Lecture 18
Virtual Memory 2
Reminder: Virtual Memory

- **Goal of virtual memory**
  - Abstraction of the storage resources of the machine
  - Protection and privacy: Processes cannot access each other’s data

- **Today’s lecture**
  - Translation Lookaside Buffer (TLB) for address translation
  - Caches with virtual memory
  - Hierarchical page table
  - Page replacement algorithm
  - Page sharing and memory mapping
  - Copy-on-Write
Translation Lookaside Buffer (TLB)

Problem: Address translation is very expensive!
Each reference requires accessing page table

Solution: *Cache translations in TLB*

- **TLB hit** ⇒ *Single-cycle translation*
- **TLB miss** ⇒ *Access page table to refill TLB*

![Diagram of TLB with virtual and physical addresses]

- **VPN** = virtual page number
- **PPN** = physical page number

**Notes:**
- Virtual address
- Fault? hit?
- Physical address
TLB Designs

- Typically 32-128 entries, 4 to 8-way set-associative
  - Modern processors use a hierarchy of TLBs
    (e.g., 128-entry L1 TLB + 2K-entry L2 TLB)

- Switching processes is expensive because TLB has to be flushed
  - Alternatively, include process ID in TLB entries to avoid flushing

- Handling a TLB miss: Look up the page table (a.k.a. “walk” the page table). If the page is in memory, load the VPN→PPN translation in the TLB. Otherwise, cause a page fault
  - Page faults are always handled in software
  - But page walks are usually handled in hardware using a memory management unit (MMU)
    - RISC-V, x86 access page table in hardware
Address Translation
Putting it all together

Virtual Address

TLB Lookup

Page Table Lookup

Protection Check

Update TLB

Protection Fault

Page Fault (OS loads page)

Physical Address (to mem)

SEGFAULT

hit

miss

\[ \text{hardware} \]
\[ \text{hardware or software} \]
\[ \text{software} \]

\[ \notin \text{memory} \]
\[ \in \text{memory} \]

denied

permitted

Resume process at faulting instruction
Using Caches with Virtual Memory

Virtually-Addressed Cache

- FAST: No virtual→physical translation on cache hits
- Problem: Must flush cache after context switch

Physically-Addressed Cache

- Avoids stale cache data after context switch
- SLOW: Virtual→physical translation before every cache access
Best of Both Worlds: Virtually-Indexed, Physically-Tagged Cache (VIPT)

OBSERVATION: If cache index bits are a subset of page offset bits, tag access in a physical cache can be done in parallel with TLB access. Tag from cache is compared with physical page address from TLB to determine hit/miss.

Problem: Limits # of bits of cache index → can only increase cache capacity by increasing associativity!
Problem: Linear Page Table Size

- With 32-bit addresses, 4 KB pages & 4-byte PTEs:
  - \(2^{20}\) PTEs, i.e., 4 MB page table per process
  - We often have hundreds to thousands of processes per machine... use GBs of memory just for page tables?

- Use larger pages?
  - Internal fragmentation (not all memory in a page is used)
  - Larger page fault penalty (more time to read from disk)

- What about a 64-bit virtual address space?
  - Even 1MB pages would require \(2^{44}\) 8-byte PTEs (35 TB!)

- Solution: Use a hierarchical page table
Hierarchical Page Table

Virtual Address

31  22  21  12  11  0

p1  p2  offset

10-bit  10-bit
L1 index L2 index

Root of the Current Page Table

(Processor Register)

Level 1 Page Table

Level 2 Page Tables

Data Pages

page in primary memory

page in secondary memory

PTE of a nonexistent page
Hierarchical Page Table: Pros & Cons

- Page table memory is proportional to amount of memory used by process
  - Assume a process only uses 8MB of virtual memory
  - Memory usage:
    - L1: $2^{10}$ entries * 4 bytes/entry = 4 KB
    - L2: 8MB/4KB = $2^{11}$ pages -> 2 page tables in L2 -> 2 * 4KB = 8KB
  - Compare to single level: 4MB -> 12KB

- Each page table walk now needs multiple memory accesses
  - But TLBs make page table walks rare
Page Replacement Algorithm

- When physical memory is full, which physical page is the victim to be evicted on a page fault?
- Goal: Minimize page faults and optimize the overall system performance
- Common page replacement algorithms
  - Least Recently Used (LRU):
    - Assumption: The least recently used page is likely to be the least needed in the near future
    - Can be expensive to implement in hardware or software, as it requires maintaining a double linked list or similar data structure to track the access order of pages
  - CLOCK Algorithm:
    - An approximation of the LRU algorithm by evicting not recently used page
Least Recently Used (LRU)

- Example implementation of LRU: Hash map and double linked list
- Large overhead of maintaining such a large map and list in a system

```
<table>
<thead>
<tr>
<th>Value</th>
<th>Pointer List</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Value</th>
<th>Pointer List</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>
```

Head (Mostly recent used)
Tail (Least used)

#Physical pages: 3
Access pattern: A, B, C, D
CLOCK Page Replacement Algorithm

- CLOCK approximates LRU by finding the **not recently used** page
  - It maintains a circular list of pages resident in memory
  - Each page has a use bit that is set to 1 when the page is accessed
  - The clock hand points to the potential victim. When a page fault occurs:

```plaintext
while (victim page not found) do:
    if (used bit of the current page == 0)
        replace current page
    else
        reset used bit of the current page
    end if
end while
```

#Physical pages: 3
Access pattern: A, B, C, D
CLOCK approximates LRU by finding the not recently used page.
- It maintains a circular list of pages resident in memory.
- Each page has a use bit that is set to 1 when the page is accessed.
- The clock hand points to the potential victim. When a page fault occurs:
  
  ```plaintext
  while (victim page not found) do:
    if (used bit of the current page == 0)
      replace current page
    else
      reset used bit of the current page
    end if
  end while
  ```

# Physical pages: 3
Access pattern: A, B, C, D
CLOCK Page Replacement Algorithm

- CLOCK approximates LRU by finding the not recently used page
  - It maintains a circular list of pages resident in memory
  - Each page has a use bit that is set to 1 when the page is accessed
  - The clock hand points to the potential victim. When a page fault occurs:

```
while (victim page not found) do:
  if (used bit of the current page == 0)
    replace current page
  else
    reset used bit of the current page
  end if
  move hand to the next
end while
```

#Physical pages: 3
Access pattern: A, B, C, D
CLOCK approximates LRU by finding the not recently used page.
- It maintains a circular list of pages resident in memory.
- Each page has a use bit that is set to 1 when the page is accessed.
- The clock hand points to the potential victim. When a page fault occurs:

```plaintext
while (victim page not found) do:
    if (used bit of the current page == 0)
        replace current page
    else
        reset used bit of the current page
    end if
move hand to the next
end while
```

*Physical pages: 3
Access pattern: A, B, C, D
CLOCK approximates LRU by finding the not recently used page:
- It maintains a circular list of pages resident in memory.
- Each page has a use bit that is set to 1 when the page is accessed.
- The clock hand points to the potential victim. When a page fault occurs:

```plaintext
while (victim page not found) do:
    if (used bit of the current page == 0)
        replace current page
    else
        reset used bit of the current page
    end if
move hand to the next
end while
```

# Physical pages: 3
Access pattern: A, B, C, D
Pros and Cons

- **LRU Algorithm:**
  - Pros: A good approximation of the optimal page replacement algorithm.
  - Cons: LRU can be expensive to implement in hardware or software, as it requires maintaining a list or similar data structure to track the access order of pages.

- **CLOCK Algorithm:**
  - Pros: More efficient to implement than LRU, only requires a circular buffer and a single reference bit per page
  - Cons: It does not maintain a precise ordering of pages based on access times
Trade-off of Different Page Sizes

- Different page sizes introduce different trade-off for:
  - Size of page tables
    - Smaller page size -> larger page table
  - #page fault with applications
    - Smaller page size -> more page fault
  - Internal fragmentation
    - Smaller page size -> less internal fragmentation
  - Time to start a process
    - Smaller page size -> quicker time to start small process
  - TLB coverage/TLB miss rate
    - Smaller page size -> low coverage and higher TLB miss rate
- General trend toward larger pages: 512 B -> 64KB (1978 -> 2000)
Page Sharing

- Sharing pages allows mapping multiple pages to the same physical page

- Useful in many circumstances
  - Multiprocessing applications that need to share data.
  - Sharing read only data for applications, OS, etc.

Reference: Stanford EE108b
Page Sharing and Memory Mapping

- Process 1 creates shared memory object with `shm_open()`
- Map the shared memory object to its virtual memory with `mmap()`

```
1. shm_open("/obj", O_CREAT...)  
2. mmap()                         
```

Diagram:
- Shared object
- Process 1 virtual memory
- Physical memory
- ` shm_open("/obj", O_CREAT...) `
Page Sharing and Memory Mapping

- Process 2 accesses the shared memory object with the name
- Map the shared memory object to its virtual memory
  - Note: the virtual memory addresses can be different for process 1 and 2.

1. `shm_open("/obj", O_RDONLY...)`  
2. `mmap()`
Copy-on-Write

- Copy-on-Write (COW)
  - Process1 and Process2 initially share the same pages
  - Only copy page if one of the processes wants to modify some page
- Pros:
  - Fast process creation
  - Efficient memory usage: Processes may share most data (e.g., .text code segment)
- Cons:
  - Increase system complexity
Copy-on-Write: Example

- Process1 creates a child process
  - The child process gets a copy of the parent’s page table
  - All pages now are read-only
  - Both processes can access the same copy of physical memory

Both page tables point to the same physical pages!
Copy-on-Write: Example

- What if the child process writes the page?
  - Protection fault
    - OS copies the page and maps it to the child’s page table
  - Child process modifies its private copy

[Diagram showing the process of copy-on-write, with arrows indicating the flow of memory access and protection faults.]
Copy-on-Write: Example

- What if the parent process writes the page?
  - Protection fault
    - OS copies the page and maps it to the parent’s page table
  - Each process modifies its private copy!

```
<table>
<thead>
<tr>
<th>Process 1 virtual memory</th>
<th>Process 1 page table</th>
<th>Physical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Child process virtual memory</td>
<td>Child process page table</td>
<td>R/W</td>
</tr>
</tbody>
</table>
```

- Copy
Summary

- TLBs make paging efficient by caching the page table
- Trade-off of different page sizes
  - Size of page table, #page fault, fragmentation, TLB coverage (TLB miss rate)
- Hierarchical page table
  - Page table memory is proportional to the amount of memory used by process
- Page replacement algorithms:
  - LRU: A good approximation of the optimal page replacement algorithm
  - CLOCK: A more efficient to implementation than LRU
- Page sharing and copy-on-write
  - Pages can be shared by processes
Thank you!

Next lecture: I/O and Exceptions