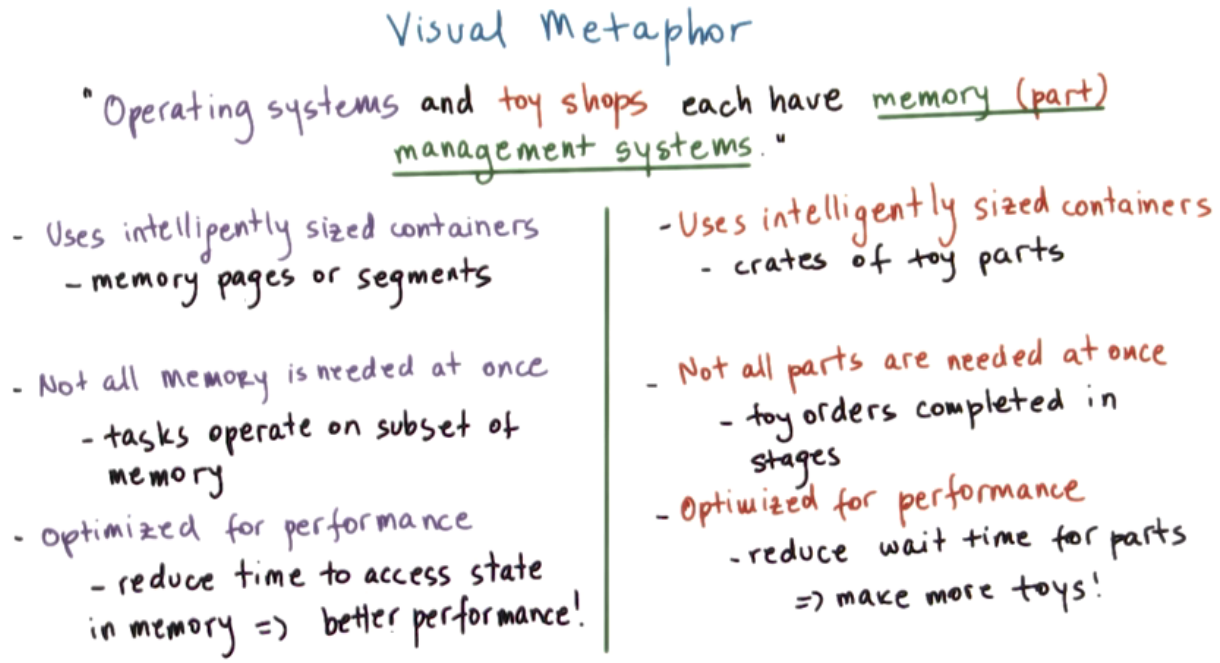


π

In this lesson, we will review the memory management mechanisms used in operating systems. We will describe how operating systems manage physical memory and also how they provide processes with the illusion of virtual memory. Let me remind you that the intent of this course is for us to review some of the basic mechanisms in operating systems and also to serve as a refresher for those of you who may have forgotten some of this content from your undergraduate classes. My goal is for us to review some of the key concepts and to make it easier for you to use additional references like the books I mentioned or some online content in case you need to fill in any blanks.

02 - Visual Metaphor

Memory vs Parts



Optimizations like TLB (Translation Lookaside Buffer)

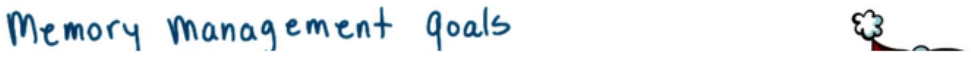
Before we get started, let's look at memory management from a high level with another visual metaphor. Here is an idea. Operating systems and toy shops each have some sort of mechanisms to manage the state that's required for their processing actions. In the case of the operating systems, this state is captured in the system memory. In the case of the Toy Shops, this state is captured in the various parts that are needed for the toy production process. Beginning with the toy shop here are a few elements required for its part management system. First, the process uses intelligently sized containers. Second, not all parts are needed at the same time. And finally the process is optimized for achieving high performance. So what do we mean by intelligently sized containers? Well, for storage purposes it's convenient to make sure that every container is of the same size so it's easier to manage those containers in and out of the storage room. Regarding the next point, you can imagine how certain types of toys like teddy bears would require fabric and threads whereas other types of toys may require wooden parts or something else. Given that toy orders are completed in stages and not every single one of the toy orders is processed at the exact same time, these parts can be brought in and out of these containers as necessary for instance. How that's done should be optimized for performance because if we can reduce the wait time for bringing the parts in and out of the containers ultimately we can make more toys. The memory management subsystems that are part of operating systems have some similar types of goals. In an operating system, memory is typically managed at granularity of pages or segments. We will discuss this more later on but the size of these containers can be an important design consideration. Like the analogy of the toy order and parts needed for the toy orders. Similarly, processes that execute in a computing system don't require all of the memory at the same time. Some subsets of the state that's needed for the computation of the tasks can be brought in an out of memory as needed

depending on what the task is performing. And finally how the state that's required for these tasks is brought in and out of memory and into memory pages and segments is optimized so as to reduce the time that's required to access that state. The end result (again) is improved performance for the system. For instance to achieve some of the performance related optimizations, memory management subsystems rely on hardware support like for instance translation lookaside buffers (TLB's), they rely on caches, and also on software algorithms such as for page replacement or for memory allocation.

03 - Memory Management Goals

In the introductory lesson processes and process management, we discussed briefly a few basic mechanisms related to memory management. The goal of this lesson is to complete our discussion with a more detailed description of OS level memory management components. Let's remind ourselves one of the roles of the OS is to manage the physical resources, in this case the memory DRAM on behalf of one or more executing processes. In order not to pose any limits on the size and the layout of an address space based on the amount of the physical memory or how it's shared with other processes, we said that we will decouple the notion of physical memory from the virtual memory that's used by the address space. In fact, pretty much everything uses virtual addresses and these are translated to the actual physical addresses where the particular state is stored. The range of the virtual addresses from V0 to Vmax here establishes the amount of virtual memory that's visible in the system and this can be much larger than the actual amount of physical memory. In order to manage the physical memory, the OS must then be able to allocate physical memory and also arbitrate how it's being accessed. Allocation requires that the OS incorporates a mechanism or an algorithm as well as data structures so that it can track how physical memory is used and also what is free among the physical memory. In addition, since the physical memory is smaller than this virtual memory, it

is likely that some of the contents that are needed in the virtual address space are not present in the physical memory. They may be stored on some secondary storage like on disk. So the OS must have mechanisms to decide how to basically replace the contents that are currently in physical memory with needed content that's on some temporary storage. So there is basically some dynamic component to the memory management process that determines when content should be brought in from disk and then which contents from memory should be stored on disk depending on the kinds of processes that are running. The second task, arbitration, requires that the OS is quickly able to interpret and verify a process memory access. That means that when looking at a virtual address, the OS should quickly be able to translate that virtual address into a physical address and to validate it to verify that it is indeed a legal access. For this, OS relies on a combination of hardware support as well as smartly designed data structures that are used in the process of address translation and validation.



The figure here illustrates that the virtual address space is subdivided into fixed size segments that we call pages. The virtual memory (not to scale these two) is divided into page frames of the same size. In terms of allocation then, the role of the OS is to map pages from the virtual memory into page frames of the physical memory. In this type of page based memory management system, the arbitration of the accesses is done by page tables. Paging is not the only way to decouple the virtual and the physical memory.

Another approach is segmentation so that would be a segment based memory management approach. With segmentation, the allocation process doesn't use fixed sized pages. Instead it uses more flexibly sized segments that can then be mapped to some regions in physical memory as well as swap in and out of physical memory. Arbitration of accesses in order to either translate or validate the appropriate access uses segment registers that are typically supported on modern hardware. Paging is the dominant method used in current OS's and we'll

primarily focus our discussion on page based memory management though we'll mention segments a little bit more later in this lesson again.

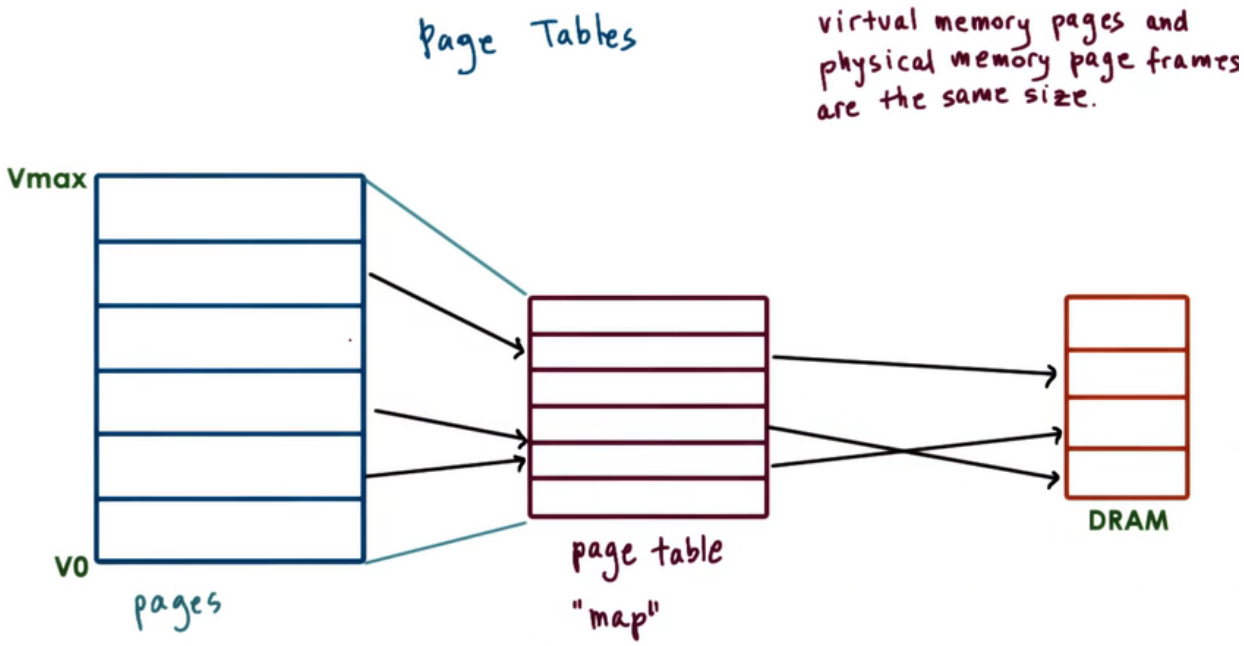
04 - Memory Management Hardware Support

As I already hinted, memory management isn't purely done by the OS alone. In order to make these tasks efficient, over the last decade the hardware has evolved to integrate a number of mechanisms to make it easier, faster, or more reliable to perform allocation and arbitration tasks regarding the memory management. First, every CPU package is equipped with a memory management unit. The CPU issues virtual addresses to the memory management unit and it's responsible for translating them into the appropriate physical address. Or potentially, the MMU can generate a fault. The faults are an exception or a signal that is generated by the MMU that can indicate one of several things. For instance, it can say that the access is illegal, like for instance that the memory address that's requested hasn't been allocated at all. Or it can indicate that there are inadequate permissions to perform a particular access. For instance that the memory reference may be part of a store instruction so the process is trying to override a particular memory location however it doesn't have a write permission for that particular access. That page is what we call write protected. Or another type of fault may be an indication that the particular page that is being references isn't present in memory and must be fetched from disk. Another way hardware supports memory management is by using designated registers during the address translation process. For instance, in a page based system there are registers that are used to point to the currently active page table, or in a segment based memory management the registers that are used to indicate the base address of the segment, potentially it's limit, its overall size of the segment, maybe the total number of segments, and similar information. Since the memory address translation happens on pretty much every memory reference, most memory management units would integrate a small cache of valid virtual to

physical mappings to perform any additional operations to access the page table or the segment and to interpret the validity of the access. This is called the Translation Lookaside Buffers or TLB. The presence of a TLB will make the entire translation process much faster since if this translation is present in this cache then there is no need. Finally, the actual generation of the physical address from the virtual address so the translation process that's done by the hardware. The OS will maintain certain data structures such as the page tables to maintain certain information that's necessary for the translation process however the actual translation the hardware performs it. This also implies that the hardware will dictate what type of memory management modes are supported. Can you support paging? Can you support segmentation? or both? So basically are there any kind of registers of this sort? It will also potentially imply what kinds of pages can there be. What is the virtual address format as well as the physical address formats since the hardware needs to understand both of these. There are other aspects of memory management that are more flexible in terms of their designs since they're performed in software. For instance the actual allocation, basically determining which portions of the main memory will be used by which process, that's done by software. Or the replacement policies that determine which portions of state will be in main memory vs on disk. So we will focus our discussion on those software aspects of memory management since that's more relevant from an OS course perspective.

05 - Page Tables

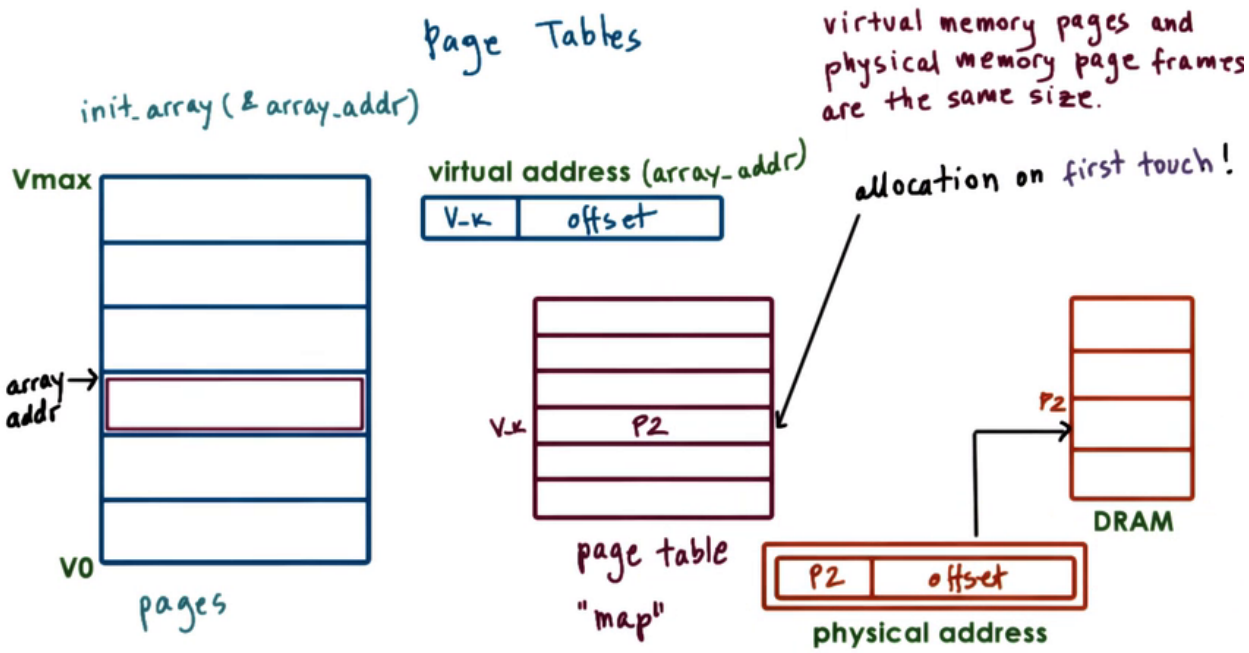
Page Tables are the mechanism used to translate virtual memory addresses into physical memory addresses. Page Tables are like a map.



As we said, pages are the more popular method to memory management. Now let's take a look at one of the major components that enables page based memory management and that's page tables. That's the component that's used to translate the virtual memory addresses into physical memory addresses. So here's a page table. For each virtual address, an entry in the page table is used to determine the actual physical location that corresponds to that virtual address. So in this way the page table is like a map that tells the OS and the hardware itself where to find specific virtual memory references. Although the sizes in this drawing are a little bit off, the sizes of the pages of the virtual memory and the corresponding page frames in physical memory are identical. By keeping the size of these two the same, we don't have to keep track of the translation of every single individual virtual address. Instead we can only translate the first virtual address in a page to the first virtual address in a page frame in physical memory and then remaining addresses in the virtual memory page will map to the corresponding offsets in the physical memory page frame. As a result, we can reduce the number of entries we have to maintain in the page table. What that means is that only the first portion of the virtual address is used to index into the page table. We call this part of the virtual address the virtual page number and the rest of the virtual address is the actual offset. The virtual page number is used as an offset into the page table and that will produce the physical frame number and that is the physical address of the physical frame in DRAM. Now to complete the full translation of the virtual address, that physical frame number needs to be summed with the offset that's specified in the later part of the virtual address to produce the actual virtual address. That resulting physical address can ultimately be used to reference the appropriate location in physical memory. Let's look at an example now. ≈

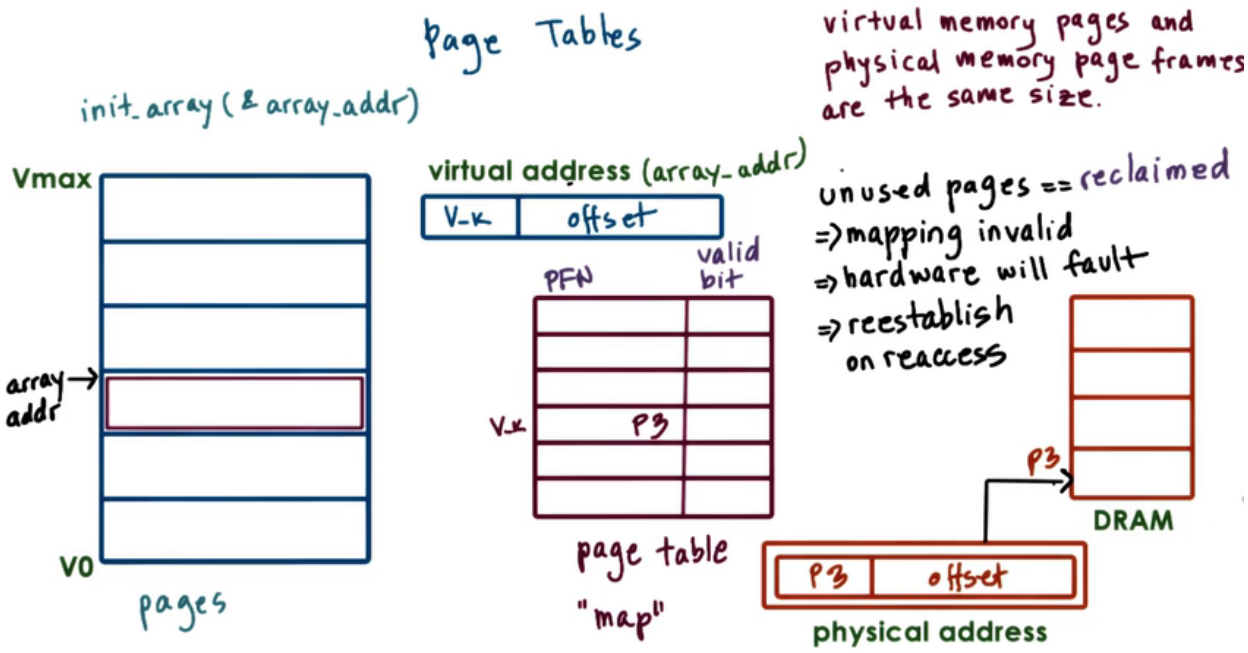
VPN (virtual page number) maps to ***PFN (physical frame number)*** and ***offset*** is added to it to produce the actual physical address.

Let's say we want to access some data structure or some array to initialize it for the very first time however we have already allocated the memory for that array into the virtual address space of the process. We've just never accessed it before. So since this portion of the address space has not been accessed before the OS has not yet allocated memory for it. What will happen the first time we access this memory is that the OS will realize that there isn't physical memory that corresponds to this range of virtual memory addresses so it will take a page of physical memory P2 in this case, a page that's free obviously, and it will establish a mapping between this virtual address (v-k and the offset address where the arrays placed in virtual memory and the physical address of page 2 in physical memory).

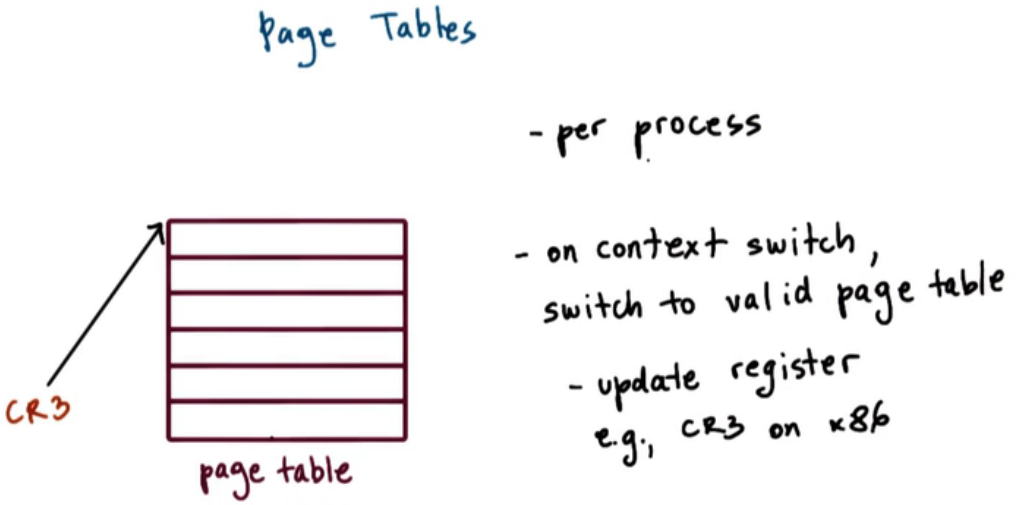


Note that i said that the physical memory for this array is only allocated when the process is first trying to access it during this initialization routine. We refer to this as allocation on first touch. The reason for this is that we want to make sure that physical memory is allocated only when it's really needed because sometimes programmers may create data structures that they don't really use.

If a process hasn't used some of it's memory pages for a long time, it is likely that those pages will be reclaimed. So the contents will no longer be present in physical memory, they will be reclaimed, they will be pushed on disk and probably some other contents will find it's way into the physical memory. In order to detect this, page table entries don't just consist of the physical frame number. Instead they also have a number of bits that tell the memory management system something about the validity of the access. For instance, if the page is in memory and the mapping is valid, then this bit is 1. If the page is not in memory, then this bit is 0 and if the hardware MMU sees that this is a bit 0 in the page table entry, it will raise a fault. It will trap to the OS. If the hardware determines that the mapping is invalid and faults, then control gets passed to the operation system. The OS at that point decides to get a number of questions. Should the access be permitted. Where exactly is the page located? Where should it be brought into DRAM? As long as a valid address is being accessed ultimately on fault, there will be a mapping that will be reestablished between a valid virtual address and a valid location in physical memory. It is likely however if the page was pushed on disk and is now being brought back into memory that it will be placed in a completely different memory location so for instance here this page is now placed in p3 and it used to be in p2. As a result, clearly the entry in the page table needs to be correctly updated.

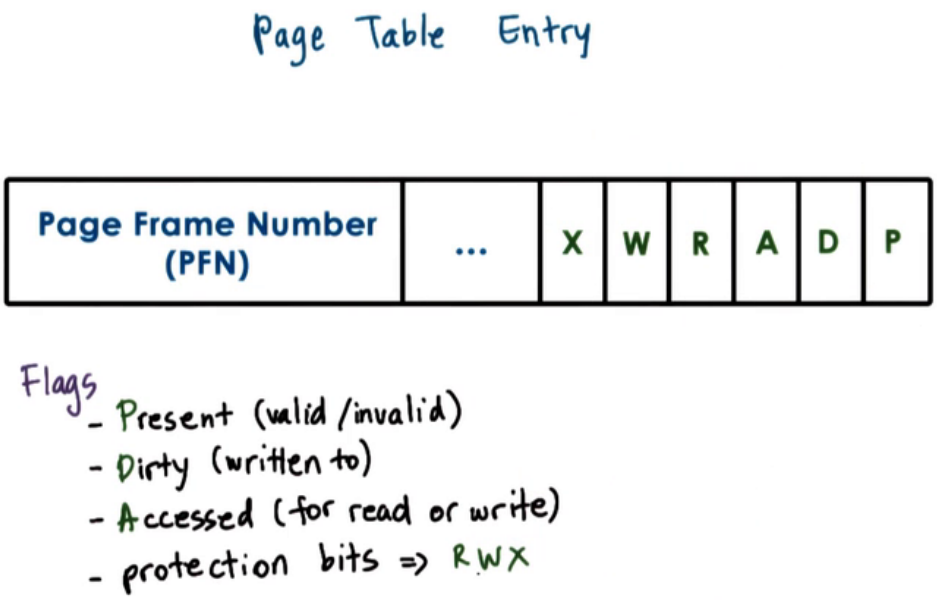


Memory brought back from disk and put back into memory in location P3 after fault detected.

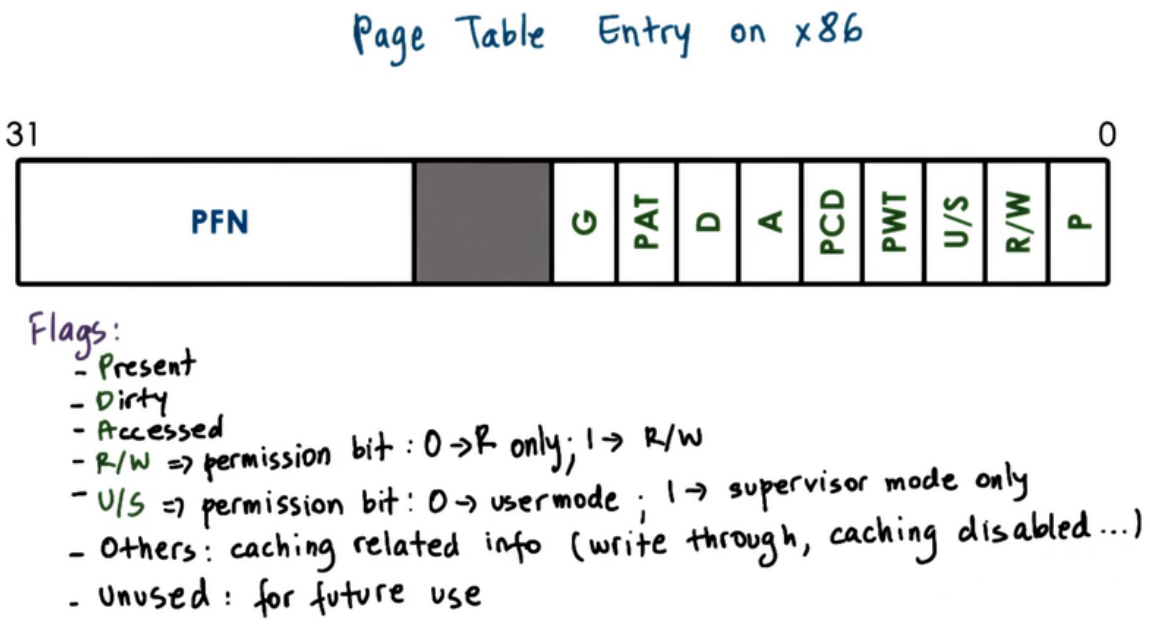


So as a final note to summarize, the OS creates a page table for every process that it runs. As a summary, the OS will maintain a page table on every single process that exists. That means that whenever a context switch is performed, the OS has to make sure that it switches to the page table of the newly context switched process. We said that hardware assists with page table accesses by maintaining a register that points to the active page table. On x86 platforms, there is a register CR3, so basically on a context switch we will have to change the contents of the CR3 register with the address of the new page table.

06 - Page Table Entry

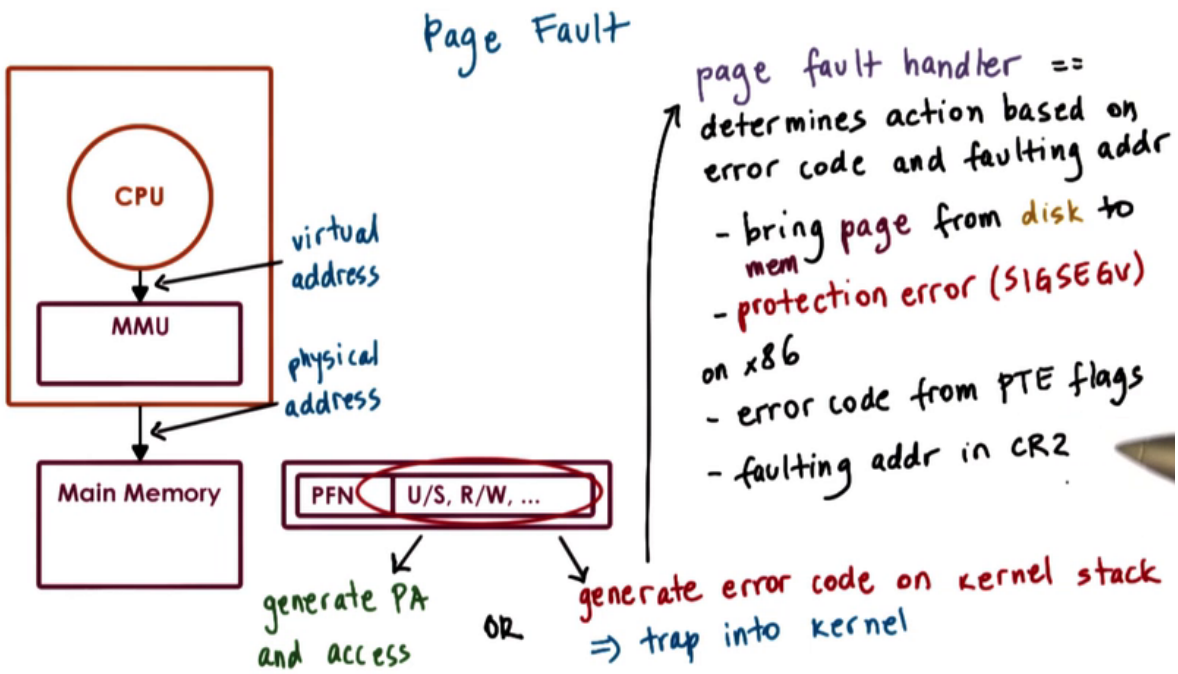


We said that every page table entry would have the physical page frame number and that it will also have at least a valid bit. This is also called the present bit since it indicates whether the content of the virtual memory are actually present in physical memory or not. There are a number of other fields that are part of each page table entry that the OS uses during memory management operations and also that the hardware understands and knows how to interpret. For instance, most hardware supports a dirty bit which gets set whenever a page is written to. This is useful for instance in file systems where files are cached in memory and then we can detect using this dirty bit which files have been written to and need to be updated on disk. Also useful is to keep to track is the access bit. This can keep track of in general whether the page has been accessed. For read or for write. Other useful information that can be maintained this part of the page table entry also would include certain protection bits. Whether a page can only be read or also written to or maybe some other operation is permissible. That was a generic discussion of a page table entry.



Here is the specifics of the pentium x86 page table entry. The flags present, dirty, and accessed, have identical meaning as in the generic page table entry we just discussed. The bit read/write it's a single bit that indicates permission. If it's value is zero, that means that that particular page can only be accessed for read only. Whereas if it's one, it means that both read and write accesses are permissible.

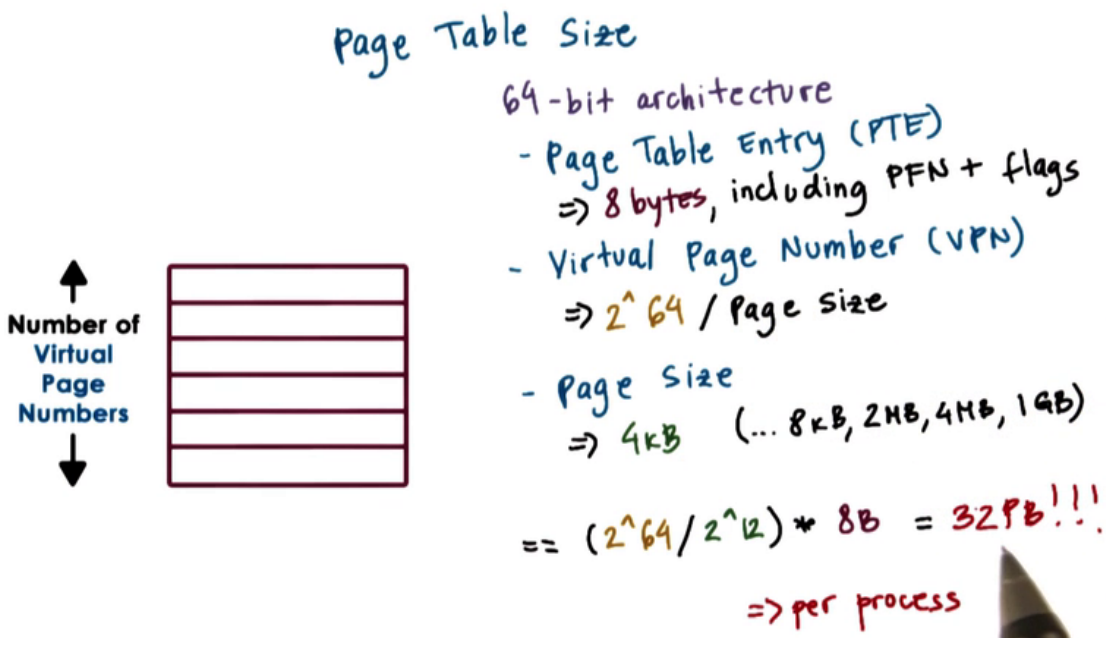
U/S is another type of permission bit which indicates whether the page can be accessed from user mode or only from supervisor mode. From when you're in the kernel basically. Some of the other flags here, they take some things regarding the behavior of the caching subsystem that the hardware has. For instance, you can specify things like whether or not caching is disabled, that some operations that's supported on modern architectures. And also, there are some parts and bits in the page table entry that are unused and hopefully in the future we will have good uses for these bits as well.



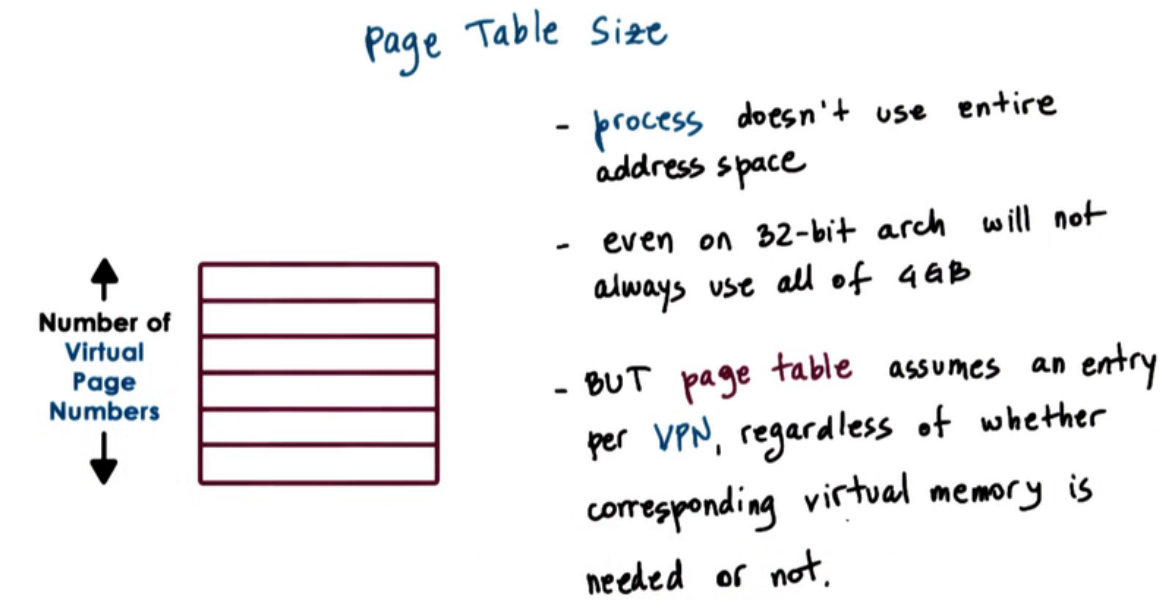
The MMU uses the page table entries not in just to perform the address translation but also relies on these bits to establish the validity of the access. If the hardware determines that a physical memory access cannot be performed, it causes a page fault. be brought in from disk, or because there is some kind of permission protection that was violated and that's why the page access If this happens, then the CPU will place an error code on the stack of the kernel and then it will generate a trap into the OS kernel. That will in turn generate a page fault handler and the page fault handler will determine what is the action that needs to be taken depending on the error code as well as the address that caused the fault. Key pieces of information in this error code will include whether or not the fault was caused because the page was not present, that it needs to is forbidden. On x86 platforms, the error code information is generated from some of the flags from the page table entry and the faulting address is also needed during the page fault handler. That one is stored in a register CR2.

07 - Page Table Size

To calculate what is the size of the page table, we know that a page table has number of entries that is equal to the number of virtual page numbers that exists in a virtual address space. For every one of these entries, we know that the page table needs to hold the physical frame number as well as some other information like the permission bits. Here's something that will make sense on a 32 bit architecture. Each of the page table entries is 4 bytes and that includes the page frame # as well as the flags. The total number of page table entries, that will depend on the total # of VPN's. And how many VPN's we can have? That's going to depend on the size of the addresses (the virtual addresses) and of the page size itself. So let's say in this example, we have a 32bit both physical memory as well as 32 bit virtual addresses. So that will be 2^32 and that will have to be divided by the actual page size. Different hardware platforms support different page sizes, but let's say we pick a common 4kb page size for this example. In that case if we do the math, you will see that the page table will be 4MB and it will be 4MB for every single process.

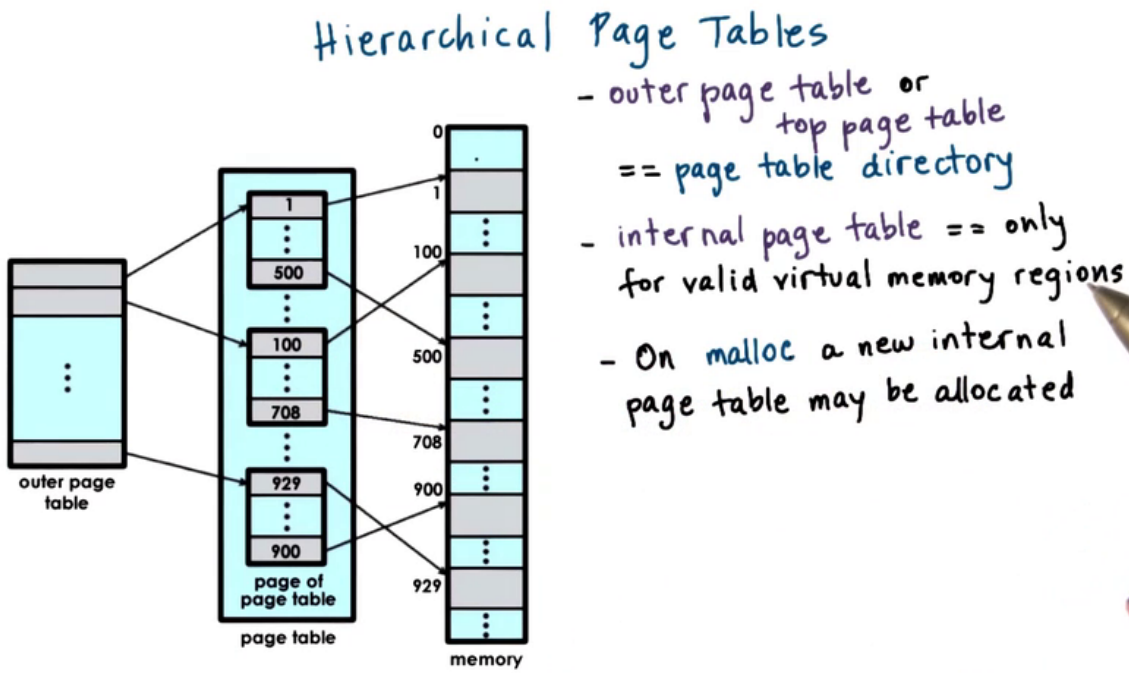


With many active processes in an OS today, this can get to be quite large. If we try to work through the same kind of example for a 64 bit architecture that say has a page table entry size of 8 bytes and let's say we use the same 4KB page size, we come up with a really scary number of 32 petabytes per process. So where does one store all of this?



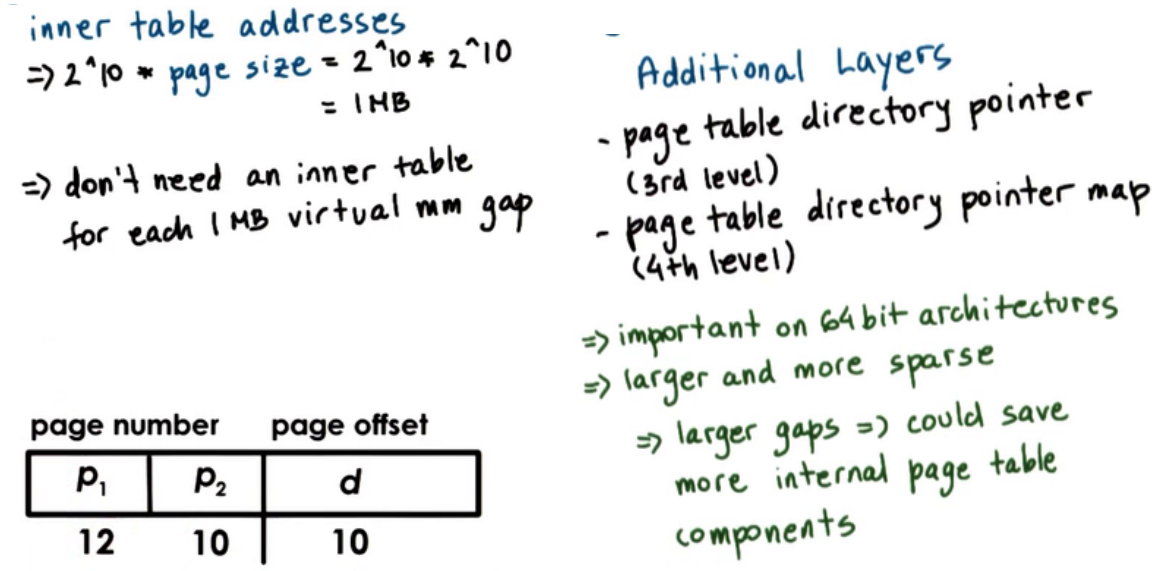
Before we answer that question it's important to know that a process will likely not use all of the theoretical virtual memory. Even in 32 bit architectures, not all of the 4MB of the virtual address space is used by every single type of process. The problem is that the page table as we described it so far, it assumes that there is an entry for every single VPN and that is regardless of whether the corresponding virtual memory region is needed by the process or not. So this page table design really explodes the requirements of the page table size and what we'll do next we'll look at some alternatives at how to represent a page table.

08 - Multi Level Page Tables

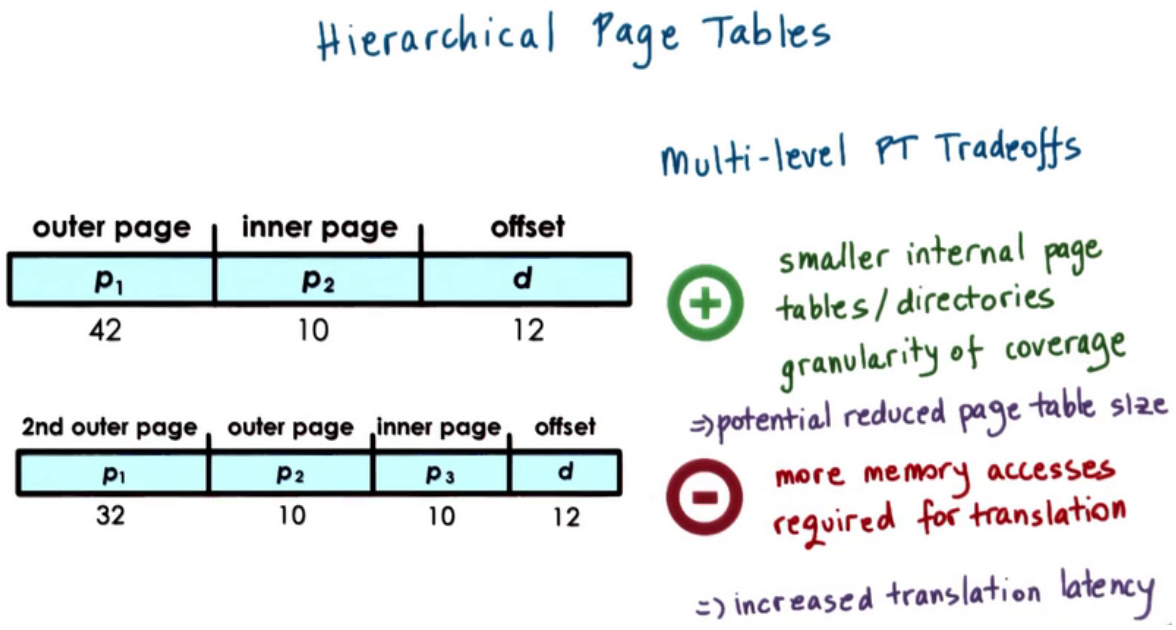


The answer to our storage issue relies on the fact that we don't really design page tables in this flat manner anymore. Instead, page tables have evolved from a flat page map to a more hierarchical multilevel structure. This figure here shows a two level page table. The outer level here is referred to a page table directory. It's elements are not pointers to actual pages, as in here, instead they're pointers to page tables. The internal page table has proper page tables as it's components that actually point to page tables. Their entries have the page frame numbers and all the protection bits for the physical addresses that are referenced by the corresponding virtual address. An important thing to note is that the internal page tables exist only for those virtual memory regions that are actually valid. So any kinds of holes in the virtual memory space will result in lack of internal page tables. So for those holes, there won't be any internal page tables allocated for them. If a process requests via malloc, additional virtual memory to be allocated to it, the OS will check and if necessary it will allocate an additional internal page table element and set the appropriate page table directory to correspond to that entry. That new internal page table entry will correspond to some portion of the newly allocated virtual memory region that the process has requested. To find the right element in this page table structure, the virtual address is split into yet another component. Using this address format, this is what we need to perform to determine the correct physical address. First, the last portion of the address is still the offset. So that's going to be used to compute the offset within the actual page, within the actual physical page. The first two components of the address are used as indexes into the page table, into the different levels of the page table hierarchy and they are ultimately going to produce the physical frame number that's the starting address of the physical region. The first portion is used as an index into the outer page table, so that will determine the page table directory entry that points to the actual page table. And then the second index is used as an index into this page table, into the internal page table. This will produce the page table entry that consists of the physical frame number and then we add that with the offset just like before and compute the physical address. In this page, the address format is such that it uses 10 bits for the internal table offset. That means that this internal page table can address 2 ^ 10 elements. So 2^10 pages can be addressed in the internal page table. Since we used 10 bits as the offset into the actual page, that means that the page itself is also 2 ^ 10 in size. Therefore if we do the math, every single internal page table can address 2 ^ 10 (the number of entries) times the page size (2 ^ 10) = 1MB of memory. What that means is that whenever there is a gap in virtual memory that's 1MB size, we don't need to allocate that internal page table. So that will reduce the overall size of the page table that's required for a particular process. This is in contrast with the single level page table design where the page table has to be able to translate every single virtual address and it has entries for every single virtual page number. So

clearly the hierarchical page table model helps in reducing the space requirements for a page table.

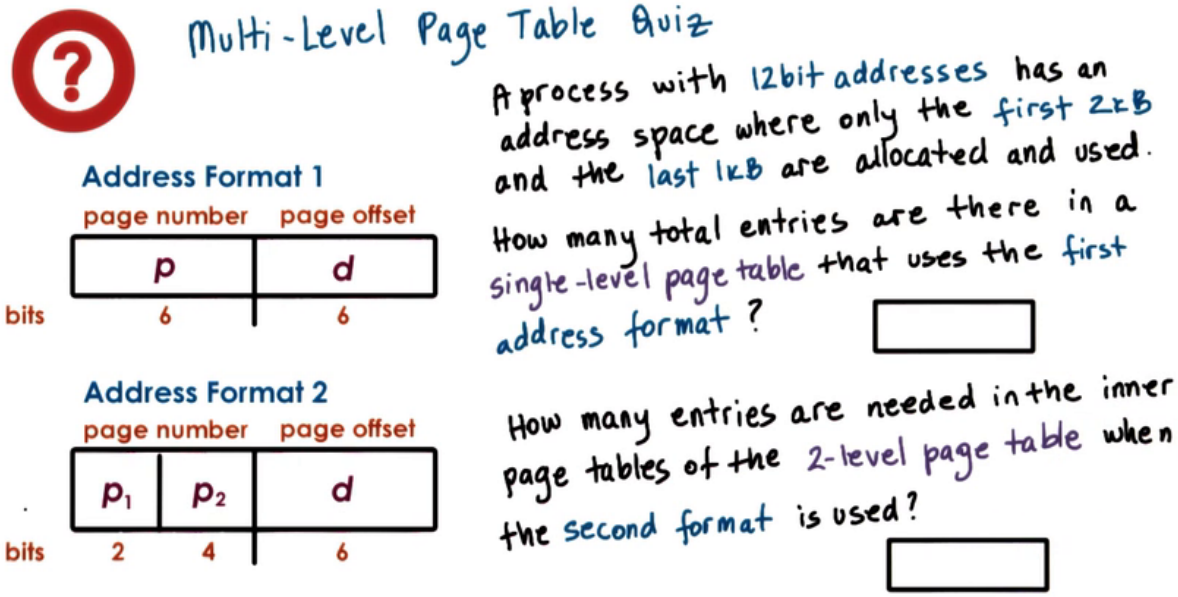


The scheme can be further extended to use additional layers using the same principle. For instance we can add another third level that can consist of pointers to page table directories. Adding yet another 4th level to this, would consist of a map of pointers to page table directories. This technique is particularly important in 64 bit architectures. There, not only are the page table requirements larger. But also the virtual addresses of processes on these 64 bit architectures tend to be more sparse. If it's more sparse, that means that it will have larger gaps in the virtual address space region. And the larger the gaps, the larger the number of internal page table components that won't be necessary as a result of the gap. In fact with a 4 level addressing, we may end of saving entire page table directories as a result of certain gaps in the virtual address space. Let's look at a quick example.



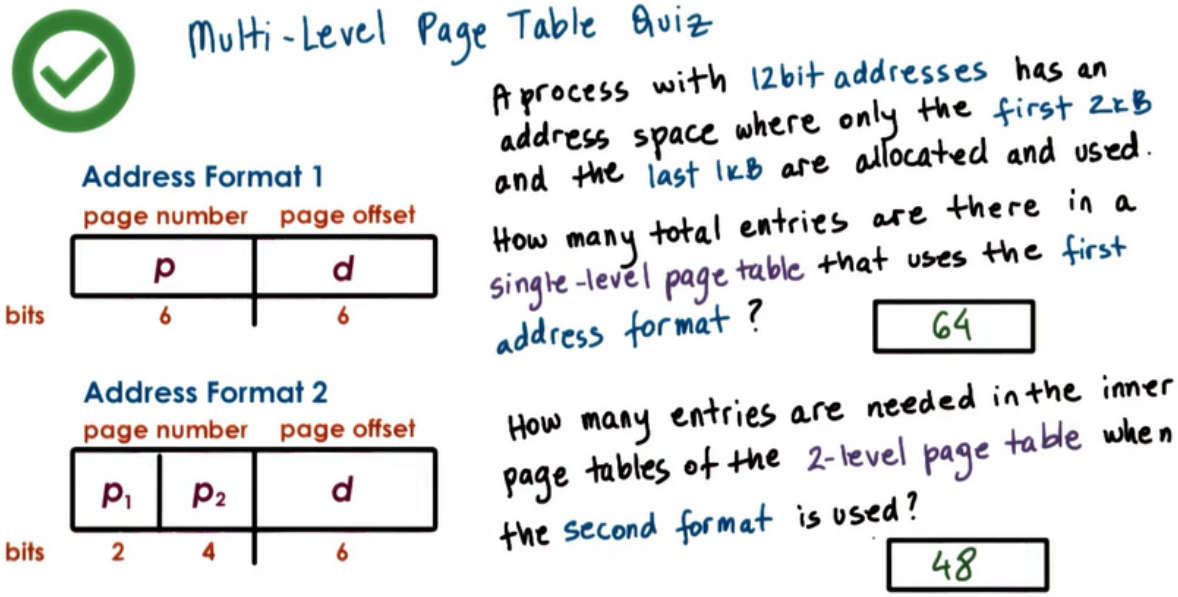
These two figures show how a 64 bit virtual address can be interpreted to determine which indexes are used into the different levels of the page table hierarchy. The top figure has 2 page table layers whereas the bottom one has 3 page table layers. In both of these figures, the offset field is the actual index into the actual physical page table. There is a trade off in supporting multiple levels in the page table hierarchy. As we add multiple levels, the internal page tables and page table directories end up covering smaller regions of the virtual address space. As a result, it is more likely that the virtual address space will have gaps that will match that granularity and we will be able to reduce the size of the page table. The downside of adding multiple levels in the page table is that there will be more memory accesses that will be required for translation since we'll have to access each of the page table components before we ultimately produce the physical address. Therefore the translation latency will be increased.

09 - Multi Level Page Table Quiz



Hints/Formulae:
# of entries in a page table = 2^(virtual page number length)
Page size = 2^(page offset length)

10 - Multi Level Page Table Quiz



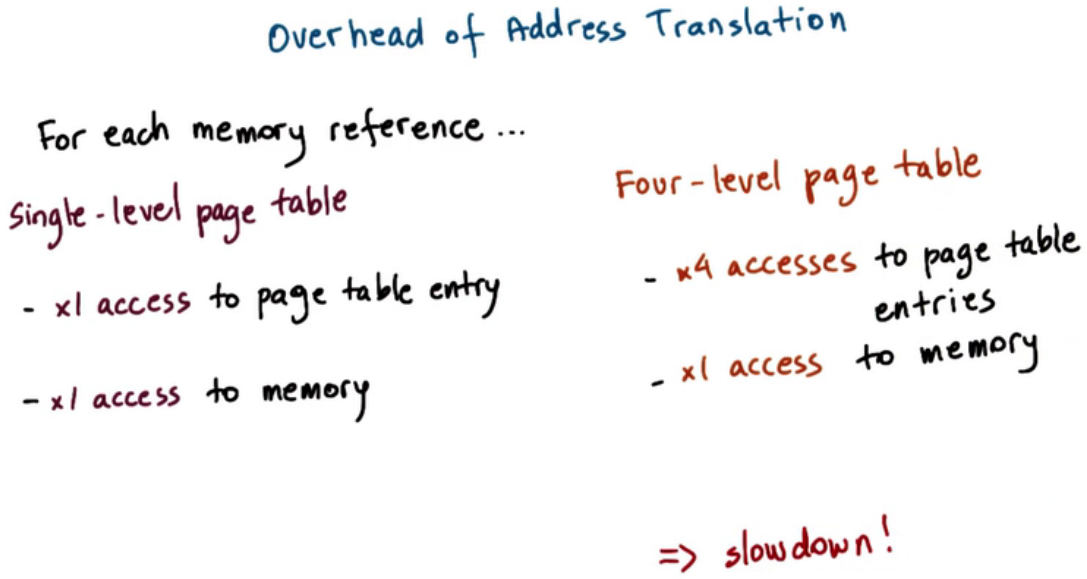
2^6 = 64 / 4 blocks of 1kb = 16 entries per inner page table
$(2^2 * 2^4) - (2^4)$ = 48

One of the first level page table entries can be eliminated since we don't have it allocated.

In both formats, the page offset is 6 bits. That means that each of the pages is 2^6. That's 64 bytes. In the case of the single page table, 6 bits are used for the virtual page number. That means that there will be 2^6 virtual pages.

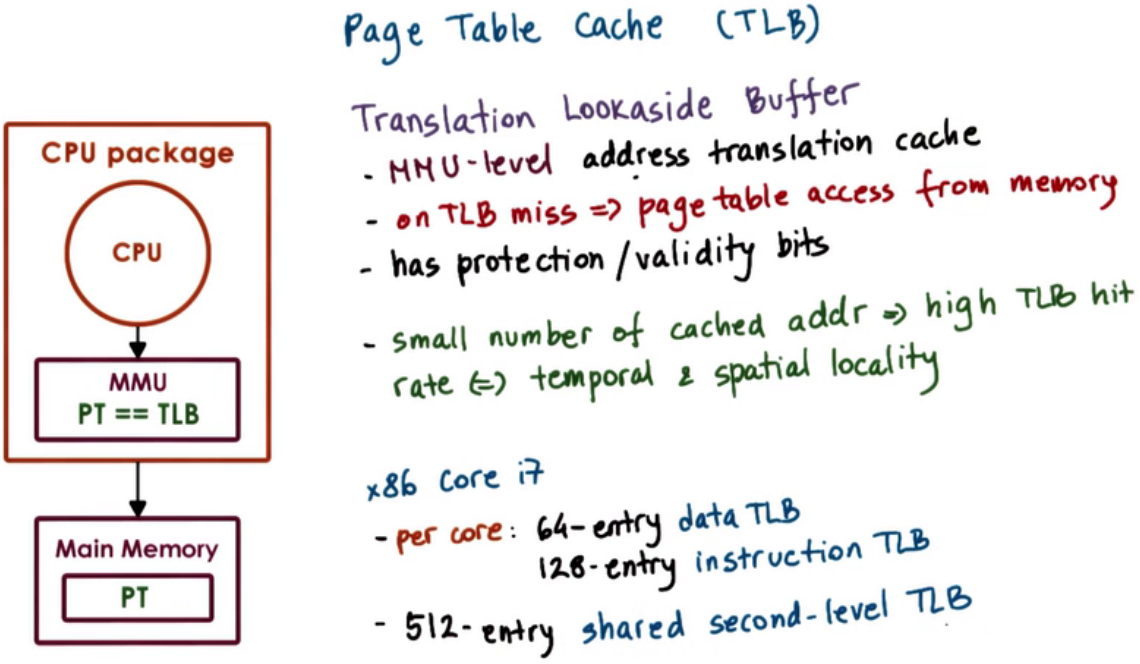
In the second address format, the first 2 bytes is used as an index into the outer page table. The page table directory and the inner 4 bits are used as an index into the inner page tables. The outer page table entries will address 2 ^ 10 virtual addresses from the virtual address space (4+6). That means every single element of the outer page table can be used to hold the translations for 1 KB of the virtual addresses. Given that the process is such that only the first 2 KB and the last 1KB of the virtual address space are allocated, that means that one of the entries in the outer page table will not really need to be populated with a corresponding inner page table. So we can save the memory that's required for that inner page table. Now the inner page table is the reuse of 4 bit index to index into the inner page table. That means that it will have 16 entries, every single one of the inner page tables will hold 16 entries, so therefore the total number of entries that are needed across the remaining inner page tables will be 48. So we reduce the page table size by 25% by choosing this multi level page table format in this particular example.

11 - Speeding Up Translation TLB



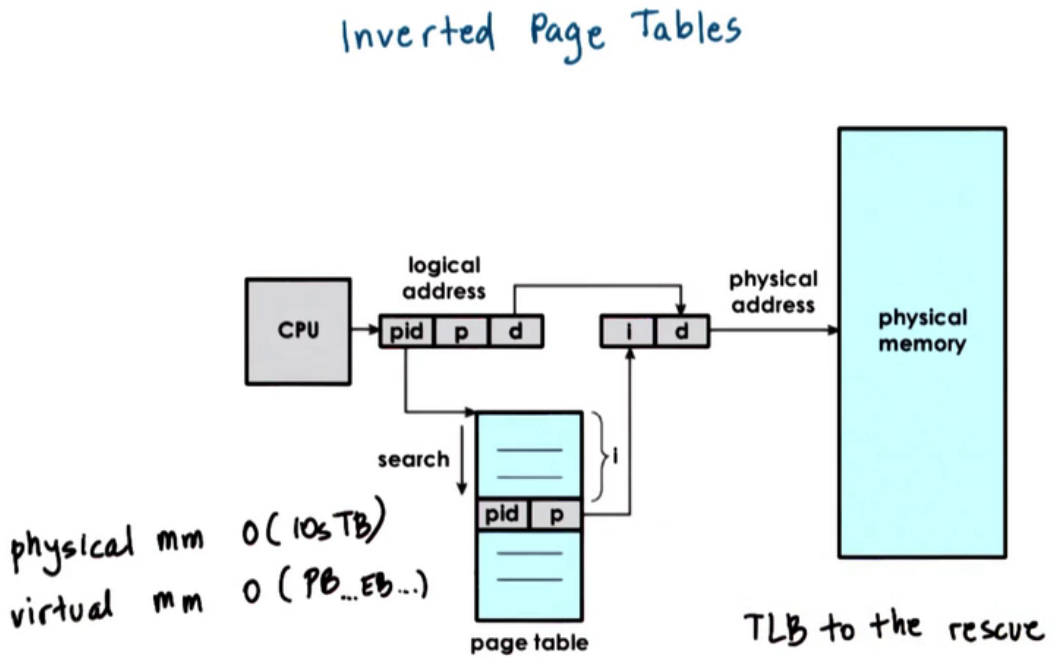
Now we know that adding levels to our address translation process will reduce the size of the page tables but it will add some overhead to the address translation process. In the simple, single level page table design, a memory reference will actually require 2 memory references. 1 to access the page table entries so that we can determine the physical frame number and the second one to actually perform the proper memory access at the correct physical address. In the 4 level page table however, we will need to perform 4 memory accesses to read the page

table entries at each level of the memory hierarchy before we can produce the physical frame number and only afterwards are we able to actually perform the proper access to the correct physical memory location. Obviously this can be very costly and can lead to a slowdown.

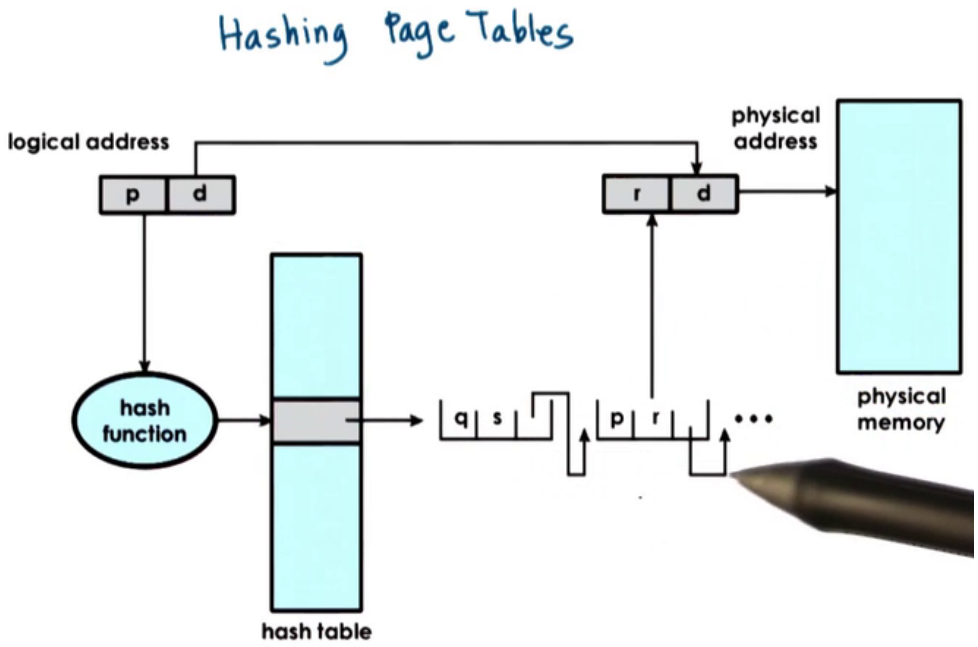


The standard technique to avoid these repeated accesses to memory is to use a page table cache. On most architectures, the MMU hardware integrated a hardware cache that's dedicated for caching address translations and this cache is called the Translation Lookaside Buffer (TLB). On each address translation, first the TLB cache is quickly referenced and if the resulting address can be generated from the TLB contents, then we have a TLB hit and we can bypass all of the other required memory accesses to perform the translation. Of course, if we have a TLB miss, the address isn't present in the TLB cache, then we have to perform all of the address translation steps by accessing the page tables from memory. In addition to the proper address translation, the TLB entries will contain all of the necessary protection and validity bits to verify that the access is correct or if necessary to generate a fault. It turns out that even a small number of entries in the TLB can result in a high TLB rate and this is because we have typically a high temporal and spatial locality in the memory references. On recent x86 platforms for instance, there is a separate TLB for data and instructions and each of those have a modest number of entries. 64 for the data. 128 for the instruction TLB. These are per core. In addition to these 2, there is also another shared second level TLB that's shared across all cores and that one is a little bit larger (512 entries). So this is for the i7 intel platforms. And this was determined to be sufficiently effective to address the typical memory access needs of processes today.

12 - Inverted Page Tables

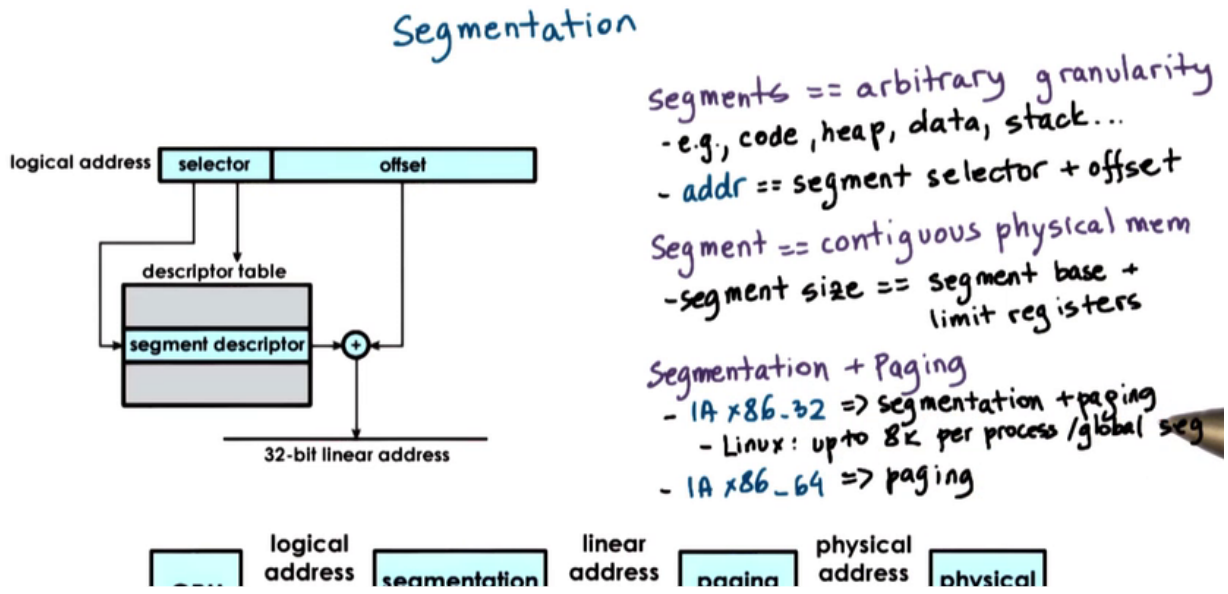


Completely different way to organize the address translation process is to create so called inverted page tables. Here the page table entries contain information one for each element of the physical memory. For instance, if we're thinking about physical frame numbers, each of the page table elements will correspond to one physical frame number. Today on the most high end platforms we have physical memory that's on the order of 10's of terabytes. Whereas the virtual memory of an address space can reach petabytes and beyond. Clearly it would be much more efficient to have a page table structure that's on the order of the available physical memory vs something that's on the order of the virtual memory that a process can have. To find the translation, the page table is searched based on the process ID and the first part of the virtual address, similar to what we saw before. When the appropriate PID and P entry is found into this page table, the index, the element where this information is stored, that will denote the physical frame number of the memory location that's indexed by this logical address. So then that is combined with the actual offset to produce the physical address that is being referenced from the CPU. The problem with inverted page tables is that we have to perform a linear search of the page table to see which one of its entries matches the PID, P information that's part of the logical address that was presented by the CPU. Since the physical memory can be arbitrarily assigned to different processes, the table isn't really ordered, there may be two consecutive entries that represent memory allocated to 2 different processes. And there really isn't some clever search technique to speed up this process. In practice, the TLB will catch a lot of these memory references so this detailed search is not performed very frequently. However we still have to perform it periodically, so we have to do something to make it a little more efficient.



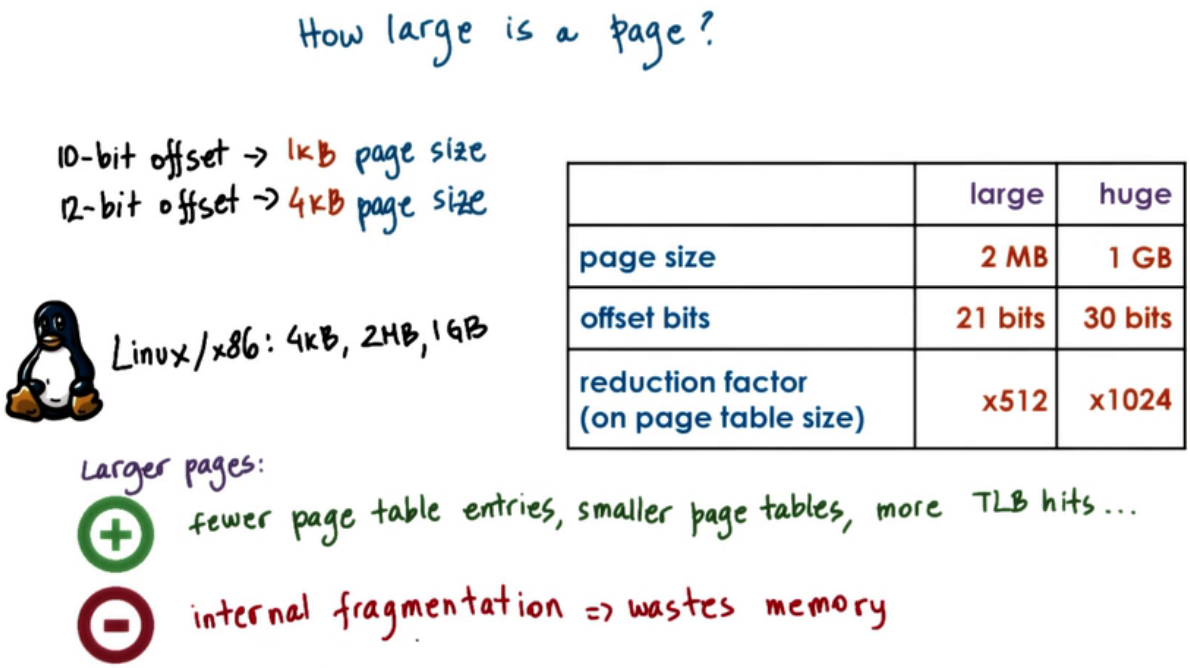
To address this issue, inverted page tables are supplemented with so called hashing page tables. In most general terms, a hashing page table looks something as follows. A hash is computed on a part of the address. And that is an entry into the hash table that points to a linked list of possible matches for this part of the address. So that allows us to basically speed up the process of the linear search, to narrow it down to a few possible entries, into the inverted page table. As a result, we speed up the address translation.

13 - Segmentation



In addition to paging, we said that virtual to physical memory mappings can be performed using segments. So the process is referred to as segmentation. With segments, the address space is divided into components of arbitrary granularity, of arbitrary size, and typically the different segments will correspond to some logically meaningful components of the address space like the code, the heap, the data, etc. A virtual address in the segmented memory mode includes a segment descriptor and an actual offset. The segment descriptor is used in combination with a descriptor table to produce information regarding the physical address of the segment and the two are combined, that information along with the offset, they're combined to produce the linear address of the memory reference. In its pure form, a segment could be represented with a continuous portion of physical memory. In that case, the segment would be defined with its base address and its limit registers which implies also the segment size. So we basically can have segments with different size using this method. In practice, segmentation and paging are used together. What this means is that the address that is produced using this process and that one we call the linear address is then passed to the paging unit, so it will be passed to a multi level, hierarchical page table to ultimately compute the actual physical address that is used to reference the appropriate memory location. The type of address translation that's possible on a particular platform that's determined by the hardware. For instance if we look at the intel platforms, the x86 platforms, on 32 bit hardware where segmentation and paging are supported for these platforms on linux allows up to 8000 segments to be available per process and another 8000 global segments. At the same time on 64 bit intel platforms, segmentation and paging are supported for backward compatibility however the default mode is to just use paging.

14 - Page Size

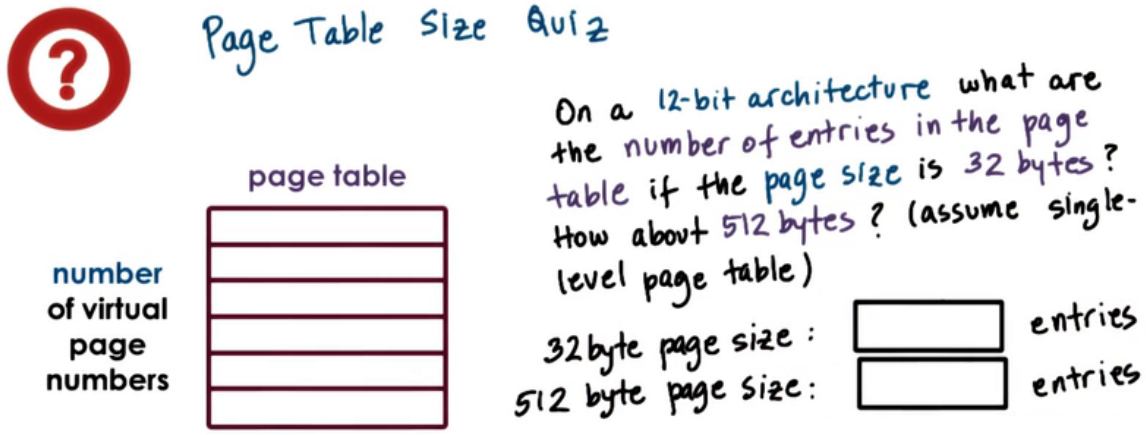


	large	huge
page size	2 MB	1 GB
offset bits	21 bits	30 bits
reduction factor (on page table size)	x512	x1024

VPN entries (page table entries, page table size) are less for larger page sizes.

So far we've glossed over any discussion of what is the appropriate page size or how large is a page. In the examples that we showed so far regarding the address formats, we used 10 bits for the offset or 12 bits for the offset. Well this offset determined what is the total amount of addresses in the page and therefore it determined the page size that we were discussing in those examples. So in the examples in which we had 10 bit offset in the address field, that mean that these 10 bits could be used to address 2 ^10 bytes in the page and therefore it mean that the page size was 1KB. Similarly, the examples that had 12 bit offset for the address format. That means that they could have addressed 2^12 or 4KB pages. But what are the page sizes in real systems. These are some examples that we cooked up. In practice, systems support different page sizes. For Linux and x86, there are several common page sizes. 4KB page size is pretty popular and that's the default in the Linux x86 environment. However page sizes can be much larger. 2MB, 1GB… The 2MB are referred to large pages as opposed to the regular 4KB ones. In addition, x86 supports "huge" pages and these are 1GB in size. In the first case, to address 2MB of content in a page, we need 21 bit for the page offset and in the case of a huge page, we need 30 bits as an offset to compute the physical address. So one benefit of using these larger page sizes is that more bits in the address are used for these offset bits and therefore fewer bits are used to represent the virtual page numbers so there will be few entries that are needed in the page table. In fact, use of these large page sizes will significantly reduce the size of the page table. Compared to the page table size that's needed when we're working with 4K pages, large pages will reduce the page table size by a factor of 512. And then switching to huge page sizes will reduce the page table size by another 512. So in summary, the benefits of larger page sizes are the fact that they require smaller page tables due to the fact that they're fewer page table entries that are needed and we can have additional benefits, often such as for instance, increased number of TLB hits, just because we'll be able to translate more of the physical memory using the TLB cache. The downside of the larger pages is the actual page size. If this large virtual memory page is not densely populated, there will be larger unused gaps within the page itself and that will lead to internal fragmentation. There will be basically wasted memory in these allocated regions of memory depending on the page size. So because of this issue, smaller page sizes of 4KB are commonly used. There are some settings like databases or in memory data stores where these large or huge page sizes are absolutely necessary and make the most sense. I should note that on different systems, depending on the OS and the hardware architecture, different page sizes may be supported. For instance, on Solaris 10 on sparc architecture the page size options are 8KB, 4MB, or 2GB.

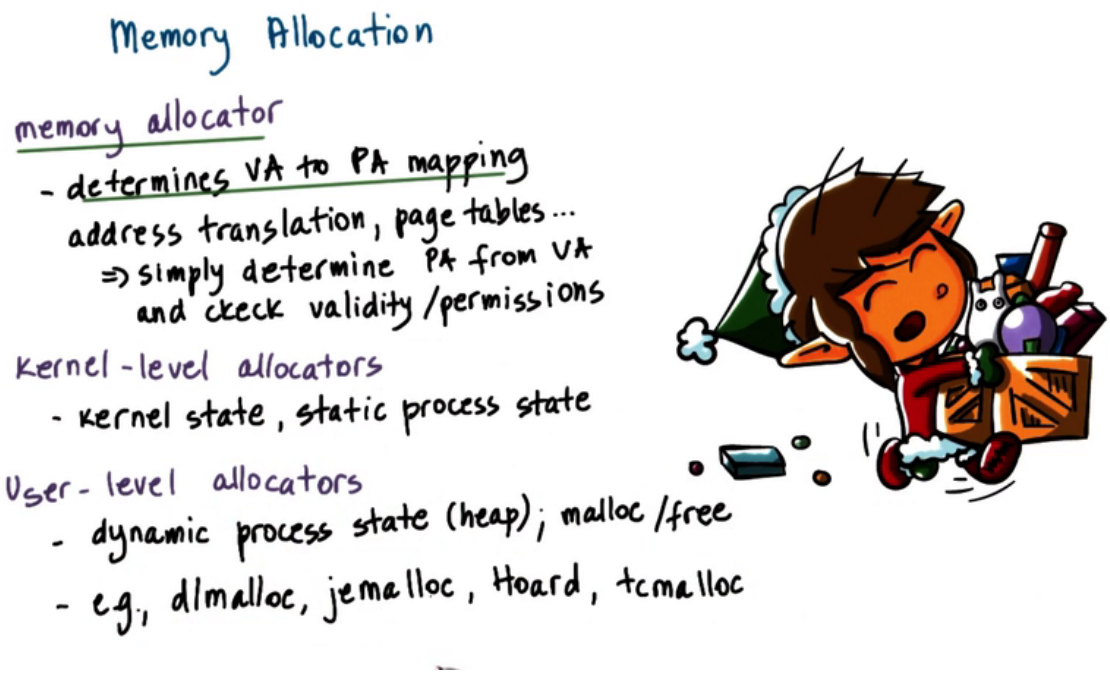
15 - Page Table Size Quiz



16 - Page Table Size Quiz

If the architecture is 12 bits that means that the addresses are 12 bits long. if the page size is 32 bytes, then we need 5 bits for the offset into that page that leave 7 bits for the virtual page number and therefore we will need 2^7 = 128 total number of entries in the page table. Using the same logic for the 512 byte pages, we will need 9 bits out of the total 12 bits for the offset into the page. And therefore we will be left 3 bits for the virtual page number. As a result the page table will need to have entries for all of the 2^3 number of virtual pages. So it will have a total of 8 entries. As you can see, this example illustrates the impact of using a larger page sizes on the requirements of the page table size.

17 - Memory Allocation

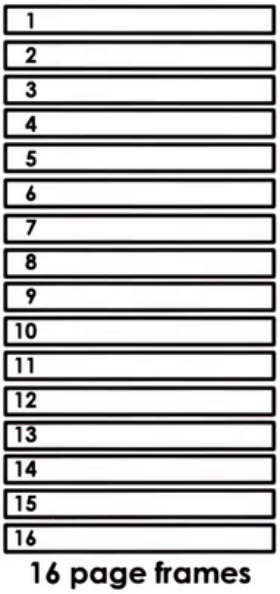


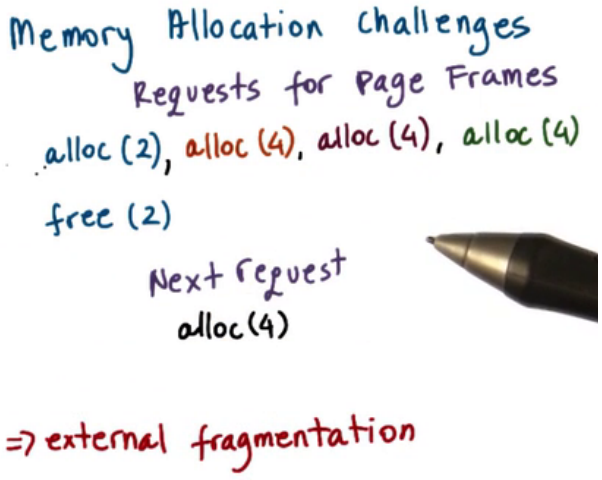
PA = Physical Address, VA = Virtual Address

So far we have described how the OS controls the processes access to physical memory but what we didn't explain was how the OS decides how to allocate a particular portion of the memory to a process in the first place. This is the job of the memory allocation mechanisms that are part of the memory management subsystem of an operating system. Memory allocation incorporates mechanisms that decide what are the physical pages that will be allocated to a particular memory region. What are the physical addresses that will correspond to a specific virtual address. Once the memory allocator establishes a mapping, the mechanisms that we discussed so far like the address translation, using page tables, they are simply used to determine a physical address from a virtual address that the process presents to the CPU. And also to perform all necessary checks regarding the validity of the access or the access permissions. Memory allocators can exist at the kernel level as well as at the user level. Kernel level allocators are responsible for allocating memory regions such as pages for the kernel, for various components of the kernel state, and also these are used for certain static state, for the processes when they're created, like for their code, stack, or initialize data. In addition the kernel level allocators are responsible for keeping track of the free memory that's available in the system. The user level allocators are used for dynamic process states, for instance the heaps, so this is state that is dynamically allocated during the process execution. The basic interface for these allocators includes malloc and free. What these calls do is that they request from the kernel some amount of memory from its free pages and ultimately release it when they're done. Once a kernel allocates some memory to a malloc call, the kernel is no longer involved in the management of that space. That memory will at that point be used by whatever user level allocator is being used and there are a number of options out there right now that have certain different trade offs, in terms of their cache efficiency, or their friendliness with respect to how

they behave in a multi-threaded environment or other aspects. We will not discuss the internals of these user level allocators in this course. Instead we will briefly describe some of the basic mechanisms that are used in the kernel allocators and the same kinds of design principles are used in the design of some of the user level allocators that are out there today.

18 - Memory Allocation Challenges

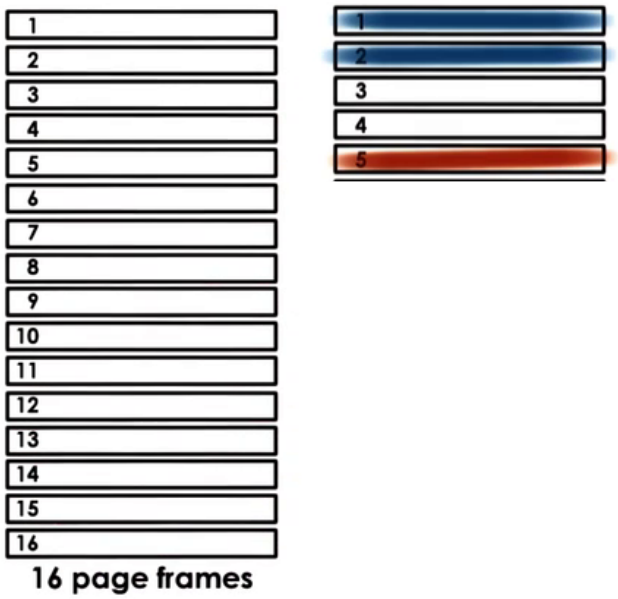




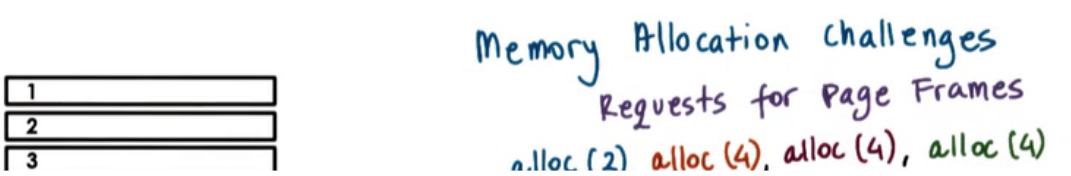
Before we talk about the kernel level allocators, I want to describe a particular memory allocation challenge that needs to be addressed. Consider a page based memory manager that needs to manage these 16 physical page frames. Let's say this memory manager takes these

requests of sizes 2 or 4 page frames and let's say it's facing the following sequence of memory requests. The first memory allocation is for request of 2 page frames and then the rest of the request is for 4 pages. So let's say the memory allocator allocates these requests in order and the end result of this will be that this will be the memory allocation, how the physical memory is used to satisfy these requests and their 2 free page frames. Let's say next the 2 pages that were initially allocated are freed. So now you likely can already imagine what the problem is. If at this point a next request comes to allocate 4 pages, there are 4 free pages in the system however this particular allocator cannot satisfy this request since these pages are not contiguous.

External fragmentation occurs where we have

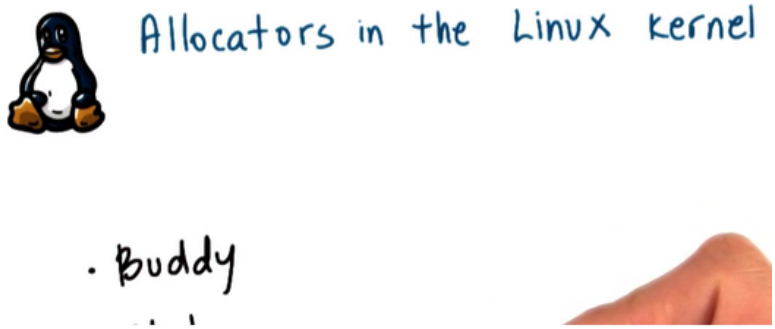


Let's say the requirement with these allocation request was for these memory pages to be contiguous so in that case this allocator cannot meet this requirement. This example illustrates a problem that's called external fragmentation. This occurs where we have multiple interleaved allocate and free operations and a a result of them we have holes of free memory that's not contiguous , and therefore requests for larger contiguous memory allocations can not be satisfied. In the previous example, the allocator had a policy in which the free memory was handed out to consecutive request in a first come first serve manner.

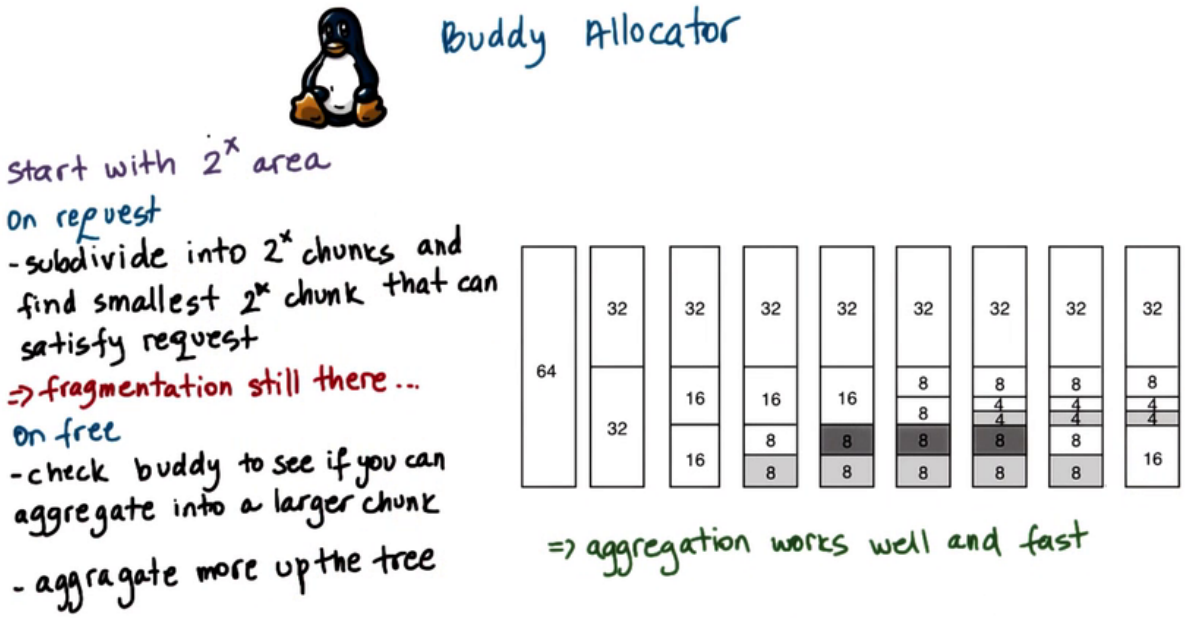


Let's consider an alternative, in which the allocator probably knows something about the request that are are coming, it knows that they will be coming for consecutive regions of 2 and 4 page frames. In the 2nd case, when the 2nd request for allocating 4 pages comes, the memory allocator isn't allocating it immediately after the first allocation, but instead it's leaving some gap. The 2nd allocation for 4 pages comes in at a granularity of 4 pages. The rest of the allocations are satisfied further below. Now when the free request comes in, these 2 first pages are freed. The system again has 4 free pages however they're consecutive. Therefore this next request for 4 free pages can actually be satisfied in the system. What we see in this example, is that when these pages are freed there was an opportunity for the allocator to call us to aggregate these adjacent areas of free pages into one larger free area. That way was more likely for the allocator to satisfy these future larger requests. This example illustrates some of the issues that an allocation algorithm needs to be concerned with. To avoid or to limit the extent of fragmentation and to allow for quick coalescing and aggregation of free areas.

19 - Linux Kernel Allocators
Allocation mechanisms: Buddy and slab allocators.



To address the free space fragmentation and aggregation issues we mentioned in the previous morsel, the linux kernel relies on two basic allocation mechanisms. The first one is called the buddy allocator. The second one is called the slab allocator.

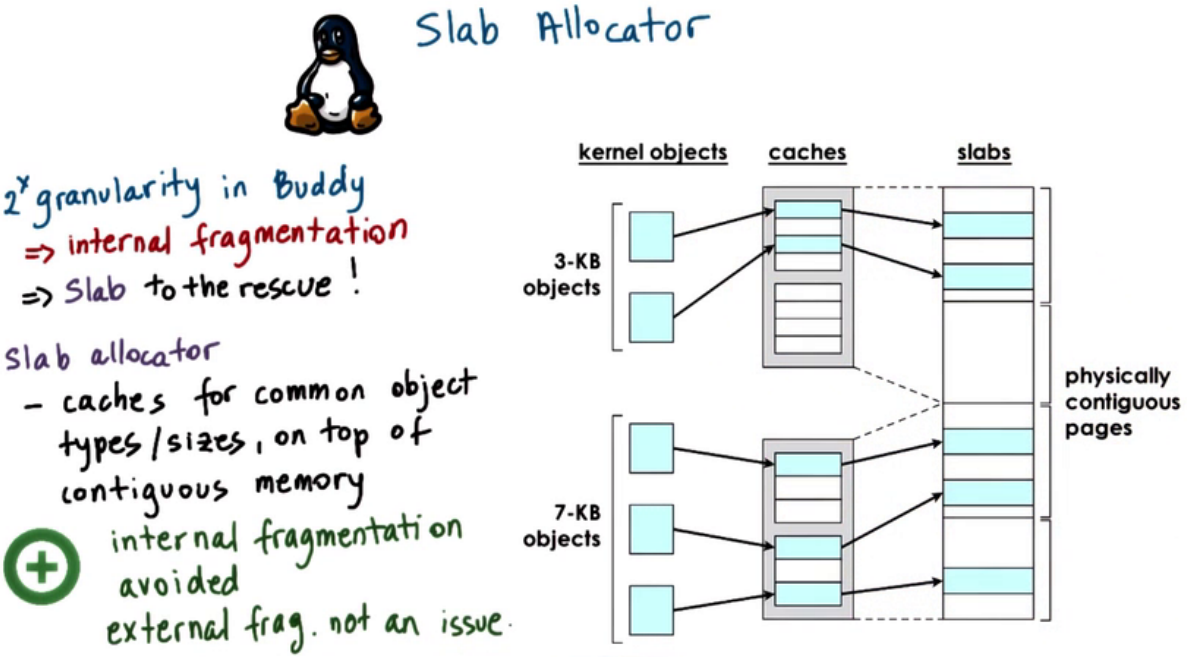


Errata

@2:34 the final bullet for the **Buddy Allocator** is misspelled. It should read **aggregate more up the tree**.

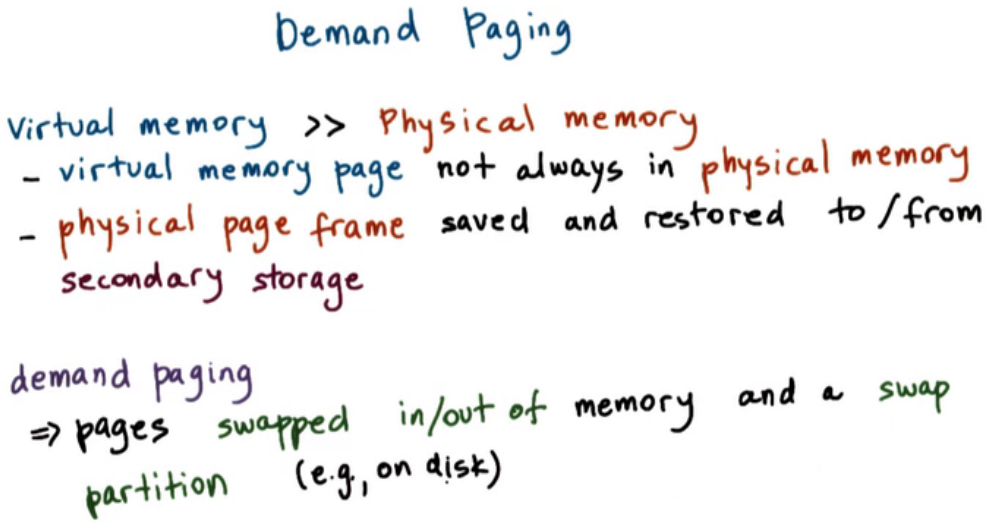
The buddy allocator starts with some consecutive memory region that's free that's of a size of power of 2. Whenever a request comes in, the allocator will subdivide this large request into

smaller chunks such that every one of them is also a power of 2 and it will continue subdividing until it finds the smallest chunk that's of a size of a power of 2 that can satisfy the request. For instance in this figure when the first request of 8 pages came in, the buddy allocator subdivided the region that was 64 (the original area). First into 2 chunks of 32 pages. Then it subdivided one of these 32 page chunks into chunks that were 16 pages each. Then it subdivided this 16 page chunk into pages that were 8 pages each. Then it turns out that this 8 page chunk satisfied the request that was for 8 pages so that was great. Subsequently, another request for 8 pages came in and then another request for 4 pages came in and for that reason this chunk of 8 pages was subdivided into 2 chunks of 4. Now when this 8 pages is freed, there will be some fragmentation here, however when the next 8 page region is freed, the algorithm will quickly be able to combine these two to produce a 16 page free space. So fragmentation still exists in the buddy allocator, but its benefits are that when a request is freed it has a way to quickly find out how to aggregate data. When this allocation of 8 pages was freed, with the buddy allocator it was very easy to figure out what is the start of the adjacent allocation. Where does the buddy of this 8 page region start? If we didn't have this information, if we didn't know if this adjacent region was also an 8 page region, we would have had to potentially scan all of these pages to determine which one is free and which one isn't so as to figure out whether we can increase this free space to 9, 10, 11, 12 or some other number of pages. But the benefit of the buddy algorithm is that the aggregation of the free areas can be performed really well and really fast. The checking of what are the free areas in the system can further be propagated up the tree to check the buddies of this 16 page free area and then the buddy of the 32 page free area and so forth. The reason why these areas are the power of two is so that the addresses of each of the buddies differ by only 1 bit. This makes it easier to perform the necessary checks when combining or splitting chunks.

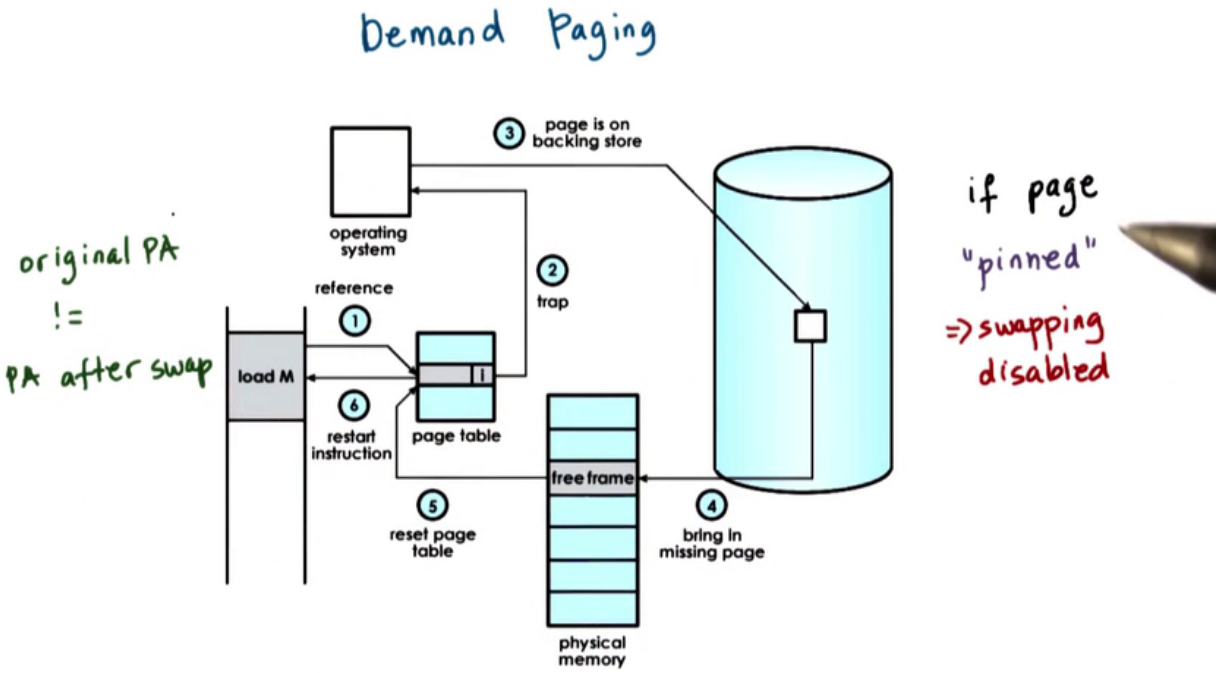


The fact that allocations using the buddy algorithm have to be made in a granularity of a power of 2 means that there will be some internal fragmentation using the buddy allocator. This is particularly a problem because there are a lot of data structures that are common in the linux kernel that are not of a size that's close to a power of 2. For instance the task data structure (task_struct) is 1.7K. To fix this issue, linux also uses the slab allocator in the kernel. The allocator builds custom object caches on top of slabs. The slabs themselves represent contiguously allocated physical memory. When the kernel starts it will precreate caches for the different object types. For instance, it will have a cache for task_struct or for the directory entry objects, then when an allocation comes for a particular object type then it will go straight to the cache and it will use one of the elements in this cache. If none of the entries is available, then the kernel will create another slab and it will pre allocate an additional portion of contiguous physical memory to be managed by the slab allocator. The benefit of the slab allocator is that it avoids internal fragmentation. These entities that are allocated in the slab, they're the exact same size as the common kernel objects. Also, external fragmentation is not really an issue. Even if we free objects in this object cache, future requests will be of a matching size and then they can be made to fit in these gaps. So the combination of the slab allocator and the buddy allocator that are used in the linux kernel, these are really effective methods to deal with both the fragmentation and also the free memory management challenges that are present regarding memory management in operating systems.

20 - Demand Paging



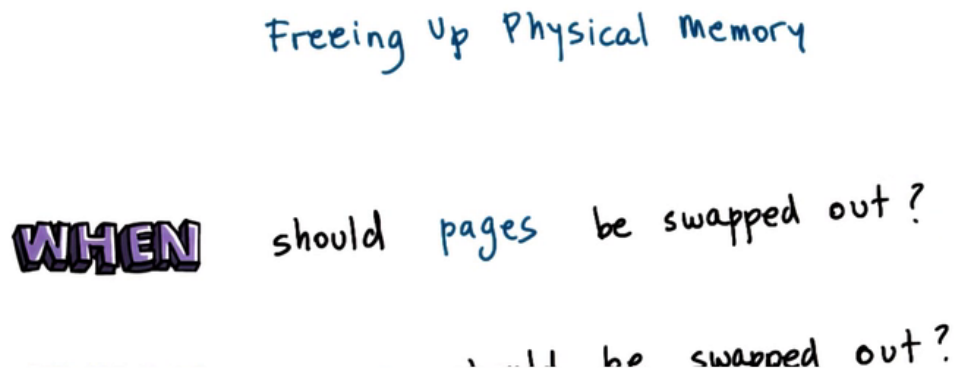
Since the physical memory is much smaller than the addressable virtual memory, allocated pages don't always have to be present in physical memory in DRAM. Instead, the backing physical page frame can be repeatedly saved and restored to and from some secondary storage like disk for instance. This process is referred to as paging or demand paging and traditionally with demand paging, pages are moved between main memory and the storage device such as disk where a swap partition resides. In addition to disk, the swap partition can be an another type of storage medium like a flash device or it can even sit in the memory of another node.



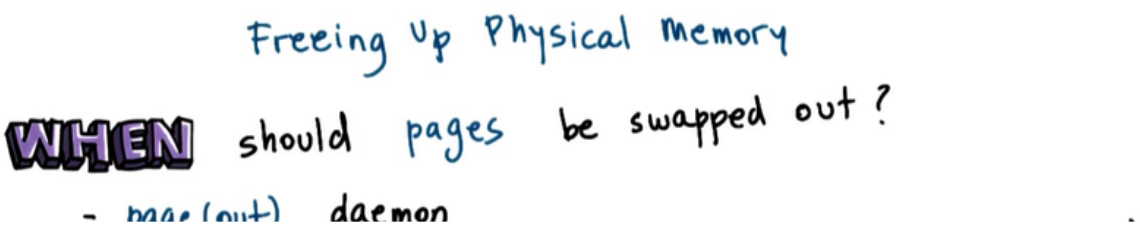
Let's see how paging works. When a page is not present in memory, it has it's present bit in the page table entry that's set to zero. When There is a reference to that page, then the memory

management unit will raise an exception and that will cause a trap into the OS kernel. On an access, the MMU will raise an exception (it's called a page fault) and this will be pushed into the OS so it will trap into the OS kernel. At that point, the OS kernel can determine that this exception is a page fault, it can determine that it had previously swapped out this memory page onto disk. It can establish what is the correct disk access that needs to be performed, and it will issue an IO operation to retrieve this page. Once the page is brought into memory, the OS will determine a free frame where this page can be placed, and it can use the page frame number for that page to appropriately update the page table entry that corresponds to the virtual address of that page. At that point, control is pushed back into the process that caused this reference and the program counter will be restarted with the same instruction so that this reference will now be made again. Except at this point, the page table will find a valid entry with a reference to this particular physical location. Note that the original physical address of this page, will very likely be different from the physical address that was established after this demand paging process was over. If for whatever reason we require a page to be constantly present in memory or if we require it to maintain this same physical address during it's lifetime then we will have to pin the page and at that point we basically disable the swapping. This is for instance useful when the CPU is interacting with devices that support direct memory access (DMA).

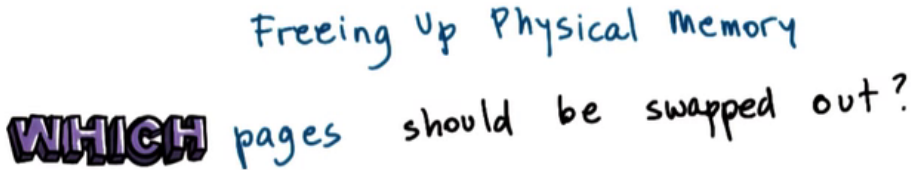
21 - Page Replacement



Moving pages between physical memory and secondary storage raises some obvious questions. When should pages be swapped out of physical memory and onto disk and also which particular pages should be swapped out?

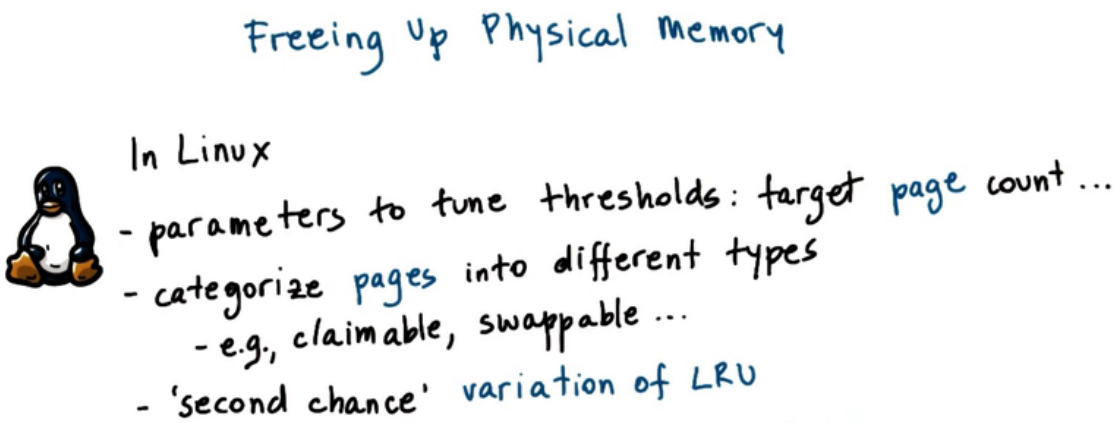


The first part is easier. Periodically, when the amount of occupied memory reaches a particular threshold, the OS will run some page (out) daemon that will look for pages that can be freed. So something that would make sense as an answer to this question would be that the pages should be swapped out when the memory usage in the system reaches some level, some high water mark, and that this paging out should be performed also when the cpu usage is below a certain threshold so as not to disrupt the execution of some applications too much.



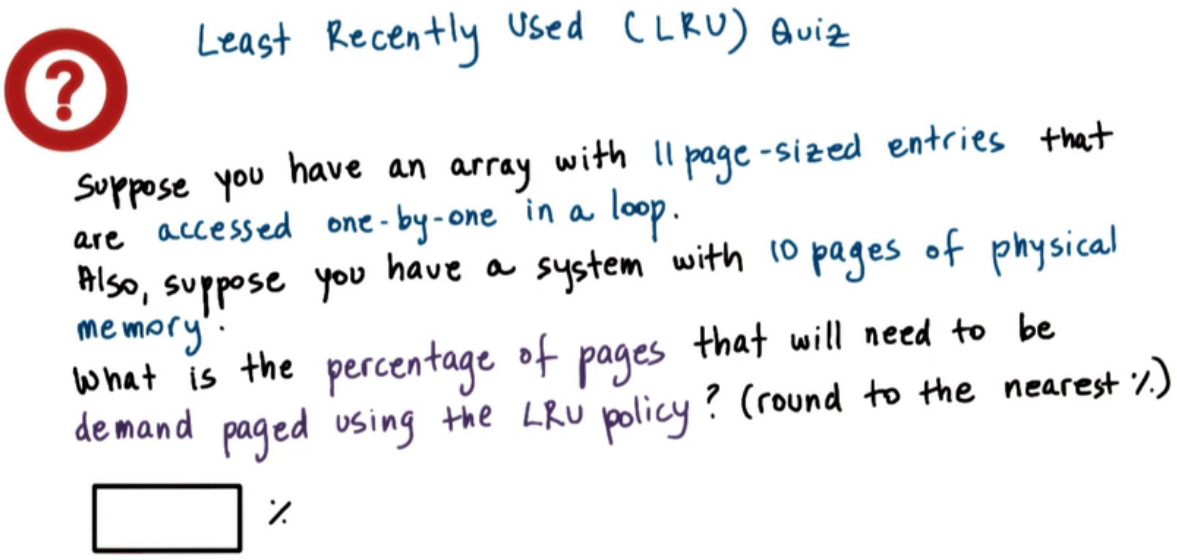
To answer the second question, one obvious answer would be that the pages that will not be used in the future are the ones that should be swapped out. The problem is how do we know which pages will vs won't be used in the future? To make some predictions regarding the page usage, OS use some historic information. For instance, once common set of algorithms is to look at how recently or how frequently has a page been used and use that to inform a prediction regarding the page's future use. The intuition is that a page that has been used most recently is more likely to be needed in the immediate future whereas a page that hasn't been accessed in a very long time is less likely to be needed. This policy is referred to as the LRU (least recently used) and it uses the access bit that's available on modern hardware to keep track of the information, whether or not the page is referenced or not. Other useful candidates for pages

that should be freed up from physical memory are the pages that don't need to be written out to disk to secondary storage and that is because the process of writing pages out to the secondary storage takes some time, consumes cycles, so we'd like to avoid the overhead of the memory management. To assist with making this decision, which pages don't need to be written out, the OS can rely on the dirty bit that's maintained by the MMU hardware that keeps track of which particular page has been modified, so not just accessed and referenced, however modified during a particular period of time. In addition, there may be certain pages, particularly certain pages containing important kernel state, or used for IO operations that should never be swapped out. Making sure that these pages are not considered by whatever replacement algorithms are executed in the OS is going to be important.



In Linux and most OS's a number of parameters are available to allow the system admin to configure the swapping nature of the system. This would include thresholds such as the ones we mentioned earlier that control when our pages swapped out, but also other parameters such as how many pages should be replaced during a period of time. Also linux categorizes the pages into different types and then that helps narrow down the decision process when it's trying to decide which pages should be replaced. Finally the default replacement algorithm in linux is a variation of the LRU process we described and it gives a second chance. It basically performs 2 scans of a set of pages before determining which ones are really the ones that should be swapped out and reclaimed. And similar types of decision can be made in other OS's as well.

22 - Least Recently Used LRU Quiz

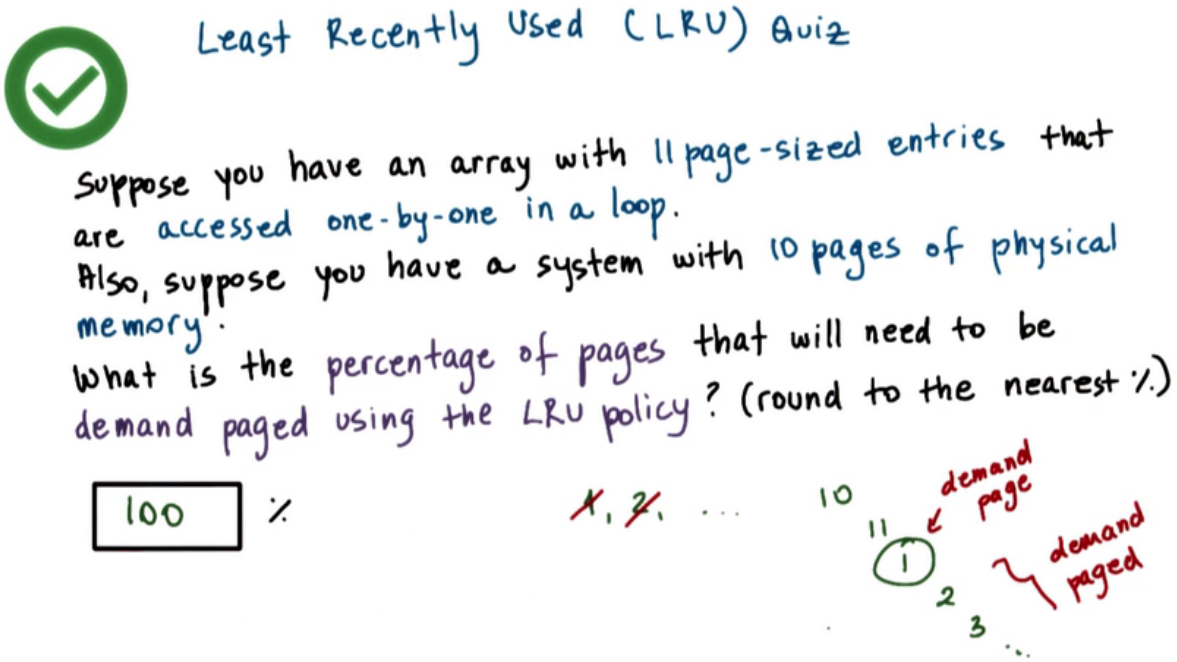


Errata

It should be further specified that **11 page-sized entries are accessed one-by-one and then manipulated one-by-one in a the loop**. Assume the following structure:

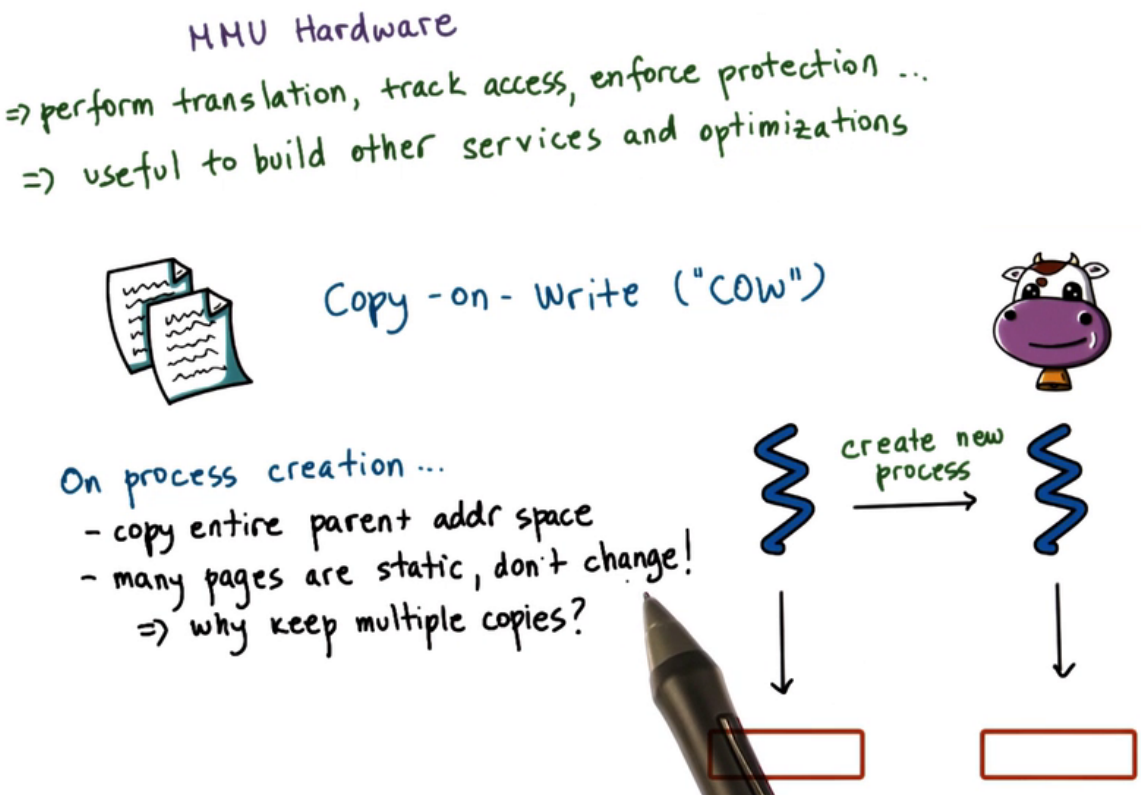
```
int i = 0;
int j = 0;

while(1) {
   for(i = 0; i < 11; ++i) {
      // access page[i]
   }

   for(j = 0; j < 11; ++j) {
      // manipulate page[i]
   }
   break;
}
```

23 - Least Recently Used LRU Quiz

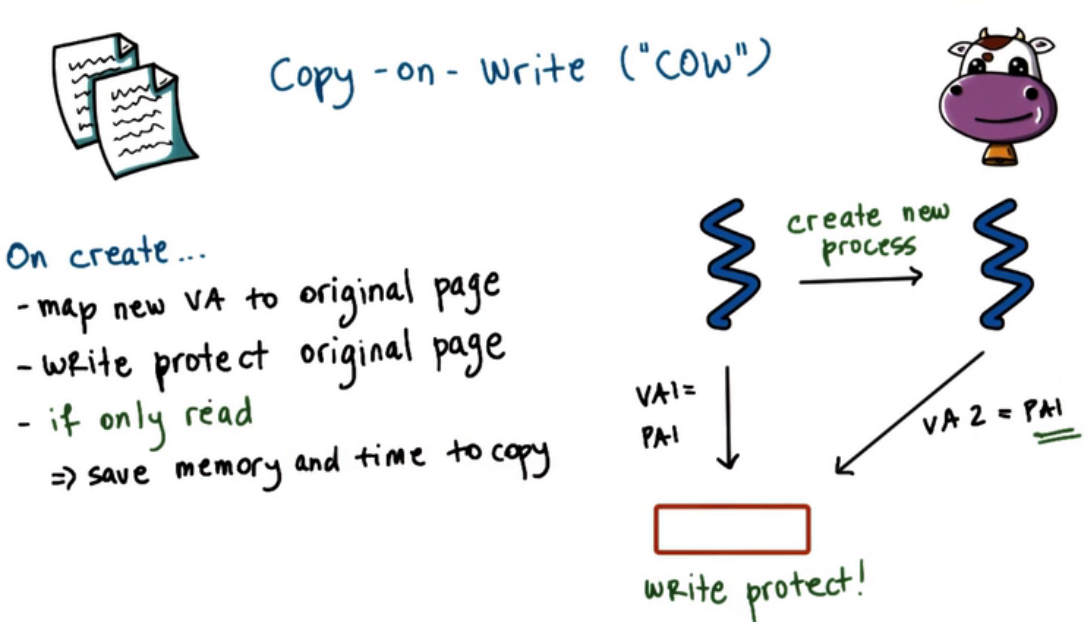


In this example, initially the first 10 pages will be loaded into memory one at a time as they're being accessed one by one. First page 1, then page 2 and then page 10 and then page 11 will need to get accessed. And that will mean that the first page, the one that's least recently used will be swapped out of memory, given that the physical memory only has 10 pages. Now the really unfortunate thing is that just as we swapped page 1 out of memory, given that the pages are accessed one by one in a loop, that exact same page is the very next page that's needed. We will have to demand page that in and given that our physical memory has 10 pages, we need to pick out another page to swap out, to replace and that's going to be page 2, given that that's the least recently used page right now. And guess what, the next page that will be needed will be exactly page 2 that we just swapped out. So the process will continue for all of the remaining pages during the execution of the program and therefore the nearest percentage of the number of pages that need to be demand paged using the LRU policy is 100. This is clearly a very pathological scenario, but what it's trying to demonstrate is that an intuitive policy such as LRU can result in really poor behavior under certain conditions. For that reason, OS can be configured to support different kinds of replacement policies that are used to manage their physical memory.

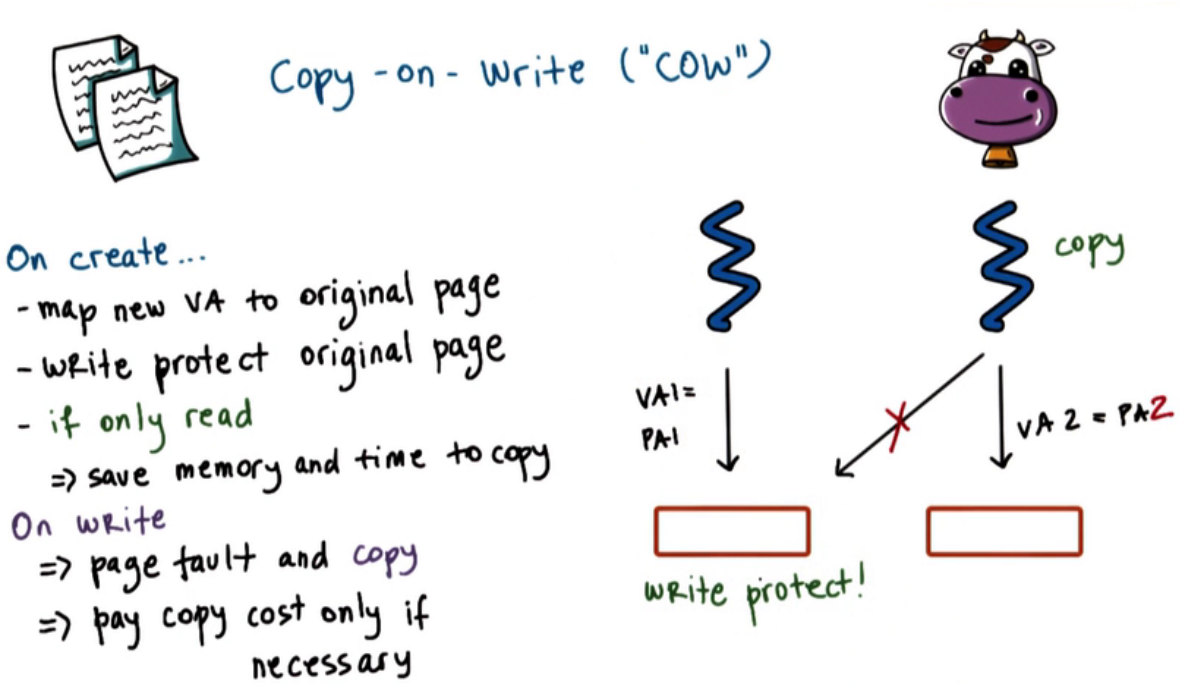
24 - Copy On Write



In our discussion about memory management so far, we saw that operating systems rely on the hardware, on the memory management unit hardware, to perform address translations and to also validate the accesses, to enforce protection and similar mechanisms. But the same hardware can also be used to build a number of other useful services and optimizations beyond just the address translation. One such mechanism is called copy-on-write or COW. Let's see what happens during process creation. When we need to create a new process, we need to recreate the entire parent process by copying its entire address space. However many of the pages are static, they don't change, so it's not clear why we should keep multiple copies.

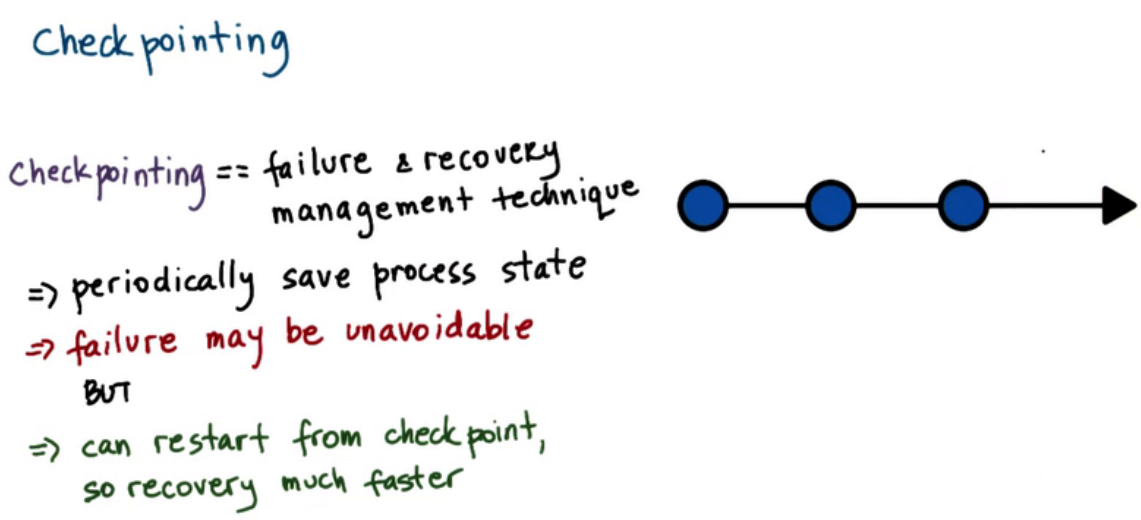


In order to avoid unnecessary copying, on creation the virtual address space of the new processes, or portions of it at least, will point, will be mapped to the original page that had the original address space content. The same physical address of the physical memory may be referred to by two completely different virtual addresses from the 2 processes. We also have to make sure to write protect the physical memory so that we can track concurrent accesses to it. If the contents of this page are indeed going to be read only, then we're going to save both on memory requirements as well as on the time that would have otherwise necessary to perform the copy. However if a write request is issued for this memory area via either one of these virtual addresses, then the MMU would detect that the page is write protected and will generate a page fault.

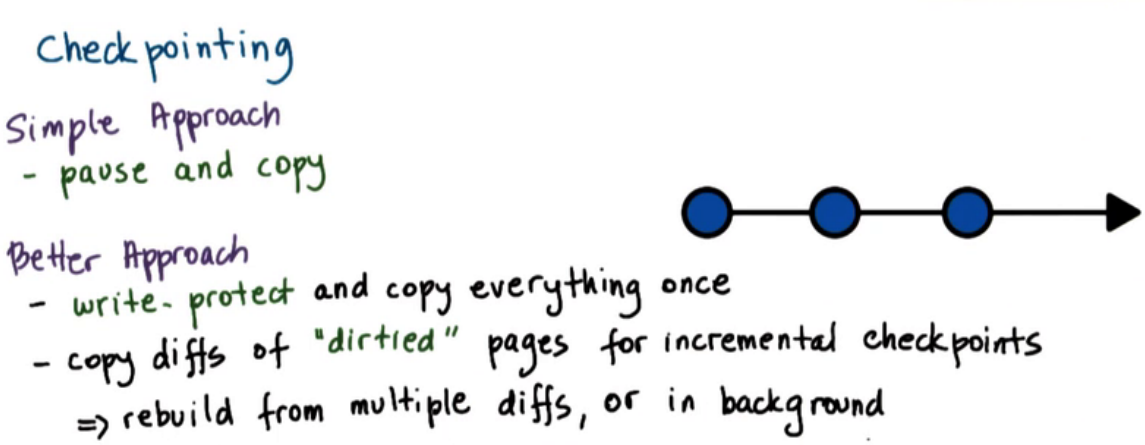


At that point, the OS will see what is the reason for this page fault, will create the actual copy, so the copy will only be performed then, will update the page tables of the two processes as necessary, so basically the page tables of the faulting process, and will in this manner, copy only those pages that need to be updated. Only those pages for which the copy cost is necessary. We call this mechanism copy on write because the copy cost will only be paid when we need to perform a write operation. There may be other references to this write protected region so whether or not the write protection will be removed once this one copy is performed will depend on who else is this page shared with.

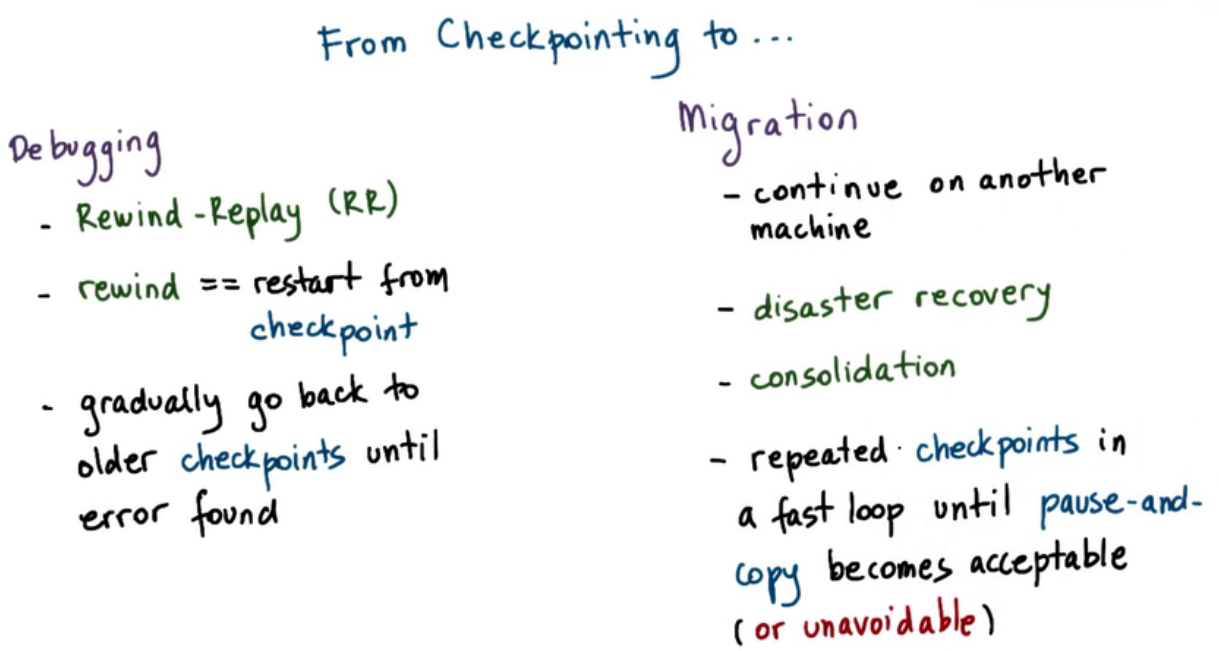
25 - Failure Management Checkpointing



Another useful operating system service that can benefit from the hardware support for memory management is checkpointing. Checkpointing is a technique that's used as part of the failure and recovery management that operating systems or system software in general supports. The idea behind checkpointing is to periodically save the entire process state. The failure may be unavoidable however with checkpointing the process doesn't have to be restarted from the beginning. It can be restarted from the nearest checkpoint point and so the recovery will be much faster.



A similar approach to checkpointing would be to pause the execution of the process and copy its entire state. A better approach will take advantage of the hardware support for memory management and will try to optimize the disruption that checkpointing will cause on the execution of the process. Using the hardware support, we can write protect the entire address space of the process and try to copy everything at once. However since the process will continue executing, we won't pause it. It will continue dirtying pages, so then we can track the dirtied pages again using the hardware mmu support, and we will copy only the diffs, only those pages that have been modified. That will allow us to provide for an incremental checkpoint. If we checkpoint using these partial diffs of just dirtied pages, we will somewhat make the recovery process more complex since we will have to rebuild the image of the process using multiple such diffs potentially or also in the background these diffs can be aggregated to produce more complete checkpoints of the process.



The basic mechanisms used in checkpointing can also be used in other services. For instance, debugging relies often in a technique called rewind replay. Here, rewind means that we will restart the execution of the same process from some earlier point. We will restart it from a checkpoint and then we will move forward and see whether we can establish what is the error, what is the bug in our program. We can gradually go back to older and older checkpoints until we find the error. Migration is another service that can benefit from similar kinds of memory management mechanisms that we described are useful for checkpointing. With migration, it's like we checkpoint the process to another machine and then we restart it on that other machine, it will continue its execution on the other location. This is useful in scenarios such as disaster recovery so as to continue the process on another machine that will not crash or in consolidation that is common in today's data centers when we try to migrate processes and migrate load onto as few machines as possible so that we can save on power and energy or utilize resources better. One way in which migration can be implemented is if we're performing repeated checkpoints in a fast loop until ultimately there is so few dirtied state from the process, something like the pause and copy approach because acceptable. Or maybe at that point, simply we really don't have another choice. The process keeps dirtying enough pages that we have to stop it in order to copy the remaining contents.

26 - Checkpointing Quiz

Checkpointing Quiz

Which one of these endings correctly completes the following statement?

"The more frequently you checkpoint ... "

- ○ the more state you will checkpoint
- ○ the higher the overheads of the checkpointing process
- ○ the faster you will be able to recover from a fault
- ○ all of the above

27 - Checkpointing Quiz

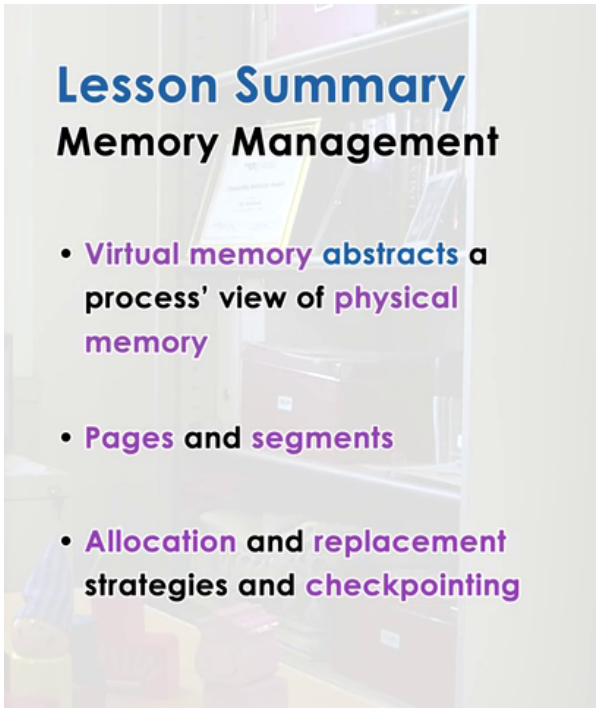
Checkpointing Quiz

Which one of these endings correctly completes the following statement?

"The more frequently you checkpoint ... "

- ○ the more state you will checkpoint
- ○ the higher the overheads of the checkpointing process
- ○ the faster you will be able to recover from a fault
- ✓ all of the above

28 - Lesson Summary



To summarize, in this lesson we looked at virtual and physical memory mechanisms in operating systems. You should now understand what are the data structures, the mechanisms, and the hardware level support that the operating system relies on when it tries to map the processes address space that uses virtual memory onto the underlying physical memory. We talked about pages and segmentation, address translation, page allocation, page replacement algorithms. We also looked at how these memory management mechanisms that are part of the operating system can used by some higher level services like checkpointing.