Chapter 17

Memory Management and Virtual Memory

You’ve already seen MMUs, TLBs, and basic paged virtual memory operations in the Systems Programming and Computer Architecture course at ETHZ. Here we take this a bit further.

17.1 Segments

Before paging, there were segments. Segments evolved from basic protection mechanisms. Today they are little used (but still present on x86 machines), although there is some evidence they are making a comeback as memories become very large.

Definition 17.1 (Base and Limit). A base and limit register pair is a couple of hardware registers containing two addresses B and L. A CPU access to an address a is permitted iff $B \leq a < L$.

Remarks:

- Base and limit registers were the first, primitive form of MMU, and at first operated entirely on physical addresses (as did user code). Each process in the system had its own base-and-limit region. To context switch, the OS reprogrammed the base and limit registers.
- Since the base address is not known to the program at compile time, programs had to be compiled to position-independent code, or a relocation register used.
- Base and limit registers still live on in the x86 architecture as a way of constraining virtual addresses.

Definition 17.2 (Relocation register). A relocation register is an enhanced form of base register. All CPU accesses are relocated by adding the offset: a CPU access to an address a is translated to $B + a$ and allowed iff $B \leq B + a < L$. 

172
Remarks:

- With relocation registers each program can be compiled to run at the same address (e.g. 0x0000).
- Relocation registers don’t allow sharing of code and data between processes, since each process has a single region of memory (and a different relocation address).
- Relocation registers allow a primitive form of swapping: a process’ single memory region can be written out to storage, and later read back in to a different physical address and resumed.

The sharing problem with relocation registers was solved by generalizing base/limit regions, and making them independent of particular processes. The result is segments.

**Definition 17.3 (Segments).** A segment is a triple \((I, B_I, L_I)\) of values specifying a contiguous region of memory address space with base \(B_I\), limit \(L_I\), and an associated segment identifier \(I\) which names the segment. Memory in a segmented system uses a form of logical addressing: each address is a pair \((I, O)\) of segment identifier and offset. A load or store to or from a logical address \((i, o)\) succeeds iff \(0 \leq o < L_i\) and the running process is authorized to access segment \(i\). If it does succeed, it will access physical address \(B_i + o\).

Remarks:

- In practice the segment identifier for an address might be specified explicitly in machine instructions, indirectly via a special processor “segment register”, or explicitly in the address value used.

**Definition 17.4 (Segment Table).** A Segment Table is an in-memory array of base and limit values \((B_i, L_i)\) indexed by segment identifier, and possibly with additional protection information. The MMU in a segmentation system holds the location and size of this table in a segment table base register (STBR) and segment table length register (STLR). Logical memory accesses cause the MMU to look up the segment id in this table to obtain the physical address and protection information.

Remarks:

- Segmentation is fast. As with pages, segment information can be cached in a TLB (and locality is much higher than with pages since segments are large and contiguous).
- Sharing is trivially easy at a segment granularity. If segments are allowed to overlap, finer-grained sharing is possible.
- The OS and hardware might have a single, system-wide segment table, or (more often) a per-process table. Context switching is fast: it just involves rewriting the STBR and STLR.
- Segment identifiers can be virtualized by having per-process segment tables indirect into a single, system-wide table. Protection is therefore applied at each process’ level, while translation of addresses is the same everywhere.
• The principal downside of segmentation is that segments are still contiguous in physical memory, which leads to external fragmentation (recall this from Systems Programming).

### 17.2 Paging

As we should already know from Systems Programming, paging solves the external fragmentation problem associated with segments, at some cost in efficiency.

**Definition 17.5 (Paging).** A paging system divides the physical address space into fixed-size regions called frames or physical pages, indexed by a physical frame (or page) number (PFN), the high bits of the address. The virtual address space is similarly divided into fixed-size regions called virtual pages, identified by virtual page number (VPN) the high bits of the virtual address. The MMU translates from virtual addresses to physical addresses by looking up the VPN in a page table to obtain a PFN. Page Table Entries (PTEs) also hold protection metadata for the virtual page, including validity information. Access via an invalid PTE causes a Page Fault processor exception. VPN-to-PFN translations are cached in a Translation Lookaside Buffer (TLB).

**Remarks:**

• Virtual pages and physical pages are almost always the same size, and are always a power-of-two in size (often 4kB).

• Segmentation and paging can be combined. Either segments become regions of virtual memory, which is then paged, or each segment itself is divided into a set of pages and paged itself.

**Definition 17.6 (Linear page table).** A Linear Page Table is a page table implemented as a simple array of PTEs, indexed by VPN.

**Remarks:**

• Linear page tables work great when there aren’t many pages – early systems often had only 8 pages per virtual address space, and sometimes simply used registers to implement the page table.

• Linear page tables grow with the size of the virtual address space, and become very large on 32-bit and 64-bit machines. They’re generally not used in modern machines.

• The DEC VAX, a 32-bit machine, did use a linear page table scheme, but ingeniously dealt with the size of the table by placing it in virtual memory [Leo87].

**Definition 17.7 (Hierarchical page table).** A Hierarchical Page Table organized in multiple layers, each one translating a different set of bits in the virtual address.

**Example 17.8 (64-bit x86 paging [Int18]).** x86 has a page size of 4096 bytes, which can be addressed using 12 bits. It happens that every page table used in x86 is also 4kB in size, and virtual addresses are 48 bits wide.
The page table has four levels, each one translating 9 bits of the virtual address leaving 12 left over for the page offset. Since each entry is 64 bits or 8 (i.e. \(2^3\) bytes) in size, to translate \(2^9\) or 512 addresses requires \(2^{9+3}\) or 4096 bits of page table – a page.

The topmost level is called the Page Map Level 4 or PML4, and translates bits 58-47 of the virtual address to (among other things) the virtual address of the next level page table, the Page Directory Page Table or PDPT. This in turn points to a Page Directory or PD, and thence to a Page Table proper or PT. This then gives the physical address of the frame.

Remarks:

- Most virtual address spaces for 64-bit machines are less than the full 64 bits in size. The address is typically sign-extended to 64 bits.
- Many architectures support two page table base registers, one for the upper portion of the virtual address space and one for the lower. This allows the kernel to run in virtual memory (using one page table) without needing to swap page tables for the kernel on a context switch.
- Not all architectures have the “everything is the size of a page” property of x86. For example, ARM page tables are very different sizes at different levels, though always a power of two bytes.

Definition 17.9 (Page table walking). The process of translating a virtual address to a physical address using a hierarchical page table is called a Page Table Walk. Most processor MMUs have a hardware table walker which is used on TLB misses to find and load the appropriate page table entry into the TLB, but others (like the MIPS [KH91]) require the OS to provide a software page table walker.

In Systems Programming, we only looked at how a hardware walker and a TLB use a hierarchical page table to map virtual addresses to physical addresses. In practice, software in the operating system always has to map virtual to physical addresses:

- A software-loaded TLB simply has no hardware to walk the page table.
- In an case, when a page fault occurs, the OS needs to map the faulting (virtual) address to the relevant PTE, so that it can figure out what kind of fault occurred and also where to find a physical page to satisfy the fault. This requires a table walk.
- When a user process \texttt{unmaps} a region of virtual memory (including when the process exits), the OS has to identify all the physical pages associated with that mapping.

And so on. The OS can do this in two ways: first, it can walk the actual hardware page table in software. Alternatively, it can keep a “shadow page table” structure which is independent of the hardware page table. This might be easier to walk, and might occupy less space (the hardware table might even be constructed lazily from this structure in response to early page faults).
However, the OS also needs to map physical addresses to virtual addresses as well. For example, when a physical page needs to be reused, once it is clean, the OS needs to identify every virtual mapping to the page and mark them invalid. This requires a data structure for reverse mapping as well. The bookkeeping involved is considerable.

**Definition 17.10** (Virtual memory region). To facilitate tracking the mappings between virtual and physical addresses, an OS typically divides an address space into a set of contiguous virtual memory regions.

**Remarks:**

- These include the “segments” of the process – not those in Definition 17.3, but the text, data, bss, stack, etc. segments used by the program loader.

- Regions also provide a convenient unit of sharing between address spaces, either at the same virtual address in all cases, or backed by a memory object that can appear at different virtual addresses in different process address spaces.

### 17.3 Segmented paging

It is possible to combine segmentation and paging.

**Definition 17.11** (Paged segments). A paged segmentation memory management scheme is one where memory is addressed by a pair (segment id, offset), as in a segmentation scheme, but each segment is itself composed of fixed-size pages whose page numbers are then translated to physical page numbers by a paged Memory Management Unit.

**Remarks:**

- This is the approach adopted by Multics [Org72], but it’s also supported (though little used) by older 16-bit and 32-bit x86 processors [Int18]. In the 64-bit x86 architecture, the segmentation registers used for this still exist, but simply hold linear offsets into a flat, paged address space.

### 17.4 Page mapping operations

Each process has its own page table. At a high level, all operating systems provide three basic operations on page mappings, which in turn manipulate the page table for a given process:

**Definition 17.12** (Map). A page map operation on an address space $A$: $A$.map$(v, p)$

- takes a virtual page number $v$ and a physical page (or frame) number $p$, and creates a mapping $v \rightarrow p$ in the address space.
Definition 17.13 (Unmap). A page unmap operation on an address space $A$:

$$A.unmap(v)$$

- takes a virtual page number $v$ and removes any mapping from $v$ in the address space.

Definition 17.14 (Protect). A page protect operation on an address space $A$:

$$A.protect(v, rights)$$

- takes a virtual page number $v$ and changes the page protection on the page.

An MMU typically allows different protection rights on pages. The rights are specified in the page table entry, so they apply to virtual addresses. For example:

- READABLE: the process can read from the virtual address
- WRITEABLE: the process can write to the virtual address
- EXECUTABLE: the process can fetch machine code instructions from the virtual address

An OS will typically change protection rights on pages not only to protect pages, but also to cause a trap to occur when the process accesses a particular page in a particular way.

### 17.5 Copy-on-write

Recall that the `fork()` operation in UNIX makes a complete copy of the address space of the parent process, and that this might be an expensive operation.

Definition 17.15 (On-demand page allocation). To avoid allocating all the physical pages needed for a process at startup time, on-demand page allocation is used to allocate physical pages lazily when they are first touched.

Algorithm 17.16 On-demand page allocation

**inputs**

- $A$ {An address space}
- $\{(v_i), i = 1 \ldots n\}$ {A set of virtual pages in $A$}

**Setup the region**

for $i = 1 \ldots n$ do

- $A.unmap(v_i)$ {Ensure all mappings in region are invalid}

end for

**Page fault**

**inputs**

- $v'$ {Faulting virtual address}
- $n \leftarrow \text{VPN}(v')$
- $p \leftarrow \text{AllocateNewPhysicalPage}()$
- $A.map(v_n \rightarrow p)$

**return**
Remarks:

- This technique uses the MMU to *interpose* on accesses to memory, and cause a trap when memory is touched for the first time. After the trap, the virtual memory has been backed and the cost of the trap doesn’t occur again.

- The general technique of “trap and fix things up” is very flexible. In particular, it is used to make `fork()` extremely efficient on modern machines via *copy-on-write*.

**Definition 17.17 (Copy-on-Write).** *Copy-on-write* or COW is a technique which optimizes the copying of large regions of virtual memory when the subsequent changes to either copy are expected to be small.

**Algorithm 17.18 Copy-On-Write**

1. **inputs**
   2. $A_p$ {Parent address space}
   3. $A_c$ {Child address space}
   4. $\{(v_i, p_i), i = 1\ldots n\}$ {A set of virtual to physical mappings in $A_p$}

**Setup**

5. for $i = 1\ldots n$ do
   6. $A_p$.protect($v_i$, READONLY )
   7. $A_c$.map($v_i \rightarrow p_i$ )
   8. $A_c$.protect($v_i$, READONLY )
   9. end for

**Page fault in child**

10. **inputs**
   11. $V$ {Faulting virtual address }
   12. $n \leftarrow VPN(V)$
   13. $p' \leftarrow$ AllocateNewPhysicalPage()
   14. CopyPageContents($p' \leftarrow p_n$ )
   15. $A_c$.map($v_n \rightarrow p'$ )
   16. $A_p$.protect($v_n$, WRITEABLE )
   17. **return**

Remarks:

- Essentially, we share the original physical pages between both address spaces, and mark both mappings read-only. When either process writes to the region, a protection fault will occur. When it does, we identify the page, allocate a new physical page, copy the contents across, and replace its mapping in the faulting process. Finally, we mark the original page mapping in the other process as writeable.

- There is no need for the processes involved to be parent or child, or for there to be only two of them. The technique is widely used, but the canonical example for this is copying the entire process address space in `fork()`.
17.6 Managing caches

We’ve already seen in Systems Programming how caches work, and how cache coherence protocols ensure varying levels of consistency among multiple caches. What about how the OS manages caches? Before looking at why and how the OS needs to manage the processor caches, let’s go through the operations it can use from software on a cache.

**Definition 17.19 (Cache Invalidate).** An **invalidate** operation on a cache (or cache line) marks the contents of the cache (or line) as invalid, effectively discarding the data.

**Example 17.20.** The x86 `invd` instruction invalidates all a processor’s caches.

**Example 17.21.** The ARMv8-A `dc iaw` instruction invalidates a specific set and way of the core’s cache. `dc ivac` invalidates any line holding a specified virtual address “to the Point of Coherency”.

Remarks:
- Invalidate operations generally don’t care about whether there is dirty data in the cache line or lines being invalidated — it just gets thrown away regardless. However, different hardware vendors use the word in different ways, so it’s a good idea to check.

**Definition 17.22 (Cache Clean).** A **clean** operation on a cache writes any dirty data held in the (write-back) cache to memory.

**Example 17.23.** The ARMv8-A `dc vac` instruction writes dirty data in any line holding a specified virtual address back “to the Point of Coherency”. It may, in addition, invalidate the line.

Remarks:
- “Clean” is a term usually associated with the ARM architecture, but it is at least unambiguous in that it leaves the cache (or line) clean, but not necessarily invalid. Rather more ambiguous is...

**Definition 17.24 (Flush).** A **flush** operation writes back any dirty data from the cache and then invalidates the line (or the whole cache).

**Example 17.25.** The x86 `wbinvd` instruction flushes all a processor’s caches. The `clflush` instruction, in contrast, flushes a single cache line.

Remarks:
- This is a reasonable definition of “Flush”, and is at least consistent with the use of the term in Intel, ARM, and MIPS documentation (and most OS people’s heads). However, don’t assume when you read “flush” in some documentation that it always means this precisely.
- ARM do use this term, though when it really matters for clarity they write “clean-and-invalidate”, which is unambiguous.

Given that caches are supposed to be transparent to software, though, why would you need these operations? Let’s look at the kinds of problems that occur, and then survey the types of caches we have as well.
17.6.1 Homonyms and Synonyms

Definition 17.26 (Cache synonyms). **Synonyms** are different cache entries (virtual addresses) that refer to the same physical addresses. Synonyms can result in cache **aliasing**, where the same data appears in several copies in the cache at the same time.

Remarks:

- Synonyms cause problems because an update to one copy in the cache will not necessarily update others, leaving the view of physical memory inconsistent.
- Worse, in a write-back cache, a dirty synonym may only be written back much later, causing an unexpected change to the contents of main memory. In a virtually-tagged cache (see below), this can even happen after virtual-to-physical page mappings have changed, creating a write to an arbitrary physical page. This is sometimes called a **cache bomb**.

Definition 17.27 (Cache homonyms). **In contrast, cache homonyms** are multiple physical addresses referred to using the same virtual address (for example, in different address spaces).

Remarks:

- Homonyms are a problem since the cache tag may not uniquely identify cache data, leading to the cache accessing the wrong data. They are a common problem in conventional operating systems. Whether homonyms and synonyms are actually a problem the OS has to deal with depends on the cache hardware in use:

17.6.2 Cache types

These days, almost all processor caches are write-back, write-allocate, and set-associative. The main differences (other than size, and inclusivity) are to do with where the tag and index bits come from during a lookup.

Definition 17.28 (virtually-indexed, virtually-tagged). A **virtually indexed**, **virtually tagged** or VIVT cache (sometimes, ambiguously, also called a “virtual cache”) is one where the virtual address of the access determines both the cache index and cache tag to look up.

Remarks:

- VIVT caches are simple to implement, and fast. However, they suffer from homonyms: since the same virtual address in different address spaces refers to different physical addresses, it may be necessary for the OS to flush the cache on every address space switch, which is expensive. Early ARM processors had this “feature”, for example.
- The problem can be alleviated by allowing cache entries to be annotated with “address space tags”.
17.6. MANAGING CACHES

Definition 17.29 (Address space tags). Address space tags, or ASIDs (Address Space Ids) are small (e.g. 5-8 bits) additional cache or TLB tags which match different processes or address space and therefore allow multiple contexts to coexist in a cache or TLB at the same time.

Remarks:

• ASIDs improve performance because the cache doesn’t need to be flushed on a context switch. In practice, VIVT caches are rare (see VIPT caches below), so their real impact is in TLBs, where they are almost ubiquitous these days.

• In practice, the “current” ASID is held in a register in the CPU or MMU which is altered by the OS kernel every time the address space switches. For example, in the MIPS R3000 this is the EntryHi register in coprocessor CP0 [?].

• ASIDs are typically small, from 5 to 8 bits in size depending on the processor architecture. Obviously there are likely to be rather more processes active in a typical system. As a result, ASIDs should be regarded as a cache for true process or address space IDs: if the OS needs to dispatch a process which doesn’t yet have an ASID assigned to it, it needs to reallocate an ASID, flush the cache to make sure no older entries for that ASID exist, and then dispatch the process.

• In the cast of a VIVT cache, ASIDs don’t solve the further problem of synonyms: if two processes share physical memory, the same data can appear in the cache twice, making it hard to retain consistency between these two copies. For this reason, VIVT caches tend to be write-through.

Definition 17.30 (Physically-indexed, Physically-tagged). A physically-indexed, physically-tagged (PIPT) cache is one where the physical address after TLB translation determines the cache index and tag.

Remarks:

• PIPT caches are easier to manage, since nothing needs to change on a context switch, and they do not suffer from homonyms or synonyms.

• The downside is that they are slow: you can only start to access a PIPT cache after the TLB has translated the address. For this reason, they are a common choice of L2 or L3 caches.

Definition 17.31 (Virtually-indexed, Physically-tagged). A virtually-indexed, physically-tagged (VIPT) cache is one where the virtual address before TLB translation determines the cache index, but the physical address after translation gives the tag.
Remarks:

- VIPT caches are a great choice for L1 D-caches these days, if they can be made to work. Ideally, the virtual index is exactly what the corresponding physical index would be. This implies that the number of index bits, plus the number of cache offset bits, in a virtual address is less than the number of bits in the page size. This in turn restricts the size of a VIPT cache.

- If this condition doesn’t hold, a VIPT cache can still work, but now the OS has to be careful about cache homonyms arising from the ambiguity now present in the cache contents. For example, in the MIPS R4400, the OS has to ensure that even and odd pages are allocated together and there is never a mapping from an even VPN to an odd PPN, or vice versa, since the cache index+offset bits are 1 bit larger than the page size.

Definition 17.32 (Physically-indexed, Virtually-tagged). A physically-indexed, virtually-tagged (PIVT) cache is one where the physical address after TLB translation determines the cache index to look up, but the virtual address before translation supplies the tag to then look up in the set.

Remarks:

- It is hard to imagine anyone building such a cache, and even harder to figure out why, but they do exist: the L2 cache on the MIPS R6000, for example, is PIVT. This requires considerable complexity on the part of the OS, and delivers questionable benefit. Thankfully, PIVT caches are extremely rare.

17.7 Managing the TLB

Like caches, the TLB is supposed to be transparent to user programs, although in practice as you have seen in Systems Programming it can be critical to know the TLB’s dimensions to improve performance, for example by tiling, blocking, and buffering large dense matrices.

Definition 17.33 (TLB coverage). The **TLB coverage** of a processor is the total number of bytes of virtual address space which can be translated by a TLB at a given point in time.

Example 17.34. The venerable Motorola MC68851 Paged MMU has a 64-entry fully-associative TLB. When configured with a page size of 8k bytes, the coverage of this TLB is $64 \times 8 = 512k$ bytes.

Remarks:

- Modern processors are quite complex, with multiple TLBs: primary and secondary TLBs (as with L1 and L2 caches), separate TLBs for instructions and data, and even different TLBs for different page sizes (regular pages, superpages, etc.).
However, looking purely at the regular page-sized TLB entries, TLB coverage has barely increased over the last 30 years. Even as far back as 2002, it was observed that TLB coverage as percentage of main memory size was dropping precipitously \cite{NIDC02}. In some ways this is not too surprising: TLBs are highly-associative caches that need to be really fast, and Content-Addressable Memory circuits (CAMs) are very energy-inefficient, but it’s a serious factor in the performance of computer software over time.

Regular processor caches on a multiprocessor machine typically try to remain coherent, in other words, they present a consistent view of the contents of main memory to the processors in the system. TLBs, by analogy, present a view of the structure of an address space to the different cores in a multiprocessor system, and so also need to be kept coherent. This this view changes less frequently than the contents of memory, it is done in software by the OS when mappings change.

**Definition 17.35** (TLB shootdown). TLB *shootdown* on a multiprocessor is the process of ensuring that no out-of-date virtual-to-physical translations are held in any TLB in the system following a change to a page table.

**Remarks:**

- Since the TLB is a cache, it should be coherent with other TLBs in the system, and with each process’ page tables.

- Shootdown is basically a way to invalidate certain mappings in every TLB, if they refer to the affected process.

- Sometimes hardware does this, but more common is for one core to send an *inter-processor interrupt* to every other core which might have such a mapping, and have the kernel on that core remove the mappings from its own TLB.

- TLB shootdown is expensive, and so operating systems try to make it faster by, for example, trying to track which TLBs might be affected by a change in a page table. If the OS can prove that given TLB cannot hold a mapping, then it doesn’t need to interrupt that TLB’s processor when the mapping needs to be invalidated.

**Bibliography**


