

## 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. Thus, we'd like to find a different solution.

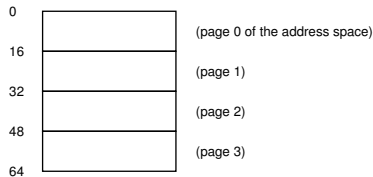


Figure 12.1: A Simple 64-byte Address Space

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 will split up our address space into fixed-sized units we call a **page**. Here in Figure 12.1 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).

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 12.2.

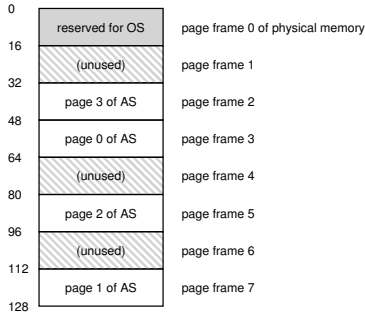


Figure 12.2: 64-Byte Address Space Placed In Physical Memory

The beauty of paging is the simplicity and ease the OS has when it is managing free physical memory. 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 the 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 Number	Physical Page Frame
0	3
1	7
2	5
3	2

As we said before, it is important to remember that this is a *per-process* data structure. If another process were to run in our example above, the OS would have to manage a different page table for it, as its virtual 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 looks like this:

```
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:

```
VPN | offset
Va5 Va4 | Va3 Va2 Va1 Va0
```

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:

```
VPN | offset
01  | 0101
```

Thus, the virtual address “20” 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 5 (binary 101). Thus, we can translate this virtual address by simply replacing the VPN with the correct PFN and then finally issue the load to physical memory:

```
VPN | offset
01  | 0101
```

becomes

```
PFN | offset
101 | 0101
```

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. Thus, our final physical address is 1010101, which is 85 in decimal, and is exactly where we want our load to fetch data from (Figure 12.2).

## 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 entirely by the OS (more modern systems).

## 12.1 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!

Because they 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 12.3 is a picture of what that might look like.

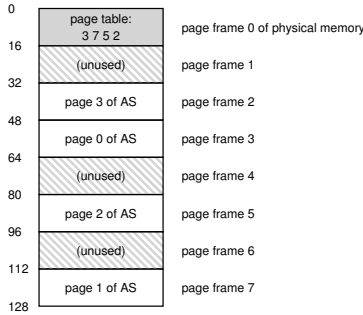


Figure 12.3: Example: Page Table in Kernel Physical Memory

## 12.2 What's Actually In The Page Table?

A small aside 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**. It is just an array. The OS indexes the array by the VPN, and thus looks up the page-table entry (PTE) at that index in order to find the desired PFN. For now, we will assume this linear page table structure; below 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 matched 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. We will come back to this bit as well.

### 12.3 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 (85). Thus, before issuing the load to address 85, 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) >> VPN_SHIFT;
AddressOfPTE = 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; `VPN_SHIFT` is set to 4 (the size of 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), the mask turns this into 010000, and 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, extract the PFN, and concatenate it with the offset from the virtual address to form the desired physical address. Finally, the hardware can fetch the desired data from memory and put it into register `eax`.

Simply put, for every memory reference (whether an instruction fetch or an explicit load or store), paging now requires us to perform one extra memory reference in order to first fetch the translation from the page table. Wow, 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 we can see what the **two** real problems are that we must solve.

#### THE CRUX: PAGING TOO SLOW, TABLES TOO BIG

Paging, as we've described, has two big problems: the page tables are **too big**, and the address translation process is **too slow**. Thus, how can the OS, in tandem with the hardware, speed up translation? How can the OS and hardware reduce the exorbitant memory demands of paging?

## 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.*

[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.*