

Lesson Preview

Memory Management

- Physical and virtual memory management
- Review of memory management mechanisms
- Illustration of advanced services

1. 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 that I mentioned or some online content, in case you need to fill in any blanks

Visual Metaphor

"Operating systems and toy shops each have memory (part) management systems."

- Uses intelligently sized containers
 - memory pages or segments

- Not all memory is needed at once
 - tasks operate on subset of memory

- Optimized for performance
 - reduce time to access state in memory => better performance!

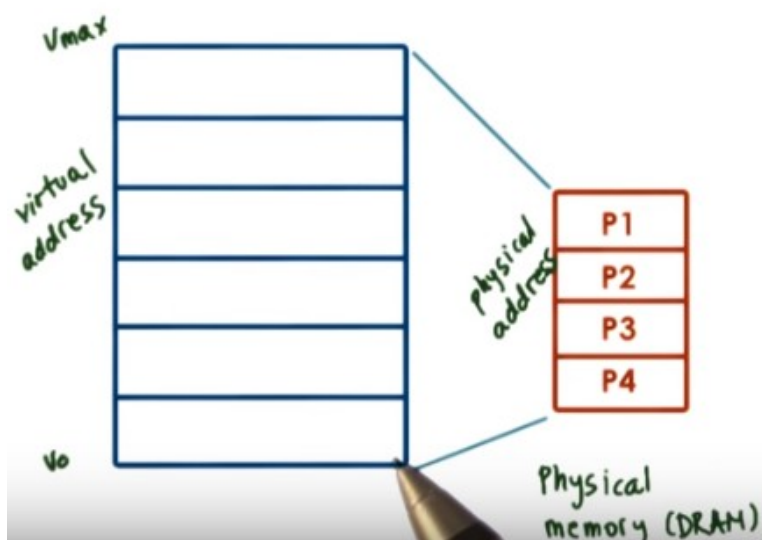
- Uses intelligently sized containers
 - crates of toy parts

- Not all parts are needed at once
 - toy orders completed in stages

- Optimized for performance
 - reduce wait time for parts
=> make more toys!

2. 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 sorts 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 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 that 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, memories typically manage the 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 with the analogy of the toy orders and the 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 task can be brought in and 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 or segments is optimized so as to reduced the time that's require to access that state. The end result again is improve 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, or TLBs. They rely on caches and also in software algorithms such as for page replacement or for memory allocation.

Memory management goals



Virtual vs. Physical
memory

Allocate

- allocation, replacement ...

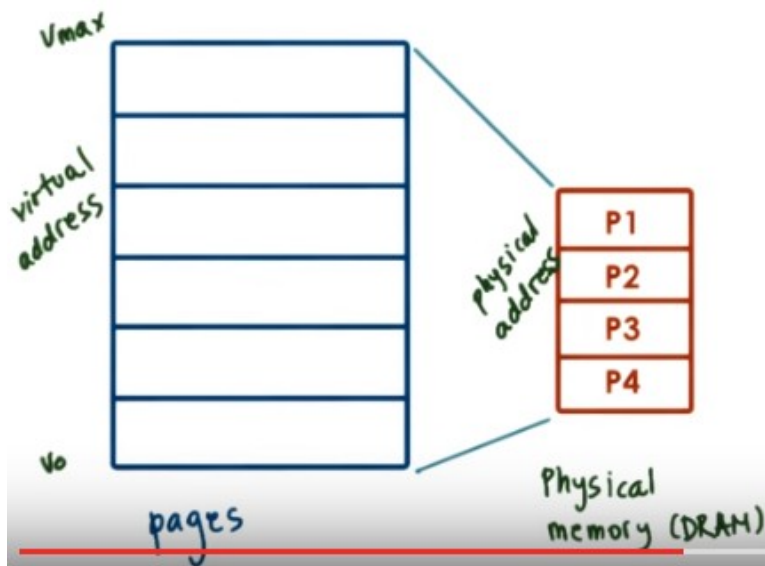
Arbitrate

- address translation and
validation



3. In the introductory lecture on 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 operating system is to manage the physical resources. In this case, the physical memory (DRAM), on behalf of one or more executing processes (我：操作系統就像機場, process 就像航空公司). In order not to post any limits on the size and the layout of an address space based on the amount of physical memory or how it shared with other processes, we said that we will decouple the notion of Physical memory from the Virtual memory that 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 V_0 to V_{max} 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 operating system must then be able to allocate Physical memory and also arbitrate how it's being accessed. Allocation requires that the operating system incorporates certain mechanisms 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 is the Physical memory 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 Physical memory. They may be stored on some secondary storage like on disk. So, the operating system 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 operating system 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 that is, indeed, a legal access. For this current operating systems rely on a combination of hardware support as well as smartly designed data structures that are used in the process of address translation and validation.

Memory management goals



Page-based
memory management

Allocate \Rightarrow pages \rightarrow page frames

Arbitrate \Rightarrow page tables

Segment-based
memory management

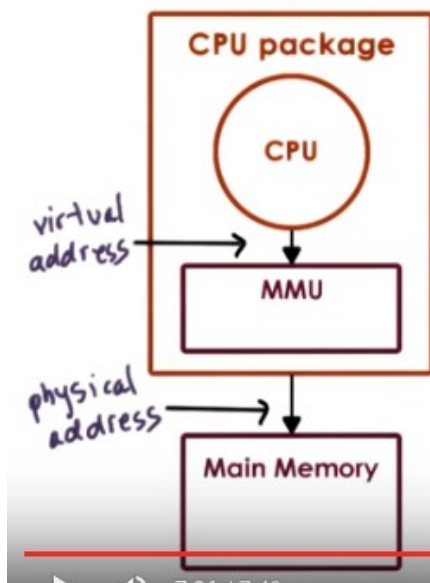
Allocate segments

Arbitrate segment registers



The figure here illustrates that the virtual address space is subdivided into fixed sized segments that are called pages. The physical memory, not to scale these two, is divided into page frames of the same size. In terms of allocation then the role of the operating system is to map pages from the Virtual memory into page frames of the Physical memory (故 allocation 的本質就是 map). In this type of Page-based memory management system the arbitration of the access is done via page tables. Paging is not the only way to decouple the Virtual and the Physical memories. Another approach is segmentations, 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 swapped 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 operating systems and we'll primarily focus our discussion on Page-based memory management. That will mention segments a little bit more later in this lesson again.

Hardware Support



MMU

- translate virtual to physical addresses
- reports faults: illegal access, permission not present in mm

Registers

- pointers to page table
- base and limit size, number of segments...

Cache - Translation Lookaside Buffer

- valid VA-PA translations: TLB

Translation

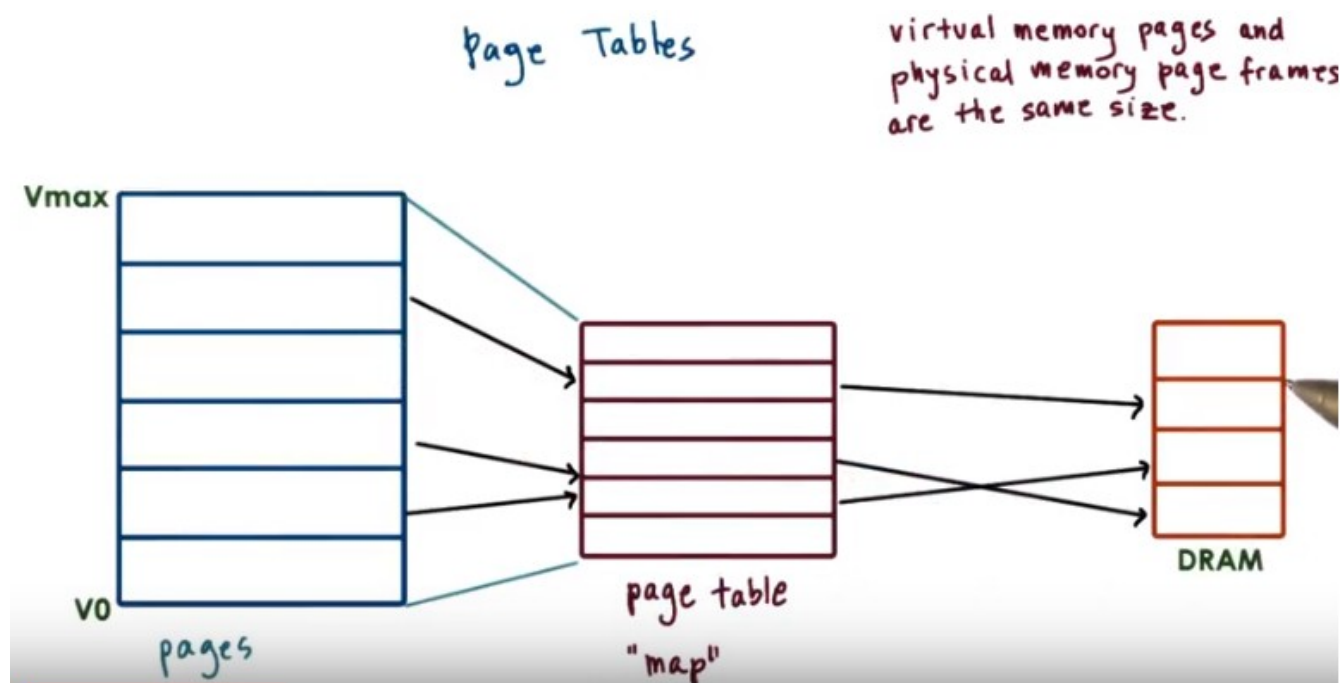
- actual PA generation done in hardware



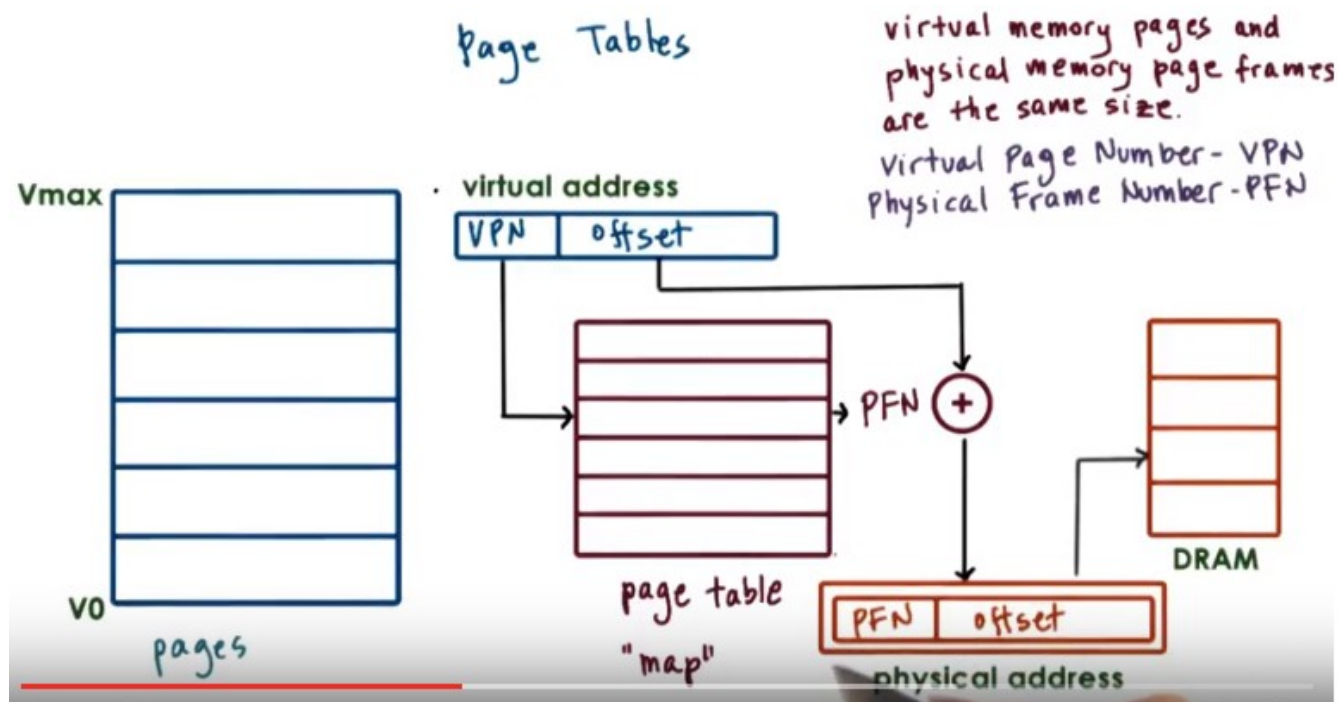
MMU: memory management unit (後面的視頻也會出現多次). MMU 就是 translator, MMU 是 hardware (由第 11 段知). TLB 就是個暫時存放 virtual address to physical address mapping 的 cache.

4. As I already hinted memory management isn't purely done by the operating system alone. In order to make these tasks efficient over the last decades the hardware has evolved to integrate a number of mechanism that 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 signal that's 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 inadequate permissions to perform a particular access. For instance, 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 right permissions for that particular access. That page is what we call right protected. Or another type of fault may be an indication that the particular. Page that's being referenced 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 its limit, so 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 address translations. This is called the translation lookaside buffer 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's no need to perform any additional operations to access the page table or the segment and to interpret. the validity of the axis. And finally, the actual generation of the physical address from the virtual

address, so the translation process, that's done by the hardware. The operating system will maintain certain data structures such as the page table, 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 kinds 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 format since the hardware needs to understand both of these? There are other aspects of memory management that are more flexible in terms of their design since they are 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 versus on disk. So we will focus our discussion on those software aspects of memory management, since that's more relevant from an operating systems course perspective.

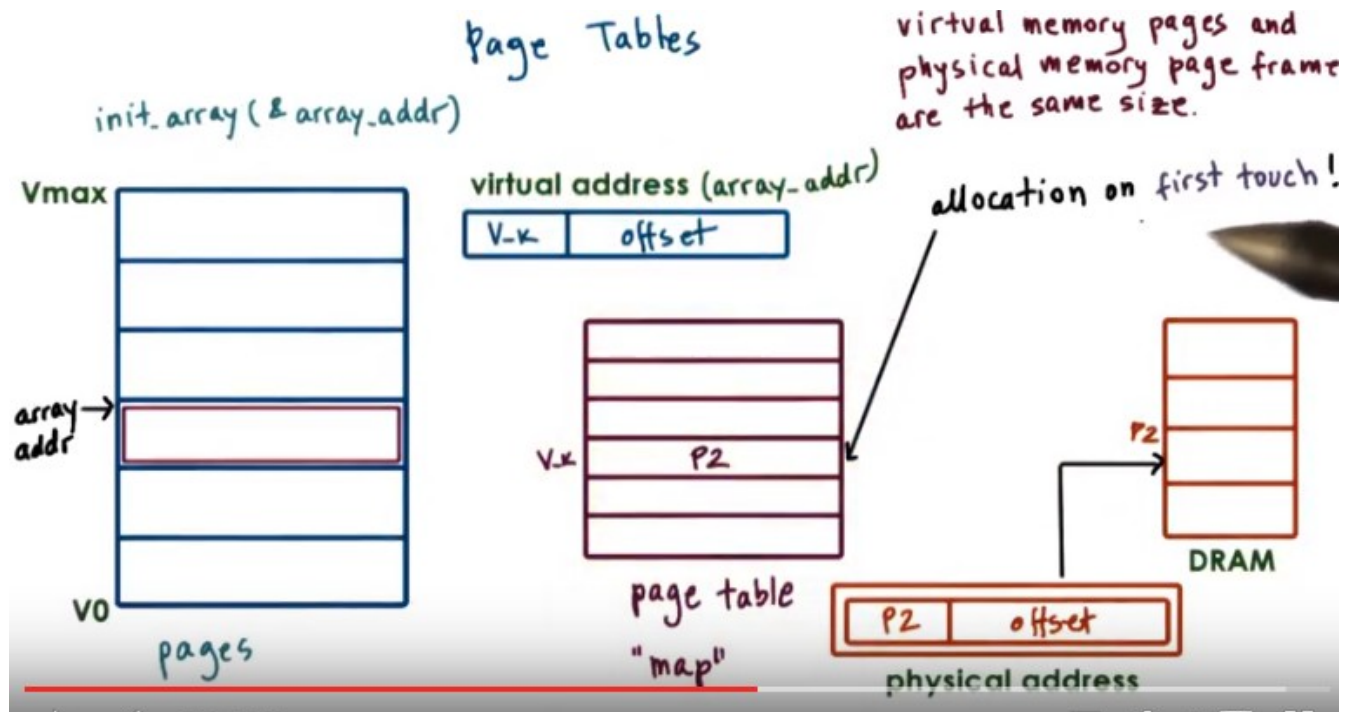


5. As we said, pages are the more popular method for memory management. Now, let's take a look at one of the major components that enables page based memory management, and that's page tables, as the component that's used to translate the virtual memory addresses into physical memory addresses. So, here's a page table. For each virtual address, and 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 operating system and the hardware itself where to find specific virtual memory references. All of 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 (即下段的 VPN) to the first virtual address in a page frame in physical memory (即下段的 PFN). And then the remaining addresses in the virtual memory page will map to the corresponding offsets (即下圖中 virtual address 和 physical address 中那個 offset) in the physical memory page frame. As a result, we can reduce the number of entries we have to maintain in the page table.



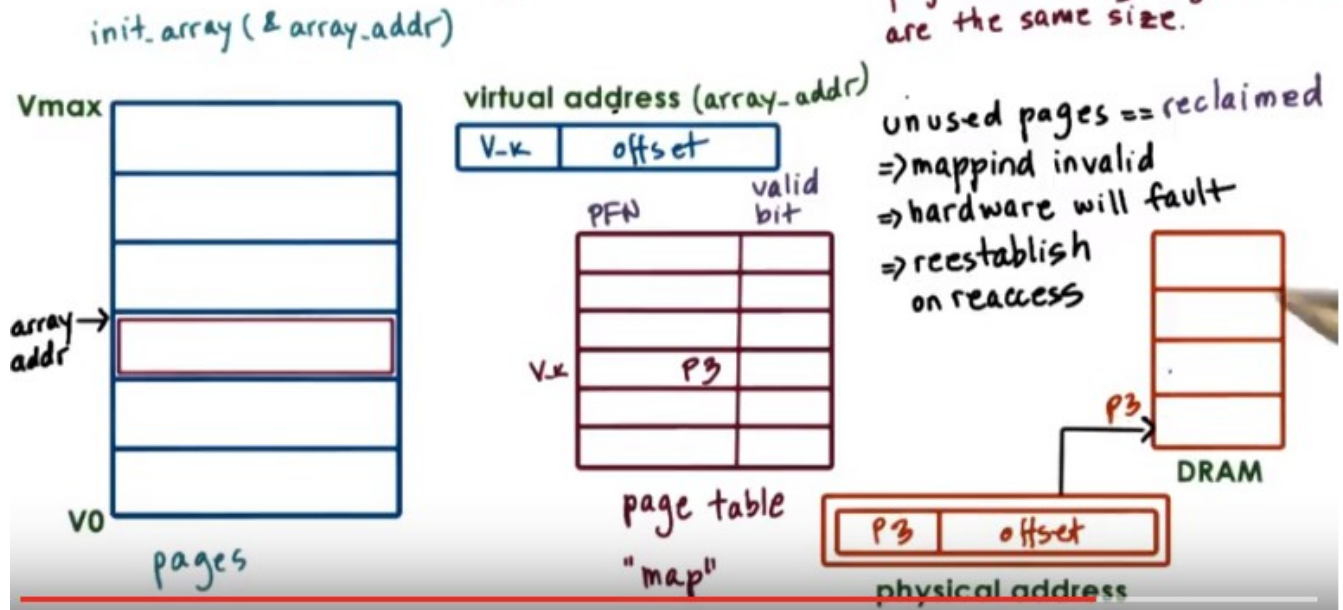
對上圖的理解(此理解對第 7 段初的計算有幫助, 後面多次驗證此理解是正確的): 上圖左邊 $V_0 \sim V_{max}$ 中的每一格即一個 page, 一個 page 對應多個存儲單元, 上圖中間的 VPN (virtual page number) 即 page 的號碼, 而 offset 即 要存的變量的 存儲單元 在這個 page 中的 offset. 在 page table 中, VPN 被轉化為 PFN (physical frame number), 而 offset 值不變地傳到 physical address 中. 最後在 DRAM 中, 根據 PFN 找到正確的 page frame, 根據 offset 找到該變量 對應的存儲單元 在這個 page frame 中的 offset.

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 (VPN), 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 (即 VPN) will produce the physical frame number (PFN). 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 sent with the offset that's specified in the later part of the physical address to produce the actual virtual address. That resulting physical address can ultimately be used to reference the appropriate location in physical memory.



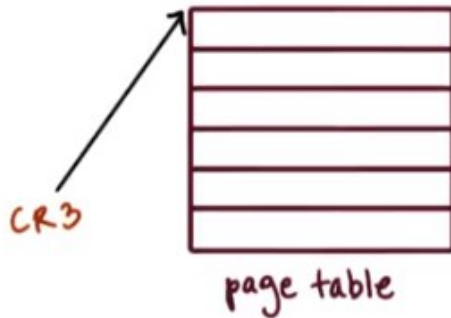
Let's look at an example now. Let's say we want to access some data structure, some array, for instance, to initialize it for the very first time. However, we have already allocated the memory for that array into the virtual address space of the process, we've just never accessed it before. So since this portion of the address base 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 operating system 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 is free, obviously. And it will establish a mapping between this virtual address, so this is the V-K and the offset address, where the array is 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 is really needed, because sometimes programmer may create data structures that they don't really use.

Page Tables



If a process hasn't used some of its memory pages for a long time, and it's likely that those pages will be reclaimed(收回, 開墾). So the contents will no longer be present in physical memory. They will be reclaimed, they(指該 memory page 中的內容) will be pushed on disks and probably some other content will find its 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 one. If the page is not in memory then this bit is zero, and if the hardware MMU see's that this is a bit zero in the page table entry it will raise a fault. It will trap to the operating system. If the hardware determines that the mapping is invalid and false, then control gets passed to the operating system. The OS at that point gets to decide a number of questions. Should the access be permitted? Where exactly is the page located? Where should it be brought into DRAM? So long as a valid address is being accessed. Ultimately in fault, there will be a mapping that will be re-established between a valid virtual address and the valid location in physical memory. It is likely however, if the page was pushed in disk and now it's 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 use to be in P2, as a result clearly the entry in the page table needs to be correctly updated.

Page Tables



- per process

- on context switch,
switch to valid page table

- update register
e.g., CR3 on x86

So as a final note to summarize, the operating system creates a page table for every process that it runs. As a summary, the operating system will maintain a page table on every single process that access. That means that whenever a context switch is performed, the operating system has to make sure that it switches to the page table of the newly context switch process. We said that hardware assist with page table accesses by maintaining a register that points to the active page table. On X86 Platforms there's a register CR3. And so basically, on a context which we will have to change the contents of the CR3 register with the address of the new page table. CR2 見第6段末.

Page Table Entry

Page Frame Number (PFN)	...	X	W	R	A	D	P
----------------------------	-----	---	---	---	---	---	---

Flags

- Present (valid/invalid)
- Dirty (written to)
- Accessed (for read or write)
- protection bits \Rightarrow RWX

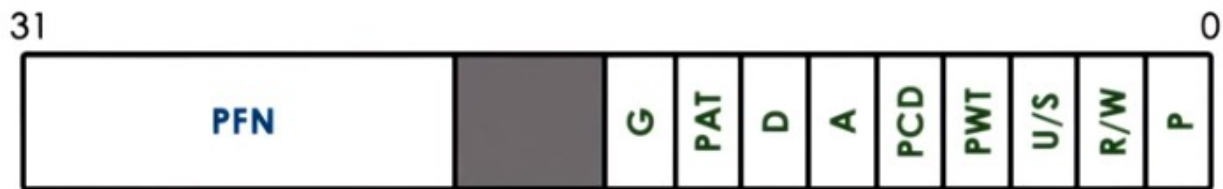
Page table 中之所以沒有 VPN, 應該是因為前面說了: The virtual page number is used as an offset into the page table. 即 page table 中每行的 index 即 VPN.

由後面知, 若一個內容不在 memory 中, 但在 disk 中(比如此內容佔內容了太久 memoy, 別的內容將它 overwrite 了並將它移到 disk 中), Present 這個 flag 也為 invalid. MMU 也會生成 page fault, 然後 OS 從

disk 中去取此內容. 第 20 段會詳細講此過程.

6. We see that every page table entry will 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 contents 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 operating system 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 track of an **accessed bit**. This can keep track of in general whether the page has been accessed, period, for read or for write. Other useful information that can be maintained as part of the page table entry also would include certain protection bits. Whether a page can be only read or also written to, or maybe some other operation is permissible.

Page Table Entry on x86

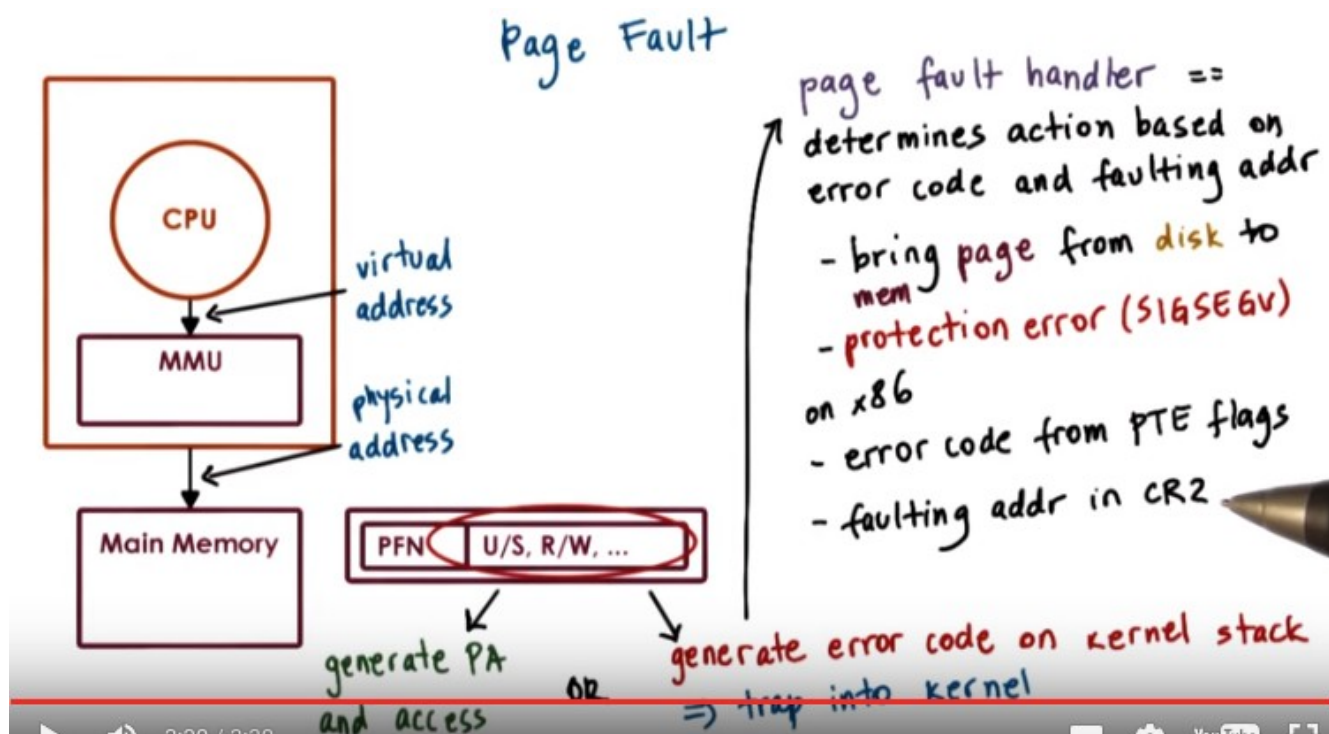


Flags:

- Present
- Dirty
- Accessed
- R/W \Rightarrow permission bit: 0 \rightarrow R only; 1 \rightarrow R/W
- U/S \Rightarrow permission bit: 0 \rightarrow usermode; 1 \rightarrow supervisor mode only
- Others: caching related info (write through, caching disabled...)
- Unused: for future use

0 \rightarrow R only 中是數字 0, 不是字母 O

So 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 its value's 0, that means that, that particular page can be accessed for read only, whereas if it's 1, that 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 dictate some things regarding the behavior of the caching subsystem that the hardware has. So, for instance, you can specify things like whether or not caching is disabled. That's an operation 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'll have good uses for these bits as well.



PTE: page table entry

The MMU uses the page table entries not 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. 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, it 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, it needs to be brought in from disk. Or because there is some sort of permission protection that was violated, and that's why the page access is forbidden. On x86 platforms, the error code information is generated from some of the flags in the page table entry. And the faulting address that's also needed

during the page fault handler, that one is stored in a register, CR2. (CR3 見第 5 段末)

Page Table Size

↑
Number of
Virtual
Page
Numbers
↓

- 32-bit architecture
- Page Table Entry (PTE)
⇒ 4 bytes, including PFN + flags
- Virtual Page Number (VPN)
⇒ $2^{32} / \text{Page Size}$
- Page Size
⇒ 4kB (... 8kB, 2MB, 4MB, 1GB)

$$= (2^{32} / 2^{12}) * 4B = 4MB$$

⇒ per process

由網上可知, 1kB 或 1MB 中的 B 表示的 byte, 而不是 bit. 32-bit architecture 意思是: 整個 $V_0 \sim V_{\max}$ 有 2^{32} 個地址, 原因: 一個 32 位的二進製數總共能表示 2^{32} 個數, 即整個 $V_0 \sim V_{\max}$ 有 2^{32} 個地址. “ $2^{32} / \text{page size}$ ”及它上一行的意思是 virtual page number 的數量等於 $2^{32} / \text{page size}$ 個, 其中 page size 為一個 page 含有的地址數(一個地址對應一個存儲單元, think of 我之前寫的我的理解) (更正: 實際上 page size 是一個 page 所佔空間大小). “ $2^{32} / 2^{12}$ ”即為 $2^{32} / 4k$, 因為 $4k = 4 * 1024 = 4 * 2^{10} = 2^{12}$. 注意 page size=4kB, 我猜測一個地址是佔 1B(即一 byte)的(後面第 7 段末的計算表明 virtual address space(即 $V_0 \sim V_{\max}$)為 2^{32} B, 第 8 段的計算也驗證了), 故 page size=4kB 意思是一個 page 中有 4k(即 2^{12})個地址. 故“ $2^{32} / 2^{12}$ ”即為“ $2^{32} / \text{page size}$ ”, 這就是前面我說到的 virtual page number 的數量, 注意它也等於 page table 中的行數, 即 page table entry (PTE)的個數, 而圖中說一個 page table entry 為 4 bytes, 所以整個 page table size 為 $(2^{32} / 2^{12}) * 4B = 4MB$.

7. To calculate 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 exist 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 is something that would make sense on a 32-bit architecture. So, each of the page table entries is 4 bytes and that includes the page frame number as well as the flags. The total number of page table entries, that will depend on the total number of VPNs. And how many VPNs we can have, that's going to depend on the size of the addresses, of the virtual addresses, and of the page size itself. So let's say in this example, we have a 32-bit, both physical memory as well as 32-bit virtual addresses. So that will be 2 to the 32nd 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 4 kilobyte page size for this example. In that case, if you do the math, you will see that the page table will be 4 megabytes and it will be 4 megabytes for every single process. With many active processes in an operating system today, this can get to be quite large.

Page Table Size



64-bit architecture

- Page Table Entry (PTE)
 \Rightarrow 8 bytes, including PFN + flags
- Virtual Page Number (VPN)
 $\Rightarrow 2^{64} / \text{Page Size}$

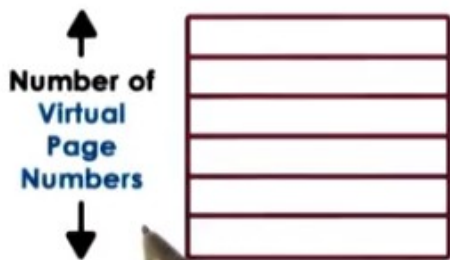
- Page Size
 \Rightarrow 4kB (... 8kB, 2MB, 4MB, 1GB)

$$= (2^{64} / 2^{12}) * 8B = 32PB!!!$$

\Rightarrow per process

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 also the same 4 kilobyte page size, we come up with a really scary number of 3 petabytes per process.

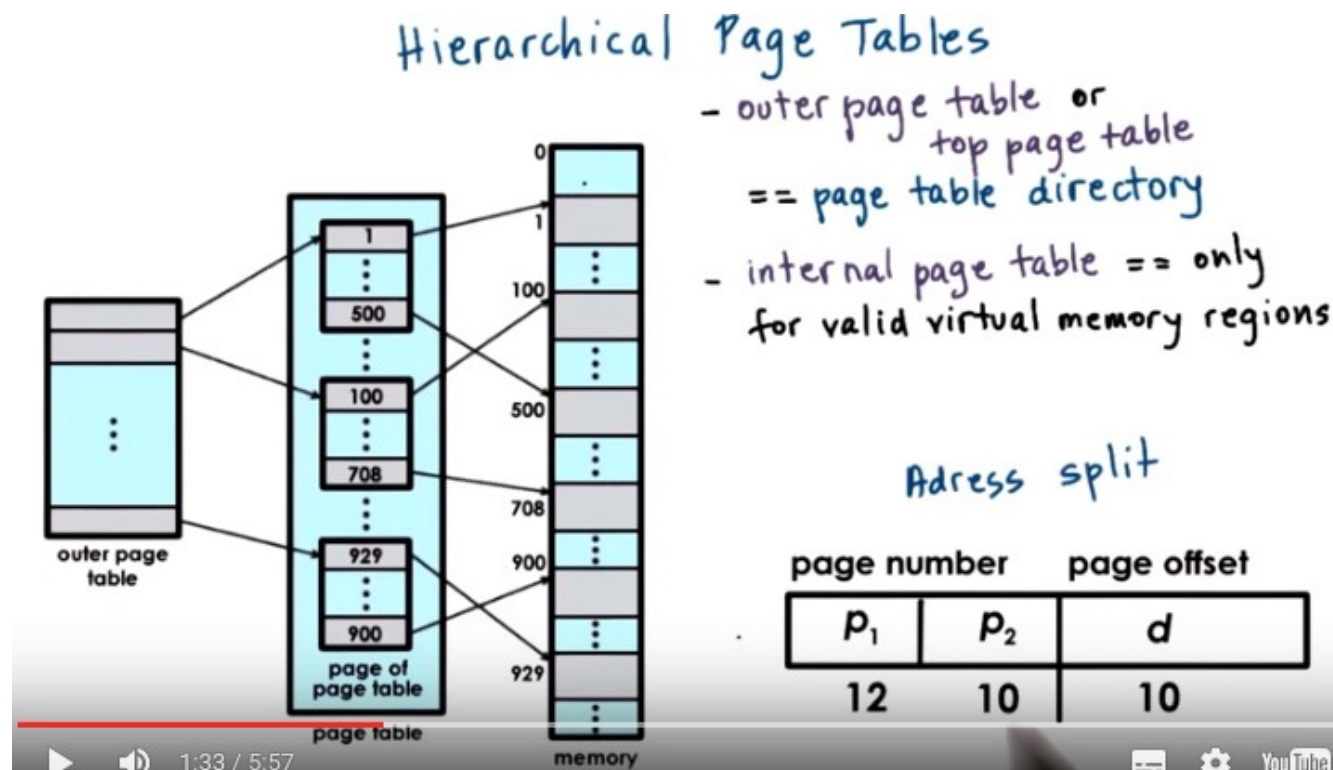
Page Table Size



- process doesn't use entire address space
- even on 32-bit arch will not always use all of 4GB
- BUT page table assumes an entry per VPN, regardless of whether corresponding virtual memory is needed or not.

1 GB = 1024 MB = 2^{10} MB = 2^{20} kB = 2^{30} B, 故 4 GB = 2^{32} B

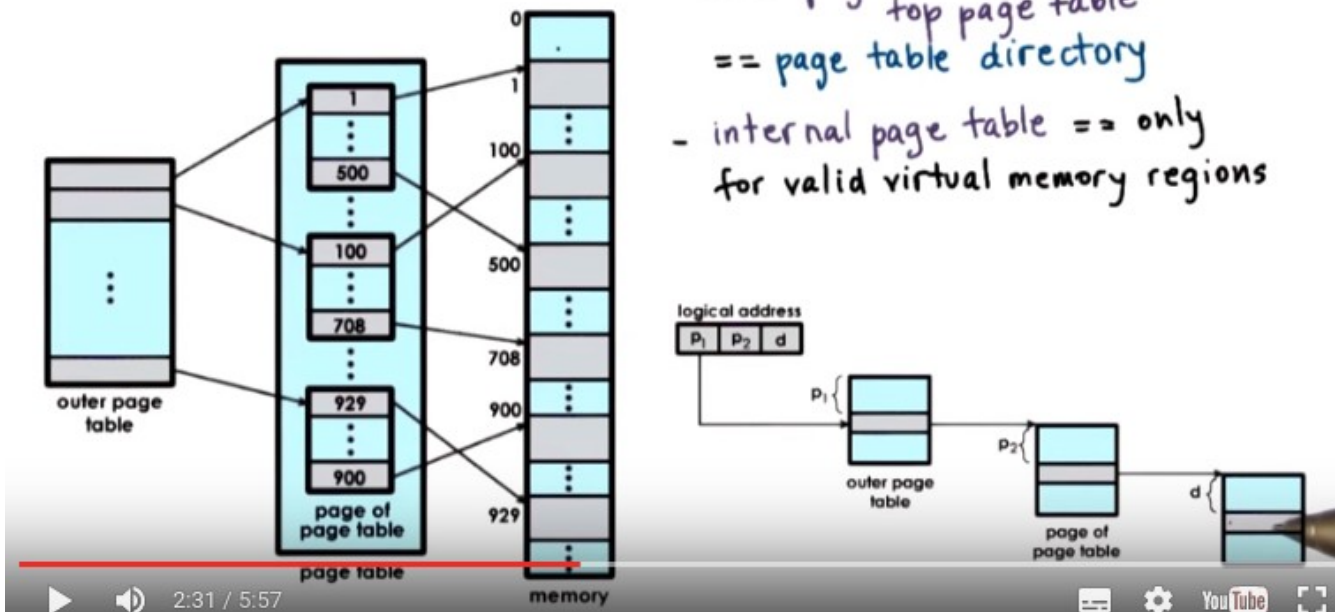
So where does one store all of this? Before we answer that question, it's important to know that a process likely will not use all of the theoretical available virtual memory. Even on 32-bit architecture, it's not all of the 4 gigabytes of virtual address space is used by every single type of process. 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 of how to represent a page table.



8. The answer to our storage issues 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 multi-level structure. This figure here shows a two level page table. The outer level here is referred to as a page table directory. Its 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 its components that actually point to page tables (應該是 pages, 因為視頻中是指到 memory 那一列的, 後面的計算也驗證了這點). Their entries have the page frame number and all the protection bits (protection bits 見第 6 段初) 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 internal page tables allocated for them. If a process requests via malloc additional virtual memory to be allocated to it (由此可知, malloc 分配的是 virtual memory), 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.

Hierarchical Page Tables

- outer page table or top page table == page table directory
- internal page table == only for valid virtual memory regions



上圖右下角的 page of page table 在左邊那三層中的中間那層。一個 p_2 所指的是 '左邊那三層中的中間那層' 中的一個小框(如 1...500 那個小框)。

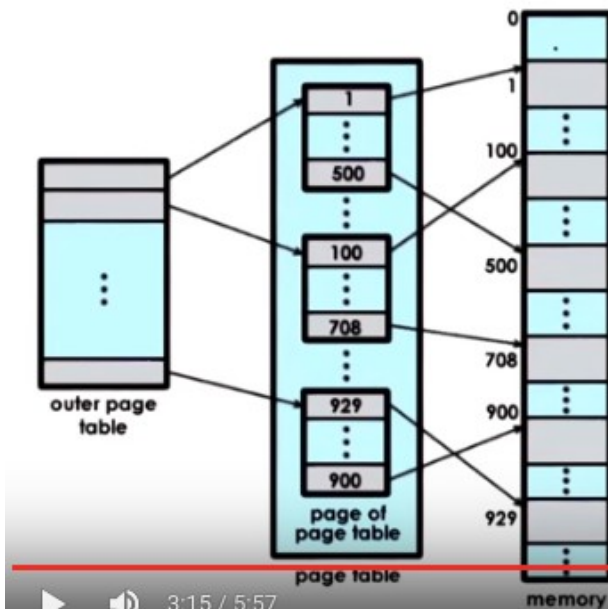
Using this address format, this is what we need to perform to determine the correct physical address. First the last portion of the address (即 d) is still the offset so that's going to be used to compute the offset within the actual page, within the actual physical page. The first two components (即 P_1 和 P_2) of the address are used as indices into the page tables, into the different levels of the page table hierarchy. And they (P_1 和 P_2) 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.

Hierarchical Page Tables

inner table addresses

$$\Rightarrow 2^{10} * \text{page size} = 2^{10} * 2^{10} = 1\text{MB}$$

\Rightarrow don't need an inner table for each 1MB virtual mm gap



page number		page offset
P_1	P_2	d
12	10	10

上圖的意思，簡單地講，就是把很多個 page table 一起放在了 '左邊那三層中的中間那層' (所以其中每個 page table 都叫 internal page table. 後面的第 10 段的 quiz 的 "[注 2]" 也證實了: 在 multi-level 情況下, 是有多個 inner page tables 的.)。注意一個 internal page table 中的每個 page 都對應一些 memory 中的地址, 這些地址在 page 中的位置由 offset 定出. 而指向這些 internal page table 的指針們, 都被放在了 '那三層中的左邊那層'.

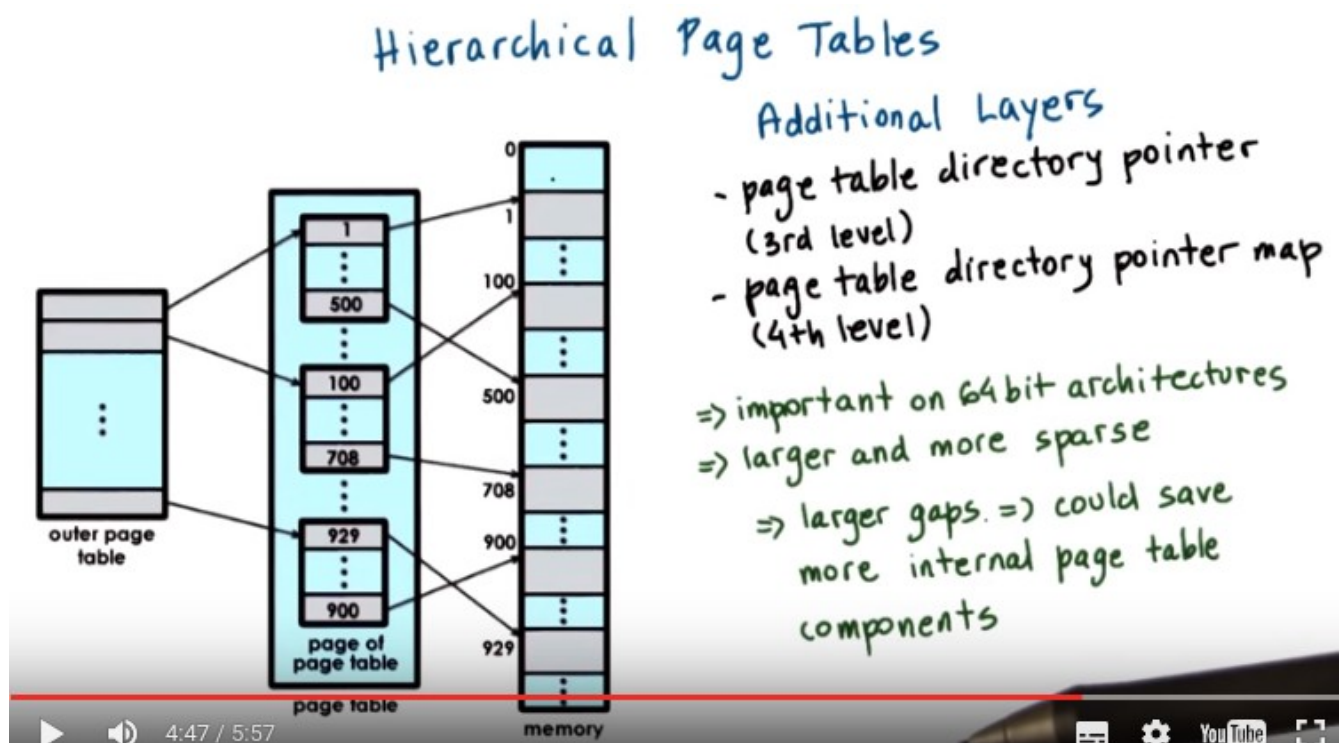
'那三層中的中間那層' 中某個 internal page table (不是 page) 在 memory 中所對應的空間沒使用時 (即 gap), 我們就不在 中間那層 分配這個 internal page table 的空間. 怎麼知道哪個 internal page table 不分配? 答: 我們有所有指向 internal page table 的指針, 它們就放在左邊那層, 通過這些指針來操作所有 internal page table. 這就是為何要 multi level.

第三遍看時, 覺得以下的紅字說得不對. 對於 outer page table, 輸入的是 VPN, 輸出的是 p_1 . p_1 指向的應該是 '左邊那三層中的中間那層' 中的一個小框 (如 1...500 那個小框, 一個小框即一個 inner page table). p_2 指向的應該是 '左邊那三層中的中間那層' 中的一個小框 (如 1...500 那個小框) 中的一個數字 (如 278), 這個數字代表一個 page, 用 offset (即 d) 可以定出具體是 page 中的哪一個存儲單元.

注意一個 p_2 指向的是 '左邊那三層中的中間那層' 中的一個小框 (如 1...500 那個小框, 這樣一個小框就是一個 page, 一個 page 中含一些地址, 這些地址的位置是由 offset (即 d) 定的). 所以右下角的表中, p_2 對應的 10 意思是 p_2 有 2^{10} 這麼多個取值的可能, 例如 p_2 可以指向 $[1, 2, 3, \dots, 2^{10}]$ 這些數中的一個. 注意 p_2 還可以指向 $[2^{10} + 1, 2^{10} + 2, \dots, 98795773868763]$ 這些數中的一個, 但只有 $[1, 2, 3, \dots, 2^{10}]$ 這片空間被分配了, $[2^{10} + 1, 2^{10} + 2, \dots, 98795773868763]$ 這片空間沒被分配, 所以暫時還指不了它們. 所以 p_2 可指向 $[1, 2, 3, \dots, 2^{10}]$ 這些數中的一個, 而這些數中的每一個都代表 memory 中的一個 page. 然後每個 page 中有 2^{10} 個 (即 d 的取值個數) offset, 此即 page size. 由第 6 段末的計算可知兩點: page size 為一個 page 含有的地址數 (一個地址對應一個存儲單元), 且一個地址是佔 1B (即一 byte). 故 page size 為 2^{10} . 故 inner table 共可表示 $2^{10} * 2^{10}$ 個地址, 這些地址佔的大小為 $2^{10} * 2^{10} \text{ B} = 1024 * 1024 \text{ B} = 1024 * 1\text{kB} = 1\text{MB}$.

下面文字(及上圖)中說了, 這 1MB 是 virtual memory 佔的(為可不是 physical memory? 因為沒分配), 不是 '左邊那三層中的中間那層' 佔的. 注意[1, 2, 3, ... 2^{10}]這片內存是為 p2 分配的. 而每片[1, 2, 3, ... 2^{10}]這樣的東西, 又被一個 p1 所指. 所有的 p1 即組成了'左邊那三層中的左邊那層'.

In this page the address format is such that it uses 10 bits for the internal page table (即左邊那三層中的中間那層) offset (即共有 2^{10} 個 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 (即一個 page 有 2^{10} 個地址, 這就是 page size). Therefore, if we do the math, we see that every single internal page table ([1, 2, 3, ... 2^{10}]這片 p2 空間(其中每一數代表一個 page)就組成一個 internal page table) can address two to the tenth, the number of entries, times the page size, that's another two to the tenth, so 1MB of memory. What that means is that whenever there is a gap in the virtual memory (不是左邊那三層中的中間那層) that's 1MB size. We don't need to allocate that internal page table (即不用分配[1, 2, 3, ... 2^{10}]這片內存), 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.

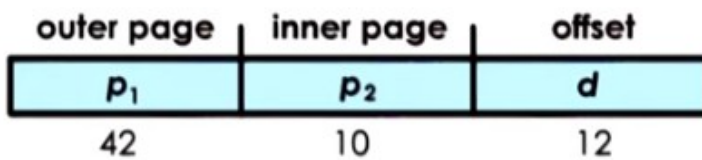


The scheme can be further extended to use additional layers using the same principle. For instance, we can add another, third level that can consist of pointers to page table directories. Adding yet another fourth level to this, which consists of a map of pointers to page table directories. This technique is particularly important on 64 bit architectures. There, not only that the page table requirements are larger, it's also the fact, is that the virtual address spaces 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 that gap. In fact, with a four level addressing, we may end up saving

entire page table directories as a result of certain gaps in the virtual address space.

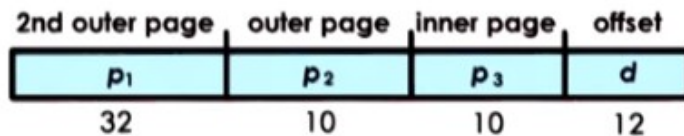
Hierarchical Page Tables

Multi-level PT Tradeoffs



smaller internal page
tables/directories
granularity of coverage

=> potential reduced page table size



more memory accesses
required for translation

=> increased translation latency

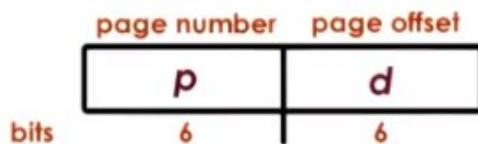
Let's look at a quick example. These two figures show how a 64-bit virtual address can be interpreted to determine which indices are used into the different levels of the page table hierarchy. The top figure has two page table layers, whereas the bottom one has three page table layers. In both of these 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.

9. Let's check your understanding of how multi level page tables work using a simple quiz. A process which uses 12bit addresses, has an address base for only the first two kilobytes, and the last one kilobyte are allocated and used. How many total page table entries are there in a single level page table that uses the first address format? As a second question, how many entries are needed for the inner page tables of the 2-level page table that uses the second address format? Write your answers in the boxes here.

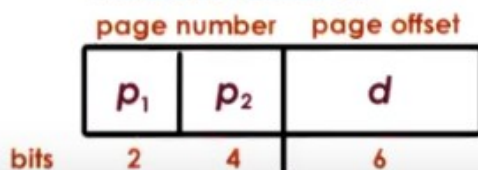


Multi-Level Page Table Quiz

Address Format 1



Address Format 2



A process with 12bit addresses has an address space where only the first 2kB and the last 1kB are allocated and used.

How many total entries are there in a single-level page table that uses the first address format?

64

How many entries are needed in the inner page tables of the 2-level page table when the second format is used?

48

前面第 6 段末的計算中說了: “32-bit architecture 意思是: 整個 $V_0 \sim V_{\max}$ 有 2^{32} 個地址”(不是本題). 故本題中的 12-bit address 意思是整個 $V_0 \sim V_{\max}$ 有 $2^{12} = 2^2 * 2^{10} = 4 * 2^{10} = 4 * 1024 = 4k$ 個地址. 而題目中說了, 這 4k 個地址中, 只有前面 2k 和後面 1k 的地址能用得上, 做實際上總共只需要 3k 的地址, 剩下那 1k 是 gap.

Address Format 1 和 Address Format 2 中, page offset 都是 6, 即一個 page 可表示 2^6 個地址.

Address Format 1 是 single level 的, 所以 $V_0 \sim V_{\max}$ 中所有地址(4k 個)都要用 page 表示出來(包括那 1k 的 gap), 一個 page 可表示 2^6 個地址, 故需要 $4k / 2^6 = 4 * 1024 / 2^6 = 2^{12} / 2^6 = 2^6 = 64$ 個 page(即 entry).

Address Format 2 中: 一個 internal page table(不是 internal page) [注 1]可以表示 $2^4 * 2^6 = 2^{10} = 1k$ 個地址. 而 p_1 有 $2^2 = 4$ 種取值方法, 每一個 p_1 的值指向一個 internal page table, 即 p_1 的每一個取值可對應 1k 個地址. 而 Address Format 2 是 multi level 的, 故只用表示實際上要用到的地址(3k 個) [注 2], 故 p_1 的 4 種取值中, 有一個取值對應的 internal page table 可以不分配, 總共只需分配 3 個 internal page table. 這 3 個 internal page table 中, 每個 internal page table 中有 2^4 個 page(即 entry), 故總共要 $3 * 2^4 = 48$ 個 entry.

[注 1] 注意題目中, Address Format 1 中說的“in a single-level page table”, 用的單數, 而 Address Format 2 中說的“in the inner page tables”, 用的複數. 這也證明我之前的理解是對的: 在 multi-level 情況下, 是有多個 inner page tables 的.

[注 2] 注意本情況中, granularity 正好符合, 即 $V_0 \sim V_{\max}$ 中有 1k 個地址的 gap, 而 p1 的每一個取值可對應 1k 個地址, 故可以將那 1k 的 gap 略過。

10. 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 first address format in the case of the single-level page table, 6 bits are used for the virtual page number. That means that there will be a total of 2^6 , so 64 different virtual pages. And in a single-level page table design, we have to have an entry for every single virtual page number, so there will be a total of 64 elements. In the second address format, the first 2 bits are used as an index into the outer page table, so the page table directory, and the inner 4 bits are used as an index into the inner page tables. But, take a look at this address format, these 2 bits, so the outer page table entries, will address 2^{10} , 4 plus 6 virtual addresses from the virtual address space. That means that 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 and the last 1 kB 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, that 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

Overhead of Address Translation

For each memory reference ...

Single-level page table

- x1 access to page table entry
- x1 access to memory

Four-level page table

- x4 accesses to page table entries
- x1 access to memory

=> slowdown!

11. Now we know that adding levels to our address translation process will reduce the size of the page tables. But it will add some overheads to the address translation process. In the simple, single level page table design, a memory reference will actually require two memory references. One 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 four level page table, however, we will need to perform four 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.

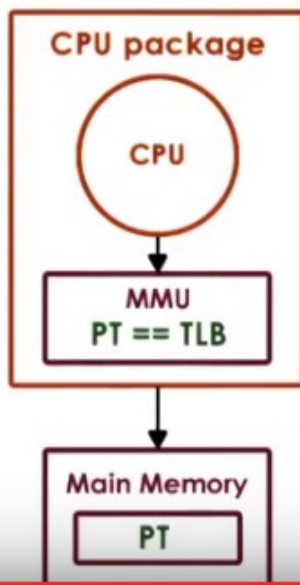
Page Table Cache (TLB)

Translation Lookaside Buffer

- MMU-level address translation cache
- on TLB miss \Rightarrow page table access from memory
- has protection/validity bits
- small number of cached addr \Rightarrow high TLB hit rate \Leftrightarrow temporal & spatial locality

x86 Core i7

- per core: 64-entry data TLB
128-entry instruction TLB
- 512-entry shared second-level TLB

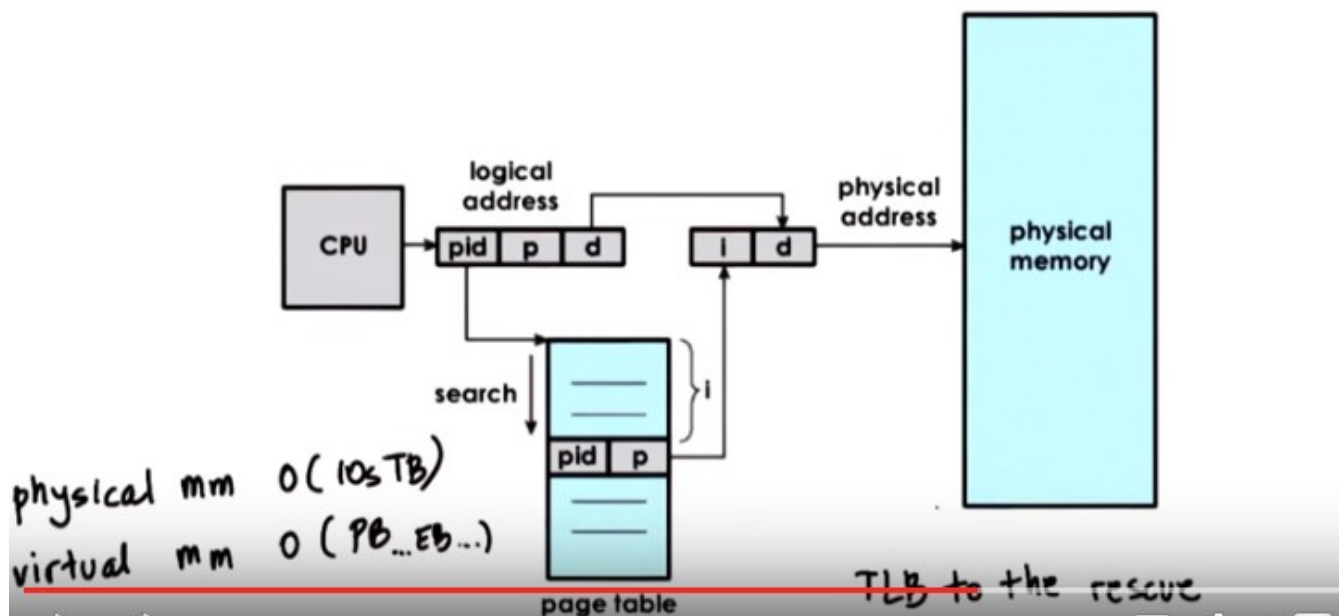


我電腦是 Core i3

The standard technique to avoid these repeated accesses to memory is to use a page table cache. On most architectures, the MMU hardware integrates a hardware cache that's dedicated for caching address translations, and this cache is called the Translation Look Aside Buffer or 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, so 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 instruction. And each of those has a modest number of entries. 64 for the data and 128 for the instruction TLB. These are per core and in addition to these two, there is also another shared second level TLB that's shared across all cores and that's, that one is a little bit larger. It has 512 entries. So this is for the i7 in platforms. And this was

determined to be sufficiently effective to address the typical memory access needs of processes today.

Inverted Page Tables



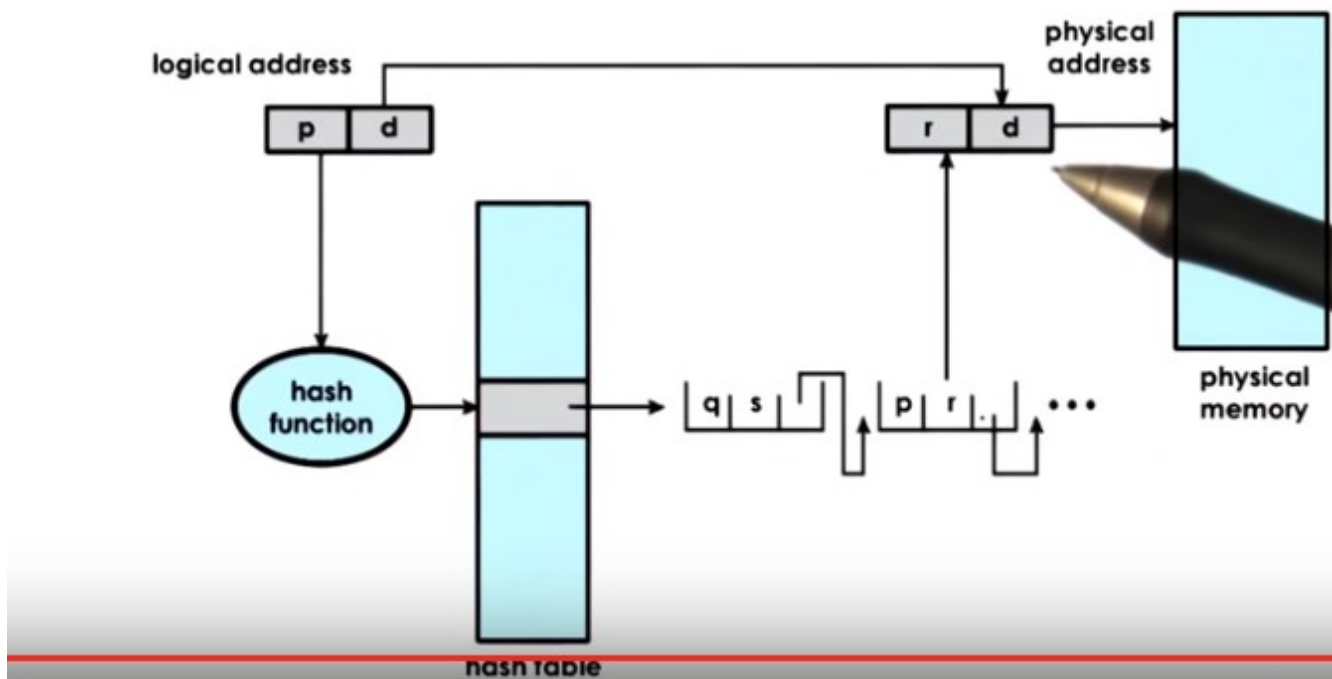
單位: EB > PB > TB

先看上圖, 再看下面文字.

12. A 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. So, 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 tens 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 versus something that's on the order of the virtual memory that a process can have. To find the translation, the page table is searched base on the process ID (pid) and the first part of the virtual address (p), 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's being co-reference 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 entry matches the pid p information

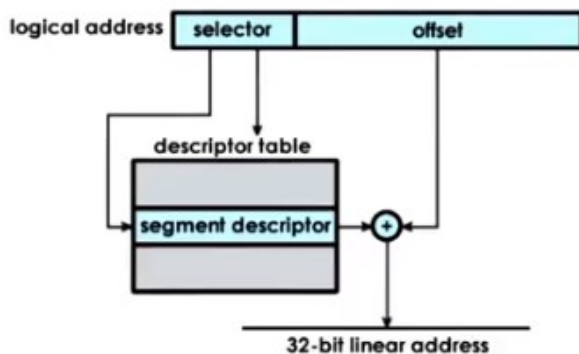
that's part of the logical address that was presented by the CPU. Since the physical memory can be arbitrarily assigned to different processes, the table isn't really ordered. There may be two consecutive entries that represent memory allocated to two different processes, and there really isn't some clever search technique to speed up this process. In practice, the TLB will catche 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 bit more efficient.

Hashing Page Tables



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 few possible entries into the inverted page table, as a result, we speed up the address translation.

Segmentation

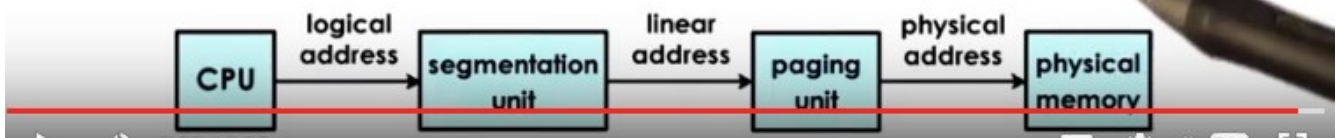


segments == arbitrary granularity
 - e.g., code, heap, data, stack...
 - $addr == segment\ selector + offset$

Segment == contiguous physical mem
 - segment size == segment base + limit registers

Segmentation + Paging

- IA x86-32 \Rightarrow segmentation + paging
 - Linux: up to 8K per process / global seg
 - IA x86-64 \Rightarrow paging



注意上圖中的 logical address, linear address, physical address

13. 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 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 contiguous portion of physical memory. In that case, the segment would be defined by 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's 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 multilevel, 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 you look at the Intel platforms, the x86 platforms, on 32 bit hardware both segmentation and paging are supported. For these platforms on Linux allows up to 8000 segments to be available per process and then 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 use just paging.

How large is a page?

10-bit offset \rightarrow 1kB page size

12-bit offset \rightarrow 4kB page size



Linux/x86: 4kB, 2MB, 1GB

Solaris/SPARC: 8kB, 4MB, 2GB

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

Larger pages:



fewer page table entries, smaller page tables, more TLB hits ...



internal fragmentation \Rightarrow wastes memory

Reduction factor 是相對於 4kB 來講的, 對於 large page size (2MB), reduction factor = $2\text{MB} / 4\text{kB} = 2 * 2^{10} * 2^{10} / 4 * 2^{10} = 2^{10} / 2 = 512$. 同理, 對於 huge page size (1GB), reduction factor = $1\text{GB} / 4\text{kB} = (1\text{GB} / 2\text{MB}) * 512 = 512 * 512$, 上圖中的 1024 應該是寫錯了, 因為下面文字中也說了: switching to huge page sizes will reduce the page table size by another 512.

14. So far we 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 use 10-bit for the offset or 12-bit 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 a 10-bit offset in the address field, that meant that these 10 bits could be used to address 2 to the 10th bytes in the page. And therefore it meant that the page size is 1 kilobyte. Similarly, the examples that had a 12-bit offset for the address format. That means that they could have addressed 4 kilobyte pages, 2 to the 12th. 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 platform, there's several common page sizes. 4 kilobyte page size is pretty popular, and that's the default in the Linux x86 environment. However, page sizes can be much larger, 2 megabytes, 1 gigabyte. The 2 megabyte pages are referred to as large pages, as opposed to the regular 4 kilobyte ones. In addition, x86 supports huge pages and these are 1 gigabyte in size. In the first case, to address 2 megabyte 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 that physical address. So **one benefit of using these larger page sizes is that more**

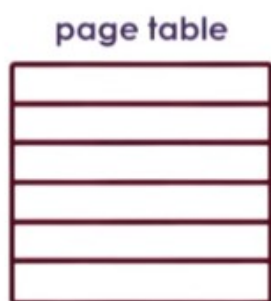
bits in the address are used for these offset bits. And therefore fewer bits are used to represent the virtual page number, so there will be fewer entries that are needed in the page table. In fact, use of these large page sizes will significantly reduce the size of the page table. Compared to the page table size that's needed when we're working with 4 kilobyte 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 there are few 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 down side of the larger pages is the actual page size. If this large virtual memory page is not densely populated, there will be a larger unused gaps within the page itself, and that will lead to, to what we call **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 4 kB 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 most sense. I should note that on different systems, depending on the operating system and the hardware architecture, different page sizes may be supported. So, for instance, on Solaris 10 on SPARC architecture, the page size options are 8 kilobytes, 4 megabytes, or 2 gigabytes.

15. Our first quiz about page tables looked at the address formats. And in the second quiz we will look at page tables again, but by looking at the page sizes. On a 12-bit architecture, what is the number of entries that is required in the page table, if the page size is 32 bytes? Also, think about what is the answer to this question, if the page size is 512 bytes. You should assume that the page table is a single-level page table. Write your answers in the text boxes.



Page Table Size Quiz

number
of virtual
page
numbers



On a 12-bit architecture what are the number of entries in the page table if the page size is 32 bytes? How about 512 bytes? (assume single-level page table)

32 byte page size: 128 entries
512 byte page size: 8 entries

32 byte: $\log 32 = 5$
 $\Rightarrow 7 \text{ bits for VPN}$
 $\Rightarrow 2^7 = 128$

512 byte: $\log 512 = 9$
 $\Rightarrow 2^3 = 8$

上圖那樣做反而不好理解, 還不如這樣做:

前面第 6 段末的計算中說了: “32-bit architecture 意思是: 整個 $V_0 \sim V_{\max}$ 有 2^{32} 個地址”(不是本題).

故 12-bit architecture 表示 整個 $V_0 \sim V_{\max}$ 有 2^{12} 個地址, 每個地址大小為 1B, 故地址共佔 2^{12}B . 而對於 32 byte page size, 一個 page 的大小為 32B, 故需要 $2^{12} / 32 = 128$ 個 page. 同理, 對於 512 byte page

size, 需要 $2^{12} / 512 = 8$ 個 page.

16. If the architecture is 12-bit, that means that the addresses are 12-bit long. If the page size is 32 bytes, then we need 5 bits for the offset into that page. That will leave 7 bits for the virtual page number, and therefore, we will need 128 to do the seven 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 with 3 bits for the virtual page number. As a result, the page table will need to have entries for all of the 2 to the 3rd number of virtual pages. So it will have 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.

Memory Allocation

memory allocator

- determines VA to PA mapping
- address translation, page tables...
 - ⇒ simply determine PA from VA and check validity / permissions

Kernel-level allocators

- kernel state, static process state

User-level allocators

- dynamic process state (heap); malloc / free
- e.g., dlmalloc, jemalloc, Hoard, tcmalloc



VA: virtual address

PA: physical address

17. So far, we have described how the operating system controls the processes' access to physical memory. But what we didn't explain was how the operating system 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 particular virtual memory regions. So 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, use of page tables, et cetera. They're 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 the user level. Kernel-level allocators are responsible for allocating memory regions, such as pages for the kernel, so for various components of the kernel state. And also these are used for certain static states, for the processes when they're created, like for their codes, stack,

or initialized datum. 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 state, for instance, for the heap. So this is state that's 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 then ultimately release it when they're done. Once the 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 friendliness with respect to how they behaved in a multithreaded 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-level 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. 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 requests of sizes two or four page frames, and let's say it's facing the following sequence of memory requests. The first memory allocation is for the request of two page frames and then the rest of the requests are for four 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 there are two free page frames.



Memory Allocation Challenges

Requests for Page Frames

`alloc(2), alloc(4), alloc(4), alloc(4)`

`free(2)`

Next request
`alloc(4)`

=> external fragmentation

Let's say next the two pages that were initially allocated or freed. So now you likely can already imagine what the problem is. If at this point a next request comes to allocate four pages, there are four free pages in the system. However, this particular allocator cannot satisfy this request since these pages are not contiguous (馬上看下句). Let's say the requirement with these allocation requests 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 as a result of them, we have basically holes of free memory that's not contiguous. And therefore, requests for largest contiguous memory allocations cannot be satisfied.

Memory Allocation Challenges

Requests for Page Frames

alloc(2), alloc(4), alloc(4), alloc(4)

free(2)


Next request

alloc(4)



16 page frames

In the previous example, the allocator had a policy in which the free memory was handed out to consecutive requests in a sort of first come, first served manner. Let's consider an alternative in which the allocator probably knows something about the requests that are coming. It knows that they will be coming for consecutive regions of two and four page frames. In the second case when the second request for an, allocating four pages comes, the memory allocator isn't allocating it immediately after the first allocation but instead is leaving some gap. The second allocation for four pages comes in at a granularity of four pages, and then the rest of the allocations are satisfied further below.



16 page frames

Memory Allocation Challenges

Requests for Page Frames

`alloc(2), alloc(4), alloc(4), alloc(4)`

`free(2)`

Next request
`alloc(4)`

\Rightarrow SUCCESS

\Rightarrow permits coalescing/aggregation of free areas

Now when the free request comes in, these two first pages are freed. The system again has four free pages. However, they're consecutive. Therefore, this next request for four 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 coalesce, to aggregate these adjacent areas of free pages into one larger free area. That way, it 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 freed areas.**



Allocators in the Linux kernel

- Buddy
- Slab

19. 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 and the second one is called the slab allocator.



Buddy Allocator

start with 2^x area

on request

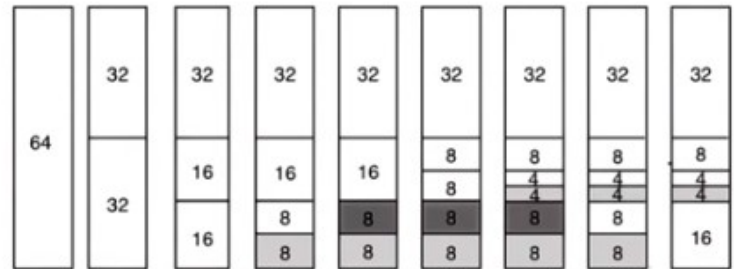
- subdivide into 2^x chunks and find smallest 2^x chunk that can satisfy request

=> fragmentation still there...

on free

- check buddy to see if you can aggregate into a larger chunk

- aggregate more up the tree



=> aggregation works well and fast

上圖的 64 意思是 64 個 pages. 上圖 request 順序: alloc(8)->alloc(8)->alloc(4)->free(8)->free(8)

The buddy allocator starts with some consecutive memory region that's free that's of a size that's a power of two. Whenever a request comes in, the allocator will subdivide this large area into smaller chunks such that every one of them is also a power of two. And it will continue subdividing until it finds the smallest chunk that's of a size that's a power of two that can satisfy the request. For instance, in this figure, when the first request of eight pages came in, the buddy allocator subdivided the region that was 64, the original area, first into two chunks of 32 pages. Then it subdivided one of these 32 page chunks into chunks that were 16 pages in each. Then it subdivided this 16 page chunk into chunks that were eight pages each. And it turned out that this eight page chunk satisfied the request that was for eight pages. So that was great. Subsequently, another request for eight pages came in, and then a request for four pages came in, and for that reason, this chunk of eight pages was subdivided into two chunks of four. Now when this eight page region is freed, there will be some fragmentation here. However, when the next eight 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 eight 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 eight page region start? If we didn't have this information, if we didn't know that the adjacent region is also an eight 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 nine, ten, 11, 12, or some other number of pages. So 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 only by 1 bit. This makes it easier to perform the necessary checks when combining or splitting chunks.



Slab Allocator

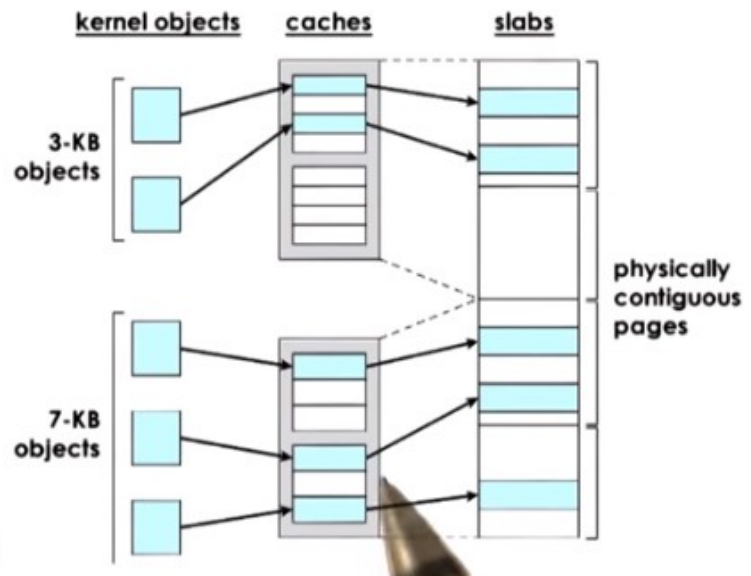
2^x granularity in Buddy
 \Rightarrow internal fragmentation
 \Rightarrow Slab to the rescue!

Slab allocator

- caches for common object types/sizes, on top of contiguous memory



internal fragmentation avoided
external frag. not an issue



Tao: Slab allocator means that (look at the figure above) we first create a cache for the 3-KB objects (and similarly a cache for the 7-KB objects). When we want to store a 3-KB object, we look into the cache and find a 3-KB object, and allocate a 3-KB space in the slab. Next time we need to allocate a 3-KB object, we put it just next to the previous 3-KB object in the slab. The same for the 7KB objects. Not sure if this is the correct understanding.

Define that allocations using the buddy algorithm have to be made in a granularity of a power of two, 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 two. 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 pre-create caches for the different object types. For instance, it will have a cache for a `task_struct` or for the directory entry objects. Therein, when an allocation comes from 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 preallocate an additional portion of contiguous physical memory to be managed by this slab allocator. The benefit of this slab allocator is that it avoids internal fragmentation. These entities that are allocated in the slab, they're of 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.

Demand Paging

Virtual memory >> Physical memory

- virtual memory page not always in physical memory
- physical page frame saved and restored to/from secondary storage

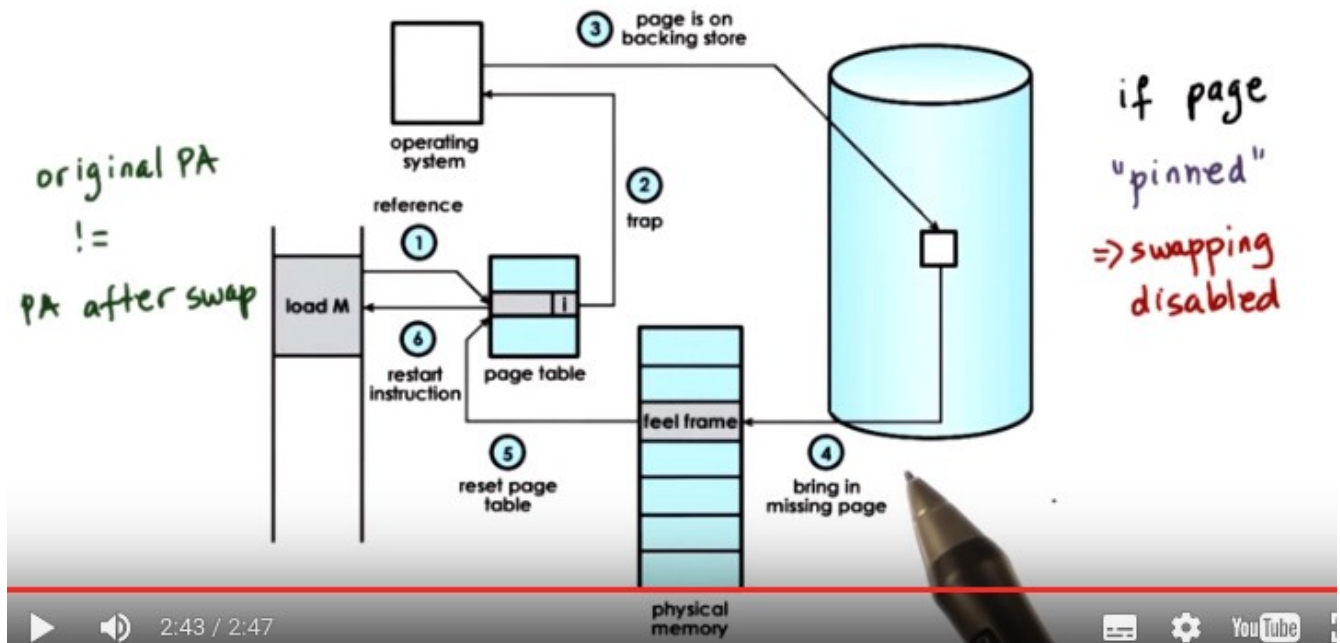
demand paging

⇒ pages swapped in/out of memory and a swap partition (e.g., on disk)

swap 的意思就是將一個內容從 memory 和 disk 之間移來移去。後面的視頻也提到過幾次 swap。

20. 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 theorem. Instead, the backing physical page frame can be repeatedly saved and restored to and from some secondary storage, like disks, for instance. And this process is referred to as paging or demand paging, and traditionally with demand paging, pages are moved between main memory and a storage device such as disk, where a swap partition resides. In addition to disk, the swap partition can be on another type of storage medium like a flash device, or it could even sit in the memory of another node.

Demand Paging



注意別忘了 demand paging 這個名字

Let's see how paging works. When a page is not present in memory, it has its **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 operating system kernel. On an access, the memory management unit will raise an exception, that's called the page fault (由此可知 page fault 是專指 page 不在 memory 而在 disk 中之情況, 但由後面的 COW 可知, page fault 不只是指這種情況), and this will be pushed into the operating system. So it will trap into the operating system 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 I operation to retrieve this page. Once this 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 instructions, 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 the the original physical address of this page will very likely be different from its 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 the same physical address during its 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 support, or DMA.

Freeing Up Physical Memory

WHEN should pages be swapped out?

WHICH pages should be swapped out?

21. 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?

Freeing Up Physical Memory

WHEN should pages be swapped out?

- page(out) daemon
- when memory usage is above threshold (high watermark)
- when CPU usage is below threshold (low watermark)

The first part is easier. Periodically when the amount of occupied memory reaches a particular threshold the operating system 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.

Freeing Up Physical Memory

WHICH pages should be swapped out?

- pages that won't be used
- history-based prediction
 - => Least-Recently Used (LRU policy)
 - Access bit to track if page is referenced
- pages that don't need to be written out
 - Dirty bit to track if modified
- avoid non-swappable pages

記住上圖中的 Access bit 和 Dirty bit

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 versus won't be used in the future? To make some predictions regarding the page usage, operating systems use some historic information. For instance, one 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 here 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 policy, 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. 即這種 page 就只放在 memory 中, 一旦從 memory 中 free 了之後, 就不存在了, 也沒人用它了. To assist with making this decision which pages don't need to be written out, the operating system 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 I operations that should never be swapped out. Then making sure that these pages are not considered by whatever replacement algorithms are executed in the operating system is going to be important.

Freeing Up Physical Memory



In Linux

- parameters to tune thresholds: target page count ...
- categorize pages into different types
 - e.g., claimable, swappable ...
- 'second chance' variation of LRU

In Linux and most OS's, a number of parameters are available to allow the system administrator to configure the swapping nature of the system. This would include thresholds such as the ones that we mentioned earlier that control when our page is 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 policy we described, and it gives a second chance. It basically performs two 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 decisions can be made in other operating systems as well.

22. We briefly discussed the Least Recently Used policy that is often used for determining which pages to swap in and out of physical memory. Consider the following problem. Suppose you have an array with 11 page-sized entries, and that all of these entries are accessed continuously in a loop, one after another. So they're accessed one by one. Also suppose that you have a system that has ten pages of physical memory. For the following system, answer this question. What is the percentage of pages that will need to be demand paged using the LRU policy? You should round up your answer to the nearest percent.



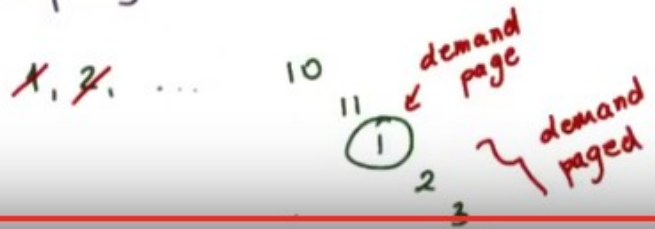
Least Recently Used (LRU) Quiz

Suppose you have an array with 11 page-sized entries that are accessed one-by-one in a loop.

Also, suppose you have a system with 10 pages of physical memory.

What is the percentage of pages that will need to be demand paged using the LRU policy? (round to the nearest %.)

100 %



Instructor Notes, 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. In this example, initially the first ten pages will be loaded into memory one at a time, as they're being accessed one-by-one. First page 1 [注 1], then page 2, then page 10. Now at this point, page 11 needs to be accessed. And that will mean that the first page, which is the one that's least recently used, needs to 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, page 1, is the very next page that's needed (即要 manipulate 它, 見上面代碼, page 1 即 page[0]). 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 reasons, operating systems can be configured to support different kinds of replacement policies that are used to manage their physical memory.

[注 1] 原 text 為 "First page 9, then page 2...", 視頻裡說的也是 9. 但我將其改為了 page 1, 理由為: 第一, 視頻中說 page 9 時, 指到的是 1. 第二, 由下文知, page 1 為 LRU. 第三, 由上面的代碼也可以看出, 最先 access 的也是 page[0], 即 page 1.



Copy-on-Write ("COW")



MMU Hardware

=> perform translation, track access, enforce protection ...
=> useful to build other services and optimizations

24. In our discussion about memory management so far, we saw that our 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 in 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.



Copy-on-Write ("COW")



On process creation...

- copy entire parent addr space
 - many pages are static, don't change!
- ⇒ why keep multiple copies?



create new process



Let's consider 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. 意思就是, 既然很多 page 是 static, 是不變的, 那為甚麼新的 process 不直接就用這個 address space, 而要去 copy 一個新的出來呢? 答曰否. 下面就要講如何不 copy.



Copy-on-Write ("COW")



On create...

- map new VA to original page
 - write protect original page
 - if only read
- ⇒ save memory and time to copy



create new process



VA1 = PA1



VA2 = PA1

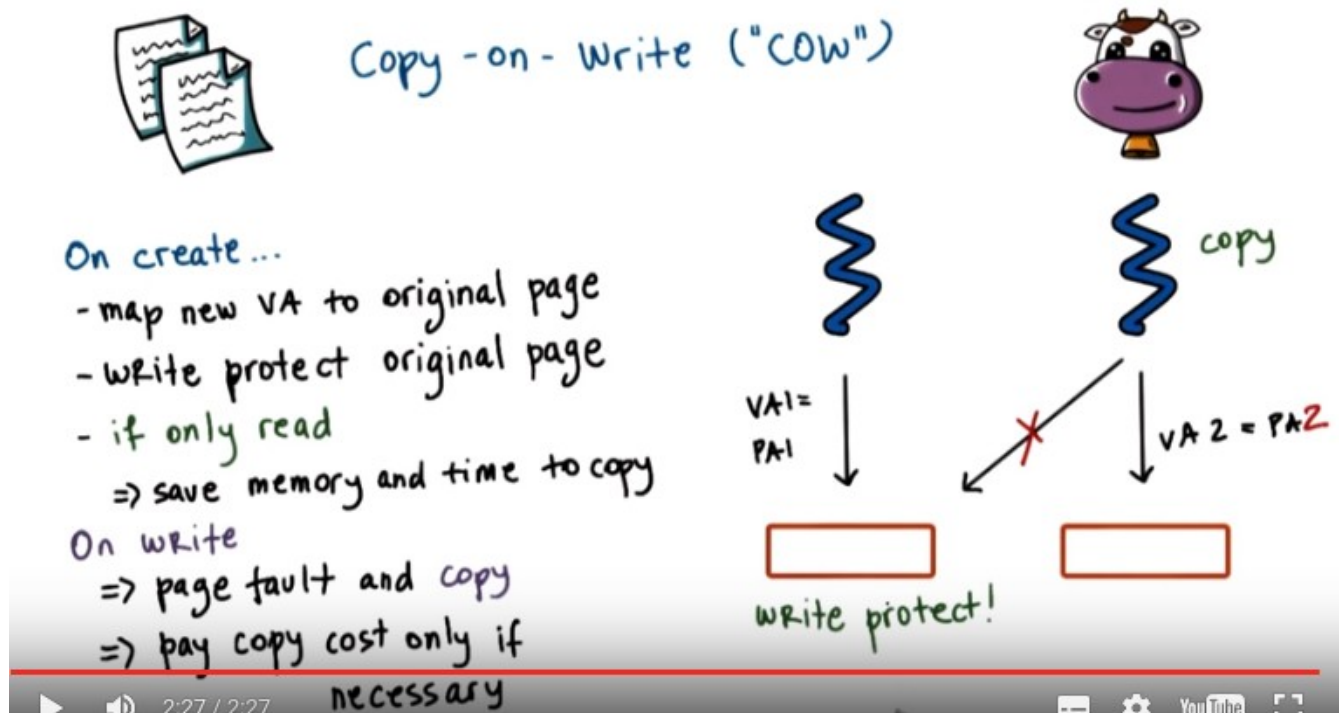


write protect!

上圖的紅框是一個 address space, 它是由一些 page 組成的.

In order to avoid unnecessary copying, on creation the virtual address space of the new process 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 two processes. We also have to make sure to write protect the physical memory so that we can track concurrent accesses to it. 以下一句話很 confusing, 根据下面 write 的情況, 可以推斷出本句的意思為: 若紅框為 read only, 則不 copy, 這樣就能節省 copy 所需的 memory 和時間. 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 been necessary to perform the copy'. 再看下面的紅字, 便恍然大悟 COW 之意思.



However, if a write request is issued for this memory area via either one of these virtual addresses (即若這兩個 process 中有一個要改紅框中的內容了, 此時就必須 copy 了!), then the MMU will detect that the page is write protected and will generate a page fault. At that point the operating system will see what is the reason for this page fault. We'll create the actual copy. So the copy will only be performed then. We'll update the page tables of the two processes as necessary. So basically the page table 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 (左邊的紅框) so whether or not the write protection will be removed once this one copy is performed will depend on who else is this page (左邊的紅框) is shared with.

Checkpointing

Checkpointing == failure & recovery management technique



=> periodically save process state

=> failure may be unavoidable

BUT

=> can restart from checkpoint,
so recovery much faster

25. Another useful operating system service, that can benefit from the hardware support from memory management is checkpointing. Checkpointing is a technique that's used as part of the failure and recovery management that operating systems or systems 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.

Checkpointing

Simple Approach

- pause and copy



Better Approach

- write-protect and copy everything once
 - copy diffs of "dirty" pages for incremental checkpoints
- => rebuild from multiple diffs, or in background

A simple 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 the 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 (at once 的意思是只做這一次全面 copy, 以後都只做局部小 copy 了. 為何要 write protect? 後面我馬上會在括號中說). However, since the process will continue executing, we won't pause it. It will continue dirtying pages. So, then we can track the dirty pages, again using the hardware MMU support, and we will copy only the diffs (之前 write protect 的目的可能就是讓用 diff 時好弄個 page fault 來提醒 OS), only those pages that have been modified. That will allow us to provide for an incremental check point. If we check point using these partial diffs of just dirty 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.

From Checkpointing to ...

Debugging

- Rewind-Replay (RR)
- rewind == restart from checkpoint
- gradually go back to older checkpoints until error found

Migration

- continue on another machine
- disaster recovery
- consolidation
- repeated checkpoints in a fast loop until pause-and-copy becomes acceptable (or unavoidable)

Consolidation: 合併, 鞏固

The basic mechanisms used in checkpointing can also be used in other services. For instance, **debugging** relies often on a technique called Rewind-Replay. Here rewind means that we will restart the execution of the same process from some earlier point. So 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** (應該是指將程序移到另一台電腦上去跑) 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 recoveries, 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. 以下的意思是,在某些情況下,我們想用 pause and copy(見上上圖中的 simple approach), 這樣做的一個方法就是不同 checkpoints 之間的時間間隔弄得很短. One way in which migration can be implemented is as if we are performing repeated checkpoints in a fast loop until ultimately, there is so few dirtied state from the process that something like the pause and copy approach becomes 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 (注意 pause and copy 中有個 pause) it in order to copy the remaining contents.

26. And to wrap up our discussion about memory management, let's take a quiz about checkpointing. 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, or all of the above.



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. The correct answer is all of the above. 第三個選項: The more frequently you checkpoint, the faster you will be able to recover from a fault. This is true because, with a frequent checkpoint you will have a recent checkpoint compared to the point of execution when the fault occurred. So you will have to replay or re-execute a less amount of time of the execution of the process. Clearly the more frequently you checkpoint, the higher the overheads of that will be. And furthermore, with frequent checkpoint, it's more likely that you will end up catching every single write to your particular page. If you spread out the checkpoints, it's possible that a single page will be written multiple times, so dirtied multiple times. And also, the more frequently you checkpoint, you will end up transmitting more state, checkpointing more state. And the reason for this is that with a frequent checkpoint, it's more likely that you will end up catching every single one of the references of the write updates to a particular page. If you spread out the checkpoints over time, it's possible that there will be repeated writes to a particular page that you will observe as a single dirty page and so you will amortize the checkpoint costs over multiple writes. With a frequent checkpoint, both the amount of the state that will be checkpoint, and in general the overheads of the process will be higher than if you do the checkpoint less recently. So this is just one of those tradeoffs where you end up gaining something, but that's going to cost you something else.

Lesson Summary

Memory Management

- Virtual memory abstracts a process' view of physical memory
- Pages and segments
- Allocation and replacement strategies and checkpointing

28. To summarize, in this lesson we look at virtual and physical memory management 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 process' 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 be used by some higher level services, like checkpointing.

29. As the final quiz, please tell us what you learned in this lesson. Also, we would love to hear your feedback on how we might improve this lesson in the future.