17

Paging: Introduction

Remember our goal: to virtualize memory. Segmentation (a generalization of dynamic relocation) helped us do this, but has some problems; in particular, managing free space becomes quite a pain as memory becomes fragmented and segmentation is not as flexible as we might like. Is there a better solution?

THE CRUX:
HOW TO VIRTUALIZE MEMORY WITHOUT SEGMENTS

How can we virtualize memory in a way as to avoid the problems of segmentation? What are the basic techniques? How do we make those techniques work well?

Thus comes along the idea of **paging** [KE+62,L78]. Instead of splitting up our address space into three logical segments (each of variable size), we split up our address space into fixed-sized units we call a **page**. Here in Figure 17.1 an example of a tiny address space, 64 bytes total in size, with 16 byte pages (real address spaces are much bigger, of course, commonly 32 bits and thus 4-GB of address space, or even 64 bits).

Thus, we have an address space that is split into four pages (0 through 3). With paging, physical memory is also split into some number of pages as well; we sometimes will call each page of physical memory a **page frame**. For an example, let's examine Figure 17.2.

Paging, as we will see, has a number of advantages over our previous approaches. Probably the most important improvement will be

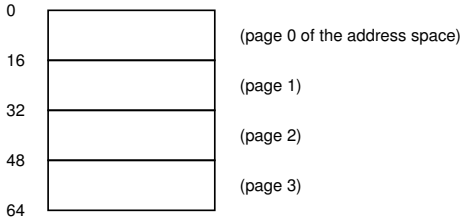


Figure 17.1: A Simple 64-byte Address Space

flexibility: with a fully-developed paging approach, the system will be able to support the abstraction of an address space effectively, regardless of how the processes uses the address space; we won't, for example, have to make assumptions about how the heap and stack grow and how they are used.

Another advantage is the *simplicity* of free-space management that paging affords. For example, when the OS wishes to place our tiny 64-byte address space from above into our 8-page physical memory, it simply finds four free pages; perhaps the OS keeps a **free list** of all free pages for this, and just grabs the first four free pages off of this list. In the example above, the OS has placed virtual page 0 of the address space (AS) in physical page 3, virtual page 1 of the AS on physical page 7, page 2 on page 5, and page 3 on page 2.

To record where each virtual page of the operating system is placed in physical memory, the operating system keeps a *per-process* data structure known as a **page table**. The major role of the page table is to store **address translations** for each of the virtual pages of the address space, thus letting us know where in physical memory they live. For our simple example above, the page table would thus have the following entries: (Virtual Page 0 → Physical Frame 3), (VP 1 → PF 7), (VP 2 → PF 5), (VP 3 → PF 2).

It is important to remember that this page table is a *per-process* data structure[1]; If another process were to run in our example above, the OS would have to manage a different page table for it, as its vir-

[1]This is generally true for most of the page table structures we will discuss; however, for some page tables, such as the **inverted page table**, there is one table for all processes.

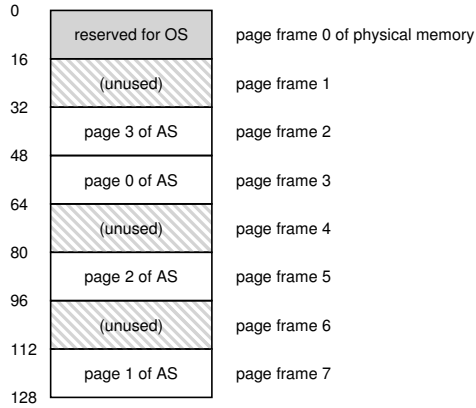


Figure 17.2: 64-Byte Address Space Placed In Physical Memory

tual pages obviously map to *different* physical pages (modulo any sharing going on).

Now, we know enough to perform an address-translation example. Let's imagine the process with that tiny address space (64 bytes) is performing a memory access:

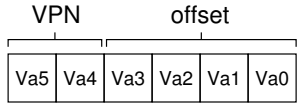
```
movl <virtual address>, %eax
```

Specifically, let's pay attention to the explicit load of the data at `<virtual address>` into the register `eax` (and thus ignore the instruction fetch that must have happened prior).

To *translate* this virtual address that the process generated, we have to first split it into two components: the **virtual page number (VPN)**, and the **offset** within the page. For this example, because the virtual address space of the process is 64 bytes, we need 6 bits total for our virtual address ($2^6 = 64$). Thus, our virtual address:

Va5	Va4	Va3	Va2	Va1	Va0

where Va5 is the highest-order bit of the virtual address, and Va0 the lowest order bit. Because we know the page size (16 bytes), we can further divide the virtual address as follows:



The page size is 16 bytes in a 64-byte address space; thus we need to be able to select 4 pages, and the top 2 bits of the address do just that. Thus, we have a 2-bit virtual page number (VPN). The remaining bits tell us which byte of the page we are interested in, 4 bits in this case; we call this the offset.

When a process generates a virtual address, the OS and hardware must combine to translate this virtual address into a meaningful physical address. For example, let us assume the load above was to virtual address 21:

```
movl 21, %eax
```

Turning "21" into binary form, we get "010101", and thus we can examine this virtual address and see how it breaks down into a virtual page number (VPN) and offset:

Thus, the virtual address "21" is on the 5th ("0101"th) byte of virtual page "01" (or 1). With our virtual page number, we can now index our page table and find which physical page that virtual page 1 resides within. In the page table above the physical page number (PPN) (a.k.a. physical frame number or PFN) is 7 (binary 111). Thus, we can translate this virtual address by replacing the VPN with the PFN and then issue the load to physical memory (Figure 17.3).

Note the offset stays the same (i.e., it is not translated), because the offset just tells us which byte *within* the page we want. Our final physical address is 1110101 (117 in decimal), and is exactly where we want our load to fetch data from (Figure 17.2).

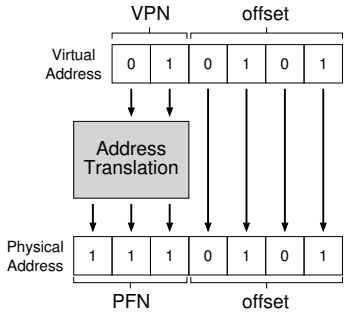


Figure 17.3: The Address Translation Process

17.1 Where Are Page Tables Stored?

Page tables can get awfully large, much bigger than the small segment table or base/bounds pair we have discussed previously. For example, imagine a typical 32-bit address space, with 4-KB pages. This virtual address splits into a 20-bit VPN and 12-bit offset (recall that 10 bits would be needed for a 1-KB page size, and just add two more to get to 4 KB).

A 20-bit VPN implies that there are 2^{20} translations that the OS would have to manage for each process (that's roughly a million); assuming we need 4 bytes per **page table entry (PTE)** to hold the physical translation plus any other useful stuff, we get an immense 4MB of memory needed for each page table! That is pretty big. Now imagine there are 100 processes running: this means the OS would need 400MB of memory just for all those address translations! Even in the modern era, where machines have gigabytes of memory, it seems a little crazy to use a large chunk of if just for translations, no?

Because page tables are so big, we don't keep any special on-chip hardware in the MMU to store the page table of the currently-running process. Instead, we store the page table for each process in *memory* somewhere. Let's assume for now that the page tables live in physical memory that the OS manages. In Figure 17.4 is a picture of what that might look like.

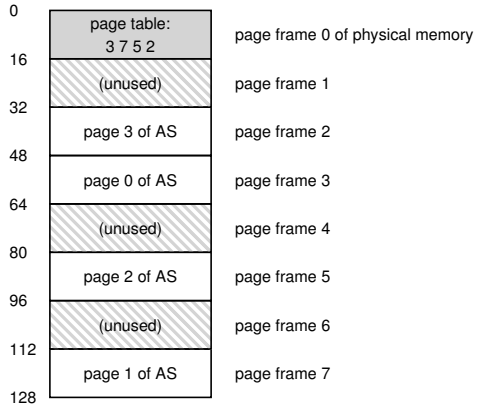


Figure 17.4: Example: Page Table in Kernel Physical Memory

17.2 What's Actually In The Page Table?

Let's talk a little about page table organization. The page table is just a data structure that is used to map virtual addresses (or really, virtual page numbers) to physical addresses (physical page numbers). Thus, any data structure could work. The simplest form is called a **linear page table**, which is just an array. The OS *indexes* the array by the VPN, and looks up the page-table entry (PTE) at that index in order to find the desired PFN. For now, we will assume this simple linear structure; in later chapters, we will make use of more advanced data structures to help solve some problems with paging.

As for the contents of each PTE, we have a number of different bits in there worth understanding at some level. A **valid bit** is common to indicate whether the particular translation is valid; for example, when a program starts running, it will have code and heap at one end of its address space, and the stack at the other. All the unused space in-between will be marked **invalid**, and if the process tries to access such memory, it will generate a trap to the OS which will likely terminate the process. Thus, the valid bit is crucial for supporting a sparse address space; by simply marking all the unused pages in the address space invalid, we remove the need to allocate physical frames for those pages and thus save a great deal of memory.

We also might have **protection bits**, indicating whether the page could be read from, written to, or executed from (e.g., a code page). Again, accessing a page in a way not allowed by these bits will generate a trap to the OS.

There are a couple of other bits that are important but we won't talk about much for now. A **present bit** indicates whether this page is in physical memory or on disk (swapped out); we will understand this in more detail when we study how to move parts of the address space to disk and back in order to support address spaces that are larger than physical memory and allow for the pages of processes that aren't actively being run to be swapped out. A **dirty bit** is also common, indicating whether the page has been modified since it was brought into memory.

A **reference bit** (a.k.a. **accessed bit**) is sometimes used to track whether a page has been accessed, and is useful in determining which pages are popular and thus should be kept in memory; such knowledge is critical during **page replacement**, a topic we will study in great detail in subsequent chapters.

Figure 17.5 shows an example page table entry from the x86 architecture [I09]. It contains a present bit (P); a read/write bit (R/W) which determines if writes are allowed to this page; a user/supervisor bit (U/S) which determines if user-mode processes can access the page; a few bits (PWT, PCD, PAT, and G) that determine how hardware caching works for these pages; an accessed bit (A) and a dirty bit (D); and finally, the PFN itself.

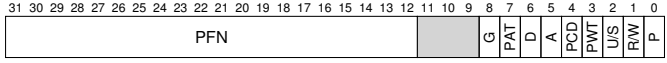


Figure 17.5: An x86 Page Table Entry (PTE)

Read the Intel Architecture Manuals [I09] for more details on x86 paging support. Be forewarned, however; reading manuals such as these, while quite informative (and certainly necessary for those who write code to use such page tables in the OS), can be challenging at first. A little patience, and a lot of desire, is required.

17.3 Paging: Also Too Slow

With page tables in memory, we already know that they might be too big. Turns out they can slow things down too. For example, take our simple instruction:

```
movl 21, %eax
```

Again, let's just examine the explicit reference to address 21 and not worry about the instruction fetch. In this example, we will assume the hardware performs the translation for us. To fetch the desired data, the system must first **translate** the virtual address (21) into the correct physical address (117). Thus, before issuing the load to address 117, the system must first fetch the proper page table entry from the process's page table, perform the translation, and then finally get the desired data from physical memory.

To do so, the hardware must know where the page table is for the currently-running process. Let's assume for now that a single **page-table base register** contains the physical address of the starting location of the page table. To find the location of the desired PTE, the hardware will thus perform the following functions:

```
VPN     = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

In our example, VPN_MASK would be set to 0x30 (hex 30, or binary 110000) which picks out the VPN bits from the full virtual address; SHIFT is set to 4 (the number of bits in the offset), such that we move the VPN bits down to form the correct integer virtual page number. For example, with virtual address 21 (010101), and masking turns this value into 010000; the shift turns it into 01, or virtual page 1, as desired. We then use this value as an index into the array of PTEs pointed to by the page table base register.

Once this physical address is known, the hardware can fetch the PTE from memory, extract the PFN, and concatenate it with the offset from the virtual address to form the desired physical address. Specifically, you can think of the PFN being left-shifted by SHIFT, and then logically OR'd with the offset to form the final address as follows:

```
offset   = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

```
1  // Extract the VPN from the virtual address
2  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4  // Form the address of the page-table entry (PTE)
5  PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7  // Fetch the PTE
8  PTE = AccessMemory(PTEAddr)
9
10 // Check if process can access the page
11 if (PTE.Valid == False)
12     RaiseException(SEGMENTATION_FAULT)
13 else if (CanAccess(PTE.ProtectBits) == False)
14     RaiseException(PROTECTION_FAULT)
15 else
16     // Access is OK: form physical address and fetch it
17     offset   = VirtualAddress & OFFSET_MASK
18     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19     Register = AccessMemory(PhysAddr)
```

Figure 17.6: Accessing Memory With Paging

Finally, the hardware can fetch the desired data from memory and put it into register `eax`. The program has now succeeded at loading a value from memory!

To summarize, we now describe the initial protocol for what happens on each memory reference. Figure 17.6 shows the basic approach. For every memory reference (whether an instruction fetch or an explicit load or store), paging requires us to perform one extra memory reference in order to first fetch the translation from the page table. That is a lot of work! Extra memory references are costly, and in this case will likely slow down the process by a factor of two or more.

And now you can hopefully see that there are *two* real problems that we must solve. Without careful design of both hardware and software, page tables will cause the system to run too slowly, as well as take up too much memory. While seemingly a great solution for our memory virtualization needs, these two crucial problems must first be overcome.

ASIDE: **DATA STRUCTURE – THE PAGE TABLE**
One of the most important data structures in the memory management subsystem of a modern OS is the **page table**. In general, a page table stores **virtual-to-physical address translations**, thus letting the system know where each page of an address space actually resides in physical memory. Because each address space requires such translations, in general there is one page table per process in the system. The exact structure of the page table is either determined by the hardware (older systems) or can be more flexibly managed by the OS (modern systems).

17.4 Summary

We have introduced the concept of **paging** as a solution to our challenge of virtualizing memory. Paging has many advantages over previous approaches (such as segmentation). First, it does not lead to external fragmentation, as paging (by design) divides memory into fixed-sized units. Second, it is quite flexible, enabling the sparse use of virtual address spaces.

However, implementing paging support without care will lead to a slower machine (with many extra memory accesses to access the page table) as well as memory waste (with memory filled with page tables instead of useful application data). We'll thus have to think a little harder to come up with a paging system that not only works, but works well. The next two chapters, fortunately, will show us how to do so.

References

[KE+62] "One-level Storage System"
T. Kilburn, and D.B.G. Edwards and M.J. Lanigan and F.H. Sumner
IRE Trans. EC-11, 2 (1962), pp. 223-235
(Reprinted in Bell and Newell, "Computer Structures: Readings and Examples" McGraw-Hill, New York, 1971).
The Atlas pioneered the idea of dividing memory into fixed-sized pages and in many senses was an early form of the memory-management ideas we see in modern computer systems.

[I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals"
Intel, 2009
Available: http://www.intel.com/products/processor/manuals
In particular, pay attention to "Volume 3A: System Programming Guide Part 1" and "Volume 3B: System Programming Guide Part 2"

[L78] "The Manchester Mark I and atlas: a historical perspective"
S. H. Lavington
Communications of the ACM archive
Volume 21, Issue 1 (January 1978), pp. 4-12
Special issue on computer architecture
This paper is a great retrospective of some of the history of the development of some important computer systems. As we sometimes forget in the US, many of these new ideas came from overseas.

Homework

In this homework, you will use a simple program, which is known as `paging-linear-translate.py`, to see if you understand how simple virtual-to-physical address translation works with linear page tables. To run the program, remember to either type just the name of the program `./paging-linear-translate.py` or possibly this `python paging-linear-translate.py`. When you run it with the [-h] (help) flag, you see:

```
Usage: paging-linear-translate.py [options]

Options:
-h, --help              show this help message and exit
-s SEED, --seed=SEED    the random seed
-a ASIZE, --asize=ASIZE
                        address space size (e.g., 16, 64k, ...)
-p PSIZE, --physmem=PSIZE
                        physical memory size (e.g., 16, 64k, ...)
-P PAGESIZE, --pagesize=PAGESIZE
                        page size (e.g., 4k, 8k, ...)
-n NUM, --addresses=NUM number of virtual addresses to generate
-u USED, --used=USED    percent of address space that is used
-v                      verbose mode
-c                      compute answers for me
```

First, run the program without any arguments:

```
ARG seed 0
ARG address space size 16k
ARG phys mem size 64k
ARG page size 4k
ARG verbose False

The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
   0x8000000c
   0x00000000
   0x00000000
   0x80000006
```

```
Virtual Address Trace
  VA  0: 0x00003229 (decimal:    12841) --> PA or invalid?
  VA  1: 0x00001369 (decimal:     4969) --> PA or invalid?
  VA  2: 0x00001e80 (decimal:     7808) --> PA or invalid?
  VA  3: 0x00002556 (decimal:     9558) --> PA or invalid?
  VA  4: 0x00003a1e (decimal:    14878) --> PA or invalid?

For each virtual address, write down the physical address it
translates to OR write down that it is an out-of-bounds
address (e.g., a segmentation fault).
```

As you can see, what the program provides for you is a page table for a particular process (remember, in a real system with linear page tables, there is one page table *per process*; here we just focus on one process, its address space, and thus a single page table). The page table tells you, for each virtual page number (VPN) of the address space, that the virtual page is mapped to a particular physical frame number (PFN) and thus valid, or not valid.

The format of the page-table entry is simple: the left-most (high-order) bit is the valid bit; the remaining bits, if valid is 1, is the PFN.

In the example above, the page table maps VPN 0 to PFN 0xc (decimal 12), VPN 3 to PFN 0x6 (decimal 6), and leaves the other two virtual pages, 1 and 2, as not valid.

Because the page table is a linear array, what is printed above is a replica of what you would see in memory if you looked at the bits yourself. However, it is sometimes easier to use this simulator if you run with the verbose flag (`-v`); this flag also prints out the VPN (index) into the page table. From the example above, run with the `-v` flag:

```
Page Table (from entry 0 down to the max size)
  [      0]   0x8000000c
  [      1]   0x00000000
  [      2]   0x00000000
  [      3]   0x80000006
```

Your job, then, is to use this page table to translate the virtual addresses given to you in the trace to physical addresses. Let's look at the first one: VA 0x3229. To translate this virtual address into a physical address, we first have to break it up into its constituent components: a *virtual page number* and an *offset*. We do this by noting down the size of the address space and the page size. In this example, the address space is set to 16KB (a very small address space) and

the page size is 4KB. Thus, we know that there are 14 bits in the virtual address, and that the offset is 12 bits, leaving 2 bits for the VPN. Thus, with our address 0x3229, which is binary 11 0010 0010 1001, we know the top two bits specify the VPN. Thus, 0x3229 is on virtual page 3 with an offset of 0x229.

We next look in the page table to see if VPN 3 is valid and mapped to some physical frame or invalid, and we see that it is indeed valid (the high bit is 1) and mapped to physical page 6. Thus, we can form our final physical address by taking the physical page 6 and adding it onto the offset, as follows: 0x6000 (the physical page, shifted into the proper spot) OR 0x0229 (the offset), yielding the final physical address: 0x6229. Thus, we can see that virtual address 0x3229 translates to physical address 0x6229 in this example.

To see the rest of the solutions (after you have computed them yourself!), just run with the `-c` flag (as always):

```
...
VA  0: 00003229 (decimal: 12841) --> 00006229 (25129) [VPN 3]
VA  1: 00001369 (decimal:  4969) --> Invalid (VPN 1 not valid)
VA  2: 00001e80 (decimal:  7808) --> Invalid (VPN 1 not valid)
VA  3: 00002556 (decimal:  9558) --> Invalid (VPN 2 not valid)
VA  4: 00003a1e (decimal: 14878) --> 00006a1e (27166) [VPN 3]
```

Of course, you can change many of these parameters to make more interesting problems. Run the program with the `-h` flag to see what options there are:

- The `-s` flag changes the random seed and thus generates different page table values as well as different virtual addresses to translate.
- The `-a` flag changes the size of the address space.
- The `-p` flag changes the size of physical memory.
- The `-P` flag changes the size of a page.
- The `-n` flag can be used to generate more addresses to translate (instead of the default 5).
- The `-u` flag changes the fraction of mappings that are valid, from 0that roughly 1/2 of the pages in the virtual address space will be valid.
- The `-v` flag prints out the VPN numbers to make your life easier.

Questions

- Before doing any translations, let's use the simulator to study how linear page tables change size given different parameters. Compute the size of linear page tables as different parameters change. Some suggested inputs are below; by using the `-v flag`, you can see how many page-table entries are filled.

  First, to understand how linear page table size changes as the address space grows:

  ```
  paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
  paging-linear-translate.py -P 1k -a 2m -p 512m -v -n 0
  paging-linear-translate.py -P 1k -a 4m -p 512m -v -n 0
  ```

  Then, to understand how linear page table size changes as page size grows:

  ```
  paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
  paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0
  paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0
  ```

  Before running any of these, try to think about the expected trends. How should page-table size change as the address space grows? As the page size grows? Why shouldn't we just use really big pages in general?

- Now let's do some translations. Start with some small examples, and change the number of pages that are allocated to the address space with the `-u flag`. For example:

  ```
  paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0
  paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25
  paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50
  paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75
  paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
  ```

  What happens as you increase the percentage of pages that are allocated in each address space?

- Now let's try some different random seeds, and some different (and sometimes quite crazy) address-space parameters, for variety:

```
paging-linear-translate.py -P 8  -a 32   -p 1024 -v -s 1
paging-linear-translate.py -P 8k -a 32k  -p 1m   -v -s 2
paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3
```

Which of these parameter combinations are unrealistic? Why?

- Use the program to try out some other problems. Can you find the limits of where the program doesn't work anymore? For example, what happens if the address-space size is *bigger* than physical memory?