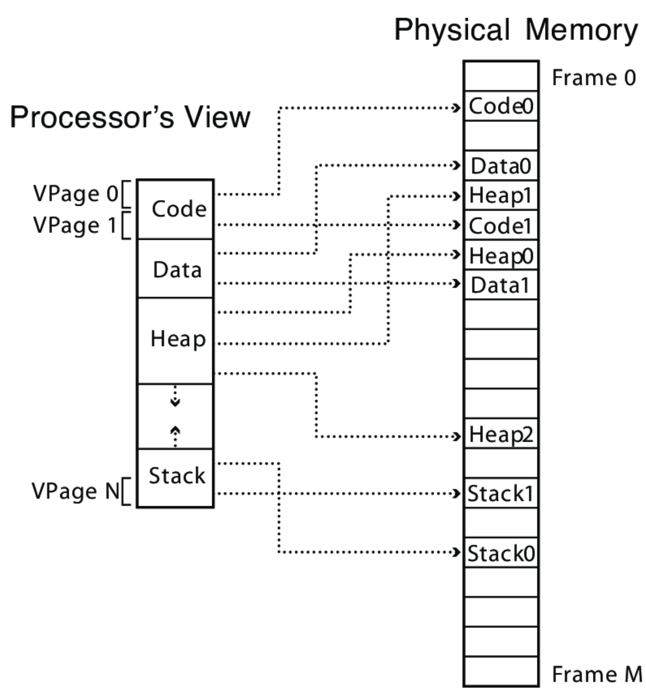
Virtual Memory: Demand Paging

Introduction

Address translation

- Distinction between virtual address space and physical memory
  - Mapping at page level providing relocation and protection
- Virtual address space convenient for programs and processes
  - Segments contiguously allocated
  - At addresses allocated by compiler
- Physical memory easy to allocate
  - Any available page frame can be mapped to any process virtual page



Introduction

Issue

- Assumption that a running process has its entire address space loaded
    - Code, data, heap, and stack segments
- Slow, especially for big processes
    - Code and data need to be loaded from disk
- Wasteful
    - Processes don't use all of their memory all the time
    - *Working set* is usually small and evolves slowly

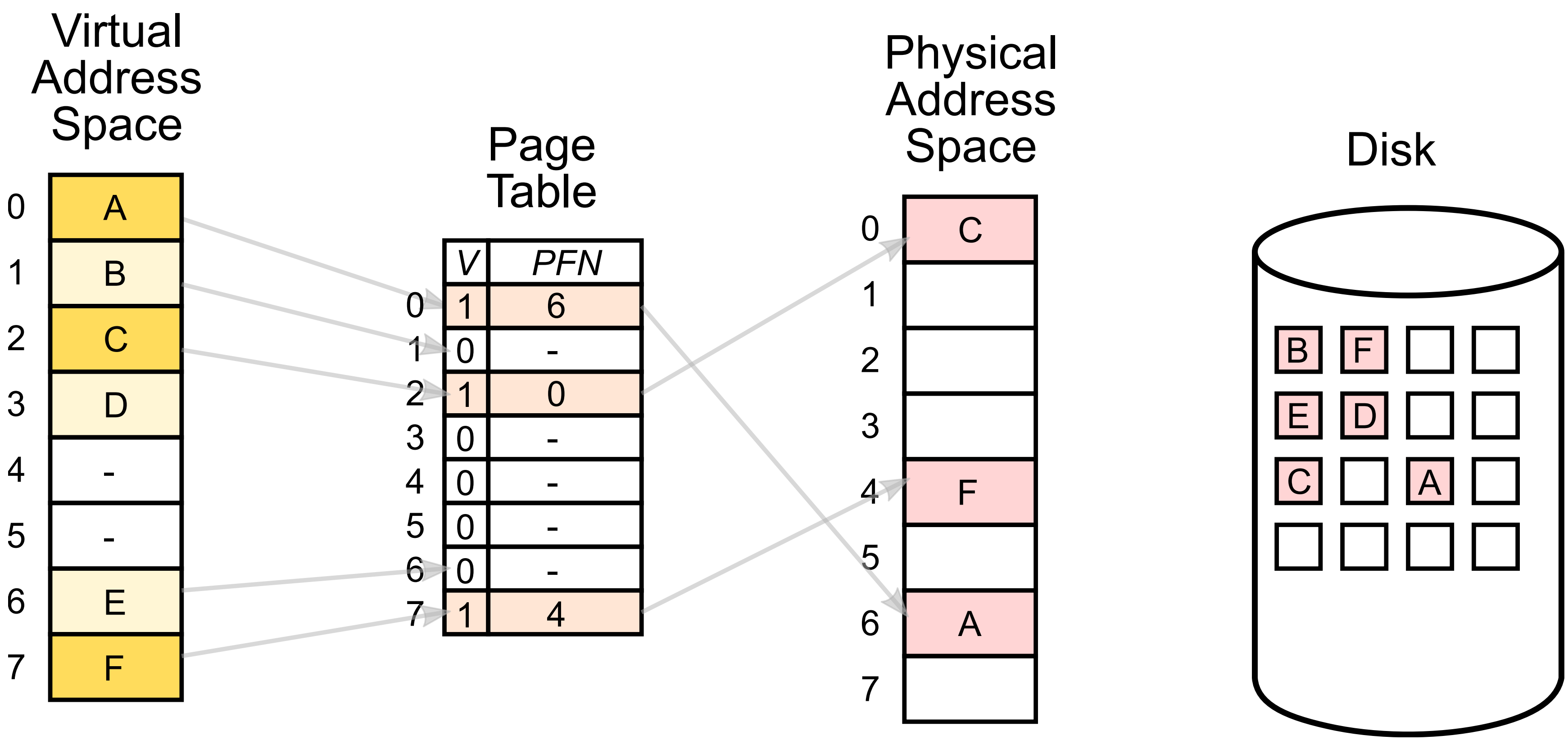
Idea

- Load pages as needed during process execution
    - Only bring in pages actually used
    - Only keep frequently-used pages in memory
- Illusion of (nearly) infinite memory, available to every process
- Multiplex virtual pages onto a limited amount of physical page frames

Demand paging

Page mapping

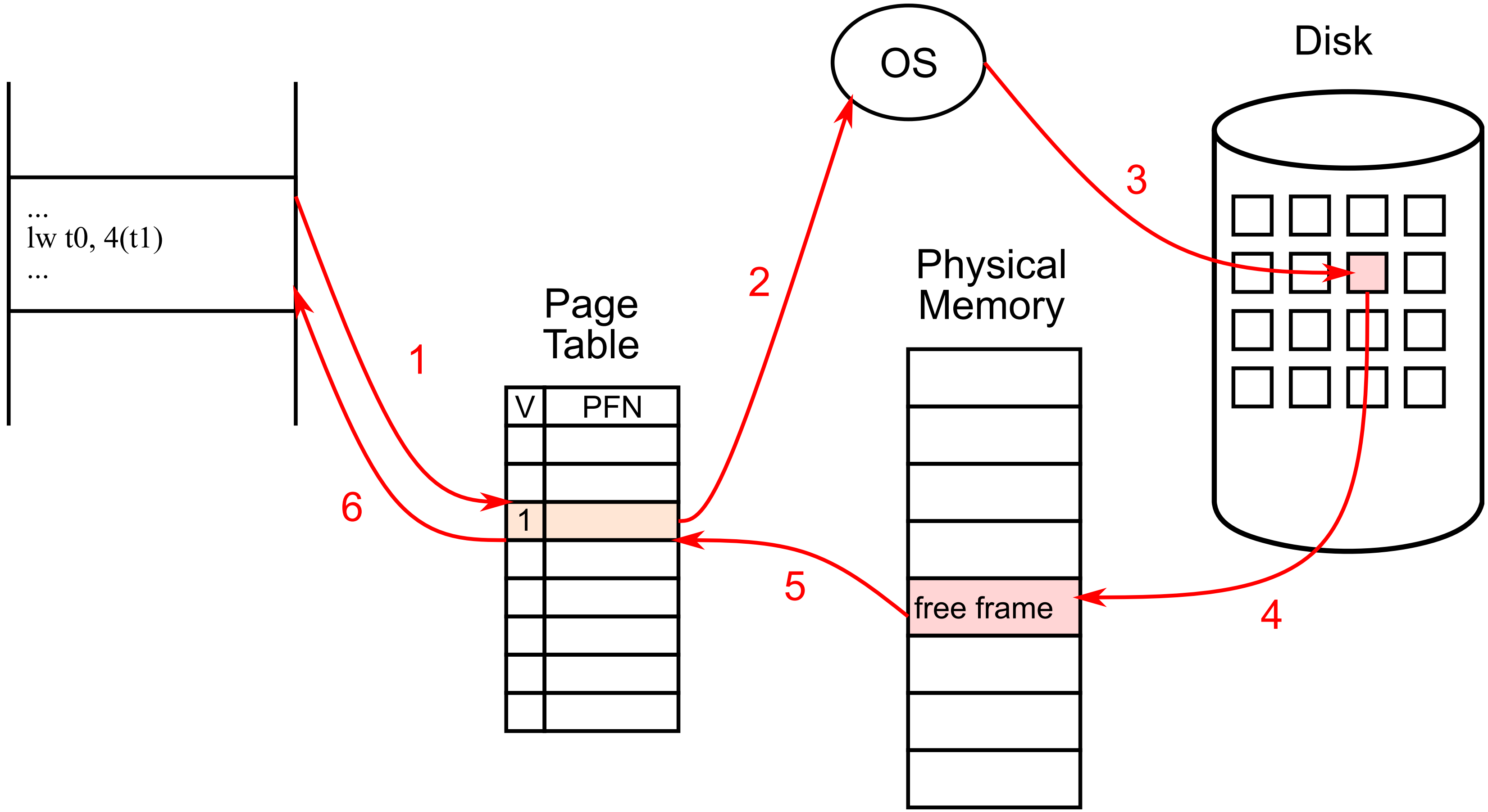
- *Resident* page: mapped in physical memory, valid PTE
- *Non-resident* page: located on disk, invalid PTE
  - Page-fault triggered if accessed
- *Unused* page



Demand paging

Page fault (overview)

- (1) Memory access, TLB access
- (2) Page invalid, trap to kernel
- (3) Locate page on disk
- (4) Swap in page in free frame
- (5) Mark page as valid
- (6) Resume process



Demand paging

Page fault (details)

- (1) Memory access from running process
  - Lookup virtual memory address in TLB
    - If TLB miss, page table walk
- (2) Page is invalid
  - Page fault, trap to kernel
- (3) Locate page on disk
  - Directly from executable if page of code
  - Reuse PTE to store the block number in swap space
- (4) Swap in page in free frame
  - Allocate page frame
    - Evict page if needed
- (4) Cont.
  - Initiate disk block read into page frame
  - Elect another process to run...
  - Disk interrupt when DMA complete
- (5) Mark page as valid
  - Purge TLB for this page
- (6) Resume process
  - At faulting instruction
  - TLB miss
  - Page table walk to fetch latest translation
  - Execute instruction and access page

Demand paging

Page frame allocation

- If memory isn't full, allocating free page frame is straight-forward
- Otherwise, need a specific strategy
  - Select old page to evict
    - *Page replacement* algorithm (e.g., FIFO, LRU, etc.)
  - Unmap old page from processes
    - Page frame can be shared among several processes
    - Find all page table entries that refer to old page
    - Set page table entries to invalid
    - Purge corresponding TLB entries
  - Write changes on page back to disk, if necessary
    - Need to detect page modifications

```c
fd = open("/path/to/file");
char *address = mmap(0, len, PROT_WRITE,
                     MAP_SHARED, fd, 0)
// Write
address[some_offset] = 'a';
...
```

Will need to write back page to disk

```c
fd = open("/path/to/file");
char *address = mmap(0, len, PROT_READ,
                     MAP_SHARED, fd, 0)
// Only read
printf("%x\n", address[some_offset]);
...
```

Can simply discard page when done

Demand paging

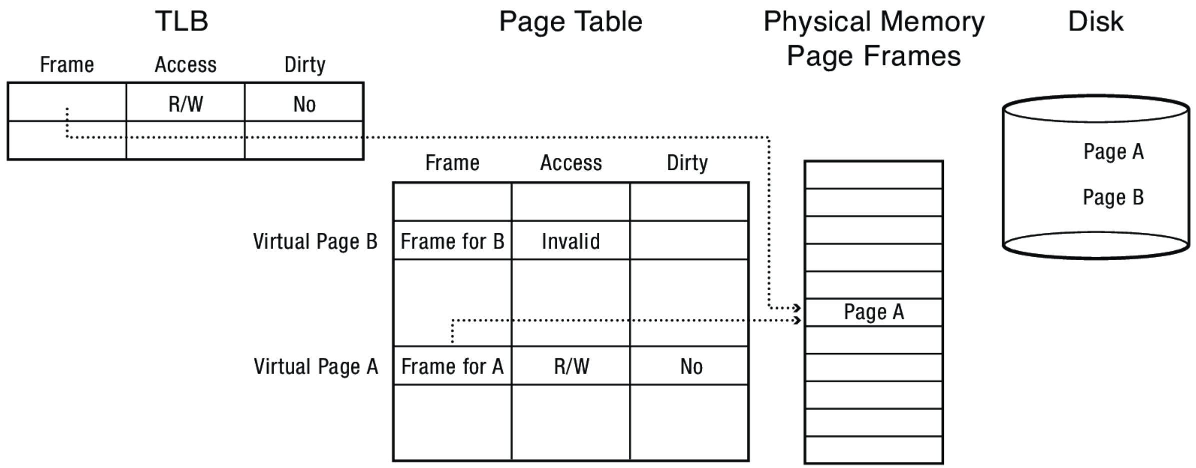
PTE bookkeeping

- On most processor architectures, every PTE has some bookkeeping support
  - **Modified bit**
    - Set by hardware in TLB entry on store instruction
    - Usually updated to PTE upon TLB eviction
  - **Use bit** (also know as *Reference bit*)
    - Set by hardware in PTE upon TLB miss
- Bookkeeping bits can be reset by OS
  - When changes to page are purged to disk
  - To track whether page has recently been used

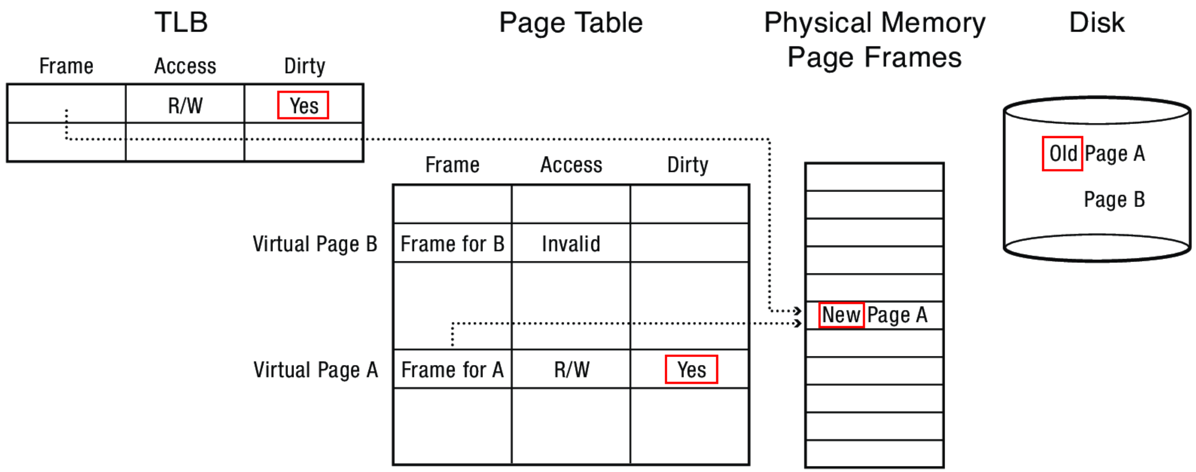
Demand paging

Tracking page modifications

Before to clean page:



After writing to page for the first time:



Page replacement

Strategies

- When memory is full, need tot select a *victim frame* to evict
- Algorithms
    - Random
        - Zero cost for bookkeeping
        - Not best, not worst, just unpredictable!
    - FIFO
    - MIN
    - LRU

Page replacement

FIFO

- Replace the page that has been loaded the longest time

Example

- 4 page frames available
- Access sequence: `A B C D A B E A B C D E`

Reference	A	B	C	D	A	B	E	A	B	C	D	E
Frame #1	A				+		E				D	
Frame #2		B				+		A				E
Frame #3			C						B			
Frame #4				D						C		

- Result: **10 page faults**

Pros

- Simple implementation

Cons

- May replace the heavily-used pages
- Suffers from *Belady's anomaly*

Page replacement

Belady's anomaly

- FIFO algorithm with 4 page frames causes 10 page faults

Reference	A	B	C	D	A	B	E	A	B	C	D	E
Frame #1	A				+		E				D	
Frame #2		B				+		A				E
Frame #3			C						B			
Frame #4				D						C		

- Now, same sequence but with only **3** page frames available...

Reference	A	B	C	D	A	B	E	A	B	C	D	E
Frame #1	A			D			E					+
Frame #2		B			A			+		C		
Frame #3			C			B			+		D	

- **9 page faults**!
- More page faults although more page frames are available...

Page replacement

MIN (aka optimal)

- Replace the page that will not be used for the longest time in the future

Example

Reference	A	B	C	D	A	B	E	A	B	C	D	E
Frame #1	A				+			+			D	
Frame #2		B				+			+			
Frame #3			C							+		
Frame #4				D			E					+

Pros

- Only 6 faults, optimal!

Cons

- Impossible to implement, only gives the ideal lower bound

Page replacement

LRU

- Replace the page that has not been used for the longest time in the past

Example

Reference	A	B	C	D	A	B	E	A	B	C	D	E
Frame #1	A				+			+				E
Frame #2		B				+			+			
Frame #3			C				E				D	
Frame #4				D						C		

Pros

- Good approximation of MIN

Cons

- Difficult to accurately implement

Page replacement

Implementing LRU

Issue

- In software, use a linked list
  - Every hit moves the page to the front of the list
  - Evict from the back of the list
  - But you can't trap back to kernel for each page access...
- In hardware
  - Impossible to manage a variable-size list...

Solution

- Approximate LRU
- Take advantage of the *use bit*!

Page replacement

Clock

- Periodically, sweep through all the pages
- If page is unused, reclaim
- If page is used, mark as unused

Second-chance

- Significant cost to reclaim *dirty* pages
- Modify clock algorithm to allow dirty pages to survive the first sweep of the clock hand