Memory Virtualization:
Paging: Summary

Jan Reineke
Universität des Saarlandes

Goals of Memory Virtualization

Transparency

- Processes are not aware that memory is shared
- Works regardless of number and/or location of processes

Protection

- Integrity: Cannot corrupt OS or other processes
- Privacy/confidentiality: Cannot read data of other processes

Efficiency

- Do not waste memory resources (minimize fragmentation)

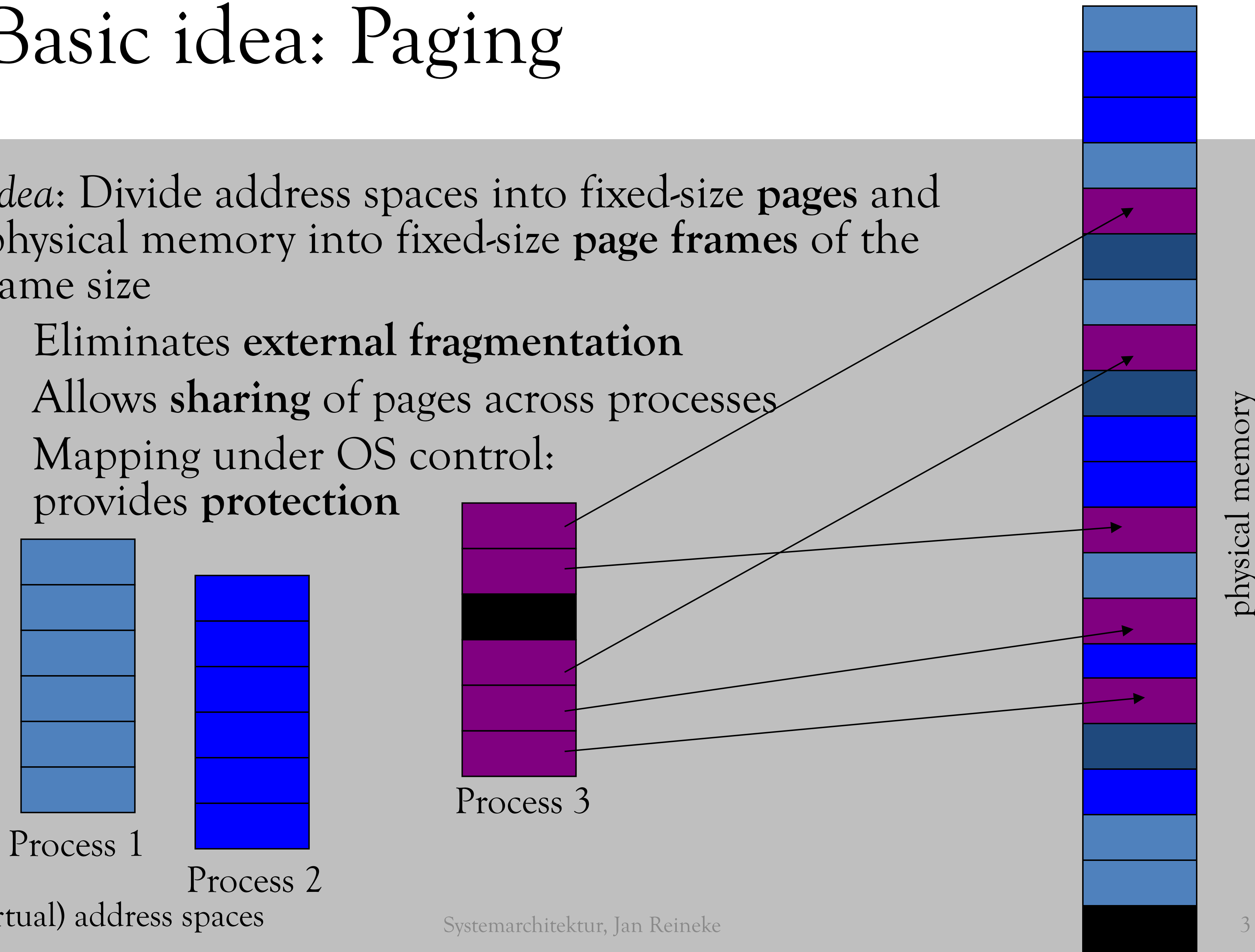
Sharing

- Cooperating processes can share portions of address space

Basic idea: Paging

Idea: Divide address spaces into fixed-size **pages** and physical memory into fixed-size **page frames** of the same size

- Eliminates **external fragmentation**
- Allows **sharing** of pages across processes
- Mapping under OS control: provides **protection**



(Virtual) address spaces

Disadvantages of "naive" paging

0. Internal fragmentation:

Wasted memory grows with larger pages

1. Substantial storage for page tables

- Simple page table: one entry for each page in address space, even if not allocated

Solution: **Multi-level page tables**

2. Additional memory access to page table upon every memory access

Solution: **Translation lookaside buffer (TLB)**

3. Size of physical memory limits memory allocation

Solution: **Swapping to disk**

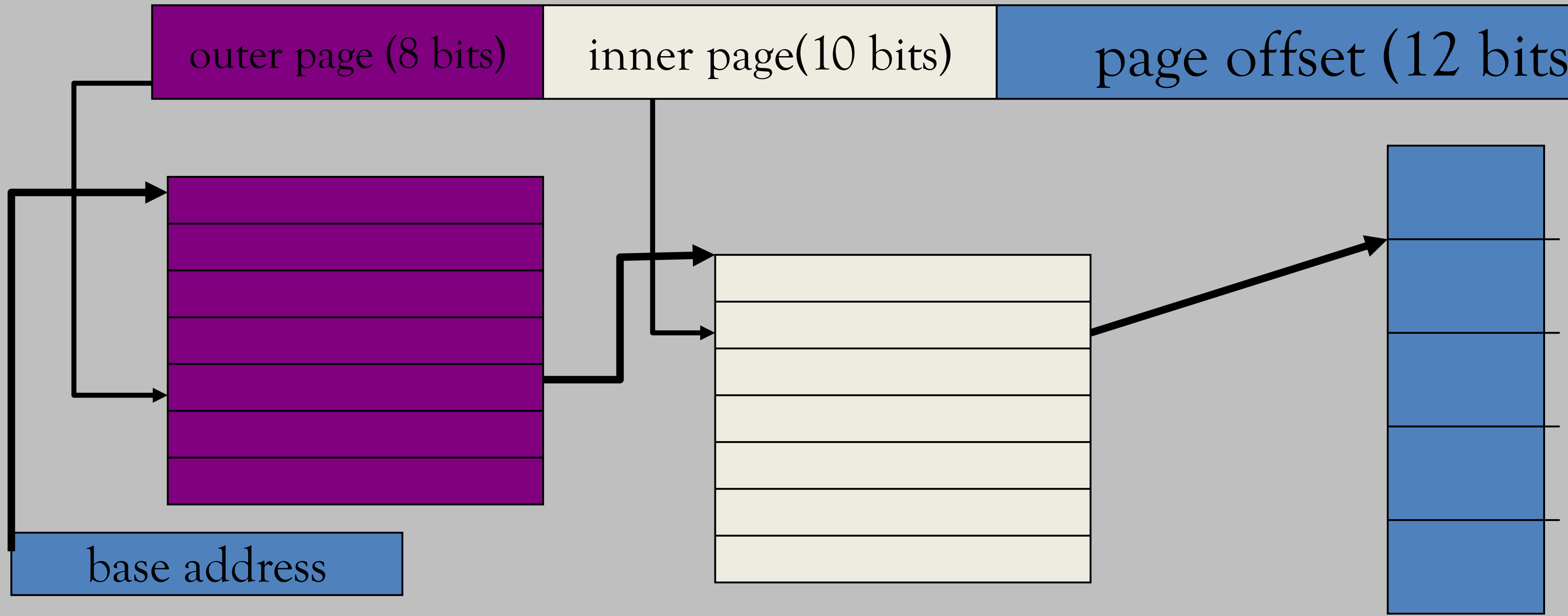
1. Multi-level page tables

Goal: Allow each page table to be allocated non-contiguously

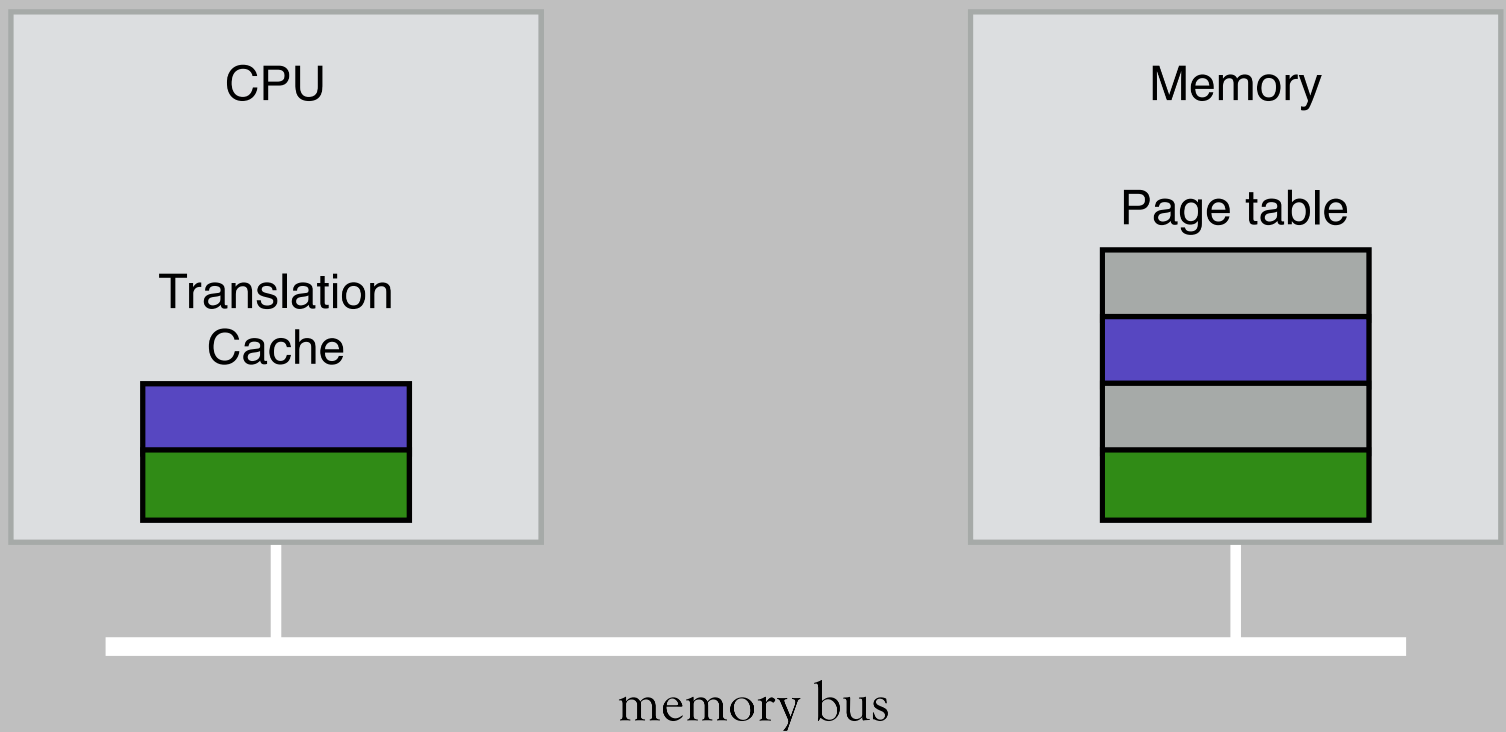
Idea: **Hierarchical page tables**

- Several translation levels, inner tables stored in pages
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)

30-bit address:



2. TLB = Translation “Cache”



TLB: **T**ranslation **L**ookaside **B**uffer

3. Swapping of unused pages

Requirements:

- **Mechanism** to manage location of each page: in memory or on disk
- **Policy** to determine *which* pages to keep in memory

3. Swapping: Mechanisms

Each page in virtual address space maps to one of three locations:

- Phys. memory: Small, fast, expensive
- Disk: Large, slow, cheap
- nowhere (not allocated)

Extend page tables with an extra bit: `present`

- Permissions (`r/w`), `valid`, `present`
- Page in memory → `present = 1`
- Page on disk → `present = 0`
  - PTE points to block on disk
  - Causes trap into OS when page is referenced: "**page fault**"

3. Swapping: Policies

Goal: Minimize number of page faults

- Page faults require milliseconds to handle (reading from disk)
- Implication: OS has plenty of time to make good decision

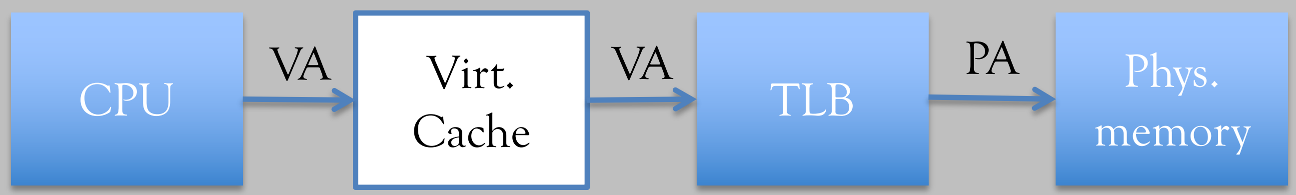
OS has two decisions:

- **Page selection**:
  When should a page (or pages) on disk be **brought into** memory?
  → Demand paging, prefetching, hints
- **Page replacement**:
  Which resident page (or pages) in memory should be **thrown out** to disk?
  → OPT, FIFO, LRU
  → efficiently implementable: CLOCK

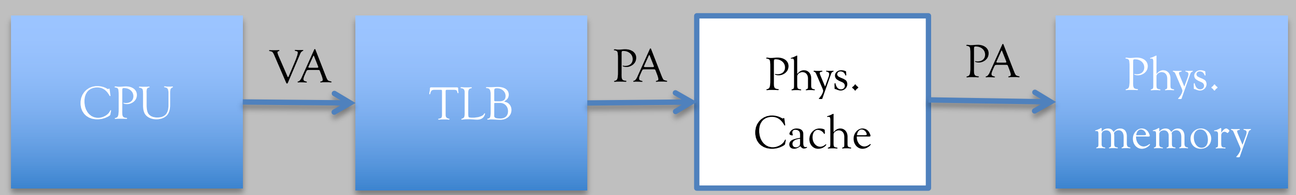
Open question: How does cache interact with virtual memory?

Alternatives:

1. Caches associate data with virtual addresses



2. Caches associate data with physical addresses



Option 1: Use virtual addresses

Advantage:

No address translation required before cache access

Disadvantages:

- Need to distinguish virtual addresses of different processes
  → ASIDs (address space identifiers)
- Aliases due to sharing: two different virtual addresses may map to the same physical page
  → Change of cache under virtual address A not visible under virtual address B
  → Possible solution: direct-mapped cache + OS ensures that aliases have same index (A and B never in cache at same time)
  → **massive restriction**!

Option 2: Use physical addresses

Advantage:

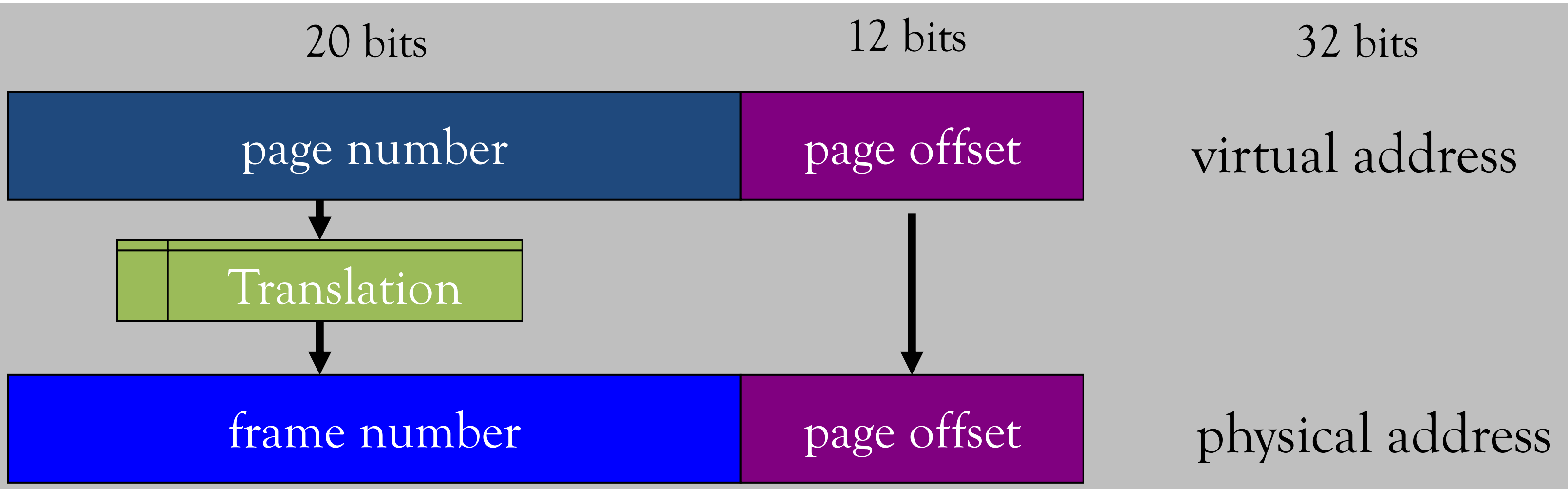
- No problems with aliases

Disadvantage:

- Must translate address **before** cache access

Approach:
"virtually-indexed, physically-tagged" caches

Virtually-indexed, physically-tagged



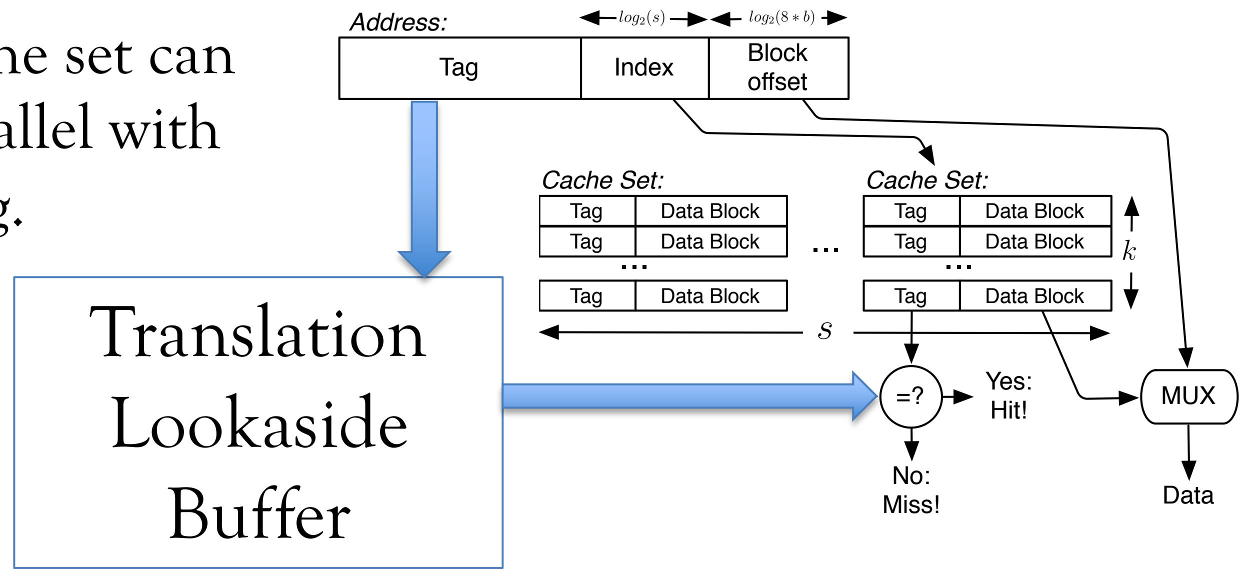
Observation: Page offset is **the same** for virtual and physical address

→ Pick cache parameters, so that page offset bits determine the index in the cache completely

→ Use TLB to translate page number into frame number in **parallel** with access to cache set

Virtually-indexed, physically-tagged

Access to tags in cache set can be performed in parallel with translation of the tag.



Constraints on cache parameters for this to work?

Assumptions:

- k is the associativity (Size of individual cache sets)
- b is the block size
- s is the number of cache sets
- Page size 4 KB → 12 bits for page offset

→ index + block offset ≤ 12

→ $\log_2 s + \log_2 b \leq 12$

→ $\log_2 s*b \leq 12$

→ $s*b \leq 4096$

→ Capacity / associativity ≤ 4096

Quiz: Possible scenarios upon a memory access

Assumption: two-level page table, physically-tagged cache

1. TLB hit
    1. Cache hit upon memory access
    2. Cache miss upon memory access
2. TLB miss
    1. Cache hit upon first access to page table
        1. Cache hit upon second access to page table
            1. Cache hit upon memory access
            2. Cache miss upon memory access
            3. Page fault: OS fetches missing page from disk
        2. Cache miss upon second access to page table
            1. …

Summary: Paging

- Complex interplay of HW and OS
  - **TLBs** for performance
    - parallel access to TLB and cache with virtually-indexed cache
  - **Multi-level page tables** against waste of memory
  - **Swapping** to provide illusion of more available memory