CSE 451: Operating Systems

Lecture 10
Paging & TLBs
Managing Page Tables

• Last lecture:
  – size of a page table for 32 bit AS with 4KB pages was 4MB!
    • far too much overhead
  – how can we reduce this?
    • observation: only need to map the portion of the address space
      that is actually being used (tiny fraction of address space)
      – only need page table entries for those portions
    • how can we do this?
      – make the page table structure dynamically extensible…
  – all problems in CS can be solved with a level of indirection
    • two-level page tables
Two-level page tables

• With two-level PT’s, virtual addresses have 3 parts:
  – master page number, secondary page number, offset
  – master PT maps master PN to secondary PT
  – secondary PT maps secondary PN to page frame number
  – offset + PFN = physical address

• Example:
  – 4KB pages, 4 bytes/PTE
    • how many bits in offset? need 12 bits for 4KB
  – want master PT in one page: 4KB/4 bytes = 1024 PTE
    • hence, 1024 secondary page tables
  – so: master page number = 10 bits, offset = 12 bits
    • with a 32 bit address, that leaves 10 bits for secondary PN
Two level page tables

Virtual address

<table>
<thead>
<tr>
<th>master page #</th>
<th>secondary page #</th>
<th>offset</th>
</tr>
</thead>
</table>

Master page table

Secondary page table

Page frame number

Physical address

<table>
<thead>
<tr>
<th>page frame #</th>
<th>offset</th>
</tr>
</thead>
</table>

Physical memory

| page frame 0 |
| page frame 1 |
| page frame 2 |
| page frame 3 |

...
Addressing Page Tables

• Where are page tables stored?
  – and in which address space?
• Possibility #1: physical memory
  – easy to address, no translation required
  – but, page tables consume memory for lifetime of VAS
• Possibility #2: virtual memory (OS’s VAS)
  – cold (unused) page table pages can be paged out to disk
  – but, addresses page tables requires translation
    • how do we break the recursion?
  – don’t page the outer page table (called wiring)
• So, now that we’ve paged the page tables, might as well page the entire OS address space!
  – tricky, need to wire some special code and data (e.g., interrupt and exception handlers)
Making it all efficient

- Original page table scheme doubled the cost of memory lookups
  - one lookup into page table, a second to fetch the data
- Two-level page tables triple the cost!!
  - two lookups into page table, a third to fetch the data
- How can we make this more efficient?
  - goal: make fetching from a virtual address about as efficient as fetching from a physical address
  - solution: use a hardware cache inside the CPU
    - cache the virtual-to-physical translations in the hardware
    - called a translation lookaside buffer (TLB)
    - TLB is managed by the memory management unit (MMU)
TLBs

• Translation lookaside buffers
  – translates virtual page #s into PTEs (not physical addr)
  – can be done in single machine cycle

• TLB is implemented in hardware
  – is a fully associative cache (all entries searched in parallel)
  – cache tags are virtual page numbers
  – cache values are PTEs
  – with PTE + offset, MMU can directly calculate the PA

• TLBs exploit locality
  – processes only use a handful of pages at a time
    • 16-48 entries in TLB is typical (64-192KB)
    • can hold the “hot set” or “working set” of process
  – hit rates in the TLB are therefore really important
Managing TLBs

• Address translations are mostly handled by the TLB
  – >99% of translations, but there are TLB misses occasionally
  – in case of a miss, who places translations into the TLB?

• Hardware (memory management unit, MMU)
  – knows where page tables are in memory
    • OS maintains them, HW access them directly
  – tables have to be in HW-defined format
  – this is how x86 works

• Software loaded TLB (OS)
  – TLB miss faults to OS, OS finds right PTE and loads TLB
  – must be fast (but, 20-200 cycles typically)
    • CPU ISA has instructions for TLB manipulation
    • OS gets to pick the page table format
Managing TLBs (2)

• OS must ensure TLB and page tables are consistent
  – when OS changes protection bits in a PTE, it needs to invalidate the PTE if it is in the TLB
• What happens on a process context switch?
  – remember, each process typically has its own page tables
  – need to invalidate all the entries in TLB! (flush TLB)
    • this is a big part of why process context switches are costly
  – can you think of a hardware fix to this?
• When the TLB misses, and a new PTE is loaded, a cached PTE must be evicted
  – choosing a victim PTE is called the “TLB replacement policy”
  – implemented in hardware, usually simple (e.g. LRU)
Selecting a page size

- Small pages give you lots of flexibility but at a high cost.
- Big pages are easy to manage, but not very flexible.
- Issues include
  - TLB coverage
    - product of page size and # entries
  - internal fragmentation
    - likely to use less of a big page
  - # page faults and prefetch effect
    - small pages will force you to fault often
  - match to I/O bandwidth
    - want one miss to bring in a lot of data since it will take a long time.
Segmentation

• A similar technique to paging is segmentation
  – segmentation partitions memory into logical units
    • stack, code, heap, …
  – on a segmented machine, a VA is <segment #, offset>
  – segments are units of memory, from the user’s perspective

• A natural extension of variable-sized partitions
  – variable-sized partition = 1 segment/process
  – segmentation = many segments/process

• Hardware support:
  – multiple base/limit pairs, one per segment
    • stored in a segment table
  – segments named by segment #, used as index into table
Segment lookups

![Diagram showing segment lookups]

- Virtual address
  - Segment #
  - Offset
- Segment table
  - Limit
  - Base
- Physical memory
  - Segment 0
  - Segment 1
  - Segment 2
  - Segment 3
  - Segment 4

Decision:
- If limit > offset (yes), proceed to limit + offset
- If limit ≤ offset (no), raise protection fault
An Early Example -- IBM System 370

24 bit virtual address

Real Memory

- 4 bits
- 8 bits
- 12 bits

simple bit operation

Segment Table

Page Table

© 2001 Brian Bershad
The Segment Table

- Can have one segment table per process
- To share memory, just share by putting the same translation into the base and bounds pair.
- Can share with different protections.
- Cross-segment names can be tough to deal with
  - Segments need to have the same names in multiple processes if you want to share pointers.
- If the segment table is big, should keep it in main memory
  - but then access is slow.
- So, keep a subset of the referenceable segments in a small on-chip memory and look up translation there.
  - can be either automatic or manual.
- Share common segments
Combining Segmentation and Paging

• Can combine these techniques
  – x86 architecture supports both segments and paging
• Use segments to *manage* logically related units
  – stack, file, module, heap, …?
  – segment vary in size, but usually large (multiple pages)
• Use pages to partition segments into fixed chunks
  – makes segments easier to manage within PM
    • no external fragmentation
    • segments are “pageable”- don’t need entire segment in memory at same time
• Linux:
  – 1 kernel code segment, 1 kernel data segment
  – 1 user code segment, 1 user data segment
  – N task state segments (stores registers on context switch)
  – 1 “local descriptor table” segment (not really used)
  – all of these segments are paged
    • three-level page tables
MIPS R3000 VM Architecture

- **User mode and kernel mode**
  - 2GB of user space
  - When in user mode, can only access KUSEG.
- **Three kernel regions; all are globally shared.**
  - KSEG0 contains kernel code and data, but is unmapped. Translations are direct.
  - KSEG1 like KSEG0, but uncached. Used for I/O space.
  - KSEG2 is kernel space, but cached and mapped. Contains page tables for KUSEG.
- **Implication is that the page tables are kept in VIRTUAL memory!**
Lookups

• Each memory reference can be 3
  – assuming no fault
• Can exploit locality to improve lookup strategy
  – a process is likely to use only a few pages at a time
• Use Translation Lookaside buffer to exploit locality
  – a TLB is a fast associative memory that keeps track of recent translations.
• The hardware searches the TLB on a memory reference
• On a TLB miss, either a hardware or software exception can occur
  – older machines reloaded the TLB in hardware
  – newer RISC machines tend to use software loaded TLBs
    • can have any structure you want for the page table
    • fast handler computes and goes. Eg, the MIPS.
The TLB

- A small fully associative cache
- Each entry contains a tag and a value.
  - tags are virtual page numbers
  - values are physical page table entries.
- Problems include
  - keeping the TLB consistent with the PTE in main memory
    - valid and ref bits, for example
  - keeping TLBs consistent on an MP.
  - quickly loading the TLB on a miss.
- Hit rates are important.
Software Loaded TLB

- The MIPS TLB contains 64 entries
  - fully associative
  - random replacement
- On a TLB miss to KUSEG, a trap occurs
  - control transfers to UTLBMISS handler
- If a miss occurs in the miss handler, a second fault occurs.
  - control transfers to a KTLBMISS handler
  - the missed VA is “pte”
  - the translation for the page table is loaded into the TLB.

```c
UTLBmiss(vm_offset_t va)
{
  PTE_t pte;
  pte = PCB.PTBR;
  pte = pte + (va / PAGE_SIZE);
  TLB_dropin(va,
      *pte); /* could miss here */
}
```
Cool Paging Tricks

• Exploit level of indirection between VA and PA
  – shared memory
    • regions of two separate processes’ address spaces map to the same physical frames
      – read/write: access to share data
      – execute: shared libraries!
    • will have separate PTEs per process, so can give different processes different access privileges
    • must the shared region map to the same VA in each process?
  – copy-on-write (COW), e.g. on fork( )
    • instead of copying all pages, created shared mappings of parent pages in child address space
      – make shared mappings read-only in child space
      – when child does a write, a protection fault occurs, OS takes over and can then copy the page and resume client
Another great trick

• Memory-mapped files
  – instead of using open, read, write, close
    • “map” a file into a region of the virtual address space
      – e.g., into region with base ‘X’
    • accessing virtual address ‘X+N’ refers to offset ‘N’ in file
    • initially, all pages in mapped region marked as invalid
  – OS reads a page from file whenever invalid page accessed
  – OS writes a page to file when evicted from physical memory
    • only necessary if page is dirty