Important from last time

- We’re trying to build efficient virtual address spaces
  - Why??

- Virtual / physical translation is done by HW and SW cooperating
  - We want to make the common case fast in HW

- Key insight: Treat a virtual address as a collection of separate elements
  - High bits identify a segment or page
  - Low bits identify offset into the segment or page
Quick Review

- **Base and bounds**
  - Maps a single contiguous chunk of virtual address space to a single contiguous chunk of physical memory
  - Fast, small HW, and safe, but inflexible and lots of external fragmentation

- **Segmentation**
  - Small number of segment registers (base-bounds per segment)
  - Each segment maps contiguous VM to contiguous physical mem
  - More flexible and less fragmentation than B+B, but same issues
Modern hardware and OSes use paging

Pages are like segments, but fixed size
  - So the bounds register goes away
  - External fragmentation goes away

Since pages are small (4 or 8 KB, often) there are a lot of them
  - So page table has to go into RAM
  - Page table might be huge
    - Accessing the page table requires a memory reference by the hardware to perform the translation
      » First, load the page table entry
      » Second, access the data the user asked for

Today we look at how these problems are fixed
**Problems**

- Page table too large to store on processor die
- Two memory references per load or store issued by the user program is unacceptable

**Solution: Cache recently-used page table entries**

- TLB == “translation lookaside buffer”
- TLB is a fixed-size cache of recently used page table entries
- If TLB hit rate sufficiently large → translation as fast as segments
- If TLB hit rate poor → load/store performance suffers badly
- Key issue: Access locality
- What is effective access time with/without TLB?
Paging Unit

- CPU issues load/store
- Memory management unit
  1. Compare VPN to all TLB tags
  2. If no match, need TLB refill
    » SW → trap to OS
    » HW → HW table walker
  3. Checks if VA is valid
    » If not, generate trap
  4. Concatenate PPN to Offset to form physical address
- This all needs to be very fast
  - Why?

![Diagram of paging unit process]

<table>
<thead>
<tr>
<th>Tag</th>
<th>PPN</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0A1FE</td>
<td>20104</td>
<td>...</td>
</tr>
<tr>
<td>104A3</td>
<td>4010D</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3D11C</td>
<td>0401B</td>
<td>...</td>
</tr>
</tbody>
</table>

Physical Address
TLB Issues

- How large should the TLB be?
- What limits TLB size?
- How does a TLB interact with context switches?
**TLB Discussion**

- **What are typical TLB sizes and organization?**
  - 16-1024 entries, fully associative, sub-cycle access latency
  - TLB misses take 10-100 cycles
  - Modern chips have multiple levels of TLB
  - Perform TLB lookup in parallel to L1 cache tag load
    » Requires virtually tagged L1 cache
  - Why is TLB often fully associative?

- **TLB “reach”**
  - Amount of address space that can be mapped by TLB entries
  - What are architectural trends in terms of:
    » TLB size and associativity
    » Main memory size
  - Example: 64-entry TLB w/ 4KB pages → 256 KB reach
  - Good news: 90-10 rule (90% of accesses to 10% of address space)
More Paging Discussion

- What are typical state bits?
  - Valid bit
  - Protection bits (read-only, execute-only, read-write)
  - Referenced bit
  - Modified bit
  
  Used to support “demand paging”

- What happens during a context switch?
  - PCB needs to be extended to contain (or point to) page table
  - Save/restore page table base register (for hardware-filled TLB)
  - Flush TLB (if PID not in tag field)
Page Table Organization

- Simple flat full table
  - Simple, fast lookups
  - Huge!!

- Hierarchical
  - Two or more levels of tables
    » Leaf tables have PTEs
  - Fairly simple, fairly fast
  - Size roughly proportional to allocated memory size

- Inverted
  - One entry per physical page
  - Entry contains VPN and ASID
  - Small table size
  - Lookups more expensive
    » Hashing helps
    » Poor memory locality

Virtual address: PID | Page# | Offset
Physical address: PPN (n) | Offset

Hashing: hash(PID | Page#)

Root Page Table
Hierarchical
Inverted Page Table
Main Memory
page number

\[ \begin{array}{c|c|c}
  p1 & p2 & d \\
\end{array} \]

page offset

outer page table

inner page table

desired page
Multi-level page tables

- Why does this save space?

- Early virtual memory systems used a single-level page table
- VAX, 386 used 2-level page table
- Sparc uses 3-level
- Alpha, IA-64, x86-64 have 4 levels
  - In many OSes, page tables other than top-level can themselves be paged
**Sharing**

- **How do you share memory?**
  - Entries in different process page tables map to same PPN

- **Questions:**
  - What does this imply about granularity of sharing?
  - Can we share only a 4-byte int?
  - Can we set it up so that only one of the sharers can write the region?
  - How can we use this idea to implement copy on write?
Initializing an Address Space

- Determine number of pages needed
  - Examine header information in executable file

- Determine virtual address layout
  - Header file determines size of various segments
  - OS specifies positioning (e.g., base of code, location of stack)

- Allocate necessary pages and initialize contents
  - Initialize page table entry to contain VPN→PPN, set valid, ...

- Mark current TLB entries as invalid (i.e., TLB flush)

- Start process
  - PC should point at valid address in code segment
  - Allocated stack space as process touches new pages
Superpages

- **Problem**: TLB reach shrinking as % of memory size
- **Solution**: Superpages
  - Permit pages to have variable size (back to segments?)
  - For simplicity, restrict generality:
    - Power of two region sizes
    - Aligned to superpage size (e.g., 1MB superpage aligned on 1MB bdy)
    - Contiguous

- **Problem**: Restrictions limit applicability. How?
Example: Superpage Usage

Virtual Addresses

<table>
<thead>
<tr>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00004000</td>
</tr>
<tr>
<td>0x00005000</td>
</tr>
<tr>
<td>0x00006000</td>
</tr>
<tr>
<td>0x00007000</td>
</tr>
</tbody>
</table>

Physical Addresses

<table>
<thead>
<tr>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80240000</td>
</tr>
<tr>
<td>0x80241000</td>
</tr>
<tr>
<td>0x80242000</td>
</tr>
<tr>
<td>0x80243000</td>
</tr>
</tbody>
</table>

Both virtual and physical ranges are contiguous

Virtual and physical ranges aligned on superpage boundary

Page table

<table>
<thead>
<tr>
<th>virtual</th>
<th>physical</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00004</td>
<td>80240</td>
<td>002</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Size: Denotes number of base pages in superpage ($2^{size}$)
Superpage Discussion

What are good candidates for superpages?
- Kernel – or at least the portions of kernel that are not “paged”
- Frame buffer
- Large “wired” data structures
  » Scientific applications being run in “batch” mode
  » In-core databases

How might OS exploit superpages?
- Simple: Few hardwired regions (e.g., kernel and frame buffer)
- Improved: Provide system calls so applications can request it
- Holy grail: OS watches page access behavior and determines which pages are “hot” enough to warrant superpaging

Why might you not want to use superpages?
Paged Segmentation

- **Problem with paging:**
  - Large VA space $\rightarrow$ large page table

- **Idea: Combine segmentation and paging**
  - Divide address space into large contiguous segments
  - Divide segments into smaller fixed-size pages
  - Divide main memory into pages
  - Two-stage address translation

- **Benefits:**
  - Reduced page table size
  - Can share segments easily

- **Problems:**
  - Extra complexity
Paged Segmentation

- **Sharing:**
  - Share individual pages → copy page table entries
  - Share entire segments → copy segment table entries

- **Protection:**
  - Can be associated with either segment or page tables

- **Implementation**
  - Segment tables in MMU
  - Page tables in main memory

- **Practice**
  - x86 supports pages & segments
  - RISCs support only paging

Sharing example:
- Two processes
- Two common segments (0&3)
Bringing It All Together

● Paging is major improvement over segmentation
  – Eliminates external fragmentation problem (no compaction)
  – Permits flexible sharing between processes
  – Enables processes to run when only partially loaded (more soon!)

● Significant issues remain
  – Page tables are much larger than segment tables
  – TLB required \( \rightarrow \) effectiveness seriously impacts performance
  – More complex OS routines for managing page table
    » But management of physical address space much easier
  – Translation overheads tend to be higher

● Question: How can we let processes run when only some of their pages are present in main memory?