# Chapter 7

# Virtual Memory

An ISA specifies an address space large enough for programs to uniquely identify every object required by the program, both instructions to be executed and data to be executed on. Back in the 1970s, it was not uncommon for the maximum number of memory locations to be $2^{16}$ so 16 bits were enough to uniquely identify each of the $2^{16}$ memory locations. In the 1980s, the maximum number of addressable memory locations grew to $2^{32}$, requiring 32-bit addresses to uniquely identify each of the $2^{32}$ memory locations. Today, 48-bit addresses are not uncommon, even though $2^{48}$ memory locations is more than a little bit absurd. (Recall that $2^{48}$ is approximately one hundred trillion, represented in decimal as 100,000,000,000,000. A lot of memory!)

The reality is that a program only operates on a small number of memory locations at any one time, needing only a small number of memory locations to specify the relevant instructions and data that will be needed for processing in the near future. Thus, it is not necessary for a program to actually require a huge number of physical memory locations, provided the address of each physical memory location can be changed as necessary to accommodate the needs of the program executing. The mechanism for doing so is the Virtual Memory Model, wherein all the memory locations the process thinks it has and the values stored at each address (instruction or data) are located on the disk. Since they are on the disk and not immediately available for access by the program, they are referred to as Virtual Memory. All the memory locations containing instructions and data needed for processing in the near future are normally located in physical memory, where they can be accessed immediately. If instructions or data are needed now and are not in physical memory, that presents a problem, which we will get to later in this chapter.

Before we get into how Virtual Memory works, let's first look at an analogy.

## 7.1 An Analogy (The Telephone)

Suppose I decide to establish a start-up company to design, build, and sell widgets, and I hire 100 engineers to make it happen. I rent space in a factory to house my 100 engineers in one large bullpen area. I provide each with various tools to do their job, a laptop, a pad of paper, pens, etc. I want to buy each employee a cell phone to use in their job, but when I check, I find that the cost of cell phones is \$10,000 each. Being a cost-conscious entrepreneur, I balk at spending a million dollars on cell phones. I know that my engineers will not be on the phone a large fraction of the day, so I really do not need to provide 100 phones. I decided that 5 cell phones are enough.

I hire Chester to manage the cell phones. I give him a bag containing the five cell phones I have purchased. Chester sits at the front of the bullpen area , observing the 100 engineers working. If one of them wants to use the phone, he reaches across his desk to pick up the phone that isn't there. Chester sees this and rushes to the engineer's desk, placing the phone just in time for the engineer to grab it. Chester now has four phones in his bag. Sure enough another engineer reaches for a non-existent phone, and Chester rushes to her desk, again just in time. As engineers finish their calls and return the phones to their desks, Chester rushes in and collects them. As long as not more than five engineers want to use the phone at the same time, the system works just fine, and I have only paid \$50,000 rather than \$1,000,000 dollars for the phones. If a sixth person wants a phone, Chester notices that one conversation has both parties silently thinking, so he grabs their phone. When they want to resume talking, Chester repeats this process from another conversation. Chester is getting a little tired, so the time to perform these tasks takes longer, requiring a longer time period to complete a conversation. If a seventh or eighth person wants to use the phone, the delay is even worse.

So what does this have to do with Virtual Memory? Answer: We have 100 engineers, but only a few need phones at any one time. In our computer system, we have a number of processes, each possessing a huge number of virtual addresses, but only a small number of those addresses need to be in physical memory at any one time. Chester manages the process of plugging in and plugging out the phones that are needed, are done, or are just idle at the moment. The Memory Management Unit of the Operating System manages the needs of addresses that need to be moved from the disk to physical memory (or back). As the number of phones required grows beyond 5 or the needs of processes for more memory grow, the time required to complete a conversation or execute a program grows. When the needs get to a certain point, the system breaks down, and all useful work stops. Operating Systems people call that thrashing. They will also tell you the best way to solve the thrashing problem is to provide more physical memory. In my start-up, the best way to solve the cell phone problem is to buy more phones.

I could go on with my analogy, but I will stop after sharing one more point. We all own cell phones. Our phone has two numbers identifying it: the phone number that people call to reach you and the actual serial number of your current cell phone. I submit that the phone number people use to call you is the virtual address of your phone. When you buy a new phone, you keep that phone number. The physical address of your current phone is the serial number of the actual device. Few people know the physical address of their phone. So, too, with your virtual and physical memory addresses. When you execute the instruction LD R1, A, A is the virtual address of the memory location containing the value you wish to load into R1. The physical address is whatever location the operating system happened to designate for A. In fact, if that location is kicked out of physical memory and later brought back, it most likely will have a different physical address.

## 7.2 Characteristics of Virtual Memory

### 7.2.1 Virtual Address Space and Physical Address Space

Every process in the computer has a large virtual address space, large enough to uniquely identify every object (instructions and data) that is needed to execute the process' program. The computer has a physical address space corresponding to the actual physical memory locations that are present in the computer system. Physical memory is partitioned into physical frames. These frames are shared among the processes. Each process' virtual memory is partitioned into virtual pages. Virtual pages and physical frames are the same size. The entire virtual address space of a process resides on the disk. Since at any point in time, only a small subset of these pages actually need to be accessed by the program, only a small subset of the pages are actually in physical memory at any instant of time.

We say a page is resident if it is in physical memory. When resident, a virtual page occupies one frame of physical memory. Every byte of the page occupies the corresponding byte of the frame. That is, byte k of the page occupies byte k of the frame for all bytes of the page. Most (not all) ISAs set the page size to 4KB. (We will discuss page size later in this chapter.) The set of pages of a process that is in physical memory at the same time is called the "working set" of the program. If the working set is too small, a lot of time is wasted moving pages as needed from disk to physical memory and back to the disk in order to have individual bytes of storage in physical memory when they are needed. If the working set is too large, more frames of memory are used than is necessary, making those frames unavailable for other processes. The set of processes whose working sets are resident is called the "balance set" of the system.

## 7.2.2 Page Table Entries (PTEs)

Every process has its own set of pages representing its virtual address space. Each page of this set has a descriptor called its Page Table Entry (PTE) which contains important information about that page of virtual memory. Figure 7.1 shows the information contained in a typical PTE specified by the ISA. The set of PTEs for the set of virtual pages comprises the process' page table.

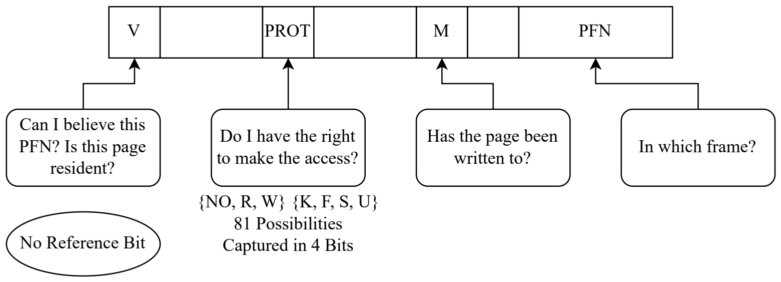
V
PROT
M
PFN
Can I believe this PFN? Is this page resident?
Do I have the right to make the access?
Has the page been written to?
In which frame?
{NO, R, W} {K, F, S, U}
81 Possibilities
Captured in 4 Bits
No Reference Bit

Figure 7.1: The Page Table Entry

### 7.2.3 The Page Table Entry

Each PTE keeps track of relevant information about its page. A valid bit tells whether or not the page is resident in physical memory, and if it is, the PTE contains the address of the physical frame that contains that virtual page. The PTE also contains a PROTECTION code, which specifies the level of privilege required in order for the process to make each allowable access.

Although access control and virtual to physical address translation are the two most important functions the virtual memory management system performs, the ISAs of some computers specify other items in the PTE that enhance the performance of the computer system.

For example, the PTE may contain a modify bit which is set to 1 if a store instruction to that page occurred since the page was brought from the disk into physical memory. If a page is to be returned to the disk and that bit is still 0, no store instruction to that page occurred since that page was brought into physical memory. That means the page on the disk is identical to the page being kicked out of physical memory, so there is no need to write that page back to the disk. Time is saved, performance is enhanced.

Some ISAs specify a bit (called the REFERENCE bit) in the PTE that is helpful in determining which frame is to be used if a page that is not resident has information that is needed to execute the current instruction. To execute that instruction, the page needs to occupy a physical frame. The REFERENCE bit helps identify which frame should be used. The current page occupying that frame would be kicked out (i.e., returned to the disk), and the page needed to execute that instruction would be brought from the disk to that frame.

### 7.2.4 Access Control and Translation

The two main functions of the virtual memory management system are (a) to be sure the processor is permitted to perform the function it wishes to perform (i.e., access control), and if so, (b) to translate the virtual address that the program has specified in the physical address that the processor will use (i.e., translation) to perform the task. Both functions require information contained in the PTE.

#### Access Control

All bytes on a page have the same access privileges. Three things contribute to permitting the access to occur: (a) the privilege level that the process is operating under, (b) the access that the processor wants to perform, and (c) the PROTECTION code in the PTE that specifies what level of privilege is required to perform the access. Figure 7.2 shows the logic needed to determine if the access is permitted. If permitted, the access is performed. If not permitted, the processor takes an exception, Access Control Violation.

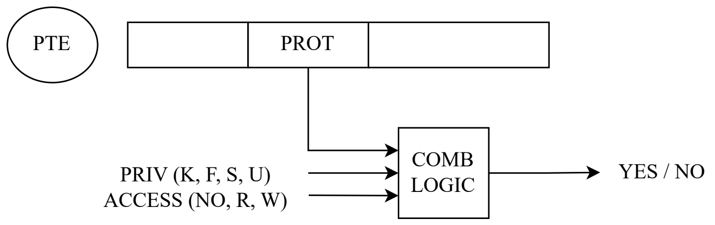

Figure 7.2: Access Control

### 7.2.5 Access Control

Accesses can be (with increasing levels of privilege required): no access, the right to execute code on the page, the right to read what is stored on the page, and the right to read and write what is stored on the page. Higher privilege is required to read what is on a page than simply to execute the code on the page. Higher privilege is required to write to the page than to simply read what is on the page.

Why does reading require a higher privilege level than executing? Answer: the owner of the code on the page may be perfectly comfortable licensing a customer to execute code on the page, but does not want that customer to read the page, and perhaps steal and sell the code he has only been licensed to execute.

Why does writing require a higher privilege level than reading? Answer: the university is perfectly happy allowing me to read my salary information, perhaps to find out if I got a raise, but does not want me to be able to give myself a raise by writing a higher salary into the appropriate memory location.

In moving from the lowest level of privilege (often referred to as "user mode") to the highest level of privilege (usually referred to as "kernel mode") the ability to access is never decreased. Most ISAs specify two levels of privilege, unprivileged and privileged. Some ISAs specify four privilege levels: User, Supervisor, Executive, and Kernel. If, for example, an ISA has these four privilege levels and for a given page, "supervisor mode" allows execute and read accesses, then "executive mode" and "kernel mode" also allow execute and read accesses. If in addition, executive mode allows write access, then kernel mode would also allow write access.

The PROTECTION code specifies the highest access permitted on the page for each privilege mode. For example, the VAX ISA has four privilege modes and three accesses: no access, read access, and write access. Based on the previous paragraph, four privilege modes, each with three possible highest access it can perform, would mean $3^4$ unique codes. Actually, these 81 protection codes can be specified with 4 bits! How come? Answer: The allowed accesses can never decrease when going from a lower to a higher privilege mode. For example, User mode can never allow accesses that are denied to Kernel mode. Only 15 of the 81 cases obey that requirement. Ergo, 4 bits are enough!

**Translation**

In order to access a memory location, it must be in physical memory. If the page is occupying a physical frame, the VALID bit of the page's PTE is 1, and the frame that the page is occupying is specified by the PTE's page frame number (PFN). Suppose the virtual address we wish to access is X, and it is on page A of the virtual address space. If the page is resident, the valid bit of page A's PTE confirms that the PTE's page frame number is the frame containing page A. The physical address can be constructed by concatenating the frame number with the page offset, as shown in Figure 7.3. Recall pages are mapped directly to frames, i.e., byte k of a page is byte k of its frame.

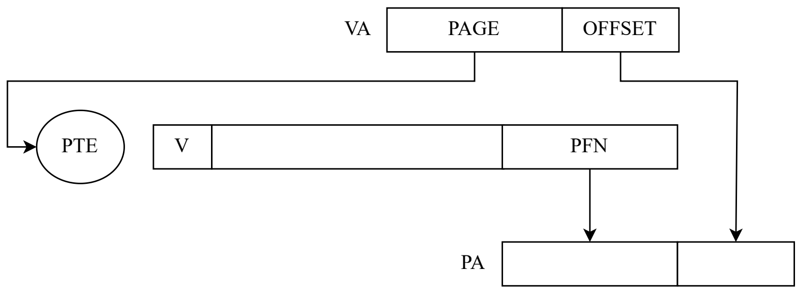

Figure 7.3: Translation

### 7.2.6 Translation

If the page is not resident, i.e., the VALID bit of the PTE is 0, then the microarchitecture initiates an exception, turning control over to the operating system to deal with the situation. The official name for the exception is Translation Not Valid (TNV) fault. More commonly, it is referred to as a "page fault."

The job of the operating system is to find a page (call it page B) to kick out of its frame in order to use that frame for storing page A, and store page B on the disk (if the modify bit is not zero), load page A in that frame, take care of some important bookkeeping, and then return control to the process that initiated the page fault. The bookkeeping consists of changing the page frame number in Page A's PTE, setting the valid bit of Page A's PTE to valid, and setting the valid bit of page B's PTE to invalid.

Several mechanisms are available to the operating system for designating which page will be removed from its frame in order to accommodate page A. One simple mechanism is the use of a REFERENCE bit in each page's PTE. When a page is brought into physical memory, its REFERENCE bit is set to 0. Periodically, all PTE reference bits are set to 0. If a page is accessed, its reference bit is set to 1. If a frame needs to be freed up to handle a page fault, the operating system chooses a page whose reference bit is 0, assuming that that page has not been accessed in a while, and is perhaps less in need of remaining in physical memory.

### 7.2.7 Summary of Virtual/Physical Memory Organization

We conclude this section with a layout of the pages of a virtual memory system.

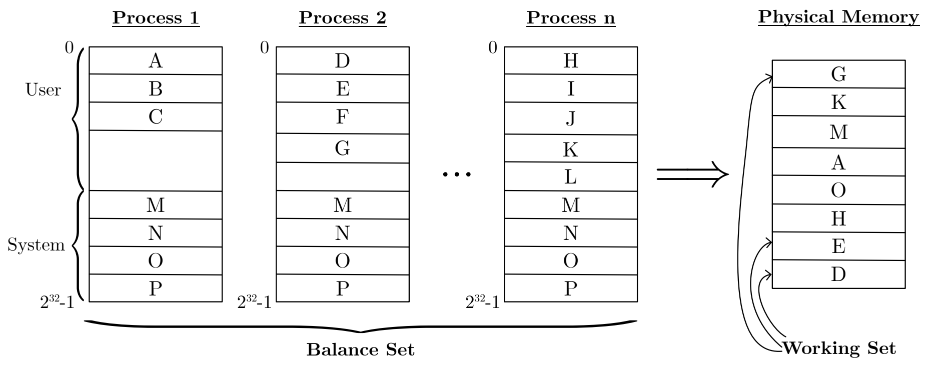

Figure 7.4: Virtual memory summary, with balance set

### 7.2.8 Virtual memory summary, with balance set

The Figure shows three processes, process 1, 2, and n, and the pages that comprise each process. Each letter represents a page of virtual memory. Physical memory consists of 8 frames. Each process has its own user space and a shared copy of the system space. System space consists of four pages (M, N, O, P); M and O are resident, M in frame 2 and O in frame 4. The Balance Set consists of processes 1,2,...n. Process 1 consists of three pages of user space (A, B, C). Only A is resident and occupies frame 3 of physical memory. Process 2 consists of four pages of user space (D, E, F, G). D,E, and G are resident in frames 7,6, and 0, respectively. D, E, and G comprise the working set of Process 2.

## 7.3 Translating VA to PA

As we have repeatedly said, programs describe memory locations by means of their virtual addresses. This gives each process the ability to reference far more memory locations than it actually has available to it. Not a problem if the virtual memory locations of the process that we want to access are actually resident in some physical memory locations. To make this work, we need to translate each virtual address to a physical address before we attempt the access.

Recall that every address, both virtual and physical, is made up of two parts. In the case of virtual addresses, the two parts are the page number and the byte within the page. In the case of physical addresses, the two parts are the physical frame number and the byte within the frame. Since the size of a virtual page is the same as the size of a physical frame, translation of a VA to a PA is simply replacing the page number with a frame number, as shown in Figure 7.3. If a page is resident, its PTE contains the physical frame number that occupies that page, and the valid bit is 1. Translation therefore consists of getting the PTE of the page and replacing the page number with the page frame number.

To get the PTE of a page, one must access the Page Table containing that PTE. A Page Table consists of a set of PTEs. A process can have multiple page tables. We will consider the simplest case wherein each process has two page tables, one specific to that process for pages of user memory and another one for pages of privileged memory, which it shares with the other processes in the Balance Set, as we saw in Figure 7.4. Each PTE in the page table specifies if the corresponding page is resident, and if so, which frame of physical memory that page occupies. The PTEs of each page table are organized sequentially in memory; i.e., first the PTE of page 0, followed by the PTE of page 1, followed by the PTE of page 2, etc. Each page table has two registers, one containing the starting address of the page table, the other specifying the number of pages in the process. The register containing the starting address of the page table is the Page Table Base Register (PBR). Usually, the first entry of the page table (i.e., the PTE of Page 0) is stored in the first bytes of a page. That is, the offset bits that determine the starting address of a Page Table are all 0s. The register specifying the number of pages of the process is the Page Table length register (PLR).

A page table can be housed in physical memory or in system virtual memory.

### 7.3.1 Case 1: The Process Page Table is in Physical Memory

Figure 7.5 shows an example of the case where the page table is in physical memory.

### 7.3.2 The Page Table is in physical memory

The process consists of n pages. The page table consists of n PTEs, starting at physical address A'00...0. A is the frame number of the first page of the Page Table. "00...0" is the page offset of the PTE of page 0. Assuming a PTE consists of 4 bytes, the address of the PTE of page 1 is A'00...100. The address of the PTE of page 3 is A'00...01100. Since PBR contains the starting address of the Page Table, the address of the PTE of page k is PBR + 4 x k. Since

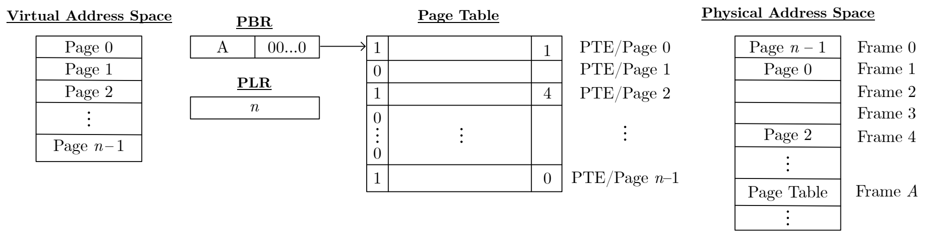

Figure 7.5: The Page Table is in physical memory

the PTEs are organized sequentially, one can access page k by indexing into the page table; i.e, the PTE of page k is contained in physical memory location A'00..0 + 4 × k. The PLR contains the value n-1, since the process comprises n pages. In Figure 7.5, Pages 0,2, and n-1 are resident, in frames 1,4, and 0, respectively.

### 7.3.3 The Translation Process for Case 1

The VA to PA translation process involves 5 steps. Assume x is on page k.

**Step 1**: Is PLR < k, the page number of the VA of x? If yes, the VA is ill-formed, the VA specifies an address that is not part of virtual address space of the process. Access is denied, and an Access Control Violation (ACV) exception is taken.

**Step 2**: Get the PTE. The address of the PTE is PBR + 4 × k. Since the Page Table is in physical memory, this step can be performed quickly and easily by indexing the Page Table with PBR + 4 × k.

**Step 3**: Check protection, verify the process' right to perform the access specified by the instruction being processed. If the access is permitted, go to Step 4. If not, the access is denied, and an ACV exception is taken.

**Step 4**: Check the PTE's valid bit. If 0, the page is not resident. A translation not valid (TNV) fault, more commonly referred to as a page fault, is taken. If 1, the page is resident, go to Step 5.

**Step 5**: Perform the translation. i.e., replace the virtual page number with the frame number obtained from the PTE.

### 7.3.4 Case 2: The Process Page Table is in System Virtual Memory

Translating a VA to a PA if the page table is in System Virtual memory is more complicated than the case where the page table is in physical memory. Then why do it, one asks. Answer: One does not need all the PTEs in physical memory at the same time, which is what you get if the page table is in physical memory. Suppose we have the virtual address space of today's x86 ISA. The address space is $2^{48}$ bytes of virtual memory. With 4KB pages that would mean $2^{36}$ pages if the process used all of virtual memory. At 4 bytes per PTE, that would mean a page table of $2^{38}$ bytes of storage. 256GB used to implement a data structure needed to implement virtual memory. Clearly beyond the pale.

On the other hand, suppose we put the process page table in system virtual space. And suppose our process virtual space out on the disk accommodates $2^{48}$ bytes of virtual storage. At 4KB per page of storage, the $2^{48}$ bytes would require $2^{36}$ pages of system virtual memory, which would need $2^{36}$ PTEs, which means $2^{38}$ bytes to store the process page table. This would require $2^{26}$ pages of systems space, which would require $2^{28}$ bytes of physical memory to store the system page table. The result: 256 MB of physical memory. Not small to be sure, but a lot smaller than the 256 GB that would be required if the process page table were stored in physical memory.

Figure **??** shows an example of the case where the page table is in system virtual memory.

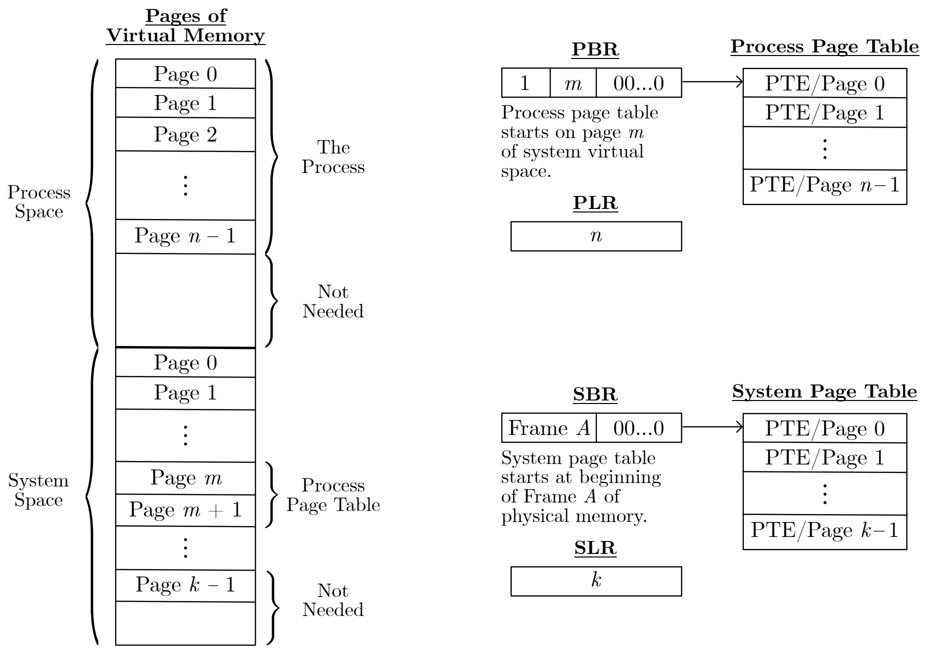

Figure 7.6: The Page Table is in System Virtual memory

### 7.3.5 The Page Table is in System Virtual memory

As before, we have PBR and PLR, which provide the same functions as in Case 1. Our example shows the usage of process space to be n pages of virtual memory. However, since we are now storing the process page table in system virtual space, we need to introduce SBR and SLR, the base register and length registers of the System Page Table. Note that the high bit of the address stored in PBR is 1, reflecting that the addresses of the PTEs of the process page table are all in system space. Note also that the process page table starts on Page M of system virtual space. Finally, note that the address in SBR is a physical address. It is the starting address of the System Page Table, which is in physical memory.

### 7.3.6 An Example of Case 2: VAX Virtual Memory

We will close out our treatment of virtual to physical address translation with an example, using the structure of the VAX Virtual Memory System.

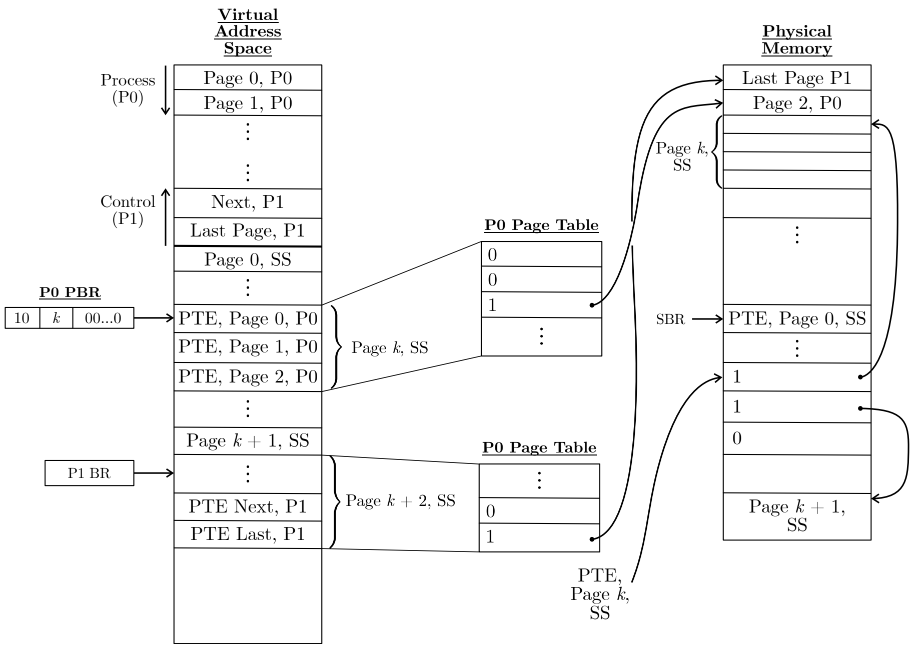

Figure 7.7: VAX Virtual Memory System

### 7.3.7 VAX Virtual Memory System

Figure 7.7 shows both the virtual address spaces of a process and the operating system, and the physical address space of the actual physical memory. For the VAX, process virtual space is partitioned into two parts, P0 space, which contains the process' program and data, and P1 space, which is mostly used for the process' user stack. P0 space grows from virtual address 0 to larger addresses. P1 space grows from virtual address $2^{31}$ -1 toward 0. System space grows from virtual address $2^{31}$ toward virtual address $2^{31}$ + $2^{30}$ -1. Three page tables are necessary to enable access to the locations in physical memory containing the contents of each desired virtual address – one page table for each of the three (P0, P1, System) virtual spaces.

The System Page Table is in physical memory. The SBR contains the physical address of the PTE of Page 0 of System Space, and SLR contains the number of pages that comprise systems space. Note that pages k and k+1 of System Virtual Space are resident.

The P0 Page Table is in System Virtual Space, starting at the beginning of Page k of System Virtual Space. Note the two highest address bits of the P0 Base Register is "10", indicating the address is in System Virtual Memory. Page 0 and Page 1 of P0 Space are not resident. Page 2 is resident in frame 1 of physical memory. The P1 Page Table is also in System Virtual Memory. The last page of P1 space is resident in frame 0 of physical memory.

### 7.3.8 The Translation Process for Case 2

With the Process Page Table in System Virtual Memory, getting the PTE of the page containing the address we wish to translate is more complicated. To translate the VA of x to the physical address where x is actually stored takes 7 steps.

**Step 1**: Let's say the VA of x is on page A of System Virtual Memory. We first check that Page A is part of the Process Space by comparing A to the contents of the PLR. If A is larger than PLR, our program has created an ill-formed page number, and an Access Control Violation exception is taken.

**Step 2**: If we have a valid page number, we compute the VA of the PTE of page A, i.e., PBR + 4 × A, and use that as the index into the process page table to obtain the PTE of Page A.

**Step 3**: Let's say the VA of the PTE of page A is on Page B of System Virtual Memory. We can get the PTE of Page B of System Virtual Memory by accessing the System Page Table, which is in physical memory. We do this by indexing into the System Page Table with index SBR + 4 × B, obtaining the physical address of the PTE of Page B of System Virtual Memory.

**Step 4**: We create the PA of the PTE of Page A by concatenating the frame number of the Page B with the offset of the PTE of Page A.

**Step 5**: Since we now have the PTE of Page A, we can check to be sure we are permitted to make the desired memory access. If the protection check fails, we take an ACV exception.

**Step 6**: If the protection check succeeds, we check the valid bit of the PTE of page A to see if the page containing X is resident. If the valid bit is 0, the page is not resident, and a page fault is taken.

**Step 7**: If the valid bit is 1, Page A is resident, and the PA of x is created by concatenating the frame number of page A (obtained from the PTE of A) with the offset bits of virtual address x.

The microarchitecture now has the PA of x and can use it to process the instruction in the program.

This process is referred to as "walking the page table." It is shown in Figure 7.8.

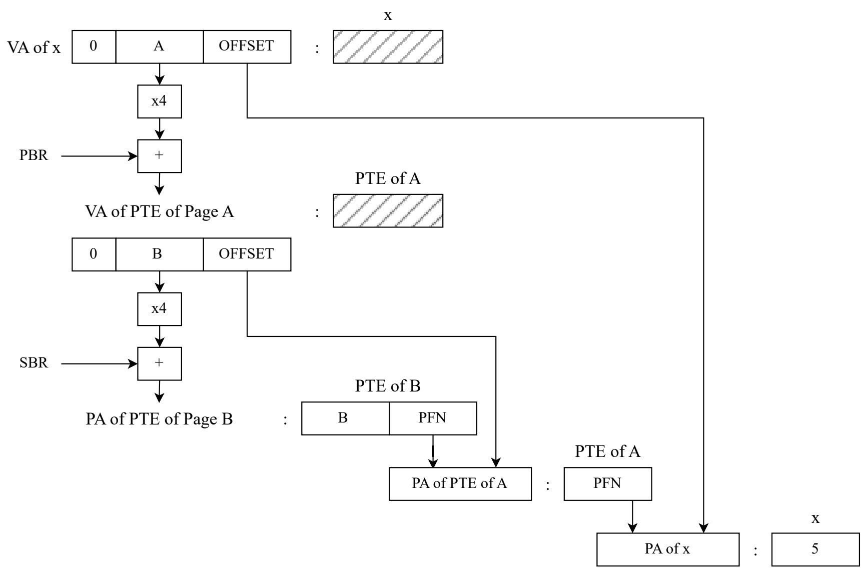

Figure 7.8: Walking the Page Table

### 7.3.9 Walking the Page Table

### 7.3.10 A Complete Example: Processing LD R1, X

We conclude this section with the processing of an instruction LD R1,X. We have changed the size of the memories to get rid of the unnecessary clutter of using large addresses when smaller addresses will work just fine for understanding the concepts. We modify the ISA as follows:

The page size will be 16 bytes instead of the VAX page size of 512 bytes. The VA will be 9 bits, instead of 32 bits – 1 bit to determine process or system space, 4 bits for the Page number, and 4 bits to identify the byte on the page.

To process the instruction LD R1, X we will need to translate the virtual address of X from 0x058 to its physical address. The VA of X tells us that X is in byte 8 of page 5 of process virtual memory.

The Physical Memory will have 128 bytes, consisting of 8 frames of 16 bytes each. The PTE will remain a 4 byte descriptor.

Our example will use 6 pages of the available 16 pages of process virtual memory, and 5 pages of the available 16 pages of System Virtual memory.

The Process Base Register has the VA 0x120. The System Base Register has the PA 0x50.

Figure 7.9 shows a memory map of our example.

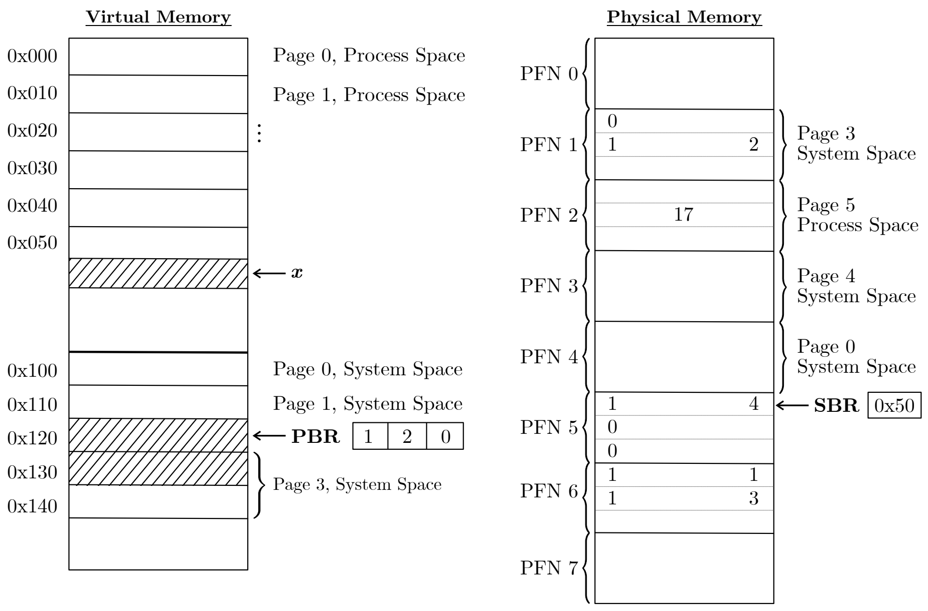

Figure 7.9: A Memory Map

### 7.3.11 A Memory Map

Note that system pages 0, 3, and 4, and process page 5 are resident. Pages 0, 3, and 4 of System Virtual Memory are in frames 4, 1, and 3, respectively. Page 5 of Process Virtual Memory is in frame 2.

The System Page Table is in frames 5 and 6. The PA of the SBR 0x50, the start of frame 5. Since the page/frame size is 16 bytes and the PTE is 4 bytes, the 5 PTEs that make up the System Page Table require all 16 bytes of frame 5 and the first 4 bytes of frame 6.

The VA of the PBR is 0x120, indicating that the process page table starts at the beginning of page 2 of System Virtual Memory.

To load X into R1, we need to access the Physical Address of X. The virtual address of X is on page 5 of Process Virtual Memory. To get the PA we need the PTE of page 5 of Process Virtual Memory. The process page table is in System Virtual Memory. The PTE of Page 5 is computed by adding PBR, the start of the Process Page Table (0x120), to $4 \times 5$ (0x14) to account for the 5 PTEs before we get to the PTE of Page 5, the 6th page. The VA of the PTE of Page 5 is therefore 0x134. This is byte 4 of Page 3 of System Virtual Memory.

Since the System Page Table is in physical memory, the PA of the PTE of Page 3 of System Virtual Memory is 0x5c. We get that physical address by adding the physical address of the SBR (0x50) to $4 \times 3$ (i.e., 0x0c) to account for the three System Virtual Memory PTEs that occupy space in the System Page Table before we get to the PTE of page 3.

The PTE of page 3 System Virtual Memory is at PA 0x5c, i.e., byte c of frame 5 of physical memory. Since that address is a physical address, we can read the PFN, i.e., 1. Since the VA of this PTE is 0x134, we concatenate the byte offset 4 to the PFN, yielding 0x14, the PA of the PTE of the page containing X. We can read the PFN from that PTE; it is 2. We concatenate the frame number (2) with the byte offset of X (8) to get the PA of X, 0x28.

We read the contents of 0x28, i.e., 17, and load that value into R1. Done!

### 7.3.12 The Translation Lookaside Buffer (TLB)

The first VAX, the VAX-11/780, took 22 clock cycles to carry out the work of walking the page table, translating a Virtual Address to a Physical Address. Since VAX instructions on average required one translation every instruction, spending 22 cycles to accomplish this would result in the VAX running like a dog, certainly not a competitive product. Actually, on average VAX instructions take ten clock cycles to execute. How can the processing of an instruction take ten cycles if one part of the processing alone always takes 22 clock cycles? Answer, most of the time walking the page table is not necessary to performing the translation. How come? Answer: The Translation Lookaside Buffer (TLB).

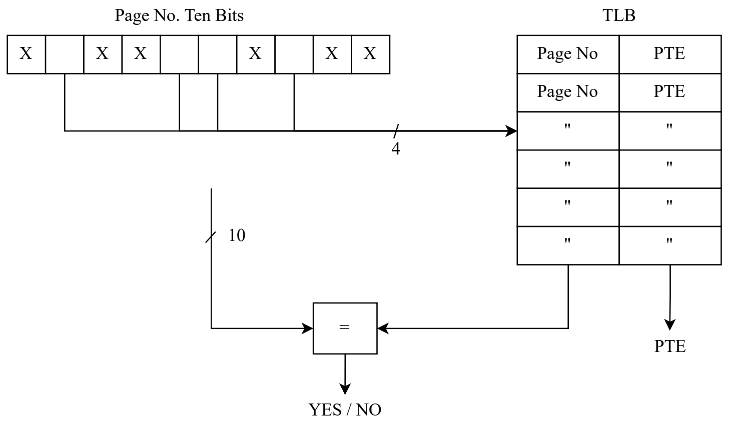

Figure 7.10: The Translation Lookaside Buffer (TLB)

### 7.3.13 The Translation Lookaside Buffer (TLB)

The Translation Lookaside Buffer is a storage structure in which each entry consists of two components, the Page Number, and the PTE. The TLB affects translation in the following way. Each time the microarchitecture encounters a Virtual Address it needs to translate, it first looks in the TLB to see if the page number and PTE are there. If it is not there, an expensive activity (in the case of the VAX-11/780, a 22 clock cycle table walk takes place, at the end of which an entry consisting of the Page Number and PTE for that page is added to the TLB. If the entry is in the TLB, the translation is done in a fraction of a clock cycle and no penalty is incurred. Measurements of the VAX running programs have reported that 95% of the times that the microarchitecture checks the TLB the entry is there. Thus, the penalty incurred in performing the translation is $0.05 \times 22$ cycles + $0.95 \times 0$ cycles. The cost is approximately one clock cycle.

**A side note:** We have earlier in this book talked about various means of accessing storage. Random Access Memory(RAM), where the current access is said to be independent of the previous access. Direct access (DASD), referring to accesses to data on a rotating disk where the access consists of two parts, the time it takes to position the disk head on the relevant track followed by the time to rotate into proper position, both very slow, and the second part, the faster swooping up information from the disk, once we have gotten the disk head and rotation taken care of. Another mechanism for accessing storage is to include part of the address in the entry that we are accessing. The TLB is an example of this. Each entry in the TLB consists of two parts, the Page number and the Page's PTE. The Page number is part of the information IN the entry that contributes part of the address. We refer to that mechanism as Content Addressable Memory (CAM). An expensive way to address such a memory is to simultaneously compare all page numbers in the TLB with the Page number of the PTE you are looking for. If you get a match, you have accessed the entry you want. There are more clever ways of using the contents to access the desired entity, but it is beyond the scope of what we wish to deal with right now.

## 7.4 Page Size

One remaining notion we have not discussed yet is the amount of information we move from the disk to physical memory and from physical memory to the disk when we take a page fault. We refer to the unit of storage as the page size. There are pros and cons associated with making the page size larger or smaller. Yet, in spite of all the discussion, almost all companies are still using the same single page size they have been using since we began dealing with this issue: 4KB.

### 7.4.1 Accessing a 4KB Page

Figure **??** shows the access to 4KB pages using the x86 32 bit address.

### 7.4.2 Accessing a 4KB page

The 32 bit virtual address is broken into three parts, bits[31:22], bits[21:12], and bits[11:0]. Each process has its own virtual address space. Each process has as part of its context, a CR3 register, which contains the starting address of

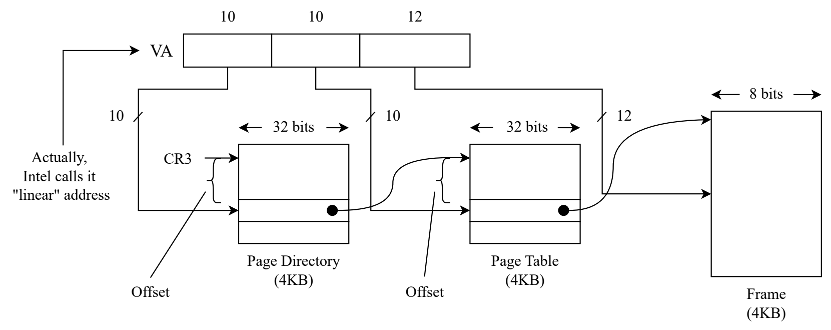

Figure 7.11: Accessing a 4KB page

a 4KB page called the Page Directory. Each process has its own Page Directory. The Page Directory contains $2^{10}$ 4 byte descriptors, addressable by bits[31:22] of the VA, and pointing to the starting address of a page table. Each page table contains $2^{10}$ PTEs for $2^{10}$ frames. Bits[21:12] of the VA identify the relevant PTE and corresponding 4KB frame. Bits [11:0] of the VA identify the byte offset of the frame. Like the VAX virtual memory, not all pages have to be resident at the same time.

### 7.4.3 Intel and Arm Have Moved to Three Page Sizes

Intel and Arm have adopted three page sizes into their ISAs. In the case of Intel, the page sizes are 4KB, 2MB, and 1GB. In the case of Arm, the page sizes are 4KB, 64KB, and 1MB. A simple example of a benefit of larger page sizes is the difference of having one PTE for a 1GB page vs $2^{19}$ PTEs for $2^{19}$ 4KB pages that span the same amount of space. Among other things, having a large number of PTEs adversely affects the effectiveness of the TLB.

### 7.4.4 Intel Page Sizes: 4KB, 2MB, and 1GB

Figure 7.12 shows the addressing structure for Intel's three page sizes. The scheme uses the low 48 bits of the VA. The bits are broken down into several fields: bits[47:39], bits[38:30], bits[29:21], bits[20:12], and bits[11:0] Each table consists of 4K bytes, made up of a frame of memory consisting of 512 8 byte descriptors. Like the old scheme, each process has its own virtual address space and a CR3 register that points to the starting address of the Page Directory. Bits[47:39] index into a page table.

From this point on, entries in a page table either identify a larger page size or operate much like the earlier scheme shown in Figure 7.11. For example, each of the 512 entries determined by bits[38:30] of the VA provide a choice between

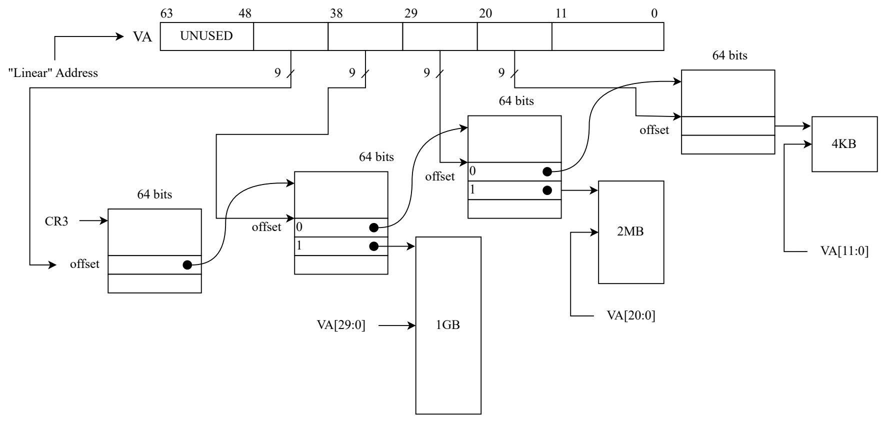

Figure 7.12: Intel Page Sizes: 4KB, 2MB, and 1GB

choosing a 1GB page with bits[29:0] providing the offset of the 1GB page, or following the earlier model of breaking units down further to smaller size.

In the same way, a page table that is indexed by bits[29:21] of the VA provides a choice between a 2MB page with offset supplied by bits[29:21] of the VA, or continuing to follow the earlier scheme of breaking down the address bits further into still smaller page sizes. To take advantage of a TLB, Intel provides separate TLBs for each page size.

### 7.4.5 Recent Research, Faruk Guvenilir

Faruk Guvenilir, a PhD graduate of ECE in 2019, came up with an elegant scheme of allowing page sizes of $2^k$ for all k from $2^{12}$ to $2^{20}$, with the added benefit of being able to design a single TLB that handles a mixture of all page sizes. The title of the paper is Tailored Page Sizes, and is published in IEEE/ACM Annual International Symposium on Computer Architecture, June 2020.