- Project 3 is due **Monday** → more code than P1 / P2
- Project 1 grades
  → Check Piazza
AGENDA / LEARNING OUTCOMES

Memory virtualization
  What are the challenges with paging?
  How we go about addressing them?
RECAP
Goal: Eliminate requirement that address space is contiguous

Idea:
Divide address spaces and physical memory into fixed-sized pages

Example page size: 4KB
PAGING TRANSLATION STEPS

For each mem reference:

1. extract **VPN** (virt page num) from **VA** (virt addr)
2. calculate addr of **PTE** (page table entry)
3. read **PTE** from memory
4. extract **PFN** (page frame num)
5. build **PA** (phys addr)
6. read contents of **PA** from memory

Assume PT is at phys addr 0x5000
Assume PTE’s are 4 bytes
Assume 4KB pages – 12 bit offset

14 bit addresses

Simplified view of page table

![Page Table Diagram]

**Example:**

READ 0x1100

\[ 0x100 + 0x0100 = 0x1100 \]

\[ \text{VPN to PPN mapping} \]

\[ \text{VPN} \rightarrow \text{PFN} \rightarrow \text{PA} \rightarrow \text{Content} \]
PROS/CONS OF PAGING

Pros
No external fragmentation
  - Any page can be placed in any frame in physical memory
Fast to allocate and free
  - Alloc: No searching for suitable free space
  - Free: Doesn’t have to coalesce with adjacent free space

Cons
Additional memory reference
  - MMU stores only base address of page table
Storage for page tables may be substantial
  - Simple page table: Requires PTE for all pages in address space
  - Entry needed even if page not allocated?
STRATEGY: CACHE PAGE TRANSLATIONS

CPU
Translation Cache

RAM
PT

memory interconnect

TLB Entry

| Tag (virtual page number) | Physical page number (page table entry) |

Fully associative

Any given translation can be anywhere in the TLB
Hardware will search the entire TLB in parallel
**TLB ACCESSES: SEQUENTIAL EXAMPLE**

- **Virt**
  - load 0x1000
  - load 0x1004
  - load 0x1008
  - load 0x100C
  -load 0x2000
  - load 0x2004

- **Phys**
  - load 0x0004
  - load 0x5000
  - load 0x5004
  - load 0x5008
  - load 0x500C
  - load 0x0008
  - load 0x4000
  - load 0x4004

**CPU's TLB**

<table>
<thead>
<tr>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

**PTBR**

<table>
<thead>
<tr>
<th>PT</th>
<th>PT</th>
<th>P1 pagetable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

**Virt**

- PTBR = 1
- Virtual address 0x1000
- Valid: 1
- VPN: 1
- PPN: 5

**Phys**

- Physical address 0x0000
- PTE: 0x0004
- (TLB hit)

- Physical address 0x5000
- PTE: 0x5004
- (TLB hit)

- Physical address 0x5008
- PTE: 0x5008
- (TLB hit)

- Physical address 0x500C
- PTE: 0x500C
- (TLB hit)

- Physical address 0x0008
- PTE: 0x0008
- (TLB hit)

- Physical address 0x4000
- PTE: 0x4000
- (TLB hit)

- Physical address 0x4004
- PTE: 0x4004
- (TLB hit)
**QUIZ 10: TLBs**

Consider a processor with 16-bit address space and 4kB page size. Assume Page Table is at 0x2000 and each PTE is of 4 bytes.

<table>
<thead>
<tr>
<th>VPN:0</th>
<th>0x0</th>
<th>0x0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td>0x0</td>
<td>0x0</td>
</tr>
<tr>
<td>0x1</td>
<td>0x1</td>
<td>0x1</td>
</tr>
<tr>
<td>0x9</td>
<td>0x9</td>
<td>0x9</td>
</tr>
<tr>
<td>0x7</td>
<td>0x7</td>
<td>0x7</td>
</tr>
<tr>
<td>0x8</td>
<td>0x8</td>
<td>0x8</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Virtual Addresses:
- 0x3000: load 0x5320, %eax
- 0x3004: load 0x4004, %ebx
- 0x3008: mul %ecx, %eax, %ebx
- 0x300C: store %ebx, 0x5324
- 0x3010: load 0x5328, %ebx

Memory accesses:
- Fetch 0x3000
  - Translate: 0x3000
    - PA is 0x9000
  - Ans: 0x2000 + 3x4 = 0x200C

- Fetch 0x5320 → 2 mem accesses
- 8th mem access = 0x7004

Total: 18 total mem accesses
**VPN:0**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>0x1</td>
<td></td>
</tr>
<tr>
<td>0x9</td>
<td></td>
</tr>
<tr>
<td>0x7</td>
<td></td>
</tr>
<tr>
<td>0x8</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**PageTable**

**Virtual Addresses**

- 0x3000: load 0x5320, %eax
- 0x3004: load 0x4004, %ebx
- 0x3008: mul %ecx, %eax, %ebx
- 0x300c: store %ebx, 0x5324
- 0x3010: load 0x5328, %ebx

**TLB**

<table>
<thead>
<tr>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>2 3</td>
<td>9</td>
</tr>
<tr>
<td>0 1</td>
<td>7 5</td>
<td>8</td>
</tr>
<tr>
<td>0 1</td>
<td>2 4</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>89</td>
</tr>
</tbody>
</table>

**Memory accesses**

- Total 12 accesses
- TLB saved 6 memory accesses
How to we replace entries in the TLB?

How do we handle context switches? → What happens to TLB?
**PERFORMANCE OF TLB?**

Miss rate of TLB: \(#\text{TLB misses} / \#\text{TLB lookups}\)

\# TLB lookups? number of accesses to a = \(2048\)

\# TLB misses?

\(=\) number of unique pages accessed

\(\geq 2\)

Miss rate?

\(\frac{2}{2048}\)

Hit rate?

\(1 - \text{miss rate}\)

---

Page size = 4 KB

Int = 4 bytes

1 page = 1024 Ints

**Would hit rate get better or worse with smaller pages?**
**WORKLOAD ACCESS PATTERNS**

**Workload A**

```c
int sum = 0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```

**Workload B**

```c
int sum = 0;
srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
```
WORKLOAD ACCESS PATTERNS

Spatial Locality
Sequential Accesses

Temporal Locality
Repeated Random Accesses
TLB REPLACEMENT POLICIES

**LRU**: evict Least-Recently Used TLB slot when needed

Fixed size TLB

Find oldest timestamp & replace that entry

---

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Last Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>20</td>
<td>timestamp</td>
</tr>
</tbody>
</table>

---

A | B | C | D | E | F | L | M | N | O | P
Workload repeatedly accesses same offset (0x01) across 5 pages (strided access), but only 4 TLB entries.

What will TLB contents be over time?
How will TLB perform?
TLB REPLACEMENT POLICIES

LRU: evict Least-Recently Used TLB slot when needed

Random: Evict randomly chosen entry

Sometimes random is better than a “smart” policy!
What happens if a process uses cached TLB entries from another process?

1. **Flush TLB on each switch**
   - **Costly** → lose all recently cached translations

2. **Track which entries are for which process**
   - Address Space Identifier
   - Tag each TLB entry with an 8-bit ASID
**TLB Example with ASID**

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x1444 ASID: 12</td>
<td>VPN = 1 ASID: 12</td>
</tr>
<tr>
<td>load 0x1444 ASID: 11</td>
<td>PPN = 2</td>
</tr>
</tbody>
</table>

**TLB:**

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virt</th>
<th>Phys</th>
<th>ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

**PTBR:**

- PT
- PT
- P1
- P2
- P2
- P1
- P1
- P2

**Pagetables (ASID):**

- P1 (ASID 11)
- P2 (ASID 12)

**ASID:**

- 11
- 12
Context switches are expensive
Even with ASID, other processes “pollute” TLB

Architectures can have multiple TLBs
  - 1 TLB for data, 1 TLB for instructions
  - 1 TLB for regular pages, 1 TLB for “super pages”
HW AND OS ROLES

If H/W handles TLB Miss:
- CPU must know where pagetables are
  - CR3 register on x86
  - Pagetable structure fixed and agreed upon between HW and OS
  - HW “walks” the pagetable and fills TLB

If OS handles TLB Miss:
- “Software-managed TLB”
  - CPU traps into OS upon TLB miss.
  - OS interprets pagetables as it chooses
  - Modify TLB entries with privileged instruction
Pages are great, but accessing page tables for every memory access is slow.

- Cache recent page translations → TLB
  - MMU performs TLB lookup on every memory access

TLB performance depends strongly on workload:
  - Sequential workloads perform well
  - Workloads with temporal locality can perform well

In different systems, hardware or OS handles TLB misses.

TLBs increase cost of context switches:
  - Flush TLB on every context switch
  - Add ASID to every TLB entry
1. What problem(s) can be solved by using ASIDs?

   TLBs need to be flushed across context switches.

2. For a hardware-managed TLB miss, which of the following statements are true?
   - HW knows where page tables
   - OS plays no role in TLB miss

3. For a software-managed TLB miss, which of the following statements are true?
   - HW raises exception on a TLB miss
   - OS moves entries in and out
DISADVANTAGES OF PAGING

Additional memory reference to page table $\rightarrow$ Very inefficient
- Page table must be stored in memory
- MMU stores only base address of page table

Storage for page tables may be substantial
- Simple page table: Requires PTE for all pages in address space
  Entry needed even if page not allocated?
SMALLER PAGE TABLES
Why are page tables so large?

Linear page tables:
- Number of virtual pages × PTE size
- = 1M virtual pages × 4 bytes
- = 4MB page table

Waste!
- 1 page
- 3 pages heap
- 4 pages stack

Virt Mem

Phys Mem
**Many invalid PT entries**

<table>
<thead>
<tr>
<th>PFN</th>
<th>valid</th>
<th>prot</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>r-x</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>rw-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>rw-</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>rw-</td>
</tr>
</tbody>
</table>

...many more invalid...

**how to avoid storing these?**

Can we use a better data structure?
Use more complex page tables, instead of just big array

Any data structure is possible with software-managed TLB

- Hardware looks for vpn in TLB on every memory access
- If TLB does not contain vpn, TLB miss
  - Trap into OS and let OS find vpn->ppn translation
  - OS notifies TLB of vpn->ppn for future accesses
OTHER APPROACHES

1. Multi-level Pagetables → HW friendly
   - Page the page tables
   - Page the pagetables of page tables…

2. Inverted Pagetables → Software managed TLBs
MULTILEVEL PAGE TABLES

Goal: Allow page table to be allocated non-contiguously

Idea: Page the page tables
- Creates multiple levels of page tables; outer level “page directory”
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)
**MULTILEVEL PAGE TABLES**

20-bit address:

- outer page (4 bits)
- inner page (4 bits)
- page offset (12 bits)

Diagram:
- Base of page directory
- Address of page table
- Empty
- VPN → PPN mapping
- Memory access

16 entries

No page table allocated

16 entries
How should logical address be structured? How many bits for each paging level?

Goal:

- Each inner page table fits within a page
- PTE size * number PTE = page size

Assume PTE size = 4 bytes
Page size = 2^12 bytes = 4KB
→ # bits for selecting inner page = 10

Remaining bits for outer page:

- 30 – 12 – 10 = 8 bits
## MULTILEVEL TRANSLATION EXAMPLE

<table>
<thead>
<tr>
<th>page directory</th>
<th>page of PT (@PPN:0x3)</th>
<th>page of PT (@PPN:0x92)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPN</td>
<td>valid</td>
<td>PPN</td>
</tr>
<tr>
<td>0x3</td>
<td>1</td>
<td>0x10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x23</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>0x80</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x59</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>
<tr>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x55</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x45</td>
</tr>
</tbody>
</table>

20-bit address:

- outer page (4 bits)
- inner page (4 bits)
- page offset (12 bits)

Translate 0x01ABC
Problem with 2 levels?

Problem: page directories (outer level) may not fit in a page

Solution:
- Split page directories into pieces
- Use another page dir to refer to the page dir pieces.

How large is virtual address space with 4 KB pages, 4 byte PTEs, (each page table fits in page) 4KB / 4 bytes in each directory

1 level:
2 levels:
3 levels:
On TLB miss: lookups with more levels more expensive

Assume 3-level page table
Assume 256-byte pages
Assume 16-bit addresses
Assume ASID of current process is 211

How many physical accesses for each instruction? (Ignore ops changing TLB)

(a) 0xAA10: movl 0x1111, %edi

(b) 0xBB13: addl $0x3, %edi

(c) 0x0519: movl %edi, 0xFF10
INVERTED PAGE TABLE

Only store entries for virtual pages w/ valid physical mappings

Naïve approach:
  - Search through data structure <ppn, vpn+asid> to find match
  - Too much time to search entire table

Better:
  - Find possible matches entries by hashing vpn+asid
  - Smaller number of entries to search for exact match

Managing inverted page table requires software-controlled TLB
Consider a virtual address space of 16KB with 64-byte pages.

1. How many bits will we have in our virtual address for this address space?

2. What is the total number of entries in the Linear Page Table for such an address space?

3. Consider a two-level page table now with a page directory. How many bits will be used to select the inner page assuming PTE size = 4 bytes?
<table>
<thead>
<tr>
<th>PPN</th>
<th>valid</th>
<th>PPN</th>
<th>valid</th>
<th>PPN</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3</td>
<td>1</td>
<td>0x10</td>
<td>1</td>
<td>0x10</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0x23</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0x80</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0x59</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0x92</td>
<td>1</td>
<td>0x45</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

20-bit address:
- outer page (4 bits)
- inner page (4 bits)
- page offset (12 bits)

translate 0xFEED0
SUMMARY: BETTER PAGE TABLES

Problem: Simple linear page tables require too much contiguous memory

Many options for efficiently organizing page tables

If OS traps on TLB miss, OS can use any data structure
   – Inverted page tables (hashing)

If Hardware handles TLB miss, page tables must follow specific format
   – Multi-level page tables used in x86 architecture
   – Each page table fits within a page
NEXT STEPS

Project 3: In progress

Next class: Swapping!