Demand paging

Mythili Vutukuru

CSE, IIT Bombay

Recap: Virtual addresses and paging

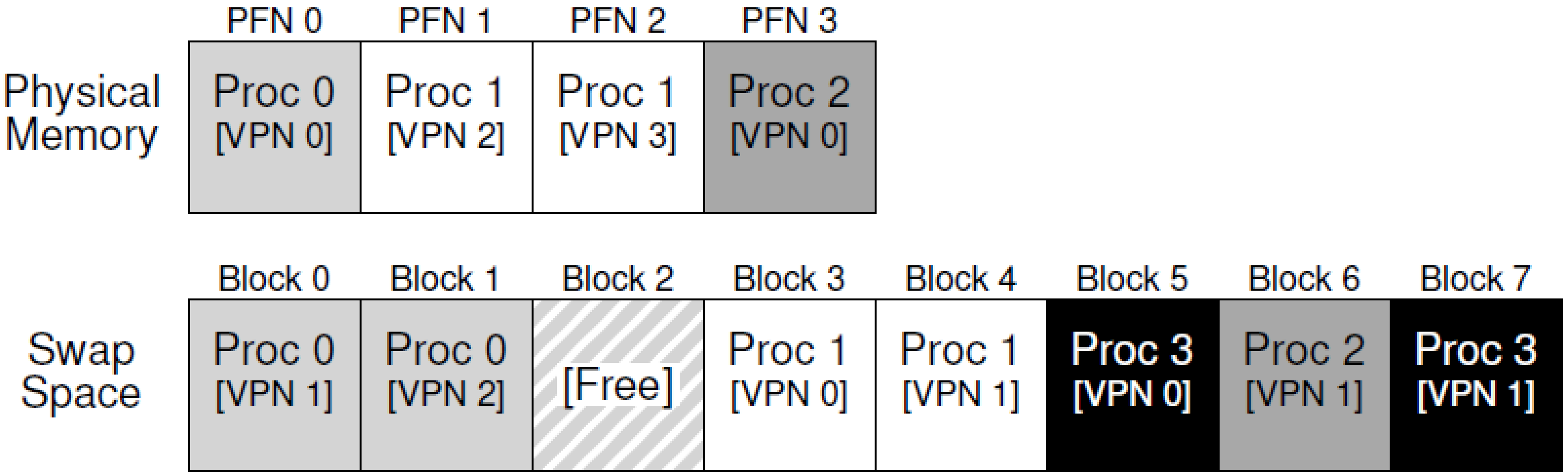
- Instructions and data of a process in memory assigned virtual addresses
  - Starting at 0 for user code (OS code also assigned high virtual addresses)
- Virtual address space of a process divided into fixed size logical pages, stored in a fixed size physical frames in memory
  - Prevents external fragmentation, cannot prevent internal fragmentation
- Page table maps logical page numbers to physical frame numbers
  - One per process, maintained by OS as part of PCB
  - Used by MMU to translate VA to PA when CPU accesses memory

Demand Paging

- Should all pages of all active processes always be in main memory?
  - Not necessary, as process will not use all of it at once
  - Not possible, with large address spaces
- Modern operating systems provide virtual memory
  - Not all logical pages are assigned frames, some memory is “virtual”
  - Why? Virtual memory of processes can be much more than physical memory in the system, OS overcommits memory
- Some pages in address space are not allocated at all (not valid)
- Some valid pages are allocated physical frames in memory, some are temporarily saved on disk, brought into DRAM when needed
- No demand paging in the simple xv6 OS

Swap space

- Special area on disk to hold pages that do not fit in DRAM
- Pages pushes to swap space when memory is full, brought into mmeory when accessed



	PFN 0	PFN 1	PFN 2	PFN 3
Physical Memory	Proc 0 [VPN 0]	Proc 1 [VPN 2]	Proc 1 [VPN 3]	Proc 2 [VPN 0]

	Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7
Swap Space	Proc 0 [VPN 1]	Proc 0 [VPN 2]	[Free]	Proc 1 [VPN 0]	Proc 1 [VPN 1]	Proc 3 [VPN 0]	Proc 2 [VPN 1]	Proc 3 [VPN 1]

Image credit: OSTEP

Tracking pages in swap space

- Invalid page addresses not in use by process, no need to store any frame number
- A valid page either has a physical frame number in memory, or a disk address in swap space, both tracked by page table

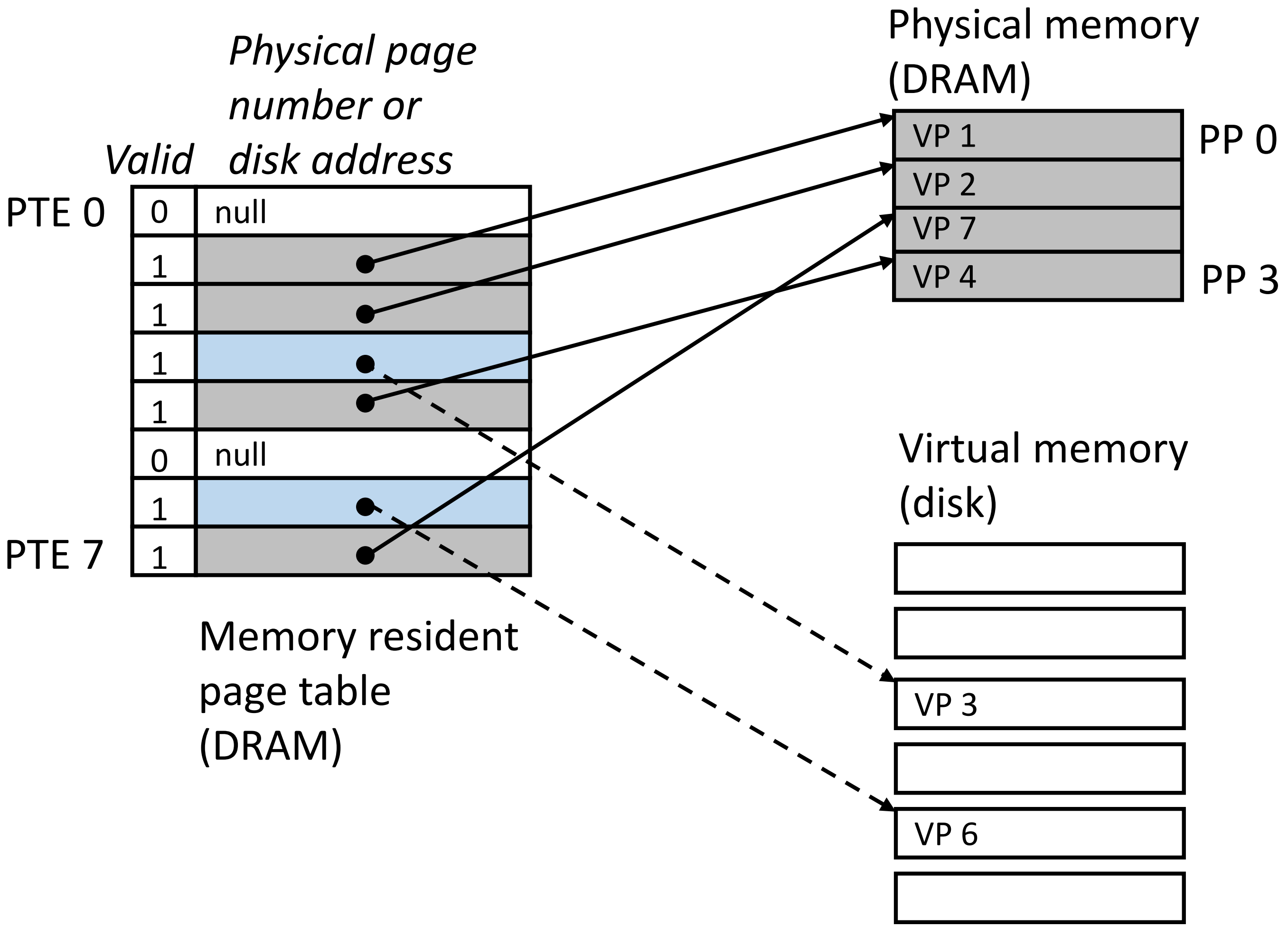


Image credit: CSAPP

Valid and present bits in page table entry

- Valid bit in PTE indicates if virtual page is in use by process
- Present bit indicates if page is allocated frame in main memory
- Valid bit and present bit both set → page in DRAM
- Valid set, present not set → page in swap space

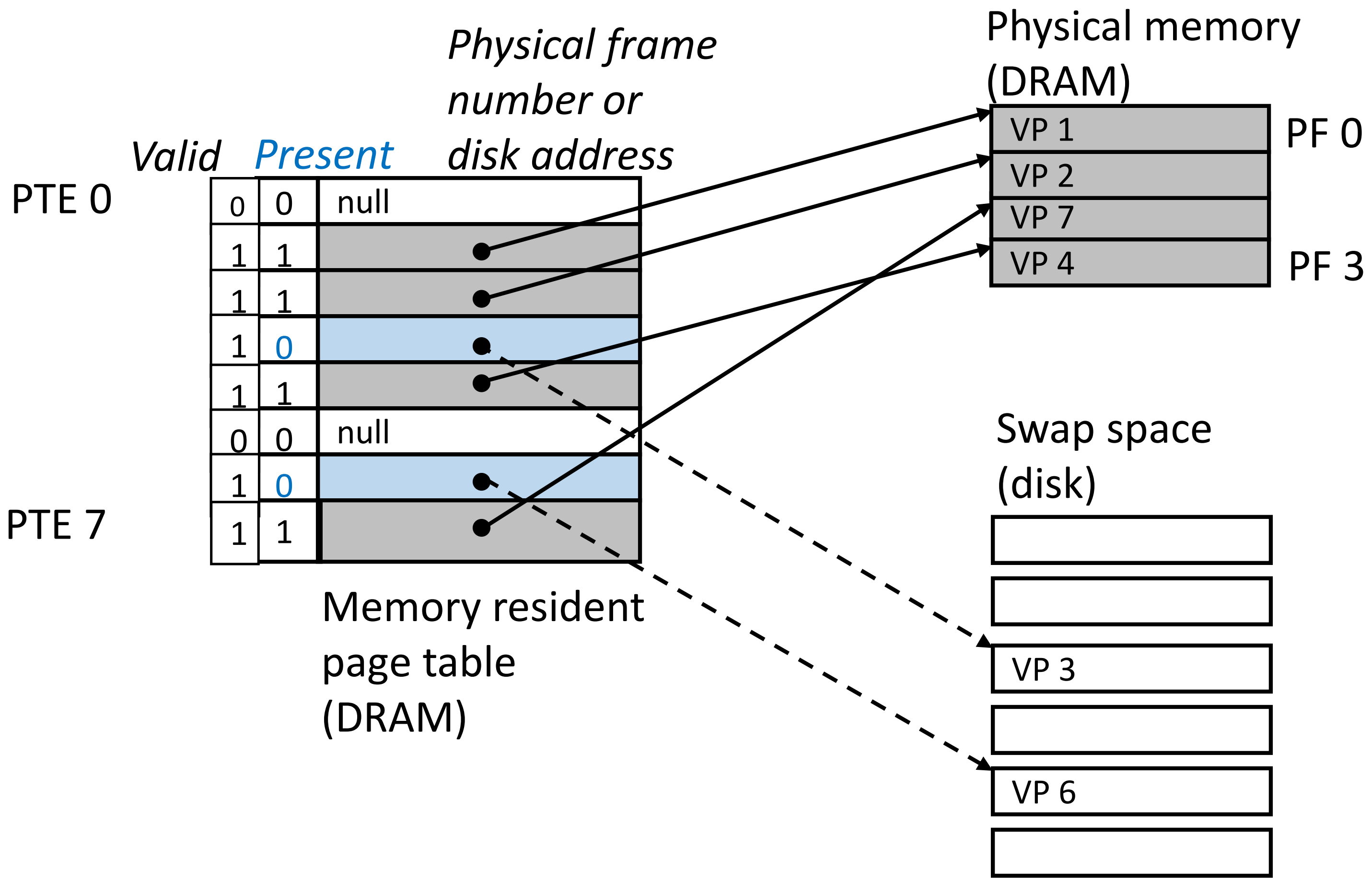


Image credit: CSAPP

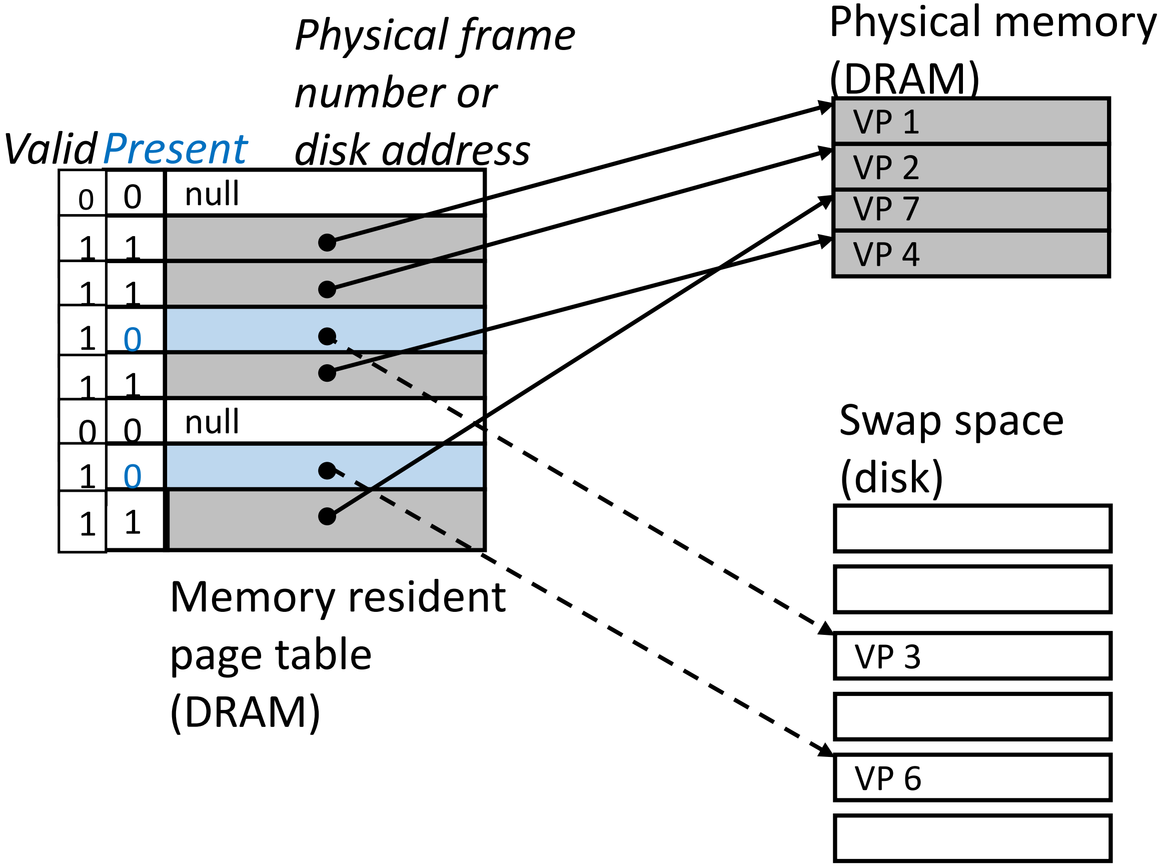
Page fault

- When MMU walks page table to translate a virtual address to physical address, the various bits in page table entry are also examined
- MMU traps to the OS (page fault) in case of any unexpected behavior
- CPU switches to kernel mode, runs page fault handler code
- How does OS handle page fault?
  - Valid bit not set → segmentation fault, terminate process
  - Any other illegal access (e.g., writing to read-only page) → terminate process
  - Valid bit set, present bit not set → OS allocates memory frame for page in DRAM, updates page table, restarts process (hopefully, all goes well now!)

Reclaiming memory

- OS usually keeps track of free pages in memory to use in page fault
- If no free frames in DRAM when servicing page fault, OS evicts a victim page from memory to swap space, allocates the freed up physical frame to faulting page
  - Copy contents of victim page from memory frame to swap space on disk
  - Copy contents of allocated page from disk into freed up memory frame
  - Update corresponding page table entries, restart process
- Page replacement policy of OS helps to identify suitable victim page
- Victim page can be from same process or from different process

CPU accesses page 3 (page fault)



Page 3 brought into memory, victim page 4 is moved to swap

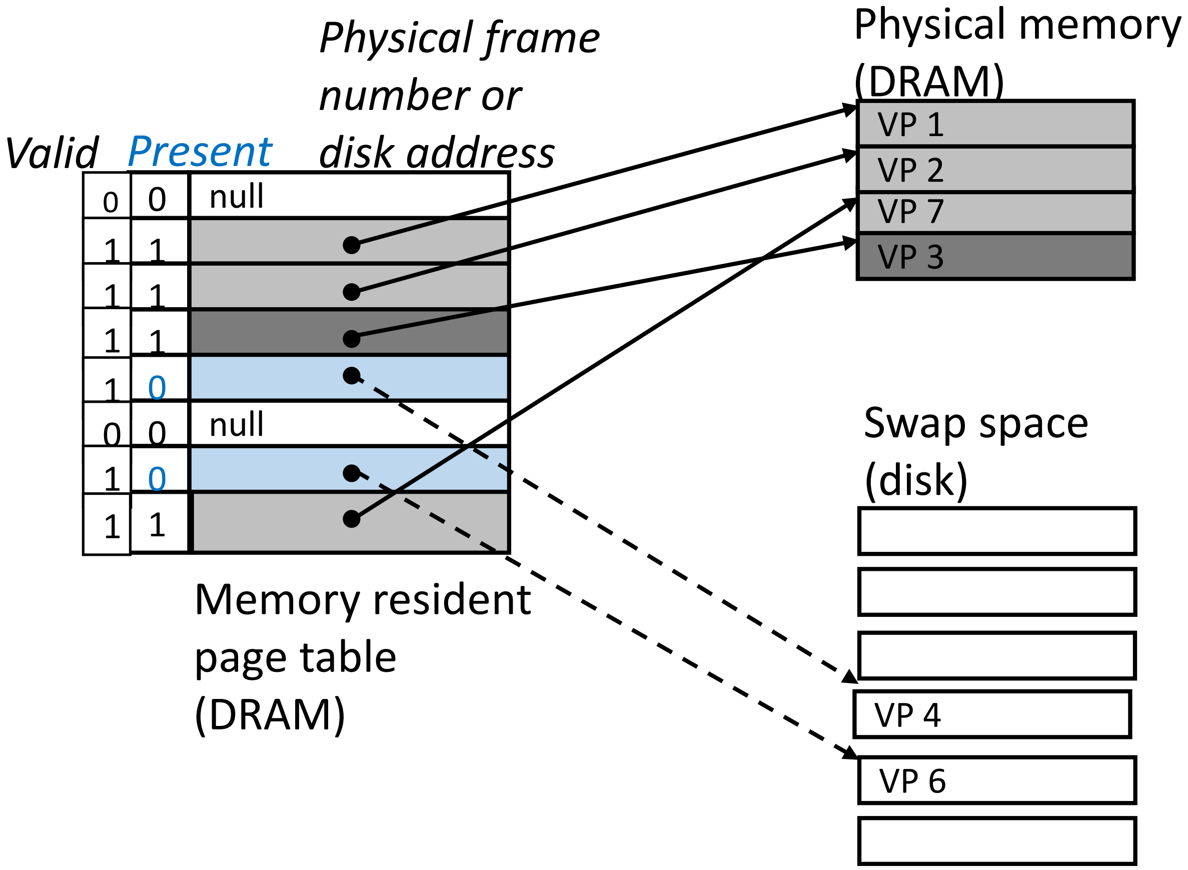


Image credit: CSAPP

File backed, anonymous, dirty pages

- Pages in the memory image of a process are of two types
  - File-backed pages contain data from files on disk (e.g., page with executable code)
  - Anonymous pages are not backed by files on disk (e.g., pages containing stack, heap)
- Further classification into dirty and non-dirty pages
- Dirty pages: pages whose content is different from their copy on disk
  - E.g., file-backed pages whose contents have changed from file
  - E.g., anonymous pages whose contents have changed since last read from swap
- PTE has information to track these types of pages, including dirty bit

Disk access during page fault

- Actions done by OS when servicing page fault depend on type of pages
- When reclaiming memory from victim page, need to copy content of victim page to swap space only if victim page is dirty
  - For other pages, can simply delete memory content, fetch from disk later
- When allocating memory frame during page fault, free memory frame must be initialized with content from disk for file-backed and non-empty anonymous pages
  - For empty anonymous page that has never been used, can just give empty frame
- Process may be blocked multiple times for disk I/O during page fault!
- Average memory access time (weighted avg across different scenarios during mem access) increases greatly if too many page faults

Summary: What happens on memory access

- CPU accesses code/data using VA
- MMU looks up TLB for VA

TLB hit, page in memory

  - If TLB hit, get PA, access memory only once to fetch code/data
  - If TLB miss, access memory to walk page table, get PTE

TLB miss, page in memory

    - If PTE is valid and present, compute PA, access memory once to fetch code/data
    - Else, MMU traps to OS

TLB miss, page not in memory

      - If invalid or illegal access, terminate process
      - If valid but page not present, allocate free memory frame (maybe by swapping out victim page), swap in contents of page (if needed), update page table, restart process
- Where do CPU caches fit in in this story?
  - Caches can be checked before address translation (virtually addressed caches) or after address translation (physically addressed caches)
  - DRAM accessed only on cache miss

Summary: TLB hit, page in memory

1. CPU accesses virtual address
2. MMU looks up page number in TLB
3. If TLB hit, page table entry is available, physical address computed
4. CPU directly accesses required code/data using physical address

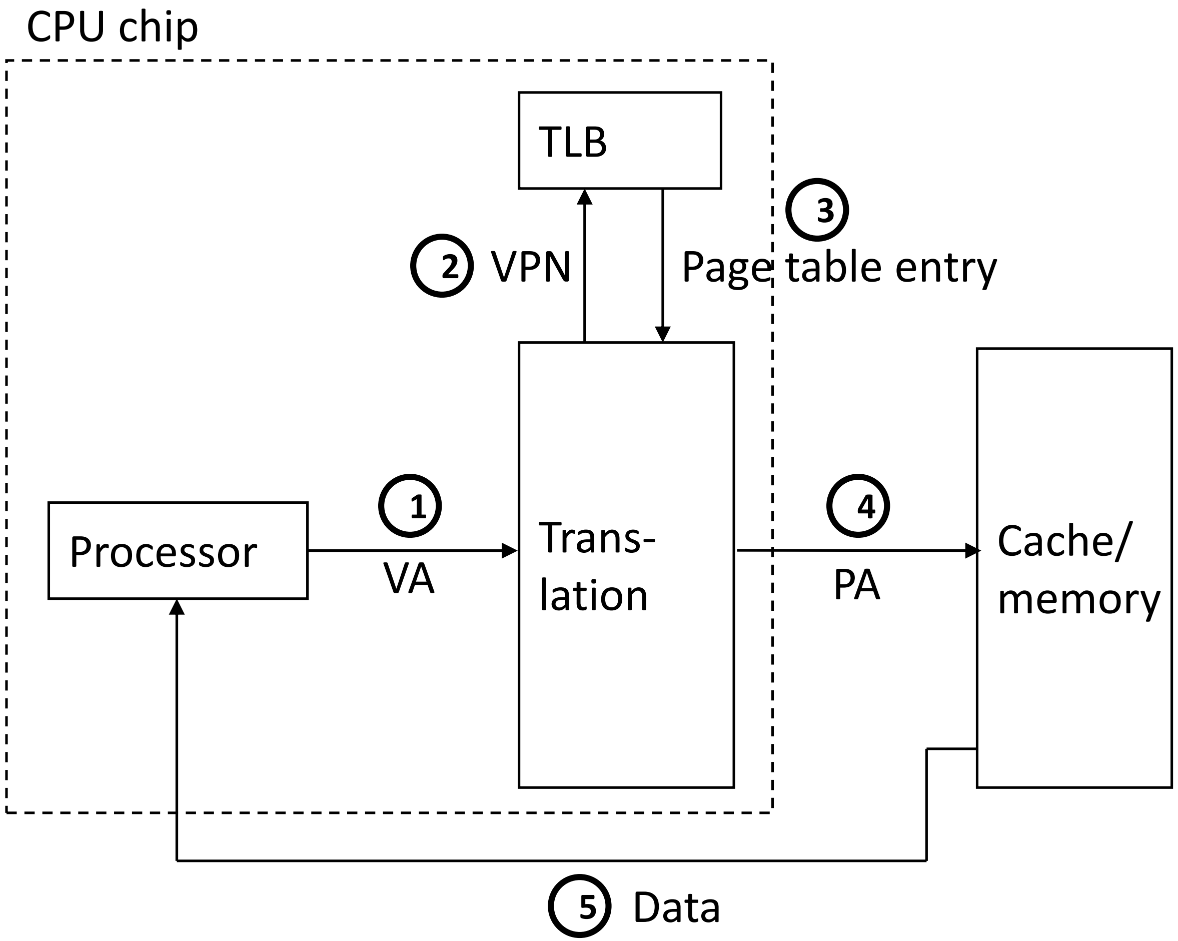


Image credit: CSAPP

Summary: TLB miss, page in memory

1. CPU accesses virtual address
2. MMU looks up page number in TLB, cannot find entry
3. MMU looks up page table in memory to find page table entry
4. Page table entry populated in TLB for future use
5. MMU computes physical address using which CPU accesses main memory

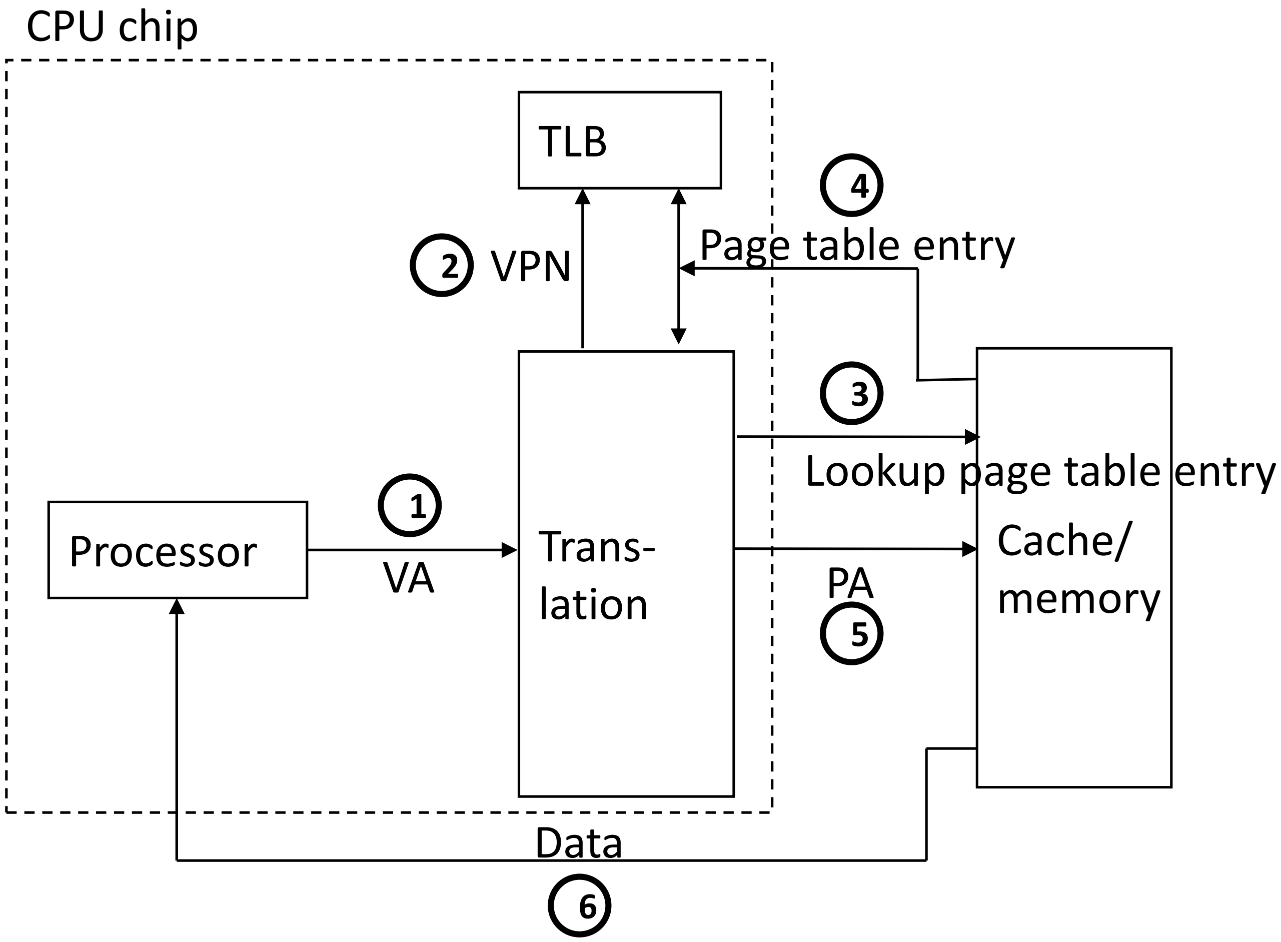
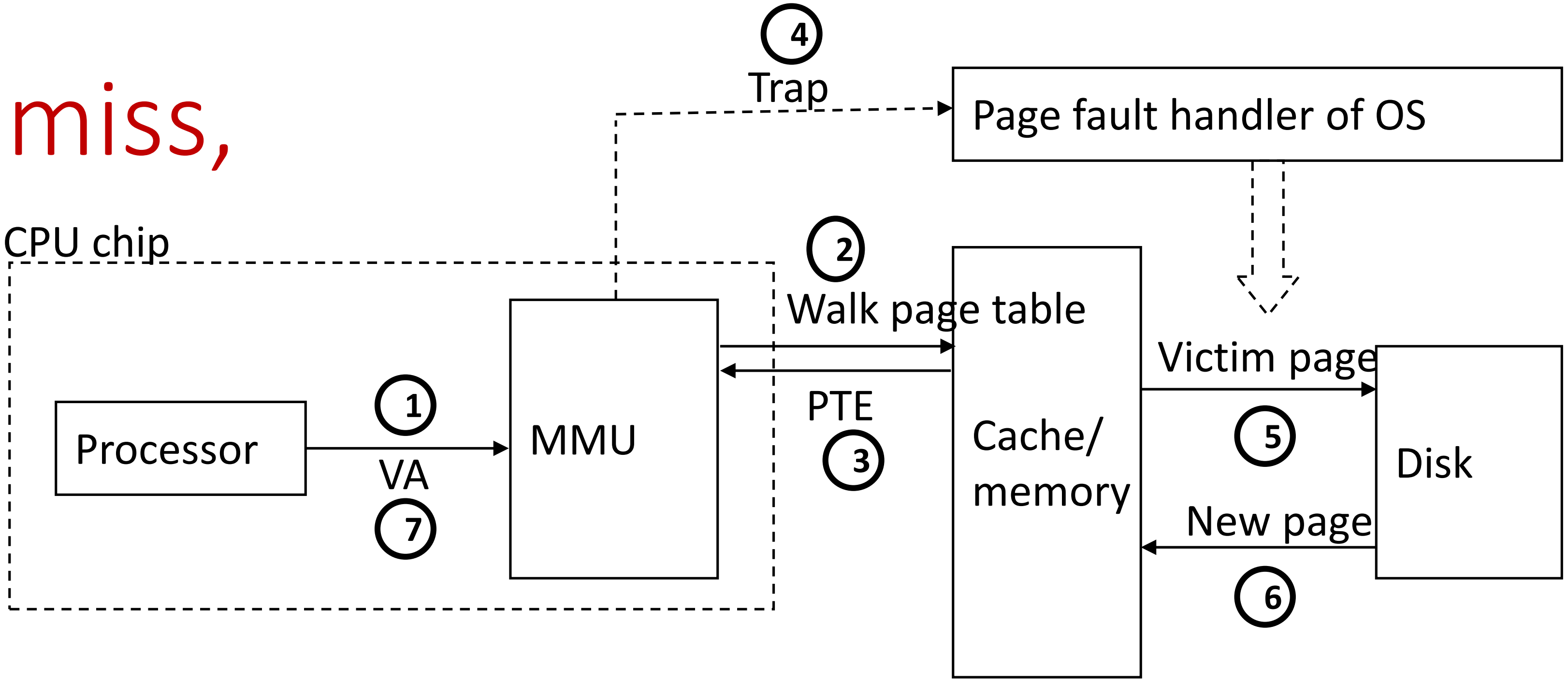


Image credit: CSAPP

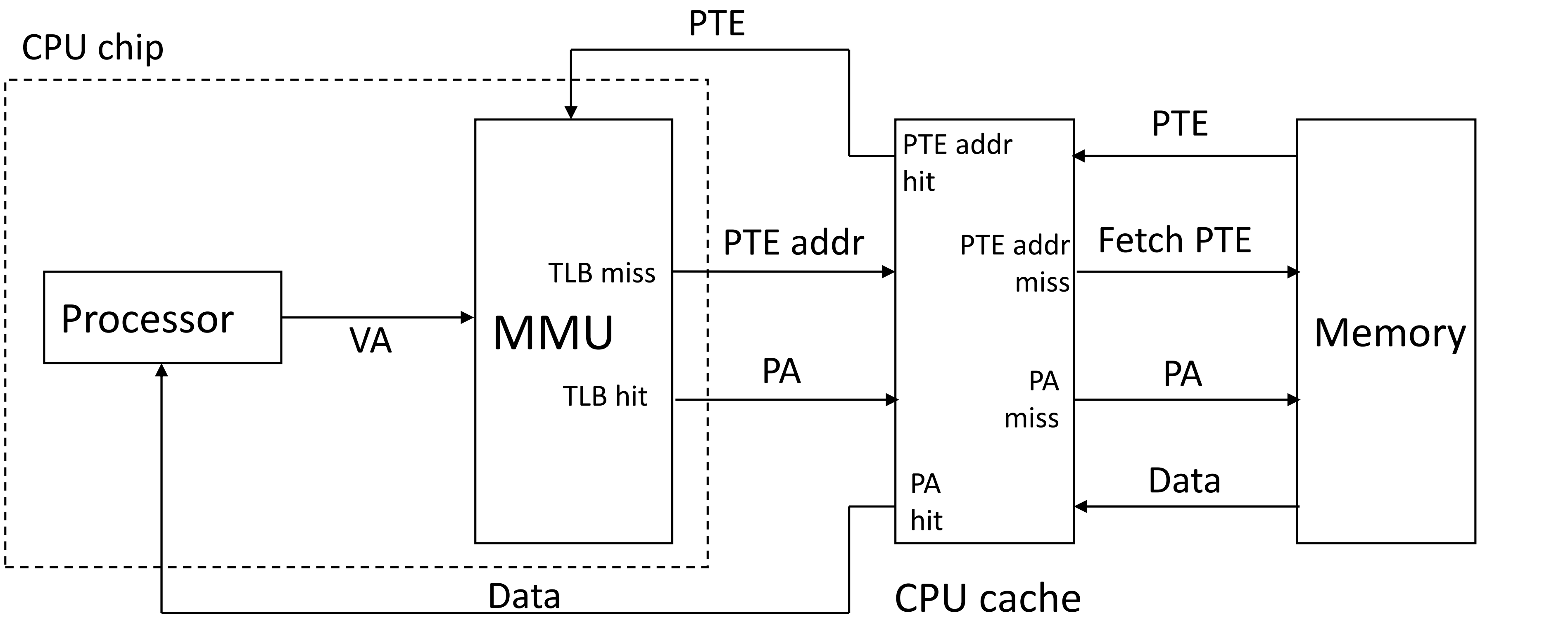
Summary: TLB miss, page fault



1. CPU accesses VA
2. TLB miss, walk page table
3. Get PTE, cannot compute PA
4. MMU traps to OS
5. OS swaps out victim page (if needed)
6. OS swaps in new page (if needed), updates PTE to reflect new mapping
7. Restart original process, memory access succeeds (hopefully!)

Image credit: CSAPP

Summary: Virtual memory and caches



CPU cache
(physically addressed,
so accessed after address translation)

Image credit: CSAPP

Thrashing

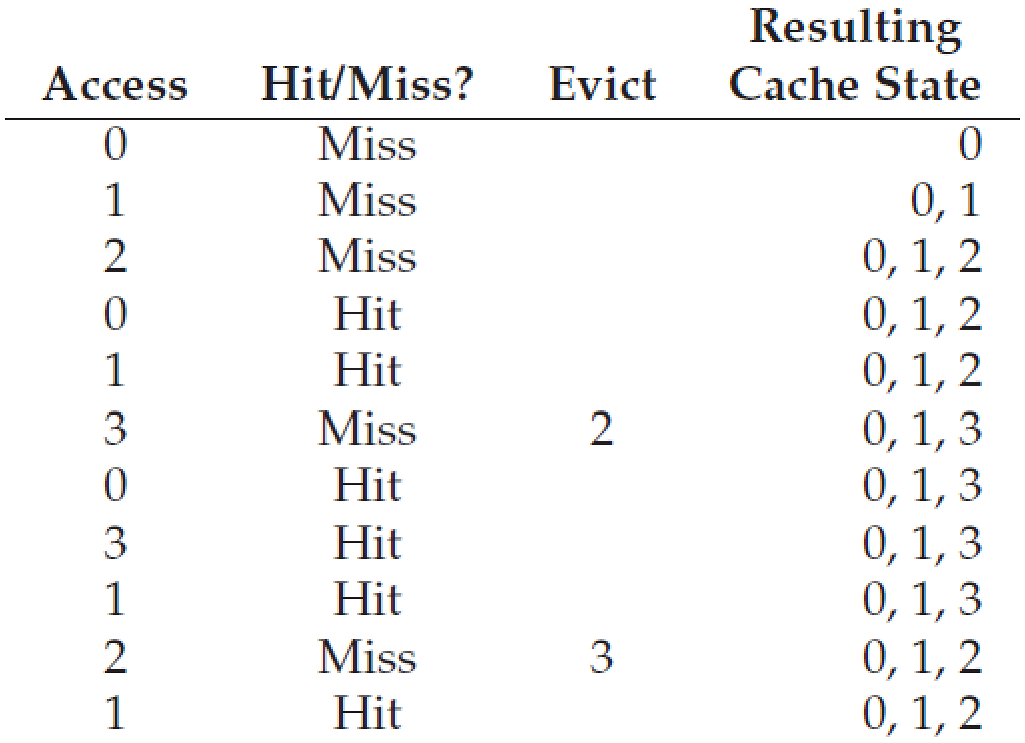
- Causes for application slowdown: CPU cache miss, TLB miss, page fault, ..
- Page fault particularly dangerous, may involve multiple disk access
- OS should allocate enough physical memory to avoid page faults, but how much?
- Every process has a working set: frequently used pages in memory image
  - Can change from time to time, based on code being executed
  - Usually smaller than total virtual memory of process
  - If memory assigned to is less than working set, frequent page faults
- Thrashing = system spends too much time servicing page faults and swapping back and forth from disk, and too little time doing useful application work, significant slowdown noticeable by users
- Solution: users can reduce working set of processes, OS can terminate some processes or clean up unnecessary memory, …

Page replacement policies

- Page replacement policy: which victim page should OS pick to evict?
- Goal: Minimize page faults, evict pages not likely to be used immediately
- Simple policy: First In First Out (FIFO) evicts pages in the order in which they have been brought into memory
    - May be suboptimal, e.g., the first assigned pages may be important pages that are in use very often, leading to another page fault in near future
- Most commonly used policy: evict the Least Recently Used (LRU) page
    - Page has not been used for sometime, so less likelihood that it will be used in future
- Optimal policy: evict page not needed for longest time in future (impractical!)

Example: Optimal policy

- Example: Process accessed 4 pages (0,1,2,3), only 3 physical frames in memory
- First few accesses are cold (compulsory) misses (if OS doesn’t assign any memory to process at start)
- Hit rate = 6/(6+5) = 54.5%
- Hit rate modulo compulsory misses 85%



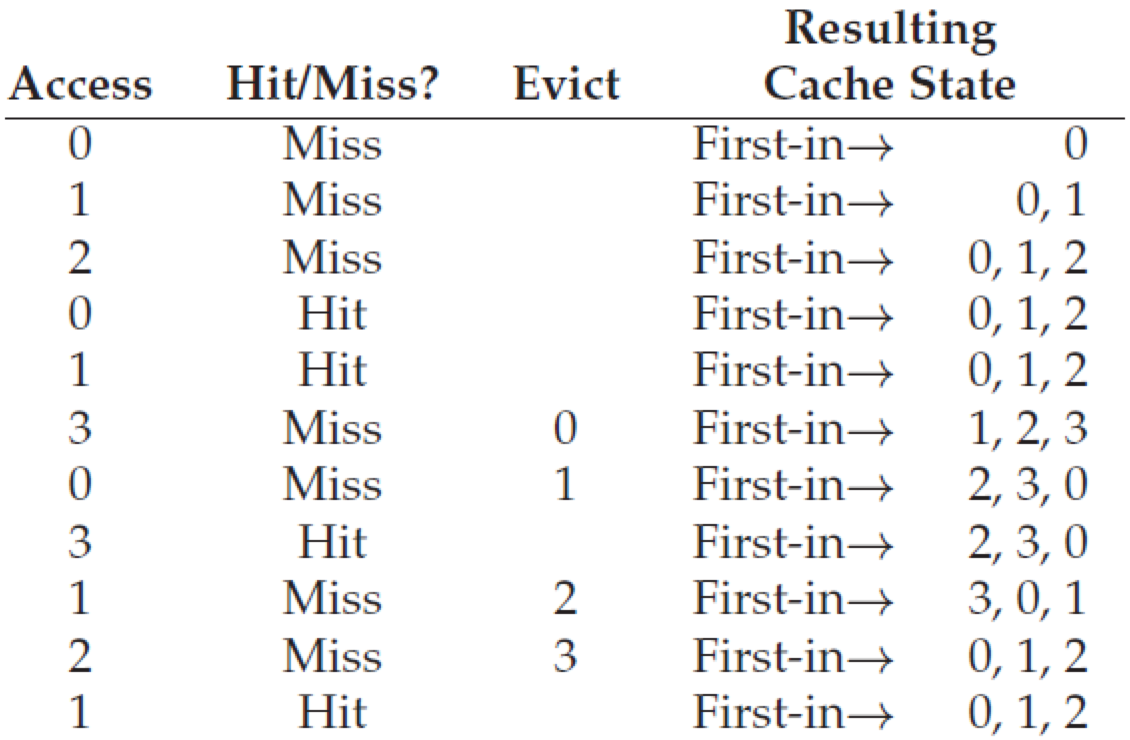
Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Figure 22.1: **Tracing The Optimal Policy**

Image credit: OSTEP

Example: FIFO

- Usually worse than optimal
- Belady’s anomaly: performance may get worse when memory size increases!

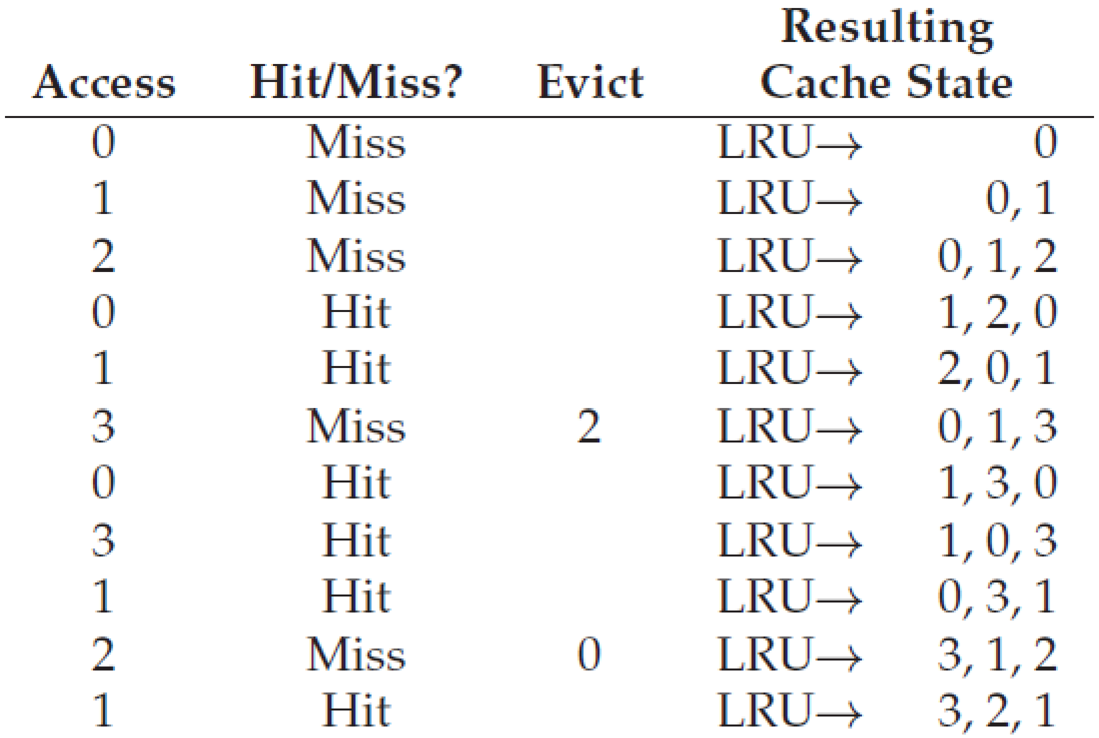


Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		First-in→ 0
1	Miss		First-in→ 0, 1
2	Miss		First-in→ 0, 1, 2
0	Hit		First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2
3	Miss	0	First-in→ 1, 2, 3
0	Miss	1	First-in→ 2, 3, 0
3	Hit		First-in→ 2, 3, 0
1	Miss	2	First-in→ 3, 0, 1
2	Miss	3	First-in→ 0, 1, 2
1	Hit		First-in→ 0, 1, 2

Image credit: OSTEP

Example: LRU

- Equivalent to optimal in this simple example
- Works well due to locality of references (recently used pages accessed again with high probability)



			Resulting
Access	Hit/Miss?	Evict	Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Figure 22.5: **Tracing The LRU Policy**

Image credit: OSTEP

LRU implementation

- How does OS know which page is LRU?
  - OS is not involved in every memory access, so doesn’t know which pages have been recently used
- Solution: MMU sets the accessed bit for every page table entry it accesses
  - Accessed bit is set implies page has been recently used
- Modern operating systems implement approximate LRU
  - Periodically, look at accessed bit of pages to classify pages into active and inactive pages
  - Pick pages that have been inactive for eviction
  - May also avoid dirty pages for eviction, since it requires extra disk write