Chapter 20: Paging: smaller tables

A problem that paging introduces is that page tables are too big and thus consume too much memory. The page tables consume a large amount of memory.

**20.1 Simple Solution: Bigger pages**

We could reduce the size of the page table in one simple way: use bigger pages. If we make the size of a page to be 16KB, then we would need 18-bit VPN and 14-bit offset. If the page table entry size is 4 bytes, since we have 2^18 entries, the size of each page table is 1MB.

There might be some problems for this approach. One is internal fragmentation where big pages lead to waste in each page.

**20.2 Hybrid Approach: Paging and Segments**

Example:

Diagram

Description automatically generated

In the page table, most of them are invalid entries. Most of them are wasted.

In our hybrid approach, instead of having a single page table for the entire address space of the process, we would have one per logical segment. In the above example, we may have 3 page tables for the code, heap and stack.

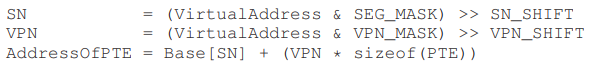
In this approach, we would use the base register to hold the physical address of the page table of a segment. The bound register is used to indicate the end of the page table.

For example, if we have 32-bit virtual address space with 4KB pages and an address space split into four segments.



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 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, the hardware uses the segment bits 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:

****

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. This method saves a significant memory.

There will still be problems. First, it still requires us to use segmentation; as we discussed before, segmentation is not quite as flexible as we would like, as it assumes a certain usage pattern of the address space; if we have a large but sparsely-used heap, for example, we can still end up with a lot of page table waste. Secondly, the approach still causes external fragmentation.

20.3 Multi-level page tables:

The idea is simple. First, we 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, we 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.

For example:

**Diagram, schematic

Description automatically generated**

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 require page-table space allocated for those regions. However, on the right, 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.

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 (minimally) has a **valid bit** and a **page frame number (PFN)**, similar to a **PTE**. If a page is valid, then at least one of the pages of the page table is valid.

The advantage of multi-level page are:

1. 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.
2. Second, if carefully constructed, each portion of the page table fits neatly within a page, making it easier to manage memory. 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.

The cost is that on a TLB miss, two loads (one for page directory, one for PTE) from memory will be required to get the right translation from the page table, while linear page table only needs one load. This is an example of a **time-space trade-off**.

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.

**A Detailed Multi-level example**

Imagine a small address space of size 16KB, 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 entries.

Table

Description automatically generated

To build a two-level page table for this address space, we start with our linear page table and break it up into page-sized unit. The full table as 256 entries. Assume each PTE is 4 bytes in size. Thus, out page table is 1KB in size (256\*4 bytes). Given that we have 64-byte pages, we can divide it into 16 PTEs.

We have 256 entries, spread across 16 pages. The page directory needs at least one entry per page of page table. Thus, it has 16 entries. We will need four bits of the VPN to index into the directory.

Diagram, table

Description automatically generated

From the **page directory index**, we can find the address of the page-directory entry. 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 do this, we need the page table index as well to index into the page table itself.

Diagram

Description automatically generated with low confidence

**More Than Two Levels**

When there are too many page directory index, we further splitting the page directory into multiple pages:

Table

Description automatically generated

**The Translation Process: Remember the TLB**

Before any of the complicated multilevel 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.

**20.4 Inverted Page Tables**

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.

We would normally use a hash table to speed up the look up in this structure.

**20.5 Swapping the Page Tables to Disk**

Some systems place page tables in kernel virtual memory because it is too big, thereby allowing the system to **swap** some of these page tables to disk when memory pressure gets a little tight.

Text

Description automatically generated