【OS】Day23(2)

【Ch19】Paging: Faster Translation(2)

19.3 Who Handles the TLB Miss?

One question that we must answer: *who handles a TLB miss?*

Two answers are possible: the hardware, or the software(OS).

In the olden days, the hardware had complex instruction sets and the people who built the hardware didn't much trust those OS people. Thus, the hardware has to know exactly where the page tables are located in memory(via a page-table base register) as well as their exact format; on a miss, the hardware would "walk" the page table, find the correct page-table entry and extract the desired translation, update the TLB with the translation, and retry the instruction.

More modern architectures have what is known as a software-managed TLB. On a TLB miss, the hardware simply raises an exception, which pauses the current instruction stream, raise the privilege level to kernel mode, and jumps to a trap handler.

When run, the ode will lookup the translation in the page table, use special "privileged" instructions to update the TLB, and return from the trap.

19.4 TLB Contents: What's In There?

Let's look at the contents of the hardware TLB in more detail. A typical TLB might have 32, 64, or 128 entries and be what is called fully associative.

Basically, this means that any given translation can be anywhere in the TLB, and that the hardware will search the entire TLB in parallel to find the desired translation. A TLB entry might look like this:

More interesting are the “other bits”. For example, the TLB commonly has a valid bit, which says whether the entry has a valid translation or not.

Also common are protection bits, which determine how a page can be accessed. For example, code pages might be marked *read and execute*, whereas heap pages might be marked read and write.

There may also be a few other fields, including an address-space identifier, a dirty bit, and so forth

19.5 TLB Issue: Context Switches

With TLBs, some new issues arise when switching between processes(and hence address spaces). Specifically, the TLB contains virtual-to-physical translations that are only valid for the currently running process; these translations are not meaningful for other processes.

As a result, when switching from one process to another, the hardware or OS must be careful to ensure that the about-to-be-run process does not accidentally use translations from some previously run process.

Let’s look at an example. Assuming another process(P2) exists, and the OS soon might decide to perform a context switch and run it. Assume here that the 10th virtual page of P2 is mapped to physical frame 170. If entries for both processes were in the TLB, the contents of the TLB would be:

VPN	PFN	valid	prot
10	100	1	rwx
—	—	0	—
10	170	1	rwx
—	—	0	—

There are a number of possible solutions to this problem. One approach is to simply flush the TLB on context switches, thus emptying it before running the next process. The flush operation simply sets all valid bits to 0, essentially clearing the contents of the TLB.

However, there is a cost: each time a process runs, it must incur TLB misses as it touches its data and code pages.

To reduce this overhead, some systems add hardware support to enable sharing of the TLB across context switches. In particular, some hardware systems provide an address space identifier(ASID) field in the TLB. We can think of the ASID as a process identifier(PID), but usually it has fewer bits.

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

Thus, with address-space identifiers; the TLB can hold translations from different processes at the same time without any confusion.

19.6 Issue: Replacement Policy

One more issue that we must consider is cache replacement. Specifically, when we are installing a new entry in the TLB, we have to replace an old one, and thus the question: *which one to replace?*

One common approach is to evict the least-recently-used or LRU entry. LRU tries to take advantage of locality in the memory-reference stream, assuming it is likely that an entry that has not recently been used is a good candidate for eviction.

Another typical approach is to use a random policy, which evicts a TLB mapping at random.