【OS】Day24

Class	Operating System: Three Easy Pieces
Date	@January 24, 2022

【Ch20】Paging: Smaller Tables

Let's 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. Linear page tables get pretty big. Assume again a 32-bit address space(2^{32} bytes), with 4KB(2^{12} bytes) pages and a 4-byte page-table entry.

An address space thus has roughly one million virtual pages in it($\frac{2^{32}}{2^{12}}$ bytes); multipled by the page-table entry size and we see that our page table is 4MB in size.

Recall also that we have one page table for every process in the system!

20.1 Simple Solution: Bigger Pages

We could reduce the size of the page table in one simple way: use bigger pages.

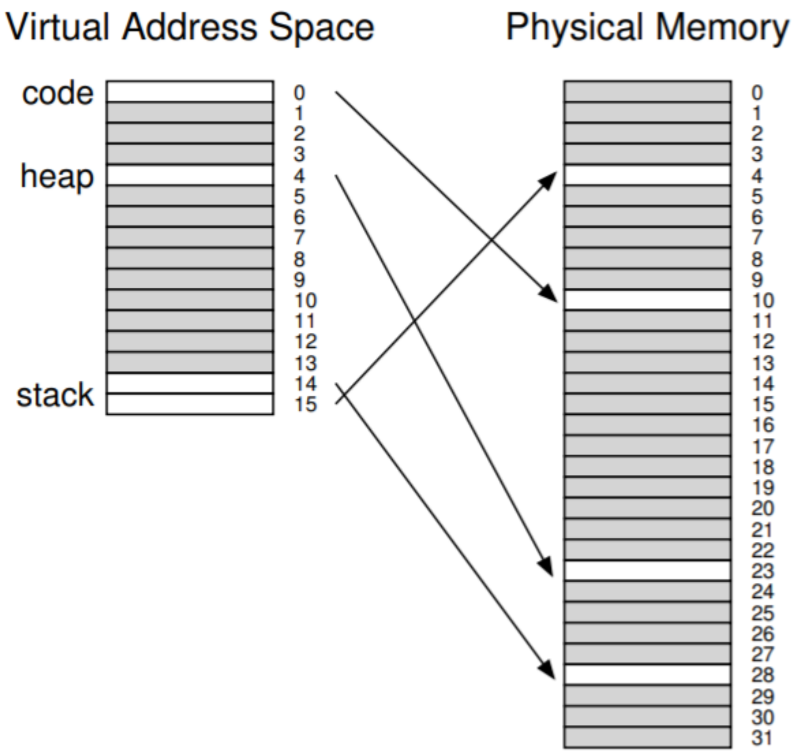
Take our 32-bit address space again, but this time assume 16KB pages. We would thus have an 18-bit VPN plus a 14-bit offset. We now have 2^{18} entries in our linear page table and thus a total size of 1MB per page table.

The major problem with this approach, however, is that big pages lead to waste within each page, also known as internal fragmentation.

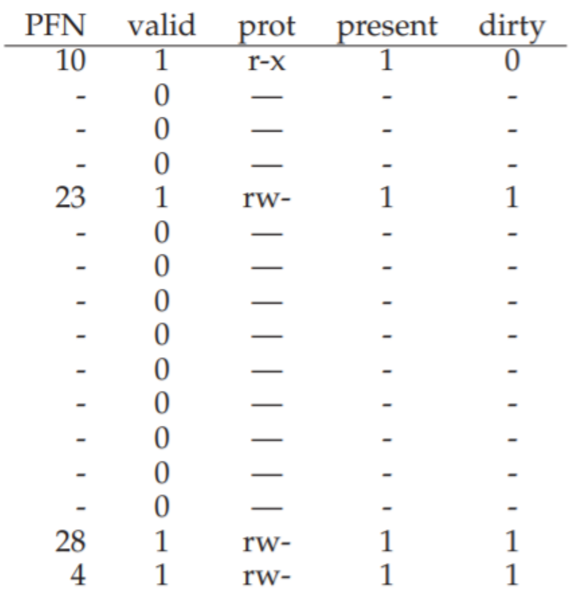
Applications thus end up allocating pages but only using little bits and pieces of each, and memory quickly fills up with these overly-large pages.

20.2 Hybrid Approach: Paging and Segmentation

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.



The page table is shown below:



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

Figure 20.2: **A Page Table For 16KB 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 28 and 4.

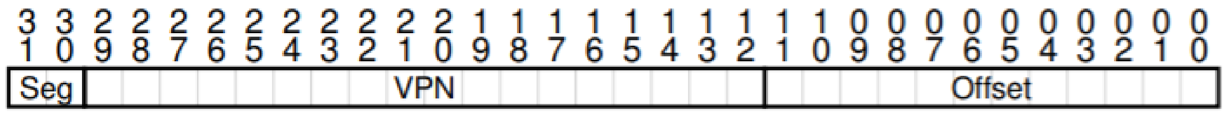
Most of the page table is unused, full of invalid entries.

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 our hybrid approach, we still have our base and limit registers 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. Assume a 32-bit virtual address space with 4KB pages, and an address space split into four segments. We will use three segments for this example: one for code, one for heap, and one for stack.



On a TLB miss, 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;
//Base[i] contains the base register for segment i
AddressofPTE = Base[SN] +(VPN * sizeof(PTE));
```

The presence of a bounds register per segment helps us prevent access page entries beyond our page table. The OS will generate an exception and likely lead to the termination of the process.