Operating Systems

Paging - Faster Translations

Hongliang Liang, BUPT

Sep. 2022

Paging: Two major challenges

1. The mapping from virtual to physical address must be **fast**
2. If the virtual address space is large, the linear page table will be large
   Solved with multi-level page tables

Challenge: Performance

Remember that dynamic address translation occurs on the fly at run-time

Too slow, each memory access requires at least **two — and up to six!**

Solution?

Solution: Basic Idea

MMU **caches** page table entries and knows how to handle **cache misses**

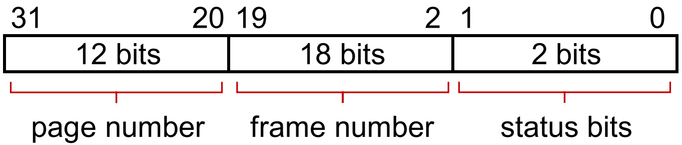
This cache is called the Translation Lookaside Buffer (TLB), a special, **small and fast-lookup** hardware cache.

Translation Lookaside Buffer is associative, high-speed memory

TLB Entries

Each entry contains

- Page number (why do we need the page number here?)
- Frame number
- Valid bit
- Other status bits from a page table entry in the physical memory



TLB Operations

Similar to operations on any cache

Page lookup

- Returns the frame number
- Fully associative memory, indexed by page number
- Performed in hardware, requires just a single cycle!

TLB Operations

Similar to operations on any cache

Page lookup

- Returns the frame number
- Fully associative memory, indexed by page number
- Performed in hardware, requires just a single cycle!

Handling TLB misses

- Lookup the page table for correct entry, fill TLB cache

TLB Operations

Similar to operations on any cache

Page lookup

- Returns the frame number
- Fully associative memory, indexed by page number
- Performed in hardware, requires just a single cycle!

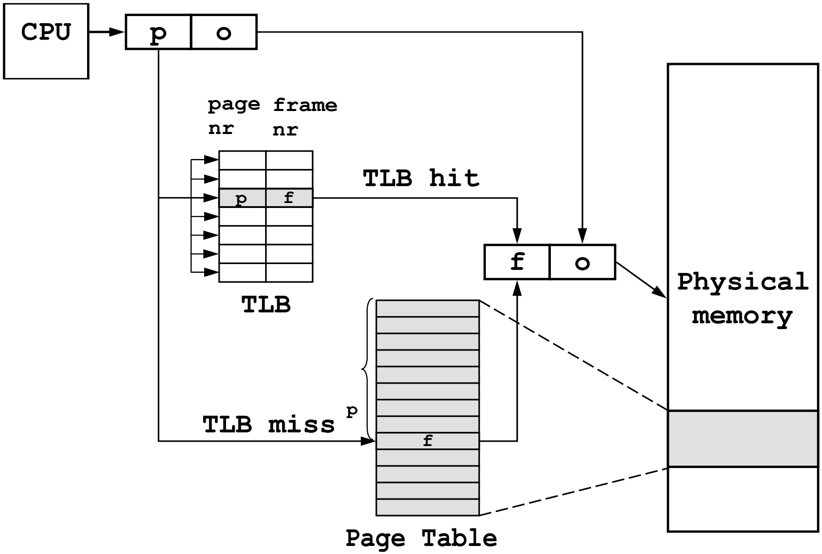
Handling TLB misses

- Lookup the page table for correct entry, fill TLB cache

Invalidating stale TLB entries

- If a page table entry is changed, the cached TLB entry must be invalidated or updated

Page Lookup with a TLB



TLB: Control Flow

```
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()
```

Handling TLB Misses

TLB misses — What if the entry is not in the TLB?

Look in the page table in memory

- Find the right entry
- Move it into the TLB
- Which TLB entry should be replaced?
- This is called the **TLB replacement policy**

Handling TLB Misses

Hardware Managed TLB (x86 CPUs)

- TLB misses are handled in hardware
- Hardware defines page table format, and uses the PTBR to locate the page table in physical memory
- TLB replacement policy fixed by hardware

Handling TLB Misses

Hardware Managed TLB (x86 CPUs)

- TLB misses are handled in hardware
- Hardware defines page table format, and uses the PTBR to locate the page table in physical memory
- TLB replacement policy fixed by hardware

Software Managed TLB (typical RISC CPUs: SPARC, MIPS)

- Hardware generates an exception called **TLB miss fault**
- OS handles TLB miss, similar to interrupt handling
- The exception handler retrieves the correct page table entry, and adds it to the TLB
- Replacement policy managed in software

Invalidating TLB Entries

When should a TLB entry be invalidated?

- On a context switch to another thread in a different address space. Why?

Invalidating TLB Entries

When should a TLB entry be invalidated?

- On a context switch to another thread in a different address space. Why?
- Prevents use of mapping in the previous address space

Invalidating TLB Entries

When should a TLB entry be invalidated?

- On a context switch to another thread in a different address space. Why?

- Prevents use of mapping in the previous address space

Option 1

- Empty the TLB, by clearing the valid bit of all entries
- New thread will generate misses until it caches enough of its own entries into the TLB

Invalidating TLB Entries

When should a TLB entry be invalidated?

- On a context switch to another thread in a different address space. Why?

- Prevents use of mapping in the previous address space

Option 1

- Empty the TLB, by clearing the valid bit of all entries
- New thread will generate misses until it caches enough of its own entries into the TLB

Option 2

- Hardware maintains an "address-space ID" tag in each TLB entry
- Hardware compares this tag to the current address space identifier, held in a specific register, on every translation
- Enables space multiplexing, no need to invalidate all entries

Hit ratio & effective memory-access time (EMAT)

Hit ratio: the percentage of times that the page number of interest is found in the TLB.

If it takes 100 ns to access memory, then a mapped-memory access takes 100 ns when the page number is in the TLB. Otherwise we must first access memory for the page table and frame number (100 ns) and then access the desired byte in memory (100 ns), for a total of 200 ns, assuming that a page-table lookup takes only one memory access.

For a 80% hit ratio, EMAT $= 0.80 \times 100 + 0.20 \times 200 = 120$ ns i.e., we suffer a 20% slowdown in average EMAT (from 100 to 120 ns).

Hit ratio & effective memory-access time (EMAT)

Hit ratio: the percentage of times that the page number of interest is found in the TLB.

If it takes 100 ns to access memory, then a mapped-memory access takes 100 ns when the page number is in the TLB. Otherwise we must first access memory for the page table and frame number (100 ns) and then access the desired byte in memory (100 ns), for a total of 200 ns, assuming that a page-table lookup takes only one memory access.

For a 80% hit ratio, EMAT $= 0.80 \times 100 + 0.20 \times 200 = 120$ ns i.e., we suffer a 20% slowdown in average EMAT (from 100 to 120 ns).

For a 99% hit ratio, which is much more realistic,
EMAT $= 0.99 \times 100 + 0.01 \times 200 = 101$ ns

Enforcing Page-Level Protection

We mentioned that page-level protection can be enforced during dynamic address translation with protection bits

Enforcing Page-Level Protection

We mentioned that page-level protection can be enforced during dynamic address translation with protection bits

Option 1:
Check the page table on **each** memory access — Slow

Enforcing Page-Level Protection

We mentioned that page-level protection can be enforced during dynamic address translation with protection bits

Option 1:
Check the page table on **each** memory access — Slow

Option 2:
Cache page-level protection bits in the TLB,
Check TLB on **each** memory access — Fast

Using TLB to Enforce Protection

TLB checks whether memory accesses are valid when performing translation

If the memory access is of an invalid type (e.g., a page in the text segment is being modified), generate a **protection fault**