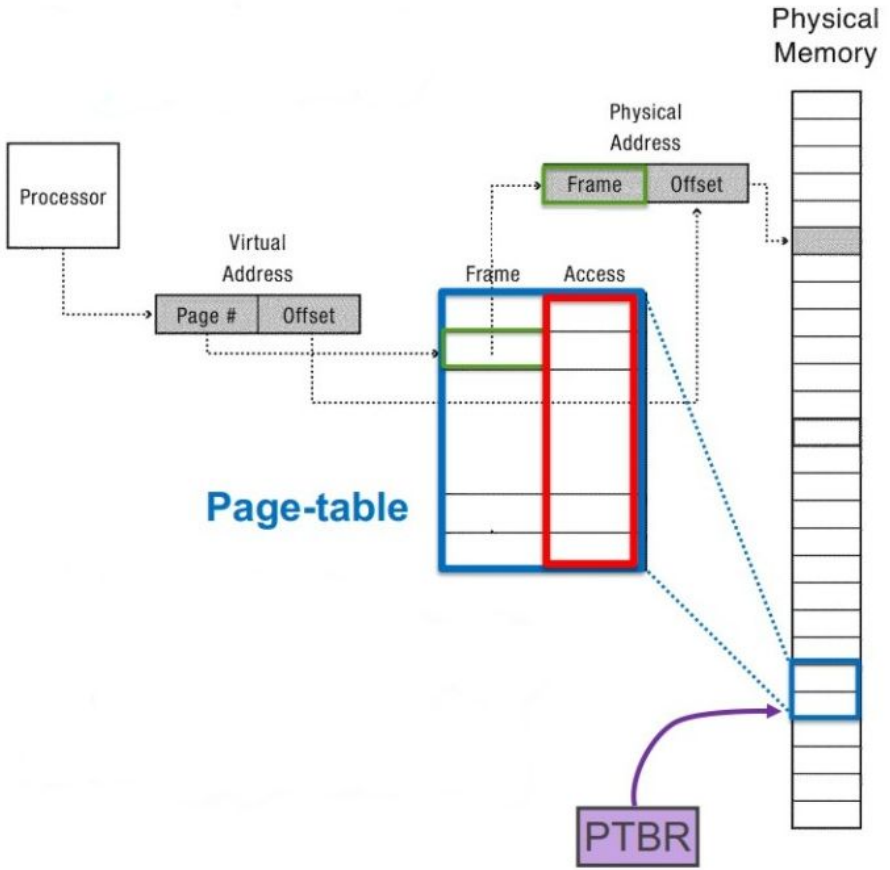
ILLINOIS TECH

College of Computing

CS 450 Operating Systems Paging and TLB

Yue Duan

Paging

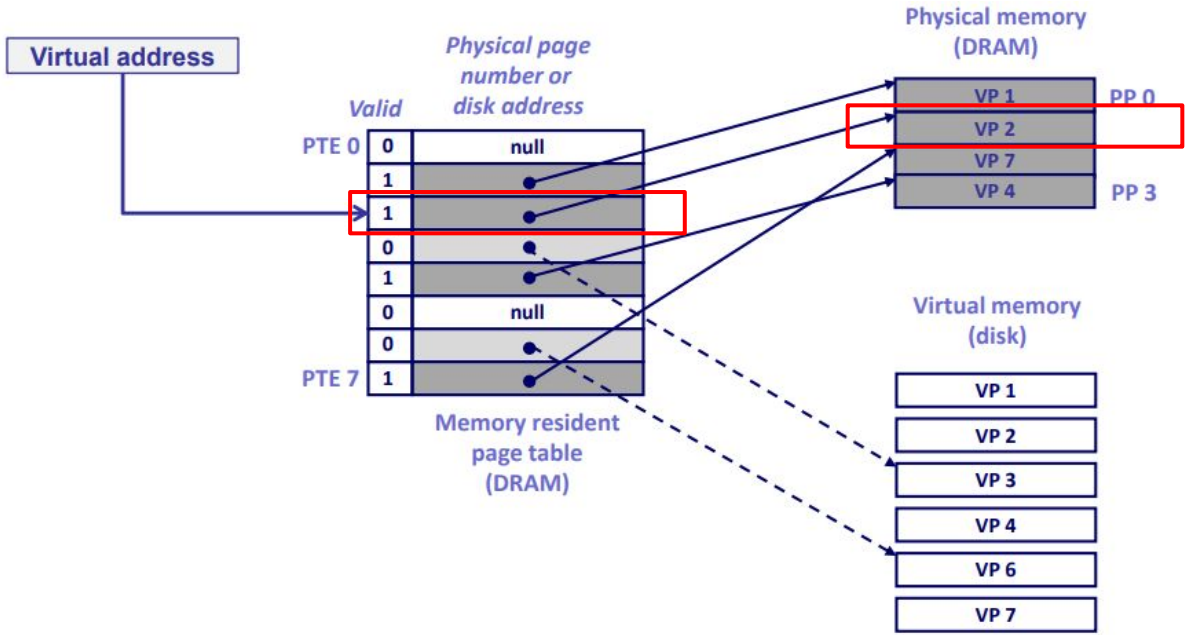

Physical
Memory
Physical
Address
Frame
Offset
Processor
Virtual
Address
Page #
Offset
Frame
Access
Page-table
PTBR

Paging Example

- Consider: 32-bit address space, pages are 4KB
  - Offset is 12 bits (because 4KB = 2^12 bytes)
  - VPN is 20 bits
- Virtual address is 0x7468
  - Virtual page is 0x7, offset is 0x468
- Page table entry 0x7 contains 0x2
  - Page frame number is 0x2
  - Seventh virtual page is at address 0x2 (2nd physical page)
- Physical address = 0x2 || 0x468 = 0x2468

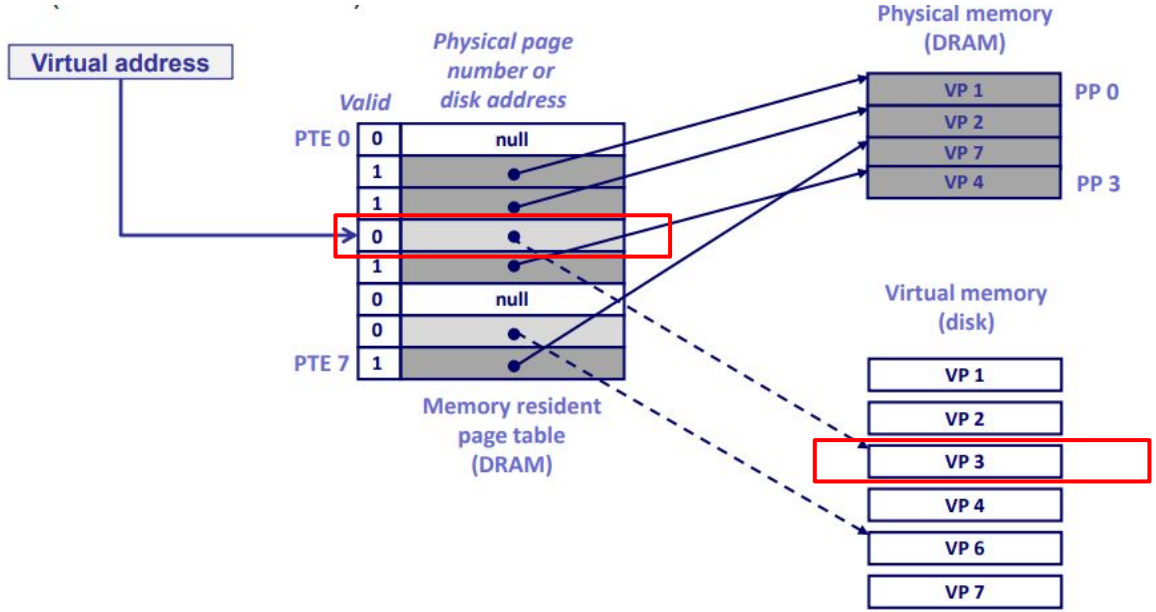
Paging Hit

- reference to VM word is in physical memory (DRAM cache hit)



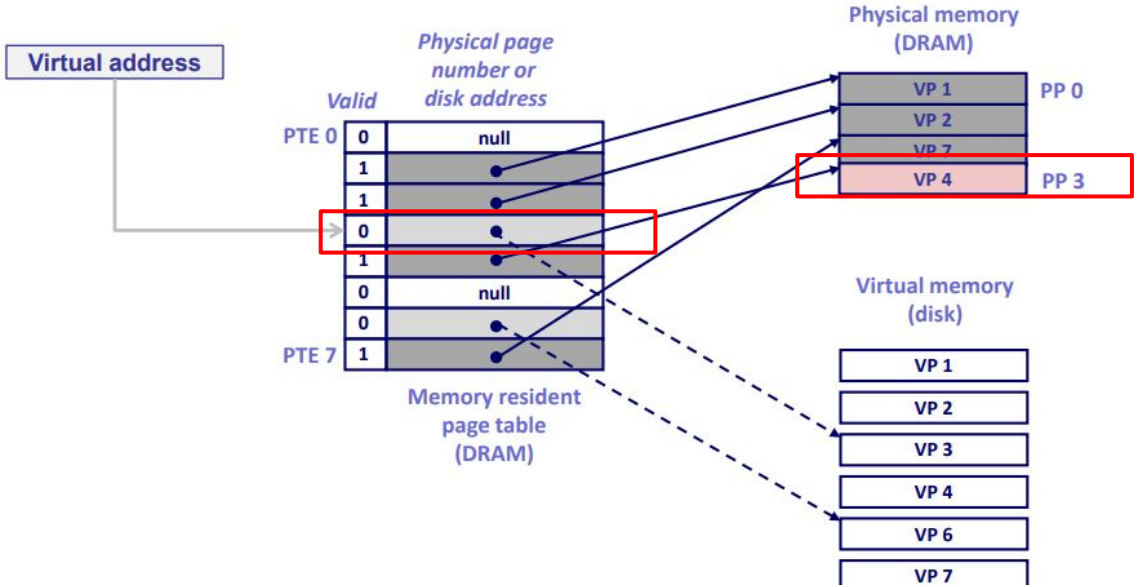
Paging Fault

- reference to VM word is NOT in physical memory (DRAM cache miss)



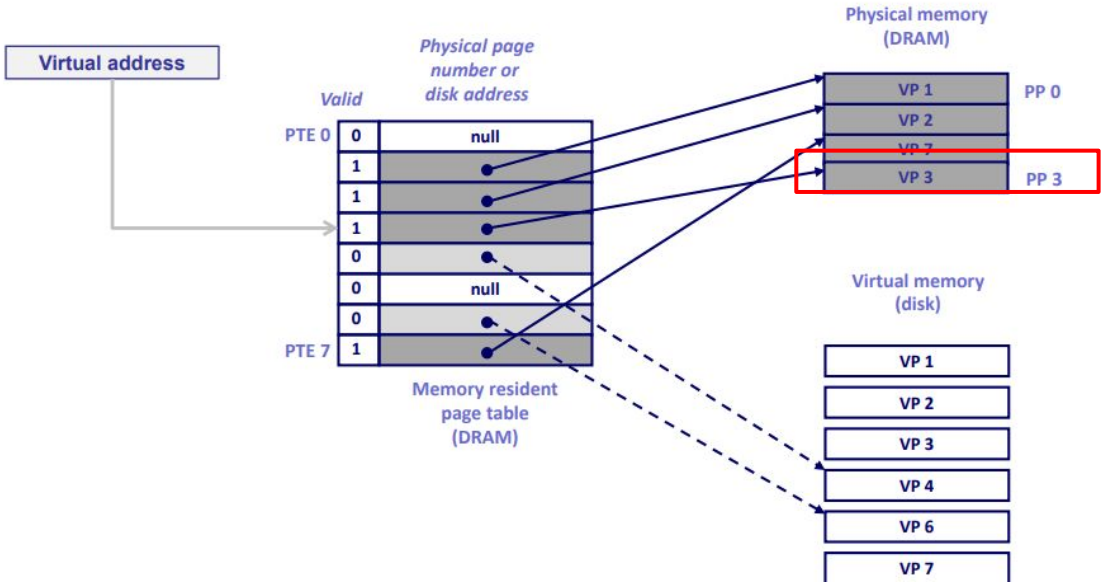
Handling Paging Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)



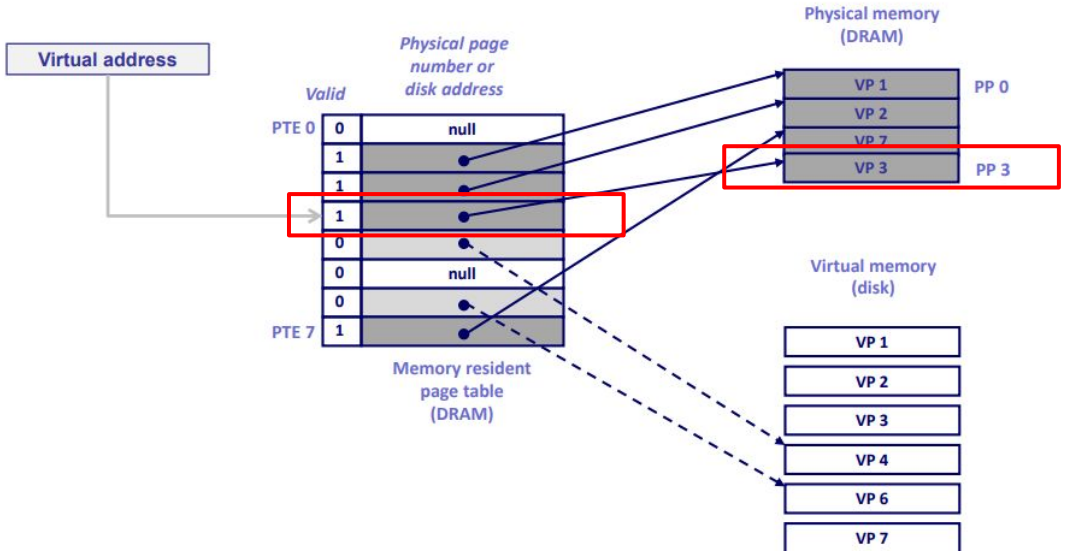
Handling Paging Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)

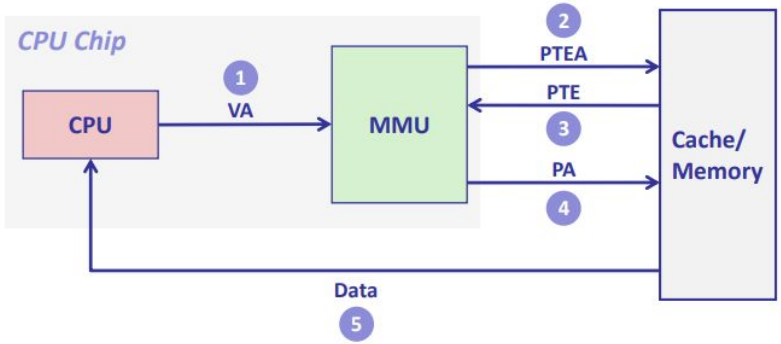


Handling Paging Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!

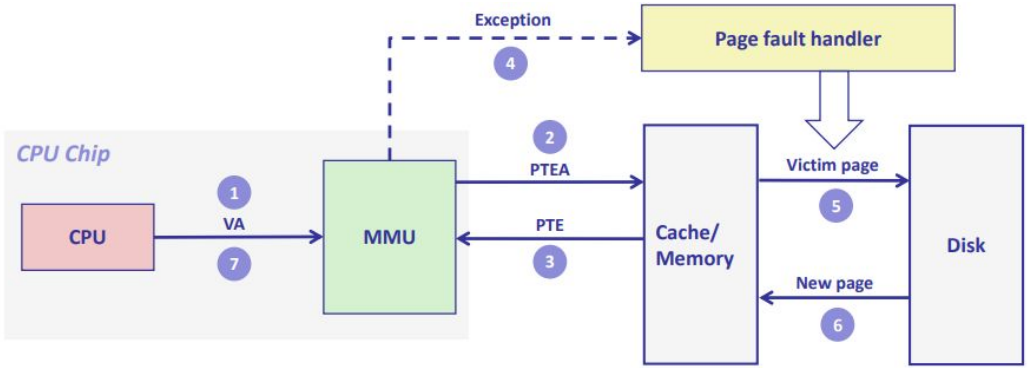


Address Translation: Page Hit



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

Address Translation: Page Fault



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim (and, if dirty, pages it out to disk)
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction

Advantages of Paging

- Easily accommodates transparency, isolation, protection and sharing
- No external fragmentation
- Fast to allocate and free page frames
  - Alloc: No searching for suitable free space; pick the first free page frame
  - Free: Doesn't have to coalesce with adjacent free space; just add to the list of free page frames
  - Simple data structure (bitmap, linked list, etc.) to track free/allocated page frames

Disadvantages of Paging

- **Inefficiency**: memory references to page table
  - Page table must be stored in memory
  - MMU stores only base address of page table
- **Storage overhead**:
  - Internal fragmentation
    - Page size may not match size needed by process
    - Make pages smaller? But then…
  - page tables may be large
    - Simple page table: requires PTE for all pages in address space
    - Entry needed even if page not allocated ?

Page Translation Steps

%rip = 0x0010

```
0x0010: movl 0x1100, %edi
0x0013: addl $0x3, %edi
0x0019: movl %edi, 0x1100
```

Assume PT is at phys addr **0x5000**
Assume PTE's are **4 bytes**
Assume **4KB pages**

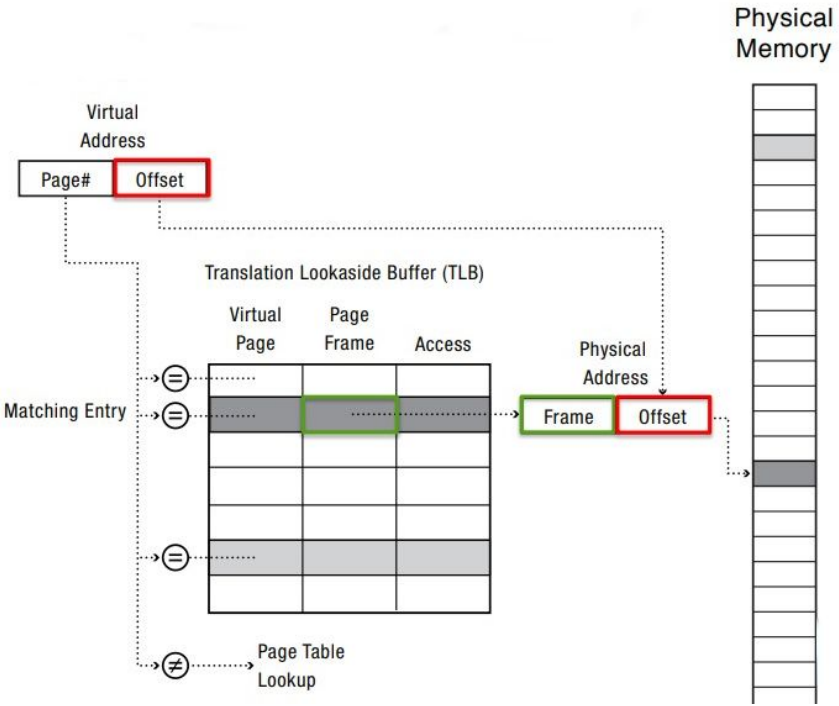
Simplified view of page table

2
0
80
99

- Physical Memory Accesses with Paging
- Fetch instruction at virtual addr 0x0010
  - Access page table to get PFN for VPN 0
  - **Mem ref 1: 0x5000**
  - Learn VPN 0 is at PFN 2
  - Fetch instruction at 0x2010 (Mem ref 2)
- Exec, load from virtual addr 0x1100
  - Access page table to get PFN for VPN 1
  - **Mem ref 3: 0x5004**
  - Learn VPN 1 is at PFN 0
  - movl from 0x0100 into %edi (Mem ref 4)

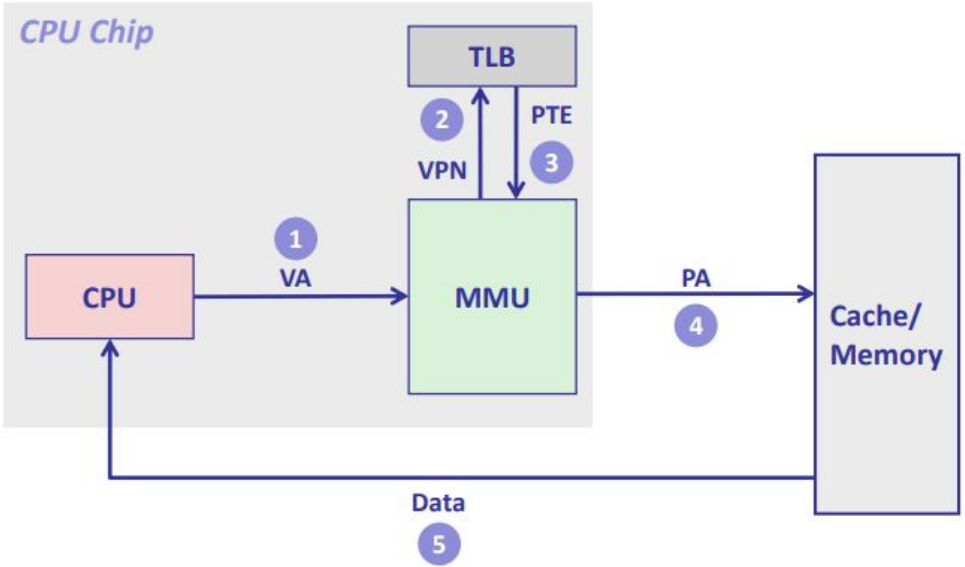
Speeding up Translation by Caching

- Translation Lookaside Buffer (TLB)
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages



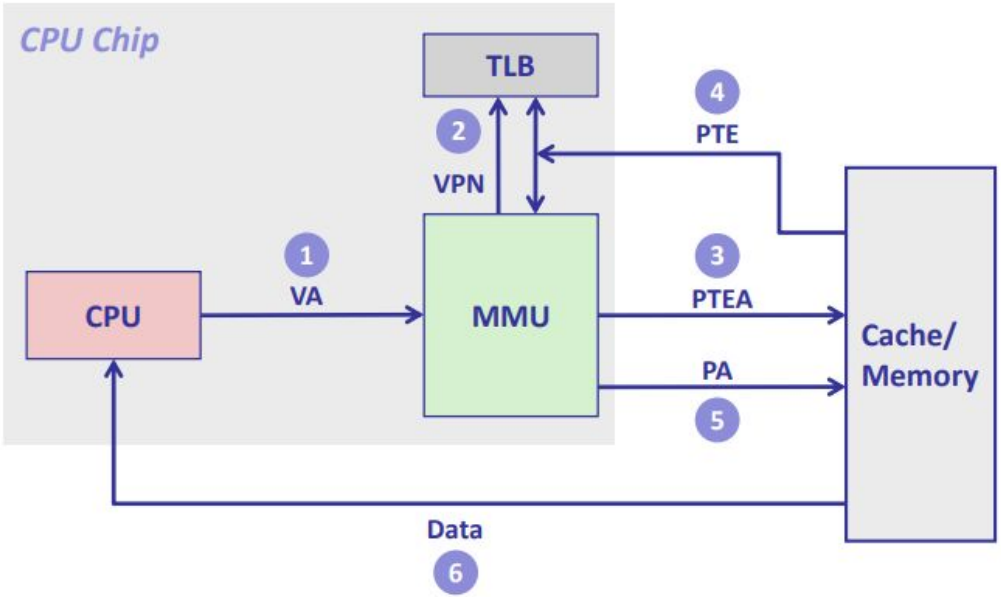
Address Translation with TLB

- Access TLB before you access memory
- A TLB hit eliminates a memory access



Address Translation with TLB

- A TLB miss incurs an additional memory access (the PTE)
- Fortunately, TLB misses are rare. Why?



TLB Entry

- TLB Entry
  - fully associative
  - any given translation can be anywhere in the TLB
  - hardware will search the entire TLB in parallel

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

Array Iterator (w/ TLB)

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

Assume 'a' starts at 0x1000

Ignore instruction fetches and access to 'i'

Assume following virtual address stream:

load 0x1000

load 0x1004

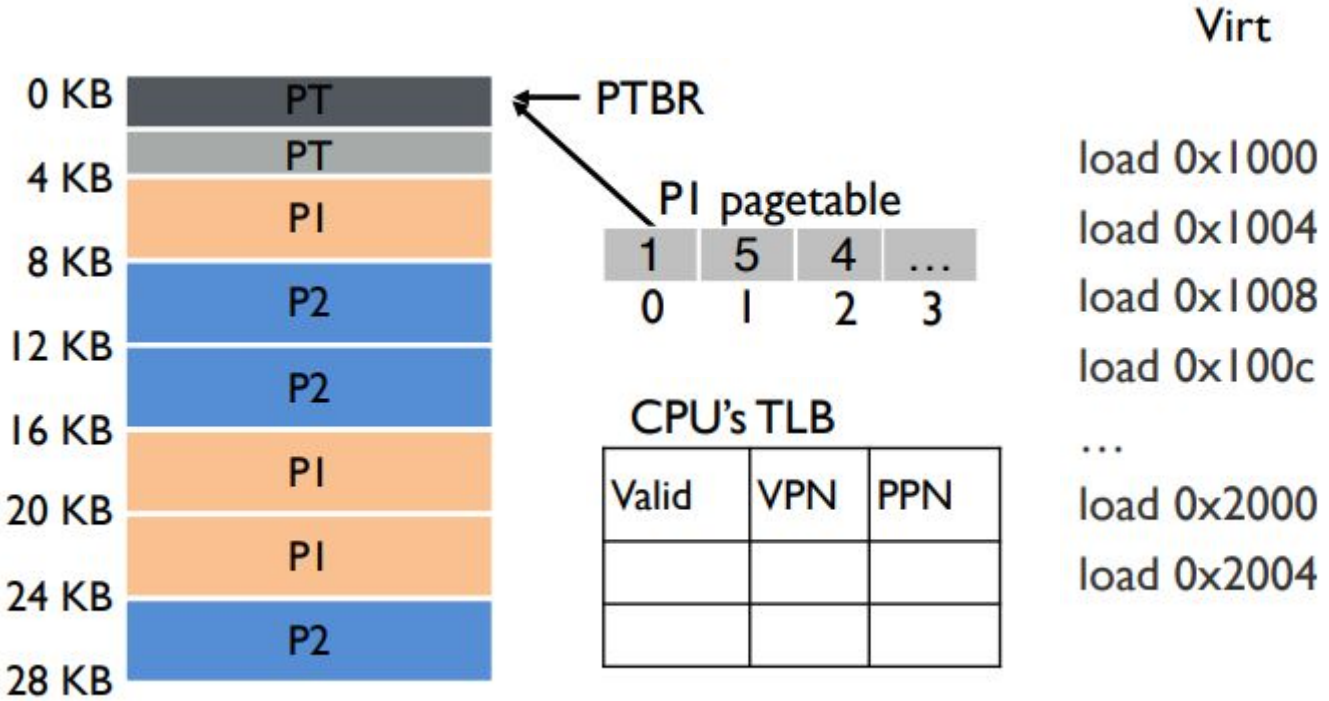
load 0x1008

load 0x100C

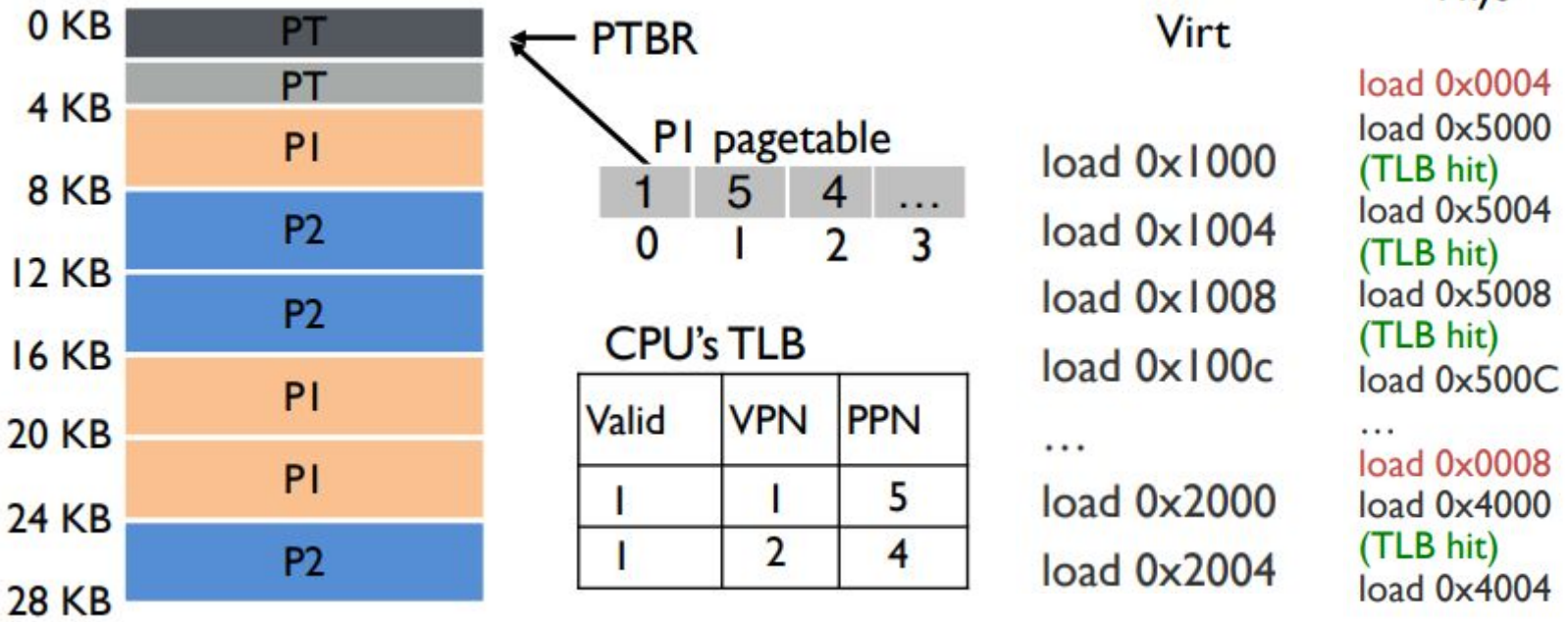
…

- What will TLB behavior look like?

Array Iterator (w/ TLB)



Array Iterator (w/ TLB)



Performance of TLB

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

Would hit rate get better or worse with smaller pages?

- Miss rate of TLB:
  # TLB misses / # TLB lookups
- # TLB lookups?
  = number of accesses to a = 2048
- # TLB misses?
  = number of unique pages accessed
  = 2048 / (elements of ‘a’ per 4K page)
  = 2K / (4K / sizeof(int)) = 2K / 1K
  = 2
- Miss rate?
  = 2/2048 = 0.1%
- Hit rate?
  (1 – miss rate) = 99.9%

Workload Access Patterns

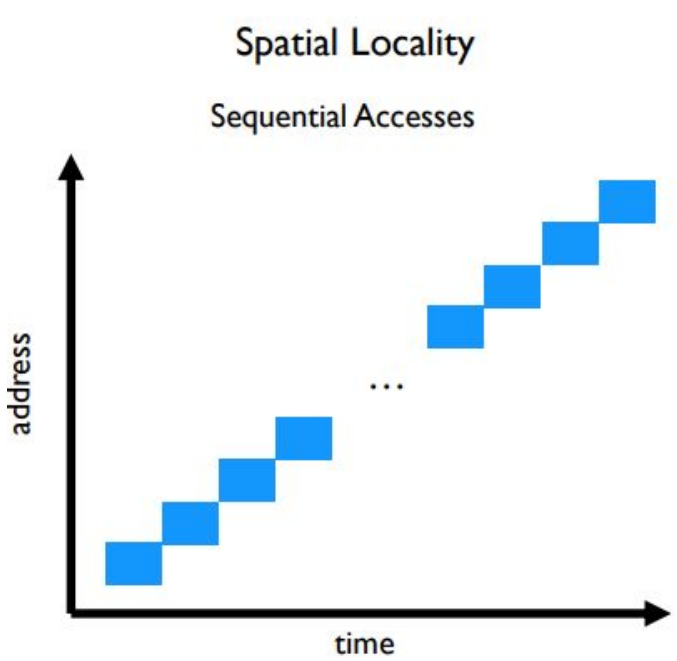
Workload A

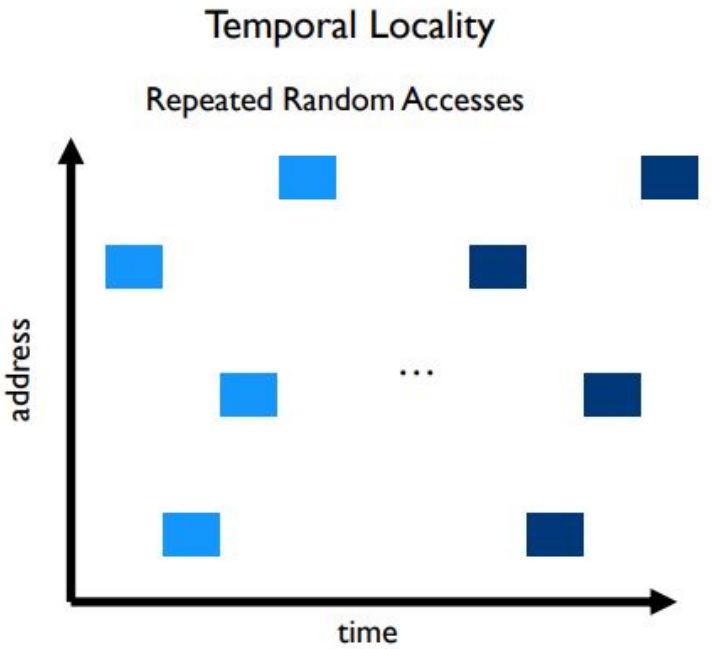
```
int sum = 0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```

Workload B

```
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





Workload Locality

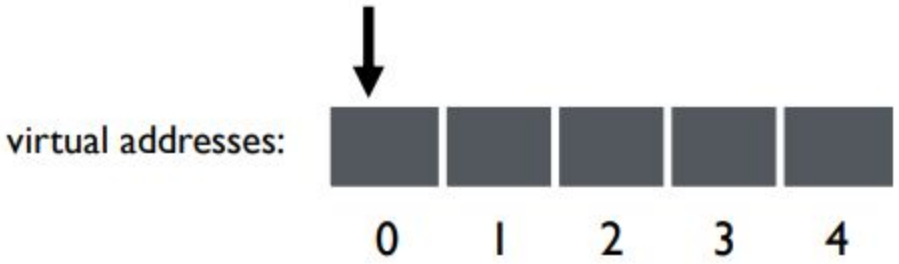
- Spatial Locality:
  - future access will be to nearby addresses
- Temporal Locality:
  - future access will be repeats to the same data
- What TLB characteristics are best for each type?

Workload Locality

- Spatial:
  - Access same page repeatedly
  - need same vpn => ppn translation
  - Same TLB entry re-used
- Temporal:
  - Access same address near in future
  - Same TLB entry re-used in near future
  - How near in future?
  - How many TLB entries are there?

TLB Replacement Policies

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



- 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?

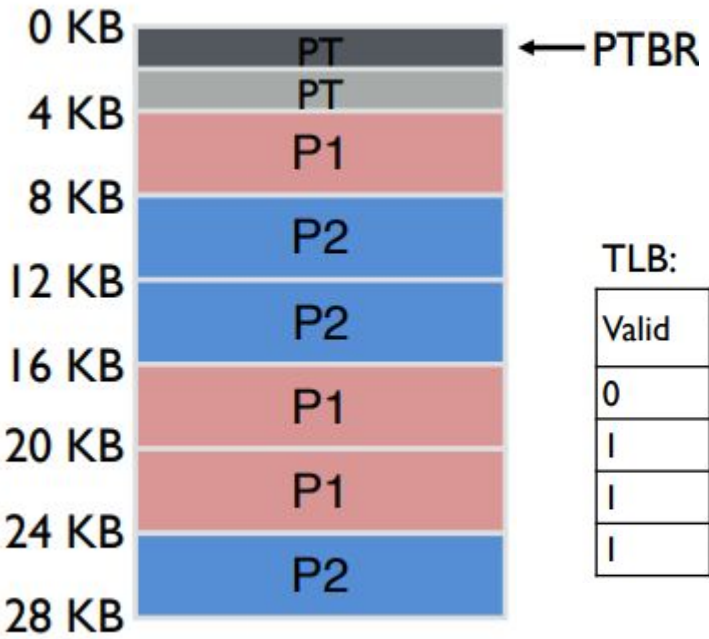
TLB Replacement Policies

- Random: Evict randomly chosen entry
- Sometimes random is better than a “smart” policy!

Context Switches

- What happens if a process uses cached TLB entries from another process?
- Flush TLB on each switch
  - Costly ⇒ lose all recently cached translations
- Track which entries are for which process
  - Address Space Identifier
  - Tag each TLB entry with an 8-bit ASID

TLB Example with ASID



1	5	4	…	P1 pagetable (ASID 11)
6	2	3	…	P2 pagetable (ASID 12)

TLB:

Valid	Virt	Phys	ASID
0	1	9	11
1	1	5	11
1	1	2	12
1	0	1	11

	Virtual	Physical
P2	load 0x1444 ASID: 12	0x2444
P1	load 0x1444 ASID: 11	0x5444

TLB Miss Handling

- 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

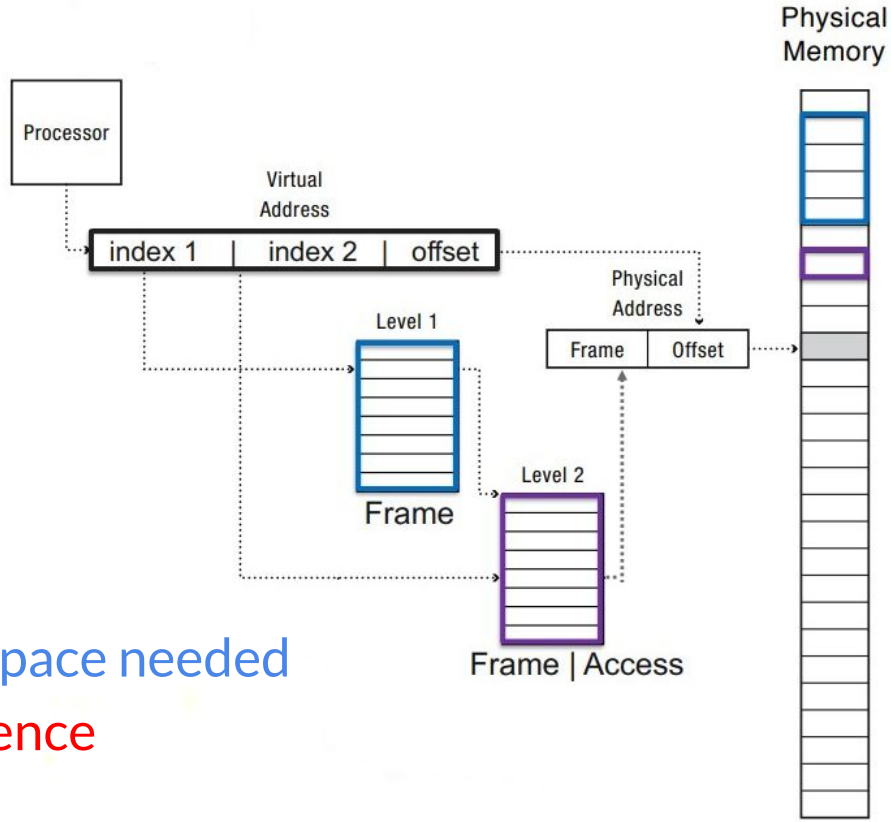
TLB Summary

- 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

Disadvantages of Paging

- **Inefficiency**: memory references to page table
  - Page table must be stored in memory
  - MMU stores only base address of page table
- **Storage overhead**:
  - Internal fragmentation
    - Page size may not match size needed by process
    - Make pages smaller? But then…
  - page tables may be large
    - Simple page table: requires PTE for all pages in address space
    - Entry needed even if page not allocated ?

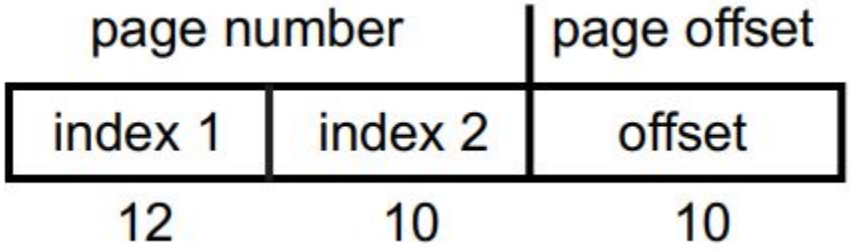
Multi-Level Page Tables



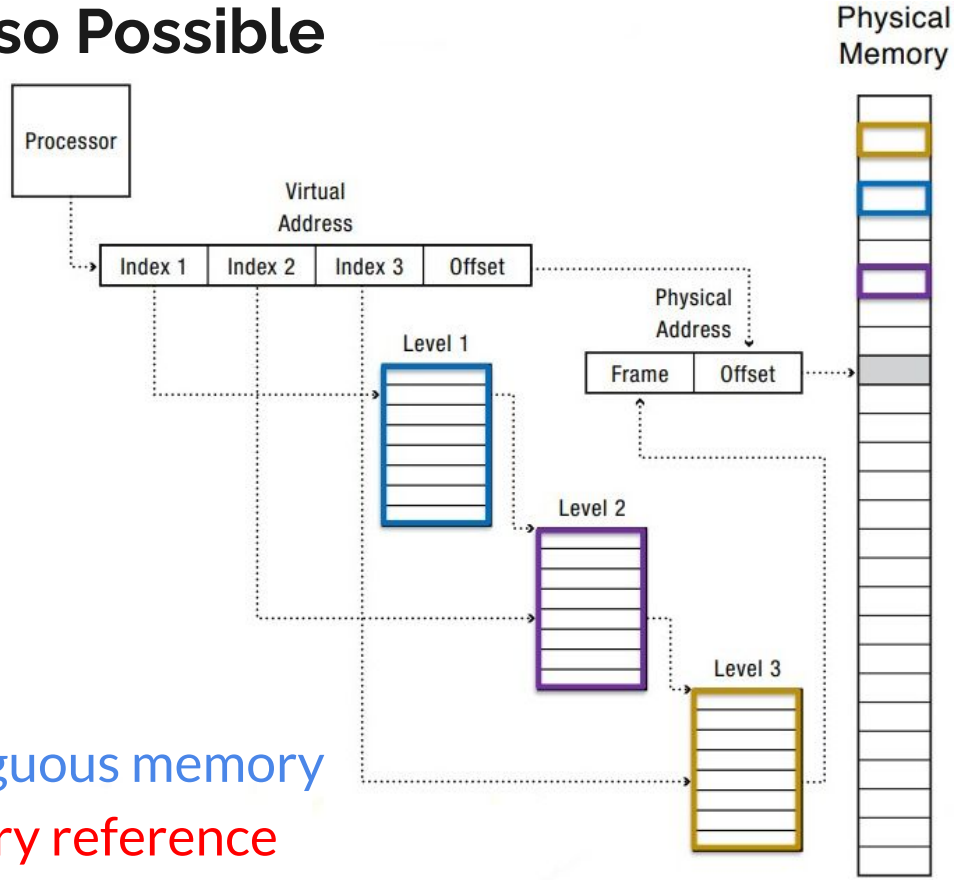
- + Allocate only PTEs in use
- + Simple memory allocation
  - no large continuous memory space needed
- − more lookups per memory reference

Two-Level Paging Example

- Assume 32-bit machine, 1KB page size
- Logical address is divided into:
  - a page offset of 10 bits (1024 = 2^10)
  - a page number of 22 bits (32-10)
- The page number is further divided into:
  - a 12-bit first index
  - a 10-bit second index



Three-Level Paging is also Possible



- + First Level requires less contiguous memory
- − even more lookups per memory reference

THANK YOU!