【OS】Day23

【Ch19】Paging: Faster Tanslations(TLBs)

Using paging as the core mechanism to support virtual memory can lead to high performance overheads.

By chopping the address space into small, fixed-sized units, paging requires a large amount of mapping information. Because that mapping information is generally stored in physical memory, paging logically requires an extra memory lookup for each virtual address generated by the program.

Going to memory for translation information before every instruction fetch or explicit load or store is prohibitively slow.

Thus, our problem: *How can we speed up address translation, and generally avoid the extra memory reference that paging seems to require?*

When we want to make things fast, the OS usually needs some help from the hardware.

To speed address translation, we are going to add what is called a translation-lookaside buffer, or TLB.

A TLB is part of the chip's memory-management unit (MMU), and is simply a hardware cache of popular virtual-to-physical address translations; thus, a better name would be an address-translation cache.

Upon each virtual memory reference, the hardware first checks the TLB to see if the desired translation is held therein; if so, the translation is performed(quickly) without having to consult the page table.

Because of their tremendous performance impact, TLBs in a real sense make virtual memory possible.

19.1 TLB Basic Algorithm

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                  // TLB Miss
11     PTEAddr = PTBR + (VPN * sizeof(PTE))
12     PTE = AccessMemory(PTEAddr)
13     if (PTE.Valid == False)
14         RaiseException(SEGMENTATION_FAULT)
15     else if (CanAccess(PTE.ProtectBits) == False)
16         RaiseException(PROTECTION_FAULT)
17     else
18         TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19         RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

The figure above shows a rough sketch of how hardware might handle a virtual address translation, assuming a simple linear page table and a hardware-managed TLB.

The algorithm the hardware follows works like this:

1. First, extract the virtual page number(VPN) from the virtual address and check if the TLB holds the translation for this VPN(Line 2).

   If it does, we have a TLB hit, which means the TLB holds the translation. Success!

   We can now extract the page frame number(PFN) from the relevant TLB entry, concatenate that onto the offset from the original virtual address, and form the desired physical address(PA), and access memory(Line 5-7), assuming protection checks do not fail(Line 4).

2. If the CPU does not find the translation in the TLB(a TLB miss), we have some work to do.

   In this example, the hardware accesses the page table to find the translation(Line 11-12), and, assuming that the virtual memory reference generated by the process is valid and accessible(Line 13, 15) updates the TLB with the translation(Line 18).

   These set of actions are costly, primarily be cause of the extra memory reference needed to access the page table(Line 12).

   Finally, once the TLB is updated, the hardware retries the instruction; this time, the translation is found in the TLB, and the memory reference is processed quickly.

The TLB is built on the premise that in the common case, translations are found in the cache. If so, little overhead is added, as the TLB is found near the processing core and is designed to be quite fast.

19.2 Example: Accessing An Array

Let's look at an example. In this example. Let's assume that we have an array of 10 4-byte integers in memory, starting at virtual address 100.

Assume further that we have *a small 8-bit virtual address space, with 16-byte pages; thus, a virtual address breaks down into a 4-bit VPN(there are 16 virtual pages) and a 4-bit offset(there are 16 bytes on each of those pages).*

The following figure shows the array laid out on the 16 16-byte pages of the system.

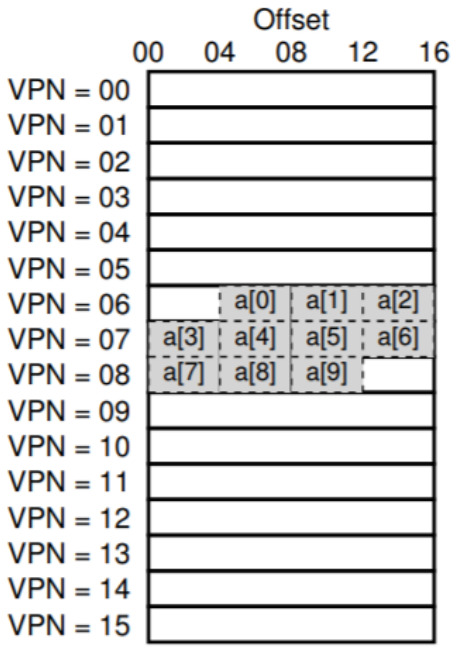

Figure 19.2: **Example: An Array In A Tiny Address Space**

The array's first entry(a[0]) begins on (VPN = 06, offset 04); only three 4-byte integer fits onto that page. The array continues onto the next page(VPN = 07), where the next four entries(a[3]...a[6]) are found. Finally the last three entries of the 10-entry array(a[7]...a[9]) are located on the next page of the address space(VPN = 08)

Let’s now consider a simple loop that accesses each array element, something that would look like this in C:

```
int sum = 0;
for(i = 0; i < 10; ++i) {
  sum += a[i];
}
```

For the sake of simplicity, we will pretend that the only memory accesses the loop generates are to the array(ignoring the variables i and sum, as well as the instructions themselves).

When the first array element(a[0]) is accessed, the CPU will see a load to virtual address 100. The hardware extracts the VPN from this(VPN = 06), and uses that to check the TLB for a valid translation. Assuming this is the first time the program accesses the array, the result will be a TLB miss.

Then the next access is to a[1], and there is some good news here: a TLB hit! Because the second element of the array is packed next to the first, it loves on the same page; because we’ve already accessed this page when accessing the first element of the array, the translation is already loaded into the TLB.

Let us summarize TLB activity when the iteration finishes: miss, hit, hit, miss, hit, hit, hit, miss, hit, hit.

Thus, our TLB hit rate, which is the number of hits divided by the total number of accesses, is 70%. Although this is not too high(indeed, we desire hit rates that approach 100%), it is non-zero, which may be a surprise.

Even though this is the first time the program accesses the array, the TLB improves performance due to spatial locality. The elements of the array are packed tightly into pages(i.e., they are close to one another in space), and thus only the first access to an element on a page yields a TLB miss.

Also note the role that page size plays in this example. If the page size had simply been twice as big, the array access would suffer even fewer misses. As typical page sizes are more like 4KB, these types of dense, array-based accesses achieve excellent TLB performance, encountering only a single miss per page of accesses.

One last point about TLB performance: if the program, soon after this loop completes, accesses the array again, we'd likely see an even better result, assuming that we have a big enough TLB to cache the needed translations: hit, hit, hit, hit, hit, hit, hit, hit, hit, hit.

In this case, the TLB hit rate would be high because of temporal locality(i.e. the quick re-referencing of memory items in time). Like any cache, TLBs rely upon both spatial and temporal locality for success, which are program properties. If the program of interest exhibits such locality(and many programs do|), the TLB hit rate will likely be high.