

Paging: Smaller Tables

We now tackle the second problem that paging introduces: page tables are too big and thus consume too much memory. We start with out a linear page table. As you might recall (or might not, this is getting pretty detailed), linear page tables get pretty big. Assume again a 32-bit address space, with 4KB pages. As above, we see that this leads to a 4MB page table. Recall also: we have one page table **per process**! Thus, with a hundred active processes (not uncommon on a modern desktop machine), we will be allocating 400MB 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.

14.1 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 (which exactly mirrors the factor of four increase in size of each page).

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.

ASIDE: MULTIPLE PAGE SIZES

As an aside, do note that many architectures (e.g., MIPS, SPARC) 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 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. Thus, the main reason for multiple page sizes is to reduce pressure on the TLB.

14.2 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 it can obtain the best of both worlds. We call such a combination a **hybrid**. For example, why eat just chocolate or just peanut-butter when you can combine the two in a lovely hybrid known as the Reese’s Peanut Butter Cup? [1]

DESIGN TIP: 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.

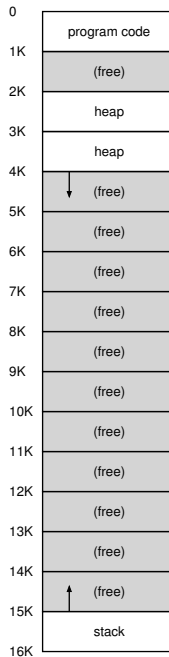


Figure 14.1: A 16-KB address space with 1-KB pages

System designers similarly 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. Let's say we have a large address space, but the currently used heap and stack are pretty small. Specifically, we use a tiny 16KB address space with 1KB pages (Figure 14.2); the linear page table for this address space

might look like what you see in Table 14.2.

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	---	-	-
90	1	rw-	1	1
91	1	rw-	1	1

Table 14.1: A Page Table for 16-KB Address Space

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 14 and 15) are mapped to physical pages 90 and 91 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 wasted space when most of the address space goes unused. Actually, don't imagine it; it's too gruesome.

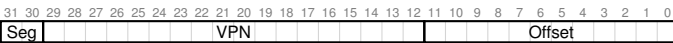
Thus, our hybrid approach. Instead of having a single page table for the entire address space of the process, why not have one per logical segment? In this example, we might thus have three page tables, one for each the code, heap, and stack portions of the address space.

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 similarly 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 to determine which base/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):

```
VPN          = (VirtualAddress & VPN_MASK) >> VPN_SHIFT;
AddressOfPTE = SegmentBaseRegister + (VPN * sizeof(PTE));
```

This equation 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).

14.3 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). We now describe this approach in more detail.

Let us 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 14.2 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.

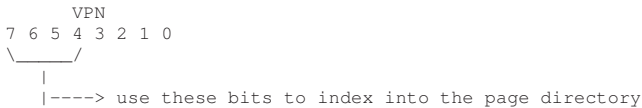
To build a two-level page table for this address space, we start with our full linear page table and break it up into page-sized units. Recall our full table (in this example) has 256 entries; let us assume each PTE is 4 bytes in size. Thus, we have a total page table of size 1 KB. Given that we have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

0000 0000	code
0000 0001	code
0000 0010	(free)
0000 0011	(free)
0000 0100	heap
0000 0101	heap
0000 0110	(free)
0000 0111	(free)
.....	... all free ...
1111 1100	(free)
1111 1101	(free)
1111 1110	stack
1111 1111	stack

Figure 14.2: A 16-KB address space with 64-byte pages

1-KB page table fits into 16 64-byte pages. Each page can hold 16 PTEs.

What we need now is a structure that can point to each of the page of the page table. We call this structure the **page directory**; it is a simple linear array with one entry per page of the linear page table. Because we have 16 pages of page table, we thus need 4 bits. We use the top four bits of the VPN to index into this directory:



The beauty of the page directory arises when there is a large region of the page table that has all invalid entries. Instead of marking them all invalid in the page table itself, we instead can mark the page directory pointer that points to this part of the page table invalid; thus, that portion of the page table does not have to be allocated any memory.

The **page directory** for the example address space above can be seen in Table 14.2.

pointer to page of PT	valid?
PhysAddr1	1
---	0
---	0
---	0
---	0
---	0
---	0
---	0
---	0
---	0
---	0
---	0
---	0
---	0
---	0
PhysAddr2	1

Table 14.2: A Page Directory

Each **page directory entry (PDE)** describes something about a chunk of the page table for the address space. In this example, we have two valid chunks of the address space (at the beginning and end), and a number of invalid mappings in-between.

At `PhysAddr1` (some physical address, or, if the page directory is page-aligned, just the physical frame number), we have the first chunk of 16 page table entries for the first 16 VPNs in the address space. It would look something like what you see in Figure 14.2.

This chunk 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 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.

A similar chunk of page table is found at `PhysAddr2`; this is for the last 16 VPNs of the address space (Figure 14.4).

Thus, for VPNs 254 and 255 (the stack), we have valid map-

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	---
-	0	---
-	0	---

Table 14.3: A Chunk of the Page Table

PFN	valid	prot
-	0	---
-	0	---
-	0	---
-	0	---
-	0	---
-	0	---
-	0	---
-	0	---
-	0	---
-	0	---
-	0	---
-	0	---
-	0	---
-	0	---
-	0	---
55	1	rw-
45	1	rw-

Table 14.4: Another Chunk of the Page Table

pings. Hopefully, what we can see from this example is how much space savings are possible with a multi-level indexed struc-

ture. In this example, instead of allocating 16 pages for a linear page table, we allocate only 3: one for the page directory, and two for the chunks of the page table that have valid mappings. The savings for large address spaces could obviously be much bigger.

Note that we haven't quite explained the entire picture of how to find the right page table entry for a given VPN. Let's take VPN 254 (to a stack page) as an example. Recall that we have a 14-bit virtual address space. Thus, an address that refers to VPN 254 might look like this:

```

VPN      offset
11111110 000000

```

Recall that we will use the top 4 bits of the VPN to index into the page directory. Thus, 1111 will choose the last entry of the page directory above. This points us to a valid chunk of the page table located at address `PhysAddr2`. We then use the second 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 entry on the page, and tells us that page 254 of our virtual address space is mapped at physical page 45. By concatenating `PFN=45` with `offset=000000`, we can thus form our desired physical address and issue the request to the memory system.

Thus, we can think of this process as splitting the VPN into two components, `VPN_PageDir` and `VPN_ChunkIndex`. The first portion (`VPN_PageDir`) is used to index into the page directory itself; the resulting page directory entry (PDE) then tells us either where to find the relevant chunk of the page table or that the region of the address space is not valid. If valid, the second portion of the VPN (`VPN_ChunkIndex`) is then used to find the desired PTE.

Assuming some kind of **page-directory base register**, which holds the address of the page directory for the current process, we can now summarize the process mathematically. The address of the page directory entry (PDE) is calculated as follows:

```
AddressOfPDE = PageDirectoryBase + (VPN_PageDir * sizeof(PDE))
```

If the PDE entry is valid, we can extract a physical address from it (`PhysAddr`) and thus calculate the address of the page table entry (PTE):

```
AddressOfPTE = PhysAddr + (VPN_ChunkIndex * sizeof(PTE))
```

Now, finally, we can use this address to access the page table and fetch the desired translation; a lot of work just to translate an address!

We thus may also see how to decide how to split the bits of the VPN into `VPN_PageDir` and `VPN_ChunkIndex`. Given a page size P , and a PTE size S , we know that a page can contain $\frac{P}{S}$ PTEs. Thus, `VPN_ChunkIndex` is chosen such that there are the right number of bits to choose $\frac{P}{S}$ PTEs ($\log_2 \frac{P}{S}$). The example above has $P = 64$ and a PTE size of 4, and thus $\frac{P}{S}$ is 16. Thus, the number of bits in the `VPN_ChunkIndex` is $\log_2(16)$ or 4. The number of bits in `VPN_PageDir` is thus the total number of bits in the VPN minus the number of bits in `VPN_ChunkIndex`.

The other benefit of multi-level structures is that they allow some freedom in the placement of pieces of the page table. So far, we have assumed that page tables live in physical memory (an assumption we will relax below); if that is the case, and we have a linear page table (an array of PTEs indexed by VPN), then 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 in the page directory which points to pieces of the page table; that indirection allows us to place those 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 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),

DESIGN TIP: 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 some penalty in terms of using more memory for the structure.

performance is obviously identical, a TLB miss suffers from a higher cost with this smaller table.

One last note: more than two levels are possible (e.g., x86 has a three-level mode). These make for even sparser trees of page tables, potentially saving even more space while again increasing the cost of servicing a TLB miss. More importantly, it allows pieces of the page directory to each fit within a page, again making allocation easier.

14.4 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 to speed up the process. The PowerPC is one example of an architecture that uses such an approach [3].

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.

14.5 Paging the Page Tables

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.

14.6 Paging: Summary

We have now seen how real page tables are built; not necessarily just as linear arrays but as more complex data structures. The trade-offs 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 (that's you).

References

[1] "Reese's Peanut Butter Cups."
Mars Candy Corporation.
Just delicious.

[2] "Virtual Memory Management in the VAX/VMS Operating System"
Hank Levy and P. Lipman.
IEEE Computer, Vol. 15, No. 3, March 1982, pp. 35-41.

[3] "Performance Implications of the PowerPC Architecture's Hashed
Page Table Utilization in Windows NT".
Mark VandenBrink.
Performance, Computing, and Communications Conference, 1997 (IPCCC
1997).