15. Translation Lookaside Buffers

Using paging to support virtual memory can lead to high performance overheads.

by chopping the address space into small, fixed-sized units(=pages), paging requires a large amount of mapping information.

Because that mapping information is generally stored in physical memory, paging logically requires an extra memory lookup for each virtual address generated by the program.

Going to memory for translation information before every instruction fetch or explicit load or store is prohibitively slow.

HOW can we speed up address translation, and generally avoid the extra memory reference that paging seems to require?

When we want to make things fast : the OS usually needs some help from the hardware

To speed address translation => a translation-lookaside buffer = TLB

Translation lookaside buffer (TLB)

- a part of the chip's memory-management unit (MMU)
- a hardware cache of popular virtual-to-physical address translations

=> address-translation cache

Upon each virtual memory reference, the hardware first checks the TLB to see if the desired translation is held therein.
= if so > the translation is performed (quickly) without having to consult the page table (which has all translations)

Use caching when possible

make the "common-case fast"
the idea behind hardware caches is to take advantage of locality
in instruction & data references.
these properties depend on the exact nature of the program

Hardware caches (as in TLB) take advantage of locality by keeping copies of memory in small, fast on-chip memory.
Instead of having to go to a slow memory to satisfy a request,
the processor can first check if a nearby copy exists in a cache.

If you want a fast cache, it has to be small, as issues like the speed-of-light and other physical constraints become relevant.
Any large cache by definition is slow.

temporal locality = re-referencing of memory items in time.

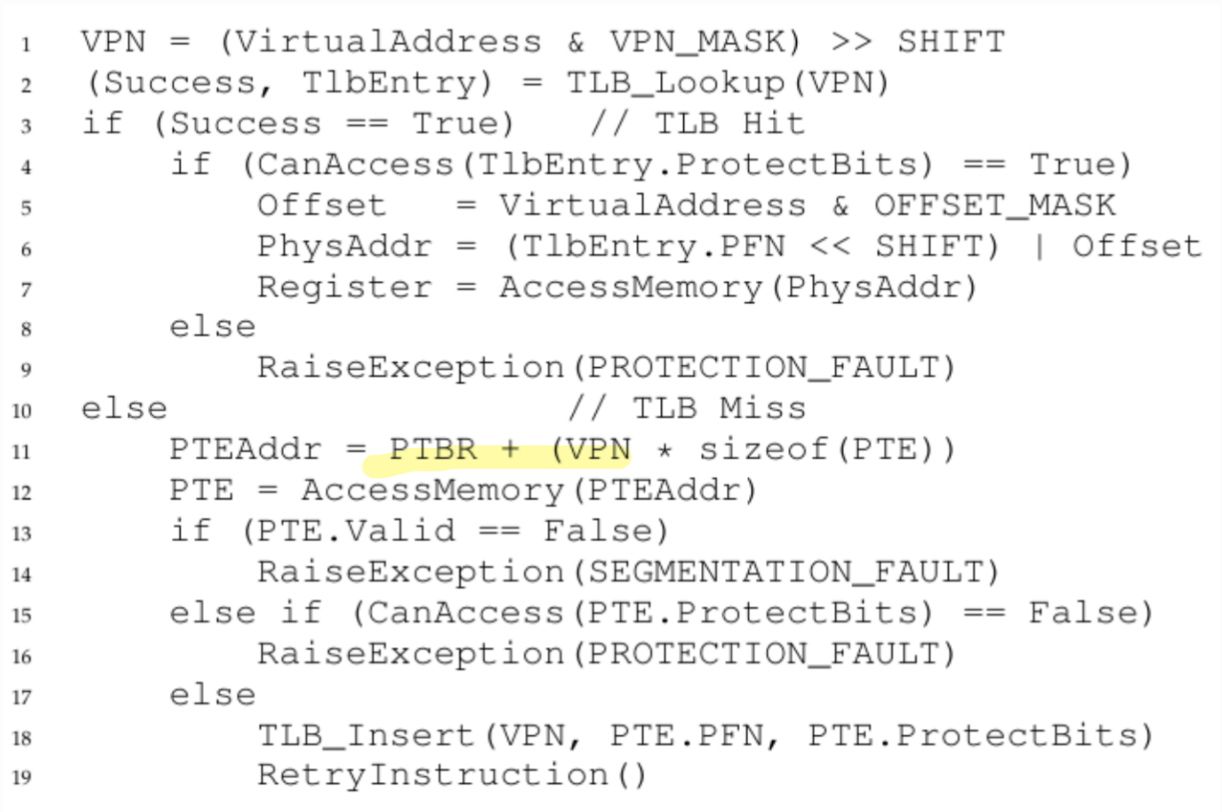
Who Handles the TLB Miss?

Hardware-managed TLBs

in the olden days, the hardware had complex instruction sets and the people who built the hardware didn't much trust those sneaky OS people.

=> hardware would handle the TLB miss entirely.

to do this, 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



```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
    else
        TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
        RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

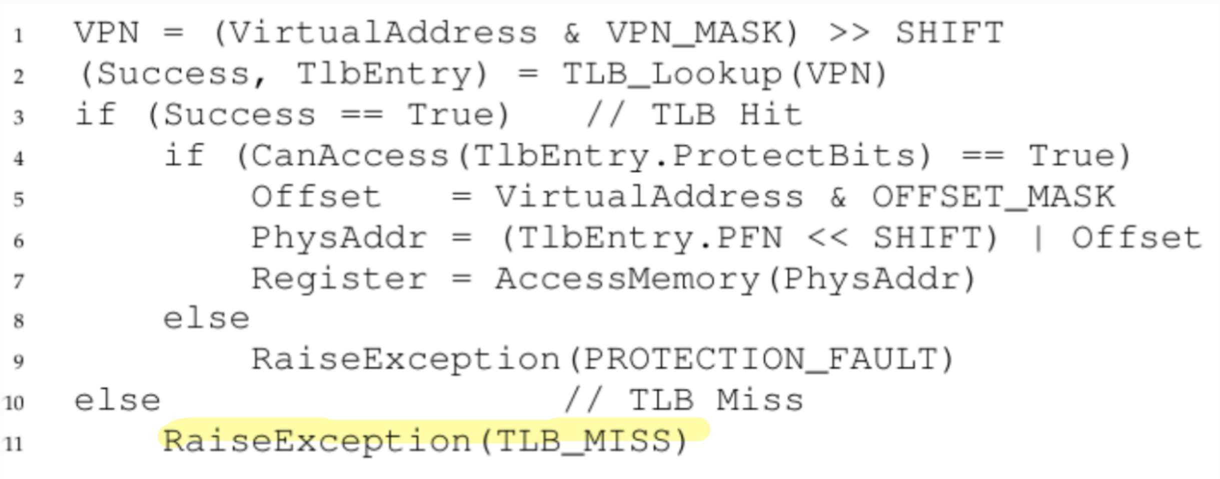
on a miss, the hardware would
1. "walk" the page table *why "walk"? it's indexed.
2. find the correct page-table entry
3. extract the desired translation, update the TLB with the translation
4. retry the instruction

*Because modern systems use **multi-level page tables** to save memory.

Software-managed TBL

more modern architectures

On a TLB miss, the hardware simply raises an exception, which



```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)    // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                      // TLB Miss
    RaiseException(TLB_MISS)
```

Figure 19.3: **TLB Control Flow Algorithm (OS Handled)**

- pauses the current instruction stream,
- raises the privilege level to kernel mode,
- jumps to a trap handler

trap handler

- code within the OS
- written with the express purpose of handling TLB misses

When this trap handler run,

1. the code will lookup the translation in the page table
2. use special "privileged" instructions to update the TLB
3. return from the trap
4. at this point, the hardware retries the instruction (resulting in a TLB hit)

TLB miss-handling trap : return-from-trap is special

Usually, in a system call, the return-from-trap should resume execution at the instruction **after** the trap into the OS,

just as a return from a procedure call returns to the instruction immediately following the call into the proedure.

when returning from a TLB miss-handling trap, the hardware must resume execution at the instruction that **caused** the trap
=> this retry thus lets the instruction run again, this time resulting in a TLB hit.

=> depending on how a trap or exception was caused, the hardware must save a different PC when trapping into the OS, in order to resume properly

When running the TLB miss-handling code,

the OS needs to be extra careful not to cause
an infinite chain of TLB misses to occur (when TLB miss-handling code is not in TLB).

Solution

1. keep TLB miss handlers in physical memory (where they are unmapped and not subject to address translation)
2. 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

Adventage of software-managed approach

flexibility : the OS can use any data structure it wants to implement the page table, without necessitating hardware change.

simplicity : the hardware doesn't do much on a miss: just raise an exception and let the OS TLB miss handler do the rest.

TLB Valid Bit != Page Table Valid Bit

page-table entry (PTE) is marked valid : the page hasn't been allocated by the process, and should not be accessed by a correctly-working program.

TBL valid bit :
simply refers to whether a TLB entry has a valid translation within it.
common initial state for each TLB entry is to be set to invalid.

TLB contents : What's in there?

TLB is fully associative
any given translation can be anywhere in the TLB
=> hardware will search the entire TLB in parallel to find the desired translation.

TLB entry : VPN | PFN | other bits

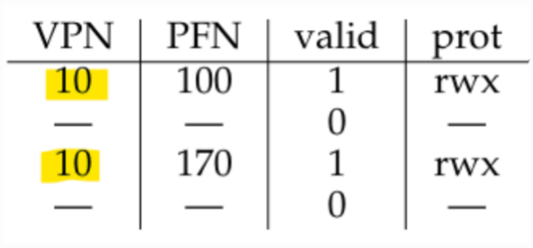
other bits

- vaild bit : whether the entry has a valid translation or not
- protection bits : determine how a page can be accesssed (read/write/exec)
- address-space identifier
- dirty bit

TLB Issues : Context Switches

the TLB contains virtual-to-physical translations that are only valid for the **currently running process**

these translations aren't meaningful for other processes.
=> when switching from one process to another, the hardware or OS (or both) must be careful to ensure that the about-to-be-run process doesn't accidentally use translations from some previously run process.



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

- 2 processes using each 10th Virtual Page => VPN 10 translates to ??, hardware can't distinguish

Solutions

1. flush the TLB on context switches
-> emptying it before runnning the next process. : simply sets all valid bits to 0

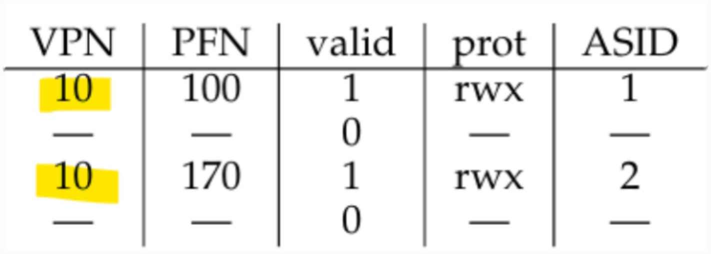
- software-based system : accomplished with an explicit (& privileged) hardware instruction
- hardware-managed TLB : the flush could be enacted when the page-table base register is changed (note that OS must change the PTBR on a context switch anyhow)

Cost : each time a process runs, it must incur TLB misses as it touches its data and code pages.
If the OS switches between processes frequently, this cost may be high.

To reduce this overhead, some systems add hardware support to enable sharing of the TLB across context switches :

2. address space identifier (ASID) / field in the TLB.
like a process identifier (PID), but usually it has fewer bits

only the ASID field is needed to differentiate otherwise identical translations.

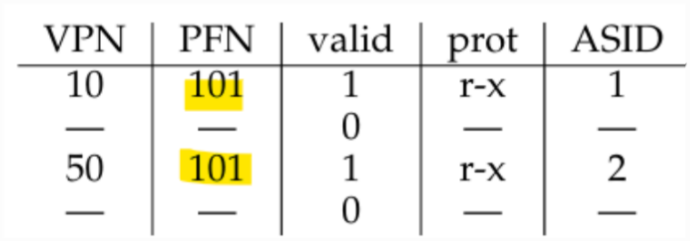


VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

with ASID, the TLB can hold translations from different processes at the same time without any confusion.

the hardware also needs to know which process is currrently running in order to perform translations, and thus the OS must, on a context switch, set some privileged register to the ASID of the current process.

2 different VPNs that point to the same physical page:



VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

sharing of code pages (in binaries, or shared libraries) is useful as it reduces the number of physical pages in use, thus reducing memory overheads.

Q.What if write enabled? => may be protection bit is set in TLB. or COW

Issue : Replacement Policy

cache replacement : when we are installing a new entry in the TLB, we have to replace an old one => which one to replace?

HOW to design TLB Replacement policy?

goal : being to minimize the miss rate (or increase hit rate)
=> improve performance.

Least-recently-used (LRU) entry

: take advantage of temporal 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.

Random

: useful due to its simplicity & ability to avoid corner-case behaviors

- LRU behaves quite unreasonably when a program loops over n+1 pages with a TLB of size n. in this case, LRU misses upon every access.

RAM Isn't always RAM

random-access memory (RAM) :
implies that you can access any part of RAM just as quickly as another. generally true because of hardware/OS features such as the TBL

HOWEVER, accessing a particular page of memory may be costly, particularly if that page isn't currently mapped by your TLB.
=> RAM isn't always RAM.

TLB can be the source of many performance problems.

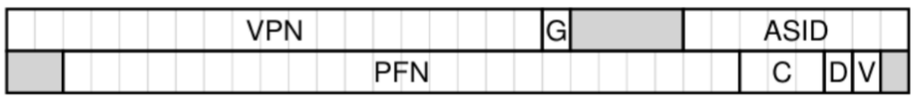


Figure 19.4: **A MIPS TLB Entry**

bits for ASID is 8 in above.
what Should the OS do if there are more than 256 (2^8) processes running at a time?
: kick out one process from TLB & reuse that ASID for new process

Wired register
: hardware register holding the number of *non-evictable* TLB slots

A wired register can be set by the OS to tell the hardware how many slots of the TLB to reserve for the OS. => for this one, no walk needed.

the OS uses these reserved mappings for code and data that it wants to access during critical times, where a TLB miss would be problematic (e.g., in the TLB miss handler)

When we use TLB well, the performance of the program will be almost as if memory isn't being virtualized at all.

However, TLB miss is critical.

physically-indexed cache : cache physical address
TLB access can easily become a bottleneck in the CPU pipeline.
address translation has to take place before the cache is accessed, which can slow things down quite a bit.

Virtually-indexed cache : cache virtual address
access caches with virtual addresses => avoiding the expensive step of translation in the case of a cache hit.
* but introduces new issues into hardware design as well