# 19

# Paging: Smaller Tables

We now tackle the second problem that paging introduces: page tables are too big and thus consume too much memory. Let's start out with a linear page table. As you might recall[1], linear page tables get pretty big. Assume again a 32-bit address space ($2^{32}$ bytes), with 4KB ($2^{12}$ byte) pages and a 4-byte page-table entry. An address space thus has roughly 1 million virtual pages in it ($\frac{2^{32}}{2^{12}}$); multiply by the page-table size and you see that our page table is 4MB in size. Recall also: we usually have one page table *for every process* in the system! Thus, with a hundred active processes (not uncommon on a modern system), we will be allocating hundreds of megabytes of memory just for page tables! Thus, we are in search of some techniques to reduce this heavy burden. There are a lot of them, so let's get going. But not before our crux:

CRUX: HOW TO MAKE PAGE TABLES SMALLER?
Simple array-based page tables (usually called linear page tables) are too big, taking up far too much memory on typical systems. How can we make page tables smaller? What are the key ideas? What inefficiences arise as a result of these new data structures?

[1] Or indeed, you might not; this paging thing is getting out of control, no? That said, always make sure you understand the *problem* you are solving before moving onto the solution; indeed, if you understand the problem, you can often derive the solution yourself. Here, the problem should be clear: simple page tables are too big.

ASIDE: **MULTIPLE PAGE SIZES**

As an aside, do note that many architectures (e.g., MIPS, SPARC, x86-64) now support multiple page sizes. Usually, a small (4KB or 8KB) page size is used. However, if a "smart" application requests it, a single large page (e.g., of size 4MB) can be used for a specific portion of the address space, enabling such applications to place a frequently-used (and large) data structure in such a space while consuming only a single TLB entry. This type of large page usage is common in database management systems and other high-end commercial applications. The main reason for multiple page sizes is not to save page table space, however; it is to reduce pressure on the TLB, enabling a program to access more of its address space without suffering from too many TLB misses. However, as researchers have shown [N+02], using multiple page sizes makes the OS virtual memory manager notably more complex, and thus large pages are sometimes most easily used simply by exporting a new interface to applications to request large pages directly.

## 19.1 Simple Solution: Bigger Pages

We could reduce the size of the page table in one simple way: use bigger pages. Take our 32-bit address space again, but this time assume 16KB pages. We would thus have an 18-bit VPN plus a 14-bit offset. Assuming the same size for each PTE (4 bytes), we now have $2^{18}$ entries in our linear page table and thus a total size of 1MB per page table, a factor of four reduction in size of the page table (not surprisingly, the reduction exactly mirrors the factor of four increase in page size).

The major problem with this approach, however, is that big pages lead to waste *within* each page, a problem known as **internal fragmentation** (as the waste is **internal** to the unit of allocation). Applications thus end up allocating pages but only using little bits and pieces of each, and memory quickly fills up with these overly-large pages. Thus, most systems use relatively small page sizes in the common case: 4KB (as in x86) or 8KB (as in SPARCv9). Our problem will not be solved so simply, alas.

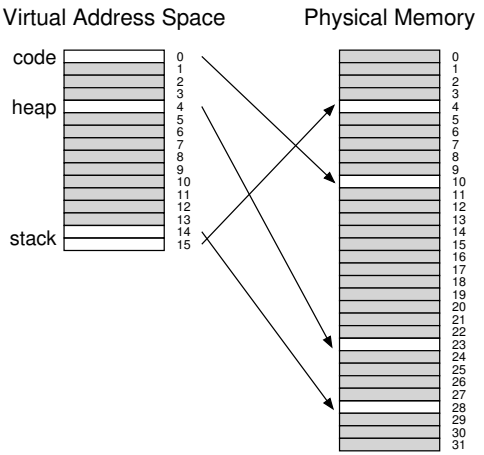



Figure 19.1: A 16-KB Address Space With 1-KB Pages

## 19.2 Hybrid Approach: Paging and Segments

Whenever you have two reasonable but different approaches to something in life, you should always examine the combination of the two to see if you can obtain the best of both worlds. We call such a combination a **hybrid**. For example, why eat just chocolate or plain peanut butter when you can instead combine the two in a lovely hybrid known as the Reese's Peanut Butter Cup [M28]?

Years ago, the creators of Multics, and in particular Jack Dennis, chanced upon such an idea in the construction of the Multics virtual memory system [M07]. Specifically, Dennis had the idea of combining paging and segmentation in order to reduce the memory overhead of page tables. We can see why this might work by examining a typical linear page table in more detail. Assume we have an address space in which the used portions of the heap and stack are small. For the example, we use a tiny 16KB address space with 1KB pages (Figure 19.1); the page table for this address space is in Table 19.1.

This example assumes the single code page (VPN 0) is mapped to physical page 10, the single heap page (VPN 4) to physical page 23, and the two stack pages at the other end of the address space (VPNs

| PFN | valid | prot | present | dirty |
|---|---|---|---|---|
| 10 | 1 | r-x | 1 | 0 |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| 23 | 1 | rw- | 1 | 1 |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| - | 0 | — | - | - |
| 28 | 1 | rw- | 1 | 1 |
| 4 | 1 | rw- | 1 | 1 |

Table 19.1: A Page Table For 16-KB Address Space

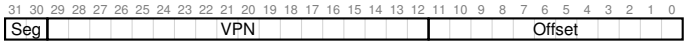
14 and 15) are mapped to physical pages 28 and 4, respectively. As you can see from the picture, *most* of the page table is unused, full of **invalid** entries. What a waste! And this is for a tiny 16KB address space. Imagine the page table of a 32-bit address space and all the potential wasted space in there! Actually, don't imagine such a thing; it's far too gruesome.

Thus, our hybrid approach: instead of having a single page table for the entire address space of the process, why not have one per logical segment? In this example, we might thus have three page tables, one for the code, heap, and stack parts of the address space.

Now, remember with segmentation, we had a **base** register that told us where each segment lived in physical memory, and a **bound** or **limit** register that told us the size of said segment. In our hybrid, we still have those structures in the MMU; here, we use the base not to point to the segment itself but rather to hold the *physical address of the page table* of that segment. The bounds register is used to indicate the end of the page table (i.e., how many valid pages it has).

Let's do a simple example to clarify. Assume a 32-bit virtual address space with 4KB pages, and an address space split into four segments. We'll only use three segments for this example: one for code, one for heap, and one for stack.

To determine which segment an address refers to, we'll use the top two bits of the address space. Let's assume 00 is the unused segment, with 01 for code, 10 for the heap, and 11 for the stack. Thus, a virtual address looks like this:




In the hardware, assume that there are thus three base/bounds pairs, one each for code, heap, and stack. When a process is running, the base register for each of these segments contains the physical address of a linear page table for that segment; thus, each process in the system now has *three* page tables associated with it. On a context switch, these registers must be changed to reflect the location of the page tables of the newly-running process.

On a TLB miss (assuming a hardware-managed TLB), the hardware uses the segment bits (`SN`) to determine which base and bounds pair to use. The hardware then takes the physical address therein and combines it with the VPN as follows to form the address of the page table entry (PTE):

```
SN           = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN          = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

This sequence should look familiar; it is virtually identical to what we saw before with linear page tables. The only difference, of course, is the use of one of three segment base registers instead of the single page table base register.

The critical difference in our hybrid scheme is the presence of a bounds register per segment; each bounds register holds the value of the maximum valid page in the segment. For example, if the code segment is using its first three pages (0, 1, and 2), the code segment page table will only have three entries allocated to it and the bounds register will be set to 3; memory accesses beyond the end of the segment will generate an exception and likely lead to the termination of the process. In this manner, our hybrid approach realizes a significant memory savings compared to the linear page table; unallocated pages between the stack and the heap no longer take up space in a page table (just to mark them as not valid).

However, as you might notice, this approach is not without problems. First, it still requires us to use segmentation; as we discussed

TIP: USE HYBRIDS
When you have two good and seemingly opposing ideas, you should always see if you can combine them into a **hybrid** that manages to achieve the best of both worlds. Hybrid corn species, for example, are known to be more robust than any naturally-occurring species. Of course, not all hybrids are a good idea; see the Zeedonk (or Zonkey), which is a cross of a Zebra and a Donkey. If you don't believe such a creature exists, look it up, and prepare to be amazed.

before, segmentation is not quite as flexible as we would like. Second, this hybrid causes external fragmentation to arise again. While most of memory is managed in page-sized units, page tables now can be of arbitrary size (in multiples of PTEs). Thus, finding free space for them in memory is somewhat complicated. For these reasons, people continued to look for better approaches to implementing smaller page tables.

## 19.3 Multi-level Page Tables

A different approach doesn't rely on segmentation but attacks the same problem: how to get rid of all those invalid regions in the page table instead of keeping them all in memory? We call this approach a **multi-level page table**, as it turns the linear page table into something like a tree. This approach is so effective that many modern systems employ it (e.g., x86 [BOH10]). We now describe this approach in detail.

The basic idea behind a multi-level page table is simple. First, chop up the page table into page-sized units; then, if an entire page of page-table entries (PTEs) is invalid, don't allocate that page of the page table at all. To track whether a page of the page table is valid (and if valid, where it is in memory), use a new structure, called the **page directory**. The page directory thus either can be used to tell you where a page of the page table is, or that the entire page of the page table contains no valid pages.

Figure 19.2 shows an example. On the left of the figure is the classic linear page table; even though most of the middle regions of the address space are not valid, we still have to have page-table space allocated for those regions (i.e., the middle two pages of the page

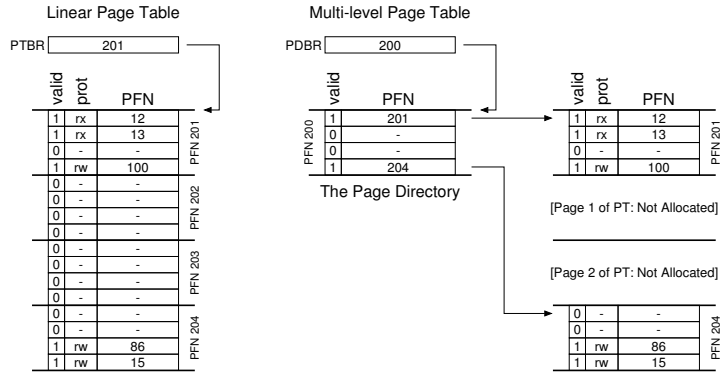



Figure 19.2: Linear (Left) And Multi-Level (Right) Page Tables

table). On the right is a multi-level page table. The page directory marks just two pages of the page table as valid (the first and last); thus, just those two pages of the page table reside in memory. And thus you can see one way to visualize what a multi-level table is doing: it just makes parts of the linear page table disappear (freeing those frames for other uses), and tracks which pages of the page table are allocated with the page directory.

The page directory, in a simple two-level table, contains one entry per page of the page table. It consists of a number of **page directory entries** (**PDE**). A PDE has a **valid bit** and a **page frame number** (PFN), similar to a PTE. However, as hinted at above, the meaning of this valid bit is slightly different: if the PDE entry is valid, it means that at least one of the pages of the page table that the entry points to (via the PFN) is valid (i.e., in at least one PTE on that page pointed to by this PDE, the valid bit in that PTE is set to 1). If the PDE entry is not valid, the rest of the PDE is not defined.

Multi-level page tables have some obvious advantages over approaches we've seen thus far. First, and perhaps most obviously, the multi-level table only allocates page-table space in proportion to the amount of address space you are using; thus it is generally compact and supports sparse address spaces.

TIP: UNDERSTAND TIME-SPACE TRADE-OFFS
When building a data structure, one should always consider **time-space trade-offs** in its construction. Usually, if you wish to make access to a particular data structure faster, you will have to pay a space-usage penalty for the structure.

Second, if carefully constructed, each portion of the page table fits neatly within a page, making it easier to manage memory; the OS can simply grab the next free page when it needs to allocate or grow a page table. Contrast this to a simple (non-paged) linear page table[2], which is just an array of PTEs indexed by VPN; with such a structure, the entire linear page table must reside contiguously in physical memory. For a large page table (say 4MB), finding such a large chunk of unused contiguous free physical memory can be quite a challenge. With a multi-level structure, we add a **level of indirection** through use of the page directory, which points to pieces of the page table; that indirection allows us to place page-table pages wherever we would like in physical memory.

It should be noted that there is a cost to multi-level tables; on a TLB miss, two loads from memory will be required to get the right translation information from the page table (one for the page directory, and one for the PTE itself), in contrast to just one load with a linear page table. Thus, the multi-level table is a small example of a **time-space trade-off**. We wanted smaller tables (and got them), but not for free; although in the common case (TLB hit), performance is obviously identical, a TLB miss suffers from a higher cost with this smaller table.

Another obvious negative is *complexity*. Whether it is the hardware or OS handling the page-table lookup (on a TLB miss), doing so is undoubtedly more involved than a simple linear page-table lookup. Often we are willing to increase complexity in order to improve performance or reduce overheads; in the case of a multi-level table, we make page-table lookups more complicated in order to save valuable memory.

[2]We are making some assumptions here, in particular that all page tables reside in their entirety in physical memory (i.e., they are not swapped to disk); we'll soon relax this assumption.

TIP: BE WARY OF COMPLEXITY

System designers should be wary of adding complexity into their system. What a good systems builder does is implement the least complex system that achieves the task at hand. For example, if disk space is abundant, you shouldn't design a file system that works hard to use as few bytes as possible; similarly, if processors are fast, it is better to write a clean and understandable module within the OS than perhaps the most CPU-optimized, hand-assembled code for the task at hand. Be wary of needless complexity, in prematurely-optimized code or other forms; such approaches make systems harder to understand, maintain, and debug. As Antoine de Saint-Exupery famously wrote: "Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away." What he didn't write: "It's a lot easier to say something about perfection than to actually achieve it."

## A Detailed Multi-Level Example

To understand the idea behind multi-level page tables better, let's do an example. Imagine a small address space of size 16 KB, with 64-byte pages. Thus, we have a 14-bit virtual address space, with 8 bits for the VPN and 6 bits for the offset. A linear page table would have $2^8$ (256) entries, even if only a small portion of the address space is in use. Figure 19.3 presents one example of such an address space.

In this example, virtual pages 0 and 1 are for code, virtual pages 4 and 5 for the heap, and virtual pages 254 and 255 for the stack; the rest of the pages of the address space are unused.

To build a two-level page table for this address space, we start with our full linear page table and break it up into page-sized units. Recall our full table (in this example) has 256 entries; assume each PTE is 4 bytes in size. Thus, our page table is 1KB (256 $\times$ 4 bytes) in size. Given that we have 64-byte pages, the 1-KB page table can be divided into 16 64-byte pages; each page can hold 16 PTEs.

What we need to understand now is how to take a VPN and use it to index first into the page directory and then into the page of the page table. Remember that each is an array of entries; thus, all we need to figure out is how to construct the index for each from pieces of the VPN.

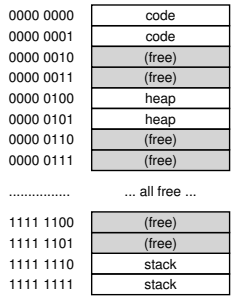
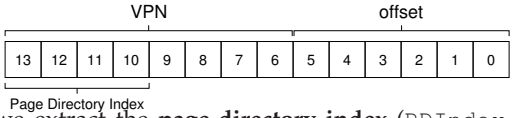


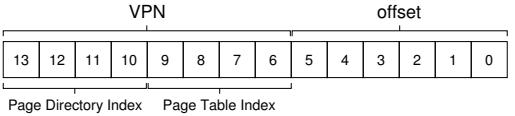

Figure 19.3: A 16-KB Address Space With 64-byte Pages

Let's first index into the page directory. Our page table in this example is small: 256 entries, spread across 16 pages. The page directory needs one entry per page of the page table; thus, it has 16 entries. As a result, we need four bits of the VPN to index into the directory; we use the top four bits of the VPN, as follows:




Once we extract the **page-directory index** (`PDIndex` for short) from the VPN, we can use it to find the address of the page-directory entry (PDE) with a simple calculation: `PDEAddr = PageDirBase + (PDIndex * sizeof(PDE))`. This results in our page directory, which we now examine to make further progress in our translation.

If the page-directory entry is marked invalid, we know that the access is invalid, and thus raise an exception. If, however, the PDE is valid, we have more work to do. Specifically, we now have to fetch the page-table entry (PTE) from the page of the page table pointed to by this page-directory entry. To find this PTE, we have to index into the portion of the page table using the remaining bits of the VPN:



| pointer to page of PT | valid? |
|---|---|
| 100 | 1 |
| —— | 0 |
| —— | 0 |
| —— | 0 |
| —— | 0 |
| —— | 0 |
| —— | 0 |
| —— | 0 |
| —— | 0 |
| —— | 0 |
| —— | 0 |
| —— | 0 |
| —— | 0 |
| —— | 0 |
| —— | 0 |
| 101 | 1 |

Table 19.2: A Page Directory

This **page-table index** (`PTIndex` for short) can then be used to index into the page table itself, giving us the address of our PTE:

```
PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
```

Note that the page-frame number (PFN) obtained from the page-directory entry must be left-shifted into place before combining it with the page-table index to form the address of the PTE.

To see if this all makes sense, we'll now fill in a multi-level page table with some actual values, and translate a single virtual address. Let's begin with the **page directory** for this example (Table 19.2).

In the figure, you can see that each page directory entry (PDE) describes something about a page of the page table for the address space. In this example, we have two valid regions in the address space (at the beginning and end), and a number of invalid mappings in-between.

In physical page 100 (the physical frame number of the 0th page of the page table), we have the first page of 16 page table entries for the first 16 VPNs in the address space. See Table 19.3 for the contents of this portion of the page table.

This page of the page table contains the mappings for the first 16 VPNs; in our example, VPNs 0 and 1 are valid (the code segment), as

| PFN | valid | prot |
|---|---|---|
| 10 | 1 | r-x |
| 23 | 1 | r-x |
| – | 0 | — |
| – | 0 | — |
| 80 | 1 | rw- |
| 59 | 1 | rw- |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |

Table 19.3: The First Page Of The Page Table (PFN 100)

are 4 and 5 (the heap). Thus, the table has mapping information for each of those pages. The rest of the entries are marked invalid.

The other valid page of page table is found inside PFN 101. This page contains mappings for the last 16 VPNs of the address space; see Table 19.4 for details.

In the example, VPNs 254 and 255 (the stack) have valid mappings. Hopefully, what we can see from this example is how much space savings are possible with a multi-level indexed structure. In this example, instead of allocating the full *sixteen* pages for a linear page table, we allocate only *three*: one for the page directory, and two for the chunks of the page table that have valid mappings. The savings for large (32-bit or 64-bit) address spaces could obviously be much greater.

Finally, let's use this information in order to perform a translation. Here is an address that refers to the 0th byte of VPN 254: `0x3F80`, or `11 1111 1000 0000` in binary.

Recall that we will use the top 4 bits of the VPN to index into the page directory. Thus, `1111` will choose the last (15th, if you start at the 0th) entry of the page directory above. This points us to a valid page of the page table located at address `101`. We then use the next 4 bits of the VPN (`1110`) to index into that page of the page table and find the desired PTE. 1110 is the next-to-last (14th) entry on the page, and tells us that page 254 of our virtual address space is mapped

| PFN | valid | prot |
|---|---|---|
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| – | 0 | — |
| 55 | 1 | rw- |
| 45 | 1 | rw- |

Table 19.4: The Last Page Of The Page Table (PFN 101)

at physical page 55. By concatenating PFN=55 (or hex `0x37`) with offset=000000, we can thus form our desired physical address and issue the request to the memory system: `PhysAddr = (PTE.PFN << SHIFT) + offset = 00 1101 1100 0000 = 0x0DC0`.

You should now have some idea of how to construct a two-level page table, using a page directory which points to pages of the page table. Unfortunately, however, our work is not done. As we'll now discuss, sometimes two levels of page table is not enough!

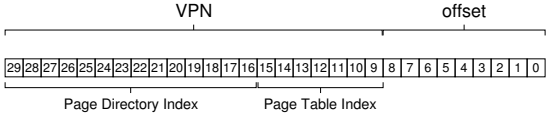
## More Than Two Levels

In our example thus far, we've assumed that multi-level page tables only have two levels: a page directory and then pieces of the page table. In some cases, a deeper tree is possible (and indeed, needed).

Let's take a simple example and use it to show why a deeper multi-level table can be useful. In this example, assume we have a 30-bit virtual address space, and a small (512 byte) page. Thus our virtual address has a 21-bit virtual page number component and a 9-bit offset.

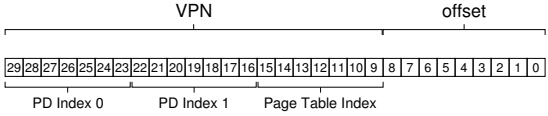
Remember our goal in constructing a multi-level page table: to make each piece of the page table fit within a single page. Thus far, we've only considered the page table itself; however, what if the page directory gets too big?

To determine how many levels are needed in a multi-level table to make all pieces of the page table fit within a page, we start by determining how many page-table entries fit within a page. Given our page size of 512 bytes, and assuming a PTE size of 4 bytes, you should see that you can fit 128 PTEs on a single page. When we index into a page of the page table, we can thus conclude we'll need the least significant 7 bits ($log_2 128$) of the VPN as an index:




What you also might notice from the diagram above is how many bits are left into the (large) page directory: 14. If our page directory has $2^{14}$ entries, it spans not one page but 128, and thus our goal of making every piece of the multi-level page table fit into a page vanishes.

To remedy this problem, we build a further level of the tree, by splitting the page directory itself into multiple pages, and then adding another page directory on top of that, to point to the pages of the page directory. We can thus split up our virtual address as follows:




Now, when indexing the upper-level page directory, we use the very top bits of the virtual address (`PD Index 0` in the diagram); this index can be used to fetch the page-directory entry from the top-level page directory. If valid, the second level of the page directory is consulted by combining the physical frame number from the top-level PDE and the next part of the VPN (`PD Index 1`). Finally, if valid, the PTE address can be formed by using the page-table index combined with the address from the second-level PDE. Whew! That's a lot of work. And all just to look something up in a multi-level table.

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                  // TLB Miss
11     // first, get page directory entry
12     PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13     PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14     PDE     = AccessMemory(PDEAddr)
15     if (PDE.Valid == False)
16         RaiseException(SEGMENTATION_FAULT)
17     else
18         // PDE is valid: now fetch PTE from page table
19         PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20         PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
21         PTE     = AccessMemory(PTEAddr)
22         if (PTE.Valid == False)
23             RaiseException(SEGMENTATION_FAULT)
24         else if (CanAccess(PTE.ProtectBits) == False)
25             RaiseException(PROTECTION_FAULT)
26         else
27             TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28             RetryInstruction()
```

Figure 19.4: Multi-level Page Table Control Flow

### The Translation Process: Remember the TLB

To summarize the entire process of address translation using a two-level page table, we once again present the control flow in algorithmic form (Figure 19.4). The figure shows what happens in hardware (assuming a hardware-managed TLB) upon *every* memory reference.

As you can see from the figure, before any of the complicated multi-level page table access occurs, the hardware first checks the TLB; upon a hit, the physical address is formed directly *without* accessing the page table at all, as before. Only upon a TLB miss does the hardware need to perform the full multi-level lookup. On this path, you can see the cost of our traditional two-level page table: two additional memory accesses to look up a valid translation.

## 19.4 Inverted Page Tables

An even more extreme space savings in the world of page tables is found with **inverted page tables**. Here, instead of having many page tables (one per process of the system), we keep a single page table that has an entry for each *physical page* of the system. The entry tells us which process is using this page, and which virtual page of that process maps to this physical page.

Finding the correct entry is now a matter of searching through this data structure. A linear scan would be expensive, and thus a hash table is often built over the base structure to speed lookups. The PowerPC is one example of such an architecture [JM98].

More generally, inverted page tables illustrate what we've said from the beginning: page tables are just data structures. You can do lots of crazy things with data structures, making them smaller or bigger, making them slower or faster. Multi-level and inverted page tables are just two examples of the many things one could do.

## 19.5 Swapping the Page Tables to Disk

Finally, we discuss the relaxation of one final assumption. Thus far, we have assumed that page tables reside in kernel-owned physical memory. Even with our many tricks to reduce the size of page tables, it is still possible, however, that they may be too big to fit into memory all at once. Thus, some systems place such page tables in **kernel virtual memory**, thereby allowing the system to **swap** some of these page tables to disk when memory pressure gets a little tight. We'll talk more about this in future notes, once we understand how to move pages in and out of memory in more detail.

## 19.6 Summary

We have now seen how real page tables are built; not necessarily just as linear arrays but as more complex data structures. The tradeoffs such tables present are in time and space – the bigger the table, the faster a TLB miss can be serviced, as well as the converse – and thus the right choice of structure depends strongly on the constraints of the given environment.

In a memory-constrained system (like many older systems), small structures make sense; in a system with a reasonable amount of memory and with workloads that actively use a large number of pages, a bigger table that speeds up TLB misses might be the right choice. With software-managed TLBs, the entire space of data structures opens up to the delight of the operating system innovator (hint: that's you). What new structures can you come up with? What problems do they solve? Think of these questions as you fall asleep, and dream the big dreams that only OS developers can dream.

## References

[BOH10] "Computer Systems: A Programmer's Perspective"
Randal E. Bryant and David R. O'Hallaron
Addison-Wesley, 2010
*We have yet to find a good first reference to the multi-level page table. However, this great textbook by Bryant and O'Halloran dives into the details of x86, which at least is an early system that used such structures. It's also just a great book to have.*

[JM98] "Virtual Memory: Issues of Implementation"
Bruce Jacob and Trevor Mudge
IEEE Computer, June 1998
*An excellent survey of a number of different systems and their approach to virtualizing memory. Plenty of details on x86, PowerPC, MIPS, and other architectures.*

[LL82] "Virtual Memory Management in the VAX/VMS Operating System"
Hank Levy and P. Lipman
IEEE Computer, Vol. 15, No. 3, March 1982
*A terrific paper about a real virtual memory manager in a classic operating system, VMS. So terrific, in fact, that we'll use it to review everything we've learned about virtual memory thus far a few chapters from now.*

[M28] "Reese's Peanut Butter Cups"
Mars Candy Corporation.
*Apparently these fine confections were invented in 1928 by Harry Burnett Reese, a former dairy farmer and shipping foreman for one Milton S. Hershey. At least, that is what it says on Wikipedia. If true, Hershey and Reese probably hated each other's guts, as any two chocolate barons should.*

[N+02] "Practical, Transparent Operating System Support for Superpages"
Juan Navarro, Sitaram Iyer, Peter Druschel, Alan Cox
OSDI '02, Boston, Massachusetts, October 2002
*A nice paper showing all the details you have to get right to incorporate large pages, or **superpages**, into a modern OS. Not as easy as you might think, alas.*

[M07] "Multics: History"
Available: `http://www.multicians.org/history.html`
*This amazing web site provides a huge amount of history on the Multics system, certainly one of the most influential systems in OS history. The quote from therein: "Jack Dennis of MIT contributed influential architectural ideas to the beginning of Multics, especially the idea of combining paging and segmentation." (from Section 1.2.1)*

## Homework

This fun little homework tests if you understand how a multi-level page table works. And yes, there is some debate over the use of the term "fun" in the previous sentence. The program is called: `paging-multilevel-translate.py`.

Some basic assumptions:

- The page size is an unrealistically-small 32 bytes
- The virtual address space for the process in question (assume there is only one) is 1024 pages, or 32 KB
- physical memory consists of 128 pages

Thus, a virtual address needs 15 bits (5 for the offset, 10 for the VPN). A physical address requires 12 bits (5 offset, 7 for the PFN).

The system assumes a multi-level page table. Thus, the upper five bits of a virtual address are used to index into a page directory; the page directory entry (PDE), if valid, points to a page of the page table. Each page table page holds 32 page-table entries (PTEs). Each PTE, if valid, holds the desired translation (physical frame number, or PFN) of the virtual page in question.

The format of a PTE is thus:

```
VALID | PFN6 ... PFN0
```

and is thus 8 bits or 1 byte.

The format of a PDE is essentially identical:

```
VALID | PT6 ... PT0
```

You are given two pieces of information to begin with.

First, you are given the value of the page directory base register (PDBR), which tells you which page the page directory is located upon.

Second, you are given a complete dump of each page of memory. A page dump looks like this:

```
page 0: 08 00 01 15 11 1d 1d 1c 01 17 15 14 16 1b 13 0b ...
page 1: 19 05 1e 13 02 16 1e 0c 15 09 06 16 00 19 10 03 ...
page 2: 1d 07 11 1b 12 05 07 1e 09 1a 18 17 16 18 1a 01 ...
...
```

which shows the 32 bytes found on pages 0, 1, 2, and so forth. The first byte (0th byte) on page 0 has the value 0x08, the second is 0x00, the third 0x01, and so forth.

You are then given a list of virtual addresses to translate.

Use the PDBR to find the relevant page table entries for this virtual page. Then find if it is valid. If so, use the translation to form a final physical address. Using this address, you can find the VALUE that the memory reference is looking for.

Of course, the virtual address may not be valid and thus generate a fault.

Some useful options:

```
-s SEED, --seed=SEED       the random seed
-n NUM, --addresses=NUM    number of virtual addresses to generate
-c, --solve                compute answers for me
```

Change the seed to get different problems, as always.

Change the number of virtual addresses generated to do more translations for a given memory dump.

Use -c (or –solve) to show the solutions.

## Questions

- With a linear page table, you need a single register to locate the page table, assuming that hardware does the lookup upon a TLB miss. How many registers do you need to locate a two-level page table? A three-level table?
- Use the simulator to perform translations given random seeds 0, 1, and 2, and check your answers using the `-c` flag. How many memory references are needed to perform each lookup?
- Given your understanding of how cache memory works, how do you think memory references to the page table will behave in the cache? Will they lead to lots of cache hits (and thus fast accesses?) Or lots of misses (and thus slow accesses)?