Paging: Faster translations (TLBs)

Recall the problem,

Question: How to speed up address translation and generally avoid the extra memory reference?

↳ using hardware support

"Translation lookaside Buffer (TLB)"

today's focus!

A TLB is a part of chip's Memory-management unit (MMU), and is simply a cache of popular virtual-to-physical address translations.

For each memory reference, hardware do the following.

Check TLB
- if translation present → perform the translation (no additional memory reference)
- o/w → access the page table
  - ↓
  - bring the corr. PTE into TLB
    - ↳ retry the instruction

```
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()
```

TLB control flow algorithm of a hardware-managed TLB.

Idea behind TLB: if the address found in TLB → TLB hit.
(less overhead)

O/W → TLB miss
(extra memory reference needed)

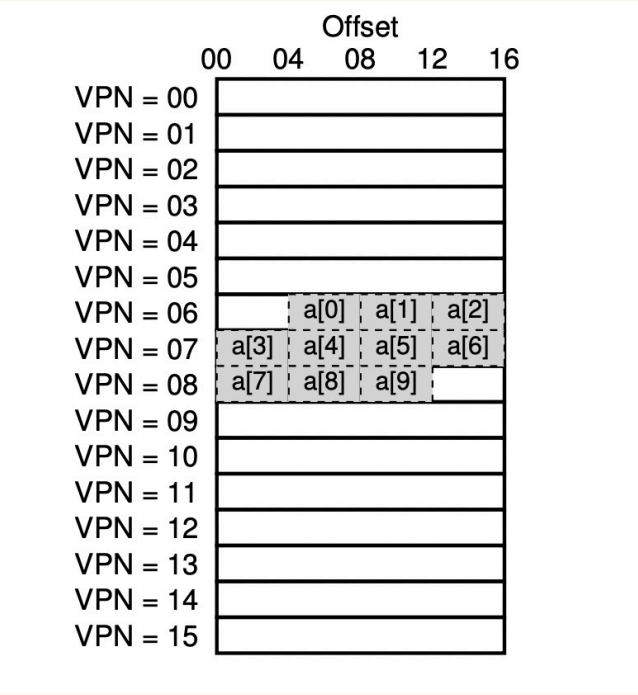
Goal: avoid TLB misses as much as possible.

example: Assume that we have an array of 10 4-byte integers in memory starting at virtual address 100.

Assumptions: virtual address space → 8 bits → 256 bytes.
page size → 16 bytes ⇒ offset = 4 bits

⇒ # pages $= \frac{2^8}{2^4} = 2^4 = 16$ pages

bits for VPN = 4 bits



a[0] → VA = 100 → 0110, 0100

VPN = 6 offset = 4

consider the foll. code that accesses each array element.

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

first access: a[0] → VA 100 (VPN = 6, offset = 4)

Check TLB → TLB miss → Insert in TLB (Translation of VPN 6)

second access → a[1]: VPN = 6, offset = 8

Check TLB → TLB hit

Similarly a[2] → hit a[3] → miss (Translation of VPN 7) a[4], a[5], a[6] → hit

a[7] → miss (Translation of VPN 8) a[8], a[9] → hit.

Summary: out of 10 access → 7 hits and 3 miss

⇒ 70% TLB hit rate

Why high TLB hit rate?

* Spatial locality: A program accessing elements of the memory which are placed close to each other.

⇒ only the first access to an element in a page yields a TLB miss.

⇒ larger size → high hit rate.

a typical page size is like, 4KB. In that case, even if the array size

is 2048, we might end up in having only 2 TLB misses (if a[0] is at the starting of the first page) $\Rightarrow$ Hit rate = $\frac{2046}{2048} \times 100 = 99.9\%$,

which is pretty good!

* Temporal locality: if the program accesses the same memory locations frequently, then it will result in fewer TLB misses.

So, if the program has the above two properties then we have high TLB hit rates.

What happens in case of TLB miss?

Who handles a TLB miss
- → Hardware managed TLB
  eg:- Intel x86 architecture
- → Software managed TLB
  eg:- MIPS/SPARC/RISC-V

On a TLB miss, hardware only raises an exception

Exception happens → current execution is paused → kernel mode → trap handler → update TLB → return-from-trap → retry instruction

here, slightly different from usual return-from-trap as the execution needs to resume from the instruction that caused the trap (not the next one)

```
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     RaiseException(TLB_MISS)
```

TLB control flow algorithm of an OS handled TLB

Hardware handled TLB

software handled TLB

How it works?

On a TLB miss, the CPU hardware (page table walker) walks the page tables in memory, finds the right PTE and inserts into TLB.
(needed in multi-level page tables which we will see later)

On a TLB miss, the CPU traps into the OS, and the OS trap handler explicitly finds the PTE in memory and writes the entry into the TLB.

Advantages

- Fast miss handling
(no OS trap/interrupt needed)

- simplicity of hardware
(CPU doesn't need a page table walker)

- transparent to OS
(OS just maintains page tables, hardware does the rest)

- flexibility for OS
(OS can define page-table format, replacement policy, etc..)

Disadvantages

- Hardware complexity
(needs dedicated logic (like, page-walker) increasing CPU design cost and power)

- less flexibility
(OS can't easily change TLB replacement policy or page-table format - it must follow the hardware defined scheme)

- high miss penalty
TLB miss → trap → OS handler → insert entry → return to user
(slower than hardware)

- OS burden

Note that, in software handled TLB systems, (like, MIPS), the OS has full freedom to design its own mapping structures.
- linear page tables, inverted page tables, hash-based structures, etc..

no fixed page table format ⇒ no point in a base register!

Caution! The trap handler should not cause an infinite chain of TLB misses!

How this happens? Imagine a system where:

virtual address → needs translation → TLB miss → go to page table.

But page table itself is stored at some virtual address (not physical)

Then to access PTE, the OS/hardware tries to translate the pagetable's virtual address. That lookup itself causes a table miss, and the chain continues forever....

How OS avoids the possibility of infinite recursion?

- it may keep TLB miss handlers in physical memory (where they are unmapped and not subject to address translation)

- or reserve some entries in the TLB for permanently-valid translations and use some of those permanent translation slots for the handler code itself (these wired translations always hit in the TLB).