Module 2, Lesson 5 “Translation Lookaside Buffers”

Hello, and welcome to another lesson on page table implementations. In previous lessons we talked about how to make page tables fit in memory. In this lesson we focus on how to speed up the page table lookups.

We will start from the observation that page table lookups are very expensive. If we don’t do caching, the memory resolution (also called translation) for every logical address issued by the CPU may require multiple memory accesses.

Caching of the page table entries is therefore absolutely critical.

Virtually all CPUs today use so-called Translation Lookaside Buffers, or TLBs, to cache page table entries, and so speed up the translation process. We will briefly explore how TLBs work, and how they may differ across CPU architectures.

In most architectures, the TLB is largely hidden from the low-level programmer, just like a memory cache. In other architectures, most prominently the MIPS, the TLB is very visible and must be explicitly managed in software. We will briefly discuss how this is done.

Recall: Address Translation for Two-Level Paging

Let’s start by reminding ourselves how the address resolution, or translation, works. In this example assume a two-level page-table, with a directory and page table pages. The page table base register points to the directory.

When the CPU issues a logical address, the MMU (memory management unit) splits off the most-significant bits of the address and uses them to index into the directory to get the page directory entry.

The page directory entry is used to find the page table page for this address.

The MMU then splits off a second portion of the virtual address and uses this one to index into the page table page to look up the page table entry, which contains the frame number.

The frame number is then concatenated with the remaining part of the logical address, and the resulting physical address is used to access the physical memory.

In reality, of course, the page directory is stored in memory.

Similarly, the page table page is stored in memory as well.

The page table base register contains the start address of the page directory in memory.

When we now translate a logical memory address, we need to make one memory access to load the page directory entry.

This gives us the location of the page table page in memory.

We need to make a second memory access to load the page table entry, in order to get the frame number.

Once we have the frame number, we concatenate it with the offset, and issue the memory reference to the translated address. In summary, we make two memory accesses in order to translate one address. The overhead for accessing a location in logical memory is therefore 200% in this case. This overhead can be much higher for hierarchical page tables with more than 2 levels or for some other implementations of page tables.

Solution: Translation Lookaside Buffers

We notice that we do all this gymnastics, with multiple memory accesses, just to locate the frame number in the page table entry.

What if, instead, we could cache the frame numbers inside the MMU, rather than having to look them up in memory?

We could maintain a table that maps the page number to the frame number.

The MMU could look up a frame number by giving the page number as input, and we could get the frame number as output if such a mapping is stored in the table. Notice that for the input we don't need to split the page number up into page table number and page number. It's all one large page number.

Now when the CPU issues a memory reference, the MMU splits off the entire page number (including page table number) and checks whether there is a match in the table. If there is, the MMU reads the frame number and constructs the physical address to reference the physical memory.

The table with the mapping from page numbers to page table entries with the frame numbers is called the "Translation Lookaside Buffer". It is very central in speeding up the translation of addresses.

Memory Translation with a TLB

Let's summarize how the memory translation works, in presence of a TLB.

First, similarly to the case without TLB, the MMU splits up the address into a page number and an offset. It then uses the page number, to find a match in the TLB.

If there is a match, then the MMU uses the frame number from the entry in the TLB to construct the physical address.

If there is no matching entry in the TLB, that is, there is a TLB miss, then the MMU needs to look up the entry in the page table. That is, the MMU needs to go to memory, possibly multiple times, to locate the page table entry, and then load the entry into the TLB. Once this has occurred, the CPU can re-issue the address, and this time there will be a match.

Keep in mind that, when we bring in a new entry into the TLB, we may have to overwrite an existing entry.

Parameter of the TLB

So, how big are these TLBs, and what are their performance characteristics?

The number of entries in a TLB can vary quite by quite a lot, ranging from about a dozen to 4,000 plus entries. Even for small TLB's, the performance improvement for address translations is significant.

This is for two reasons: First, each entry in the TLB represents an entire page of memory, rather than individual memory locations. In addition, programs exhibit a significant amount of locality of reference. By this we mean that if the program referenced a set of pages in the recent past, it will likely reference the same set of pages, with small variations, in the near future. Since we have the page table entries for recent references in the TLB already, it will be unlikely that we will have many TLB misses in the near future.

What is the benefit of a TLB look up versus a full-fledged page-table lookup? Well, the lookup latency for TLB entries is about one clock cycle.

If we have to look up the entries in the page table, this can take up to 100 times longer.

Therefore, the goal is to have very low miss rates. If we have a system with a 1 clock-cycle lookup latency in the TLB and a miss penalty of 100 clock cycles, then a 1% miss rate effectively doubles the lookup latency. This is significant.

Freeing TLB Entries

In systems with hardware managed TLB's, the programmer just sees the page table. The TLB works behind the curtain, so-to-say, to speed up the page table lookups. As a result, the TLB can be largely ignored by the programmer. There are two situations where the programmer needs to be aware of the hardware-managed TLB. The first situation is when regions in logical memory are freed, and the mapping between page and frame needs to be deleted.

Let's look at the following example, where the programmer manipulates the page table in the MMU to map a page to a frame by filling in the appropriate entries in the page table.

When at some later point in time the CPU references the page, the MMU loads the page-table entry into the TLB.

Now, if after some time the system frees the page, say because the memory is released, then the mapping from the page to the frame must be deleted as well. The programmer does this by invalidating the page table entry in the page table.

This is done by overwriting the value of the page table entry in memory. Because this is just a memory operation, the TLB is not aware of it. The frame now is not mapped to a page, and we can make the frame available for use elsewhere.

Now we have a problem, because the TLB is not aware that the mapping between the page and the frame was deleted. This is a problem because, if the page is allocated again in the future, it will be most likely be mapped to a different frame.

Since the TLB is not aware of this change, it will keep mapping to the old frame.

The programmer therefore needs to explicitly delete the TLB entry. This is called "flushing" the TLB entry. Once the entry is flushed, the obsolete mapping is deleted, and the TLB is again consistent with the content of the page table. Depending on the CPU, the programmer can flush the specific TLB entry. On some CPUs she may have to flush the entire TLB.

How many TLBs?

A second situation where the TLB may become visible to the programmer is during address space switches, typically when the system switches from one process to another. This is related to whether the CPU manages one TLB per address space, or one TLB for the entire system.

Let's look at the case of one TLB per address space. In this case, the TLB maps the page number to the page table entry with the frame number. Page numbers are unique within the same address space, but not across address spaces, of course.

Therefore, if we switch from one address space to another, the programmer needs to flush the TLB. In some architectures, such as the x86, this is not a problem because, in order to switch address space, the programmer needs to load the location of the new page table into the page table base register. Loading a new value into the page table base register automatically flushes the TLB.

Alternatively, the CPU may have a single TLB system-wide. Now we need to make sure that the TLB entries are unique across all address spaces.

One way to do this is to prefix the page number with the address space ID of the page. Now, the combination of address space ID and page number is unique, and the system can operate with a single TLB.

Software-Managed TLBs: Paging - MIPS Style

Until now we have assumed that the TLB is managed in hardware and is therefore largely hidden from the programmer.

In some CPU's – mostly hard-core RISC CPUs, such as the MIPS – the content of the TLB is managed in software. In other words, it is up to the programmer to manage the content of the TLB.

Let's see how this works by observing the translation process from a logical (or virtual) address to the corresponding physical address.

First, we split up the virtual address into the page number (called virtual page number in MIPS parlance) and the address within the page, which we call the offset. The address space ID, for which we can use the process ID, is stored in a designated register and is copied from there.

In order to translate this address with help of the system-wide TLB, we prefix the page number with the address space ID. The address space ID for which we can use the process ID is stored in a designated register and is copied from there.

We use the combination of address space ID and page number to look up the entry in the TLB.

In this case we find a match.

The output of the TLB gives us (in addition to a few flags, which we don't need to care about for now) the frame number (called physical frame number in MIPS parlance).

We concatenate the frame number with the offset to get the physical address.

Let's repeat the same process for the case where we don't have a matching entry in the TLB.

This time we cannot find a match.

This causes a so-called TLB REFILL exception, which calls the operating system refill exception handler.

It is up to the programmer to have the exception handler get the page table entry from somewhere; the hardware does not care from where. For example, the programmer may maintain an array of page table entries, or a hash table, or whatever appropriate data structure to represent the page table.

In this programmer-implemented data structure the exception handler finds the page table entry, which includes the frame number. The exception handler now issues instructions to load this entry into the TLB.

The TLB refill exception handler returns, and the CPU re-issues the memory reference.

This time there is a matching entry in the TLB, and the translation continues by concatenating the frame number and the offset to get the physical address.

Memory Translation – MIPS Style

Let's summarize the important points of MIPS-style software-managed TLBs.

The fundamental principle of RISC architectures in general is to do as much as possible with as little hardware as possible.

This is the case also for memory management: There is no MMU other than the software managed TLB. Everything else, including the page table, whether it is naïve or hierarchical, or inverted, or hashed, is implemented in software.

As long as matching entries can be found in the TLB, memory references are translated by the TLB. When the TLB cannot translate, because there is no matching entry, a special TLB refill exception is raised, and the TLB refill exception handler, which is a function provided by the programmer, takes over. The handler locates in memory and loads into the TLB the matching entry. When the exception returns, the memory reference is re-issued, and now the address can be successfully translated.

software-managed TLBs: TLB Refill Exception

How would one implement a TLB REFILL exception?

The first thing such an exception would do is to figure out whether the virtual address is valid. Is the virtual address within the range of the valid addresses? If not, throw a new exception to indicate that the system tried to make an invalid memory reference. Eventually, such an exception would percolate up to the user as a "Segmentation Fault" or something similar.

If the address is valid, the exception handler needs to construct the corresponding TLB entry. For this it locates the page table entry the page table data structure, which has been defined in software. Once the new TLB entry is ready to be loaded into the TLB, an available spot in the TLB needs to be found.

If the TLB is full, an existing entry in the TLB needs to be discarded, thus making sure that there is at least one free slot in the TLB.

Now we can load the new TLB entry into one of the free slots of the TLB. The exception can now return, and when the memory reference is re-issued by the CPU, a matching entry will be found, and the translation completed.

The MIPS TLB: TLB Entry Fields

To complete the discussion of the TLB on the MIPS and in order to further de-mystify what goes on in a TLB, let's look at the structure of the MIPS TLB. The TLB in the MIPS has no more than a few dozen entries of the following kind:

Each entry has an input and an output.

The input, which is the part of the entry that is used for matching, contains the address space ID and the page number.

In addition, it contains a bit that controls whether the address ID must be considered in the matching or not. For example, operating system code may be shared across all address spaces, and so it makes no sense to match the address space ID.

On the output side we have the frame number, and a number of flags that control details of the memory reference. The "N" flag, for example, controls whether this particular memory location can be stored in the cache. Similarly, the "D" flag, controls whether the CPU can write to the page. If a write is issued to a write-protected page, an exception is thrown.

Finally, the "V" flag, for valid, is used, as we will learn in a later lesson, to indicate whether the frame is located in memory, or whether it needs to be paged in from secondary storage.

TLBs: summary

Let's summarize. In this lesson we learned about Translation Lookaside Buffers or TLBs, which are used in most CPUs to speed up the page table lookup by caching page table entries.

We observed that page table lookups require in many cases multiple memory accesses, and so they are very expensive.

One solution is to cache a sufficient number of page table entries such that the MMU only very rarely needs to go to memory to issue page table lookups. This cache is called the TLB. In practice, moderately small TLBs with a few dozen entries, are sufficient to significantly reduce the number of page table lookups that go out to memory.

In most CPUs the TLB is managed by the hardware and is largely hidden from the programmer. We learned that there are a few cases where the programmer must be aware of the TLB, most prominently during process switch and when virtual memory gets released. In CPUs with software-managed TLBs on the other hand, the programmer has to explicitly manage the entries in the TLB.

Finally, we learned how the MIPS has a rather extreme case of software-managed TLBs, where the TLB makes for the entire MMU. The hardware is limited to matching entries in the TLB and to throwing a TLB refill exception whenever no match can be found. Everything else, including the data structure that stores the page table, must be implemented by the programmer.

In the following lessons we will learn how paging works in the x86 architecture, and how the MMU can be used to implement on-demand swapping of memory to and from secondary storage if needed.

Thank you for Watching!