Virtual Memory 2

Today

- Handling bigger address spaces
- Speeding translation
Considerations with page tables

Two key issues with page tables

- **Mapping must be fast**
  - Done on every memory reference, at least 1 per instruction

- **With large address spaces, large page tables**
  - 32b addresses, 4KB page →
    - 12b offset, 20b page number ~ 1 million PTE ($2^{20}$)

  - 64b addresses & 4KB page → … $2^{52}$ pages ~ $4.5 \times 10^{15}$!!!
Considerations with page tables

Two key issues with page tables

- Mapping must be fast
  - ...

- With large address spaces, large page tables
  - ...

- Simplest solutions
  - Page table in registers
    - Fast during execution, $$$ & slow to context switch
  - Page table in memory & Page Table Base Register
    - Fast to context switch & cheap, but slow during execution
Page table and page sizes

- Bigger pages
  - Smaller page tables
  - But more internal fragmentation

- Smaller pages
  - Less internal fragmentation
  - Less unused program in memory
  - But ... larger page tables
  - more I/O time, getting page from disk ... seek and rotational delays dominate
    - Getting a bigger page would take as much time
Page table and page sizes

- Avg process size $s$ bytes, page size $p$ bytes
  - Num. of pages needed per process $\sim s/p$
- Page table entry size $e$ bytes
  - $s/p*e$ bytes of page table space

- $Overhead = s/p*e + p/2$

- Finding the optimum
  - Take first derivative respect to $p$, equating it to zero
    
    $$-se/p^2 + 1/2 = 0 \quad p = \sqrt{2se}$$

- $s = 1$MB $\quad e = 8$ bytes $\Rightarrow$ Optimal $p = 4$KB
Separate instruction & data spaces

- One address space – size limit
- Pioneered by PDP-11: 2 address spaces, Instruction and Data spaces
  - Double the space
  - Each with its own page table & paging algorithm
A hybrid approach – Pages & segments

- As in MULTICS
  - Instead of a single page table, one per segment
  - The base register of the segment points to the base of the page table

- But
  - Segmentation is not as flexible as we may want
  - Fragmentation is still an issue
  - Page tables can be of any size, so here we go again
Hierarchical or multilevel page table

- Another approach – page the page table!
  - Same argument – you don’t need the full PT

- Example
  - Virtual address (32b machine, 4KB page): 20b page # + offset
  - Since PT is paged, divide page #:
    Page number (10b) + Page offset in 2^{nd} level (10b)

- Pros and cons
  - Allocate PT space as needed
  - If carefully done, each portion of PT fits neatly within a page
  - More effort for translation
  - And a bit more complex
Three-level page table in Linux

- Designed to accommodate the 64-bit Alpha
  - To adjust for a 32b processor – middle directory of size 1
Inverted page tables

- Another way to save space – inverted page tables
  - Page tables are index by virtual page #, hence their size
  - Inverted page tables – one entry per page frame
    - But to get the page you are still given a VPN
      - Straightforward with a page table, but
    - Linear with inverted page tables – too slow mapping!

Traditional page table
with an entry per
each $2^{52}$ pages

1GB physical
memory has $2^{18}$
4KB page frames

Indexed by
virtual page
Inverted and hashed page tables

- Slow inverted page tables … hash tables may help
  - Since different virtual page number might map to identical hash values – a collision-chain mechanism

1GB physical memory has $2^{18}$ 4KB page frames

Hash table

Indexed by hash on virtual page

Virtual page | page frame

And of course caching, a.k.a. TLB …
And now a short break ...

Before the Internet - xkcd
Speeding things up a bit

- Simple page table 2x cost of memory lookups
  - First into page table, a second to fetch the data
- Two-level page tables triple the cost!
  - Two lookups into page table and then fetch the data
- And two is not enough …

- How can we make this more efficient?
Speeding things up a bit

- Ideally, make fetching from a virtual address almost as efficient as from a physical address.

- Observation – locality of references (lot of references to few pages).

- Solution – hardware cache inside the CPU
  - Translation Lookaside Buffer (TLB)
  - Cache the virtual-to-physical translations in HW
    - A better name would be address-translation cache
  - Traditionally managed by the MMU
TLBs

- Translates virtual page #s into page frame #s
  - Can be done in single machine cycle
- Implemented in hardware
  - A fully associative cache (parallel search)
  - Cache tags are virtual page numbers
  - Cache values are page frame numbers
    - With this + offset, MMU can calculate physical address
  - A typical TLB entry might look like this
    - VPN | PFN | Other bits
TLBs hit

\[ \text{VPN} = (\text{VirtAddr} \times \text{VPN} \_\text{MASK}) \gg \text{SHIFT} \]

\( (\text{Success, TlbEntry}) = \text{TLB}\_\text{Lookup}(\text{VPN}) \)

If (Success == True) // TLB Hit

if (CanAccess(TlbEntry.ProtectBits) == True)

\[ \text{Offset} = \text{VirtAddr} \& \text{OFFSET\_MASK} \]

\[ \text{PhysAddr} = (\text{TlbEntry.PFN} \ll \text{SHIFT}) | \text{Offset} \]

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
        TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
        RetryInstruction()
Managing TLBs

- Address translations mostly handled by TLB
  - >99% of translations, but there are TLB misses
  - If a miss, translation is placed into the TLB

- Who manages the TLB miss?
  - Hardware, the memory management unit – MMU
    - Knows where page tables are in memory
      - OS maintains them, HW access them directly
  - E.g., Intel x86
Managing TLBs

- Software TLB management
  - E.g. MIPS R10k, Sun’s SPARC v9

- Idea
  - OS loads TLB
  - On a TLB miss, faults to OS
    - OS finds page table entry
    - removes an entry from TLB
    - enter new one
    - restart instruction

- Must be fast
  - CPU ISA has instructions for TLB manipulation
  - OS gets to pick the page table format
Managing TLBs

- OS must ensure TLB and page tables are consistent
  - When OS changes protection bits in an entry, it needs to invalidate the line if it is in the TLB
- When the TLB misses, and a new process table entry is loaded, a cached entry must be evicted
  - How to choose a victim is called “TLB replacement policy”
  - Implemented in hardware, usually simple (e.g., LRU)

- Could you have a TLB miss and still have the referenced page in memory?
  - Yes, that’s a “soft miss”; all that’s needed is to update TLB
Managing TLBs

- What happens on a process context switch?
  - Need to invalidate all the entries in TLB! (flush)
    - A big part of why process context switches are costly
  - *Can you think of a hardware fix to this?*
  - Add an Address Space Identifier field to the TLB
An example TLB

- From MIPS R4000 – software-managed TLB
  - 32b address space with 4KB pages
    - 20b VPN and 12b offset
  - But TLB has only 19b for VPN!
    - *User addresses will only come from half the address space – the rest is for the kernel*
    - So 19b is enough

![TLB Diagram]

- VPN
- ASID
- PFN
- G
- C
- D
- V
An example TLB

... 

– VPN translates to up to a 24b physical frame number and can thus support systems with up to 64GB of physical memory

– How is that?!

– G is for pages globally shared (so ASID is ignored)
– C is for coherence; D is for dirty and V is for valid
– Since it is software managed, OS needs instructions to manipulate it
  • TLBP – probes
  • TLBR – reads
  • TLBWI and TLBWR to replaces a specific or a random entry
Effective access time

- Associative Lookup $= \varepsilon$ time units
- Hit ratio - $\alpha$ - percentage of times that a page number is found in the associative registers (ratio related to TLB size)

Effective Memory Access Time (EAT)

$$EAT = \alpha \times (\varepsilon + \text{memory-access}) + (1 - \alpha) \times (\varepsilon + 2 \times \text{memory-access})$$

$\alpha = 80\%$ $\varepsilon = 20$ nsec $\text{memory-access} = 100$ nsec

$$EAT = 0.8 \times (20 + 100) + 0.2 \times (20 + 2 \times 100) = 140$$ nsec
Next time

- Virtual memory policies
- Some other design issues