## Recall: Address Translation Comparison

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Segmentation</strong></td>
<td>Fast context switching: Segment mapping maintained by CPU</td>
<td>External fragmentation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Paging (single-level page)</strong></td>
<td>No external fragmentation, fast easy allocation</td>
<td>Large table size ~ virtual memory Internal fragmentation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Paged segmentation</strong></td>
<td>Table size ~ # of pages in virtual memory, fast + easy allocation</td>
<td>Multiple memory references per page access</td>
</tr>
<tr>
<td><strong>Two-level pages</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Inverted Table</strong></td>
<td>Table size ~ # of pages in physical memory</td>
<td>Hash function more complex</td>
</tr>
</tbody>
</table>
Recall: Paging Tricks: Examples

Demand Paging
- Swapping for pages
- Keep only active pages in memory
- When exception occurs for page not in memory, load from disk and retry

Copy on Write
- Remember fork() – copy of address space
- Instead of real copy, mark pages read-only
- Allocate new pages on protection fault

Zero-Fill On Demand
- New pages should be zeroed out – slow!
- Instead, pages start invalid – create new zero page when accessed
Diversion: Protection without HW

Example: **emulator**
- Add any permissions checks, but very slow

Faster: Programming language restrictions (think Java)

Faster: binary translation / software fault isolation
- Insert checks before every dangerous operation
  - Load, store, ...
- Either compile/link-time or runtime
- Can separately verify checks are present
Diversion: Protection without HW

OS that runs only "managed" languages:
Singularity (Microsoft Research; defunct?)
  – Like C#, Java, ...
  – Which don't allow pointers to other process's addresses or the OS's addresses

No address translation/protection – rely on language runtime to limit what programs can access
Diversion: Protection without HW

Binary Translation
- MIPS: ~5% overhead
- (but with caveats)

Separate base and bound for code, data

All jumps and loads/stores use particular registers
- Checks before putting any values into these registers
- Checks are of the form address & mask == value

Static checker that assembly follows rules
Caching: Definition

**Cache**: repository for copies that can be accessed quicker than originals

Goal: Improve **performance**

Lots of things can be cached

- Memory (61C), Address Translations (today), Files, File Names, ...
Why cache? Much faster storage
Why cache? Address Translation

Virtual Address:

<table>
<thead>
<tr>
<th>Virtual Seg #</th>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base0</td>
<td>Limit0</td>
<td>V</td>
</tr>
<tr>
<td>Base1</td>
<td>Limit1</td>
<td>V</td>
</tr>
<tr>
<td>Base2</td>
<td>Limit2</td>
<td>V</td>
</tr>
<tr>
<td>Base3</td>
<td>Limit3</td>
<td>N</td>
</tr>
<tr>
<td>Base4</td>
<td>Limit4</td>
<td>V</td>
</tr>
<tr>
<td>Base5</td>
<td>Limit5</td>
<td>N</td>
</tr>
<tr>
<td>Base6</td>
<td>Limit6</td>
<td>N</td>
</tr>
<tr>
<td>Base7</td>
<td>Limit7</td>
<td>V</td>
</tr>
</tbody>
</table>

Page Table:

<table>
<thead>
<tr>
<th>Page #</th>
<th>Access</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>page #0</td>
<td>V,R</td>
<td></td>
</tr>
<tr>
<td>page #1</td>
<td>V,R</td>
<td></td>
</tr>
<tr>
<td>page #2</td>
<td>V,R,W</td>
<td></td>
</tr>
<tr>
<td>page #3</td>
<td>V,R,W</td>
<td></td>
</tr>
<tr>
<td>page #4</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>page #5</td>
<td>V,R,W</td>
<td></td>
</tr>
</tbody>
</table>

Physical Address:

<table>
<thead>
<tr>
<th>Physical Page #</th>
<th>Offset</th>
</tr>
</thead>
</table>

Three DRAM accesses per access?
- Even three cache accesses per access is too slow

Solution: Translation cache (called **TLB**)

Three DRAM accesses per access?
Average Access Time

Average access time =

$\text{(Hit Rate} \times \text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Time})$

Intuition: Caching is good when most accesses are for a small portion of the items

Most accesses $\rightarrow$ Relatively high hit rate

Small portion $\rightarrow$ Hits can be faster than misses
  - Small data storage is faster
Locality

Temporal Locality (Time)
- Recently accessed items will be accessed soon

Spatial Locality (Space)
- Items adjacent to recently accessed items will be accessed soon
Recall: The Memory Hierarchy

Goal: speed of fastest technology with capacity of cheapest

- Processor
  - Control
  - Datapath
    - Registers
    - On-Chip Cache

- Second Level Cache (SRAM)
- Main Memory (DRAM)
- Secondary Storage (Disk)
- Tertiary Storage (Tape)

<table>
<thead>
<tr>
<th>Level</th>
<th>Speed (ns)</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>1s</td>
<td>100s</td>
</tr>
<tr>
<td>Datapath</td>
<td>10s-100s</td>
<td>Ks-Ms</td>
</tr>
<tr>
<td>Second Level Cache</td>
<td>100s</td>
<td>Ms</td>
</tr>
<tr>
<td>Main Memory</td>
<td>10,000,000s (10s ms)</td>
<td>Gs</td>
</tr>
<tr>
<td>Secondary Storage</td>
<td>10,000,000,000s (10s sec)</td>
<td>Ts</td>
</tr>
</tbody>
</table>
Types of Cache Misses

**Compulsory** ("cold start"): first access to a block
- Insignificant in any long-lived program

**Capacity**: not enough space in cache

**Conflict**: memory locations map to same cache location

**Coherence** (invalidation): memory updated externally
- Example: multiple cores, I/O
Types of Cache Misses

**Compulsory** ("cold start"): first access to a block
- Insignificant in any long-lived program

**Capacity**: not enough space in cache
- Not affected by changes in cache design (mostly)

**Conflict**: memory locations map to same cache location

**Coherence** (invalidation): memory updated externally
- Example: multiple cores, I/O
Types of Cache Misses

**Compulsory** ("cold start"): first access to a block
- Insignificant in any long-lived program

**Capacity**: not enough space in cache

**Conflict**: memory locations map to same cache location

**Coherence** (invalidation): memory updated externally
- Example: multiple cores, I/O

Eliminated by increasing cache size (enough)
Types of Cache Misses

**Compulsory** ("cold start"): first access to a block
- Insignificant in any long-lived program

**Capacity**: not enough space in cache

**Conflict**: memory locations map to same cache location

**Coherence** (invalidation): memory updated externally
- Example: multiple cores / I/O

Eliminated by increasing associativity
Finding Blocks in (Memory) Cache

*Index* used to lookup **set** of candidates

*Tag* used to identify actual copy
- None in set match $\rightarrow$ cache miss

*Block* is minimum unit of caching
- Data select $\rightarrow$ part of block to get
Review: Direct Mapped Cache

Example: $2^{10}$ byte capacity; $2^{5}$ byte blocks; 32-bit addresses
- Upper $(32 - 10)$ bits are tag
- Lower 5 bits are offset
- Remaining bits are index
Review: Direct Mapped Cache

Example: \(2^{10}\) byte capacity; \(2^5\) byte blocks; 32-bit addresses
- Upper \((32 - 10)\) bits are \textit{tag}
- Lower 5 bits are \textit{offset}
- Remaining bits are \textit{index}

<table>
<thead>
<tr>
<th>31</th>
<th>9</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Tag</td>
<td>Cache Index</td>
<td>Byte Select</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Valid Bit</th>
<th>Cache Tag</th>
<th>Cache Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x50</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte</th>
<th>31</th>
<th>63</th>
<th>1023</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>* *</td>
<td>* *</td>
<td>* *</td>
</tr>
<tr>
<td>Byte 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte 32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte 33</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte 992</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte 1023</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Review: Direct Mapped Cache

Example: $2^{10}$ byte capacity; $2^5$ byte blocks; 32-bit addresses

- Check *index* to find set of *one* potential block
- Compare *tag* to verify block
- Use *byte select* to choose byte within block
Review: Set Associative Cache

*N-way set associative*: $N$ entries per cache index

Example: 2-way set associative
Review: Fully Associative Cache

**Fully associative:** every block can hold address

- Compare cache tags to **all blocks**
- No index

---

Ex: 0x01
Review: Where can a block go?

- Example: Block 12 placed in 8 block cache

**32-Block Address Space:**

Direct mapped:
block 12 can go only into block 4
(12 mod 8)

Set associative:
block 12 can go anywhere in set 0
(12 mod 4)

Fully associative:
block 12 can go anywhere
Review: Which block to replace?

Direct mapped – no choice

Associative:
  - Random
  - LRU (least recently used)

Empirical miss rates for CPU caches:

<table>
<thead>
<tr>
<th>Size</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LRU</td>
<td>Rand</td>
<td>LRU</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>
Review: Write Policies

**Write-through**: Hold in cache until removed
   - Simple to implement
   - Writes can become bottleneck

**Write-back**: Always write to cache
   - Dirty bit
   - Better for repeated writes
   - Reads may trigger write
   - Coherence policy
Caching in Address Translation

- CPU
- Virtual Address
- TLB
  - Cached?
    - Yes
    - No
  - Save Result
- Translate (MMU)
- Data Read or Write (untranslated)
- Physical Address
- Physical Memory
Caching in Address Translation

Locality in page accesses?
  – Yes! Same reasons as CPU cache locality.

TLB – cache of page table entries

TLB design options:
  – Associativity?
  – Hierarchy?
  – Size?
Recall: What happens in the MMU?

Reads the page table, triggers interrupts

Always: translations are **cached** – what about when nothing cached yet?

**Option 1: Hardware traversal (example: x86)**
- Hardware reads page tables
- Invoke *page fault handler* if invalid/non-present PTE

**Option 2: Software traversal (example: MIPS)**
- Invoke software handler
TLBs and Context Switches

Do nothing?
- Might use old process's address space

Option 1: Invalidate entire TLB

Option 2: ProcessID in TLB
- Called "tagged TLB"

Same problem if translation table changes
- Examples: swapping, copy-on-write
TLB and page table changes

Consider **copy-on-write**.

Process A calls `fork()`.

OS marks its pages as read-only.

After returning from `fork` it tries to write to its stack.

Triggers protection fault. OS makes a copy of the stack page, updates page table.

Restarts instruction.
TLB and page table changes

Consider using copy-on-write.

Process A calls fork().

OS marks its pages as read-only.

After returning from fork it tries to write to its stack.

Triggers protection fault. OS makes a copy of the stack page, updates page table.

Restarts instruction.

What if TLB has the read/write version of the page cached?
TLB and page table changes

Consider copy-on-write.

Process A calls fork().

OS marks its pages as read-only.

After returning from fork it tries to write to its stack. **Triggers protection fault.**

OS makes a copy of the stack page, updates page table. **Restarts instruction.**

What if TLB has the read-only version of the page cached?
How does hardware invalidate TLB entries?

Could maybe keep track of where the page table is and watch for writes to it... but this is really complicated with two (or more)-level page tables and tagged TLBs...

Instead, the **operating system** needs to do it!

- Means the TLB is *not a transparent cache (for the OS)*
TLB and page table changes

Consider *copy-on-write*.

Process A calls fork().

OS marks its pages as read-only. **Tells MMU to clear those TLB entries.**

After returning from fork it tries to write to its stack.

Triggers protection fault. OS makes a copy of the stack page, updates page table. **Tells MMU to clear that TLB entry.**

Restarts instruction.
Designing a TLB

Critical path of every memory access
- Caches need physical addresses. (Why?)
- Very high miss time

So low associativity?
- Problem: conflict misses
- Code + data + stack indexes easily coincide:
  - code @ $0x0040 \ 0000$
  - data @ $0x1000 \ 0000$
  - stack @ $0x7FFF \ 0FFC$
Typical TLB Organization

128-512 entries

Often highly associative

- Sometimes very small, less-associative cache in front (e.g. 4-16 entry direct mapped "TLB slice")

Virtual address → physical address + other info

- Or virtual address + Process ID
- physical address + other info ~ Page Table Entry
### TLB Example: MIPS R3000

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Dirty</th>
<th>Ref</th>
<th>Valid</th>
<th>Access</th>
<th>ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFA00</td>
<td>0x0003</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>R/W</td>
<td>4</td>
</tr>
<tr>
<td>0x0040</td>
<td>0x0010</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>0x0041</td>
<td>0x0011</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R</td>
<td>0</td>
</tr>
</tbody>
</table>

**Page numbers**

Permissions + other bits from page table
- Including info to help paging (later)

**ASID = Address Space ID ~ Process ID**
- Entry only valid when that "PID" running
R3000 TLB Overhead

<table>
<thead>
<tr>
<th>Inst Fetch</th>
<th>Dcd/ Reg</th>
<th>ALU / E.A</th>
<th>Memory</th>
<th>Write Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>I-Cache</td>
<td>RF</td>
<td>Operation</td>
<td>WB</td>
</tr>
<tr>
<td>E.A.</td>
<td>TLB</td>
<td>D-Cache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Half-cycle TLB

One cycle extra latency on every instruction
Overlapping TLB and cache (1)

Idea: cache access shouldn't need physical page # immediately

<table>
<thead>
<tr>
<th>virtual address</th>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>physical address</td>
<td>tag / page #</td>
<td>index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>byte</td>
</tr>
</tbody>
</table>

Cache access starts with index + byte

TLB access finishes in time to compare tag
Overlapping TLB and cache (2)

Larger cache?
- Can only overlap set lookup

Alternative: Virtual addresses in caches
Putting Everything Together: Address Translation

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Page Table Pointer

Page Table (1st level)

Page Table (2nd level)

Physical Address:
- Physical Page #
- Offset

Physical Memory:
Putting Everything Together: TLB

Virtual Address:

Virtual P1 index | Virtual P2 index | Offset

Page Table (1st level)

Page Table (2nd level)

TLB:

Physical Memory:

Physical Address:

Physical Page # | Offset

Virtual Address:

Offset

Virtual P2 index

PageTablePtr

Physical Address:

Physical Memory:
Putting Everything Together: Cache

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (1st level)

Page Table (2nd level)

TLB:

Physical Address:
- Physical Page #
- Offset

Physical Memory:

Cache:
- tag
- index
- byte

Tag:
- block:
- ...
Other Examples of Caching

Demand paging (later)
- Leave unused parts of process memory on disk/SSD

File systems
- Contents of files
- Directory structure

Networking
- Hostname to IP address translations
- Web pages
Other Examples of Caching

Demand paging (later)
- Leave unused parts of process memory on disk/SSD

File systems
- Contents of files
- Directory structure

Networking
- Hostname to IP address translations
- Web pages
Logistics

Homework 1 due today

Project 1 Checkpoint 1 due today

Midterm next week class time
  – Up to and including demand paging
Break
The Working Set Model

Theory: Programs transition through sequence of "working sets" – subsets of their address spaces
Cache Behavior and Working Sets

![Graph showing hit rate versus cache size]

- **Hit Rate**
  - Y-axis: 0 to 1

- **Cache Size**
  - X-axis

- **Note:**
  - When the new working set fits, the hit rate increases significantly.
Zipf/Power Law Model

$P_{\text{access}}(\text{rank}) = \frac{1}{\text{rank}}$

"Heavy-tailed" distribution

- Lots of rare items – lots of *unavoidable* misses
- A few *very common* items – caching very helpful
Demand Paging

Main memory as cache for SSD/disks

Idea: *Illusion* memory has the capacity of disk
Illusion of "infinite" memory

How? **Transparent layer of indirection**
- the page table + page fault handlers
Illusion of "infinite" memory

How?
- Transparent layer of indirection
  - the page table + page fault handlers

Virtual Memory
4 GB

Page Table

Physical Memory
512 MB

Disk
500GB

For performance? No – Disk is 10000x slower (latency) than DRAM.

For correctness? Yes. Just looks like slower memory.
Memory as a Cache for Disk

Block size: 1 page

Associativity: fully associative

Write policy: write-back (disk writes are slow)

Replacement policy: LRU approximation *(later)*
Illusion of "infinite" memory

How? **Transparent layer of indirection**  
- the page table + page fault handlers
Illusion of "infinite" memory

How?

Transparent layer of indirection
- the page table + page fault handlers

Virtual Memory
4 GB

Page Table

Physical Memory
512 MB

Disk
500 GB

For performance? No – Disk is 10000x slower (latency) than DRAM.

For correctness? Yes. Just looks like slower memory.
Recall: x86-32 Page Table Entry

10/10/12-bit split of virtual address

top-level page tables called directories

<table>
<thead>
<tr>
<th>Page Frame Number</th>
<th>Free (OS)</th>
<th>0</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>P</th>
<th>W</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Physical Page Number)</td>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- PFN: physical page number of page or next page table
- P: Present bit (= Valid)
- W: Writable
- U: User-accessible
- A: Accessed – set when page is accessed
- D: Dirty – set when page is modified
- L: If 1, points to 4MB "hugepage" instead of next page table
- PWT: Write-through caching behavior (for memory-mapped IO)
- PCD: Disable caching (for memory-mapped IO)
Recall: x86-32 Page Table Entry

10/10/12-bit split of virtual address

top-level page tables called directories

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>0</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>P</th>
<th>W</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- PFN: physical page number of page or next page table
- P: Present bit (= Valid)
  - = 0 if page is on disk (or actually invalid)
- P: Present bit (= Valid)
- W: Writable
- U: User-accessible
- A: Accessed – set when page is accessed
- D: Dirty – set when page is modified
- L: If 1, points to 4MB "hugepage" instead of next page table
- PWT: Write-through caching behavior (for memory-mapped IO)
- PCD: Disable caching (for memory-mapped IO)
Recall: x86-32 Page Table Entry

10/10/12-bit split of virtual address

top-level page tables called directories

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- PFN: physical page number of page or next page table
- P: Present bit (= Valid)
- W: Writable
- U: User-accessible
- A: Accessed – set when page is accessed
- D: Dirty – set when page is modified
- L: If 1, points to 4MB "hugepage"
- PWT: Write-through caching behavior (for memory-mapped IO)
- PCD: Disable caching (for memory-mapped IO)

Dirty bit for write-back! (Could also be done in SW)
Recall: x86-32 Page Table Entry

10/10/12-bit split of virtual address

top-level page tables called directories

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>0</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>P</th>
<th>C</th>
<th>W</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

- PFN: **physical** page number of page or next page table
- P: Present bit (= Valid)
- W: Writable
- U: User-accessible
- A: Accessed – set when page is accessed
- D: Dirty – set when page is modified
- L: If 1, points to 4MB “hugepage” instead of next page table
- PWT: Write-through caching behavior (for memory-mapped IO)
- PCD: Disable caching (for memory-mapped IO)

Accessed bit to help approximate LRU
Demand Paging: Page Replacement

Process accesses page on disk

Memory Management Unit (MMU) traps to OS – *Page Fault*

OS chooses a page to replace
   - If it's modified (D=1), writes it back to disk

OS load new page into memory

OS update page table entries (replaced + new page)
   - Invalidates TLB for them, too

OS restart process from just before access
Demand Paging: Page Replacement

Process accesses page on disk

Memory Management Unit (MMU) traps to OS – \textit{Page Fault}

OS chooses a page to replace
- If it's modified ($D=1$), writes it back

\textbf{OS load new page into memory}

OS update page table entries (replaced + new page)
- Invalidates TLB for them, too

OS restart process from just before access

\textbf{Takes a long time!}

Put process on wait queue while waiting for disk
Demand Paging: Page Replacement

Process accesses page on disk

Memory Management Unit (MMU) traps to OS – **Page Fault**

OS chooses a page to replace
  - If it's modified (D=1), writes it back to disk

OS load new page into

OS update page table entries (replaced + new page)
  - Invalidate old TLB entries for them, too

OS restart process from just before access
Summary: TLB

Cache for page-table entries

Lookup by virtual page number

Possibly "tagged" with process ID

**Explicit** invalidation by OS
Summary: Demand Paging

Illusion of infinite memory

Memory is just a cache for program's real data on disk

Much more tommorrow