Memory Management Unit

From OSDev Wiki

Discourse on Memory Management Units and Virtual Memory systems in contemporary architectures

A very simple overview of the theory involved in using virtual address spaces as a general rule. Article does not focus on any one architecture, but seeks to model a generic CPU with an MMU.

A general discourse on Virtual Memory systems

As a rule, a chipset (motherboard) would tend to have N bytes of physical memory. Physical memory is "real" memory which should be globally visible to all processors. Under normal operation, or rather, when the CPU is operating without its Paged **Memory Management Unit** turned on, any address the CPU encounters will bypass the (P)MMU and go directly out onto the address bus.

I'd like to move directly into the idea of a TLB, and how "paging" works, etc. Many processor architectures of the day specify a set of behaviours that the processor will exhibit when the OS software activates the processor's PMMU. But what *is* a Memory Management Unit? A Memory Management Unit is a cache of translation information. When a processor allows multiple "virtual", independent address spaces to be used on a machine such that the CPU sees one stretch of "virtual" memory which can be mapped to any physical page, there must be some form of tabling, or other record-keeping of which physical frame each virtual page should cause the processor to eventually access on the address bus.

To clarify: A processor with a MMU that provides virtual memory has an on-chip cache of "translations". Each "translation record/entry" tells the CPU the mapping of one virtual address to one physical address. Let us imaging this on-chip cache as a big lookup array of entries that are of this form:

```
// Abstract model of a TLB.

typedef uintptr_t vaddr_t;
typedef uintptr_t paddr_t;

// Flag to mark an entry in the modelled hardware TLB as having been set
#define TLB_ENTRY_FLAGS_INUSE

struct tlb_cache_record_t
{
   vaddr_t entry_virtual_address;
   paddr_t relevant_physical_address;
   uint16_t permissions;
};

// Instance of a hardware Translation Lookaside Buffer.
struct tlb_cache_record_t   hw_tlb[CPU_MODEL_MAX_TLB_ENTRIES];
```

Your processor's TLB is essentially a hash lookup table of entries that tell what physical address each page refers to. When you enable paging, every address reference is sent out to the TLB for lookup. The CPU does something like this internally:

```
// Model routine for a TLB lookup.

int tlb_lookup(vaddr_t v, paddr_t *p)
{
   for (int i=0; i<CPU_MODEL_MAX_TLB_ENTRIES; i++)
   {
      if (hw_tlb[i].flags & TLB_ENTRY_FLAGS_INUSE && hw_tlb[i].entry_virt
      {
         *p = hw_tlb[i].relevant_physical_address;
         return 1;
      };
   };
   return 0;
}
```

If the TLB contains an entry for that virtual address, that virtual address's recorded physical address is returned. The CPU *does not* care about the actual state of the REAL translation in memory! *You* are responsible for ensuring that the information in the processor's TLB is correct. Let us pretend that our processor's TLB has an entry in it that records the virtual address 0xC0103000 as pointing to the physical address 0x11807000. Assume that your kernel has changed this information in the page tables in RAM; Your writing to physical RAM does not affect the on-chip TLB. UNLESS you tell the processor to flush that TLB entry for 0xC0103000 from its TLB, the next time 0xC0103000 is referenced, the CPU will look into the TLB, and go right on ahead to send out the physical address 0x11807000 that the TLB *says* the address corresponds to.

A processor architecture would normally, then provide an instruction to invalidate TLB entries, either en mass, or one by one, or however the CPU designers decided. Let's try to model a TLB flush. In our model CPU architecture, there is an instruction that software can issue which will invalidate one virtual address. It is called: TLBFLSH. An OS would invoke this on our model architecture by doing something like this:

```
asm volatile ("TLBFLSH   %0\n\t"::"r" (virtual_address));
```

And on to our model:

```
// Modelled function for a flush of the TLB modelled earlier on.

void tlb_flush_single(vaddr_t v)
{
   for (int i=0; i<CPU_MODEL_MAX_TLB_ENTRIES; i++)
   {
      if (hw_tlb[i].flags & TLB_ENTRY_FLAGS_INUSE && hw_tlb[i].entry_virt
      {
         ht_tlb[i].flags &= ~TLB_ENTRY_FLAGS_INUSE;
         return;
      };
   };
}
```

Now please understand that a CPU will, as long as you enable its MMU, *always* use the TLB before sending an address out onto the address bus. That is, once you enable the MMU, until you take it off, you have essentially "trapped" yourself in a virtual address space. Unless you can edit that virtual address space by editing the page tables, and invalidating the TLB entries for virtual addresses, you have no means of changing where your kernel can read from/write to in RAM. Enabling paging means that ALL of your data and instruction fetch addresses will pass through the TLB first.

The TLB *is* the MMU. Understand that. The MMU is the *TLB*. The page tables that you construct are not in any way part of the CPU's MMU. In fact, many architectures will never even look at your software contructed tables. They will only look at the TLB. How then, does the TLB get filled with entries if a processor does not walk the software constructed page tables? You put entries into the TLB. That's how.

There are two main types of MMU implementations; or rather, most MMU implementations can be seen in two broad categories: (1) Those that require software to manually edit the processor's on-chip TLB cache entries and personally ensure *full* coherency, and (2) those that only require software to invalidate stale entries, and that will of their own volition, go out of their way to search for a new entry when a virtual address is to be translated.

Those MMU implementations that scan some form of OS-constructed tables are called "Hardware Assisted TLB-Loading" enabled MMUs. As a rule, a CPU need not concern itself with trying to decide which TLB entries to fill for you; That's your job. But some CPUs are nice enough to even go ahead and scan software page-tables for and automatically gain a new translation. Now it's about time to explain what a "Translation Fault" is.

A translation fault is what occurs when a processor searches its on-chip TLB for a translation record for a virtual address and does not find one. Note well, I didn't say that a translation fault is when the CPU has searched both the TLB *and* the software-constructed page-tables. The original translation fault occurs when there is no TLB entry for a virtual address that has been referenced in the current code.

Depending on the processor architecture in question, this translation fault will cause one of two reactions:

```
(1) The processor will trap into the OS and ask it to manually search its own address-space translation records
(2) The processor will initiate a "Page table Walk" where it automatically starts walking tables from a specific
```

That is, not all processor architectures will scan for translation information for you. On architectures which will walk software-constructed translation tables, the format for these tables tends to be very strictly specified: "This bit should be here, and the physical address should be here, and X must be in spot Y, etc, etc". An example is the famous x86 processor architecture. There do exist processor architectures which will just immediately trap into the OS when there is no TLB entry for a vaddr; In this case, the OS has the liberty of deciding for itself what format it keeps process-specific translation information in; No specification will tell you how to format your per-process translation information. You are responsible for keeping track of process virtual address spaces and also for scanning that information on translation fault.

And now we should understand how MMUs work. And thus, we should understand the idea of virtual address spaces, and why you need to invalidate TLB entries, etc. Note well that there are architectures that have some very funky translation-table/page-table formats: like the PowerPC. It uses a hash table instead, which is nothing like the x86 page tables. This wikipedia article (http://en.wikipedia.org/wiki/Memory_management_unit) talks about different processors' MMU implementations.

Theory Concretion: A look at the x86 "Self-referencing Page Directory trick"

Since questions about this are asked all the time, it is best to simply explain it very clearly and get it out of the way.

This section of the article looks specifically at the x86-32 architecture and seeks to explain the "self-referencing page directory" trick. On x86-32, the processor takes after the translation fault model above which faults after *both* not finding an entry in the TLB *and* not finding a translation entry in the software constructed page tables which it walks for you since it's just nice like that. Like any other architecture with an MMU model that has hardware-assisted TLB loading (that is, the processor walks the page table for you), you are required to give the processor the address of the top level table that begins describing the address space's translations so it will know where to start walking from. This is essentially the page-directory's physical address in CR3. The processor walks and as it does so, it *interprets* the data at the addresses it finds as either Page-Directory Entries, or Page Table Entries.

Get that very solidly, please: bytes in RAM are not purposeful; they do not inherently imply any meaning. Page tables are just bytes in RAM. You could very well decide to give a page table to a network card for a DMA transfer. Just as you could put the address of a network frame into CR3 and have the processor walk that. It's very possible that your blunderous CR3 value would have, contiguous from it at the right offsets, the right bits set (PRESENT, WRITE, etc etc), and some frame address such that the CPU may even walk all the way to what it sees as a page table and find an entry to translate your fault address. This does not mean that this translation data the CPU interpreted as translation data was indeed translation data. The CPU takes your byte address in RAM and walks from there, *interpreting* those bytes as translation information.

Take the following example, then: You have a page directory at physical address 0x12345000. This page directory would of course have 1024 entries, numbered 0 to 1023. Since I want to jump into the meat of the subject, let's imagine that this page directory at 0x12345000's last entry, index 1023, points back to 0x12345000.

That is, pdir[1023] == 0x12345xxx, where 'xxx' represents the permission bits. Let us also imagine that pdir[0] points to a frame at 0x12344000, which the processor would interpret as a page table. This table has of course, it own 1024 page table entries. They map of course, the virtual addresses 0x0 to 0x3FFFFF to various frames in RAM. So right now, our example page directory looks like this:

```
[Page directory at 0x12345000]:
entry 0000 | phys: (0x12344 << 12) | perms 0bxxxxxxxxxxxx |
entry ...
entry ...
entry 1023 | phys: (0x12345 << 12) | perms 0bxxxxxxxxxxxx |

[Page table at 0x12344000 that pdir[0] points to]:
entry 0000 | phys: (0x34567 << 12) | perms 0bxxxxxxxxxxxx |
entry ...
entry ...
entry 512  | phys: (0x72445 << 12) | perms 0bxxxxxxxxxxxx |
```

This setup is such that: pdir[0] == 0x12344xxx, pdir[0], ptab[0] == 0x34567xxx, pdir[0], ptab[512] == 0x72445xxx, pdir[1023] == 0x12345xxx.

Let us simulate a page table walk to find out what physical address entry{pdir[0], ptab[512]} maps to; i.e, what physical address virtual address 0x200000 is mapped to.

1. Processor encounters an instruction that references 0x200xxx while paging is turned on. This reference must pass through the MMU.
2. Assume this entry was not found in the TLB. Fault #1 occurs. On x86-32, this causes the processor to walk the page tables rather than to trap into the OS.
3. The CPU takes the virtual address, 0x00200xxx and splits it up, 10 bits, 10 bits, and 12 bits.
4. The CPU now knows that it must index into entry 0x0 of the page directory pointed to by CR3, and then index into entry 0x200 (512) of the page table pointed to by the page directory in CR3.
5. CPU begins page table walk. The OS has written 0x12345xxx into CR3. The CPU assumes that this is the address of a valid page directory and sends out (0x12345000 + 0x0 * sizeof(pdir_entry_t)) onto the address bus as a uint32_t fetch. That computes to the physical address 0x12345000 being sent out on the address bus.
6. CPU gets the 4 bytes from the memory controller on the data bus, and interprets the 4bytes it just got from our page directory index 0 as a page directory entry.
7. CPU sees that this page directory entry is 0x12344xxx, as in our example above. 'xxx' are the permission bits. The CPU checks these, and since we're just gansta like that, they're correct, and the entry is PRESENT.
8. CPU now, having validated the permission bits, will extract the physical address from the page directory entry. It will get 0x12344000.
9. CPU now confident that 0x12344000 is the start of a page table, decides to index into this page table by computing the offset from the base, 0x12344000. It needs to get index 0x200. It then computes: (0x12344000 + (0x200 * sizeof(ptab_entry_t)), and sends the result out onto the address bus.
10. Memory controller returns the 4 bytes at 0x12344800.
11. CPU interprets this as a page table entry, and checks the bits. Once again, we are some real Gs, so the permissions bits turn out to be valid.
12. CPU is now at the leaf level, and extracts the frame address for the virtual address 0x200xxx. This turns out to be 0x72445000.
13. CPU evicts some entry from the TLB to make space for this new translation data that maps 0x200000 in the current address space to the frame 0x72445000.
14. Program execution continues. Note that translation fault number two, what people call the x86 Page Fault, never occurs since the CPU *did* find a translation from its walk.

Now that we have a firm grasp of how a page table walk works, and what the idea of "interpretation" is, let us simulate a walk of a self-referencing page directory entry.

1. Processor encounters an instruction that references 0xFFFFFxxx while paging is turned on. This reference must pass through the MMU. This is also our self-referencing entry, as you can see from the example page-table setup above.
2. Assume this entry was not found in the TLB. Fault #1 occurs. On x86-32, this causes the processor to walk the page tables rather than to trap into the OS.
3. The CPU takes the virtual address, 0xFFFFFxxx and splits it up, 10 bits, 10 bits, and 12 bits.

4. The CPU now knows that it must index into entry 0x3FF (1023) of the page directory pointed to by CR3, and then index into entry 0x3FF (1023) of the page table pointed to by the page directory in CR3.
5. CPU begins page table walk. The OS has written 0x12345xxx into CR3. The CPU assumes that this is the address of a valid page directory and sends out (0x12345000 + 0x3FF * sizeof(pdir_entry_t)) onto the address bus as a uint32_t fetch. That computes to the physical address 0x12345FFC being sent out on the address bus.
6. CPU gets the 4 bytes from the memory controller on the data bus, and correctly interprets the 4bytes it just got from our page directory index 1023 as a page directory entry.
7. CPU sees that this page directory entry is 0x12345xxx, as in our example above: this entry references itself! But the CPU does not check that; it simply checks the permissions, and gets ready to interpret that address in the page directory entry as a page table's physical address. 'xxx' are the permission bits. The CPU checks these, and since we're just gansta like that, they're correct, and the entry is PRESENT.
8. CPU now, having validated the permission bits, will extract the physical address from the page directory entry. It will gets 0x12345000 as a result of our self-referencing trick, and prepares to read *AGAIN* from the page directory, instead of reading from a page table. It will *interpret* the page directory's bytes now as a page table.
9. CPU now confident that 0x12345000 is the start of a page table, decides to index into this page table by computing the offset from the base, 0x12345000. It needs to get index 0x3FF. It then computes: (0x12345000 + (0x3FF * sizeof(ptab_entry_t)), and sends the result out onto the address bus.
10. Memory controller returns the 4 bytes at 0x12345FFC.
11. CPU interprets this as a page table entry, and checks the bits. Just like last time, since we're reading the same bytes, the permissions turn out to be valid.
12. CPU is now at the leaf level (or so it thinks), and extracts the frame address for the virtual address 0xFFFFFxxx. This turns out to be 0x12345000, since entry 0x3FF (1023) in the page directory (which the CPU is currently interpreting as a page table) points to the page directory, 0x12345000.
13. CPU evicts some entry from the TLB to make space for this new translation data that maps 0xFFFFF000 in the current address space to the frame 0x12345000.
14. Program execution continues. Note that translation fault number two, what people call the x86 Page Fault, never occurs since the CPU *did* find a translation from its walk.
15. More importantly, the program that sent out the address, 0xFFFFF000, is about the receieve the data that was at physical frame 0x12345000, since the CP walked to that extent, and has a TLB entry that maps it as such. This program can now read and write from/to the page directory simply by accessing offsets from 0xFFFFFxxx. Accessing 0x12345000 will get page directory entry 0. Accessing 0x12345004 will get entry 1, and so on.

And now that we understand how the self-referencing page table trick can be used to read from and write to the current page directory, we'll examine how to read from and write to the page tables in the current process next. A note: 0xFFFFF000 is a valid address to any program unless you map it as SUPERVISOR, or in other words, you leave the USER bit unset. Otherwise programs in userspace will be able to edit their own page tables. Imagine a program deciding to start mapping pages in its address space to the kernel at physical address 0x100000? It could very well now decide to zero out the kernel in RAM.

Retrieved from "http://wiki.osdev.org/index.php?title=Memory_Management_Unit&oldid=17446"
Category: Memory management

- This page was last modified on 1 January 2015, at 14:22.
- This page has been accessed 15,093 times.