# EC 440 – Introduction to Operating Systems

**Orran Krieger (BU)**
**Larry Woodman (Red Hat)**

# Last Lecture review: page replacement

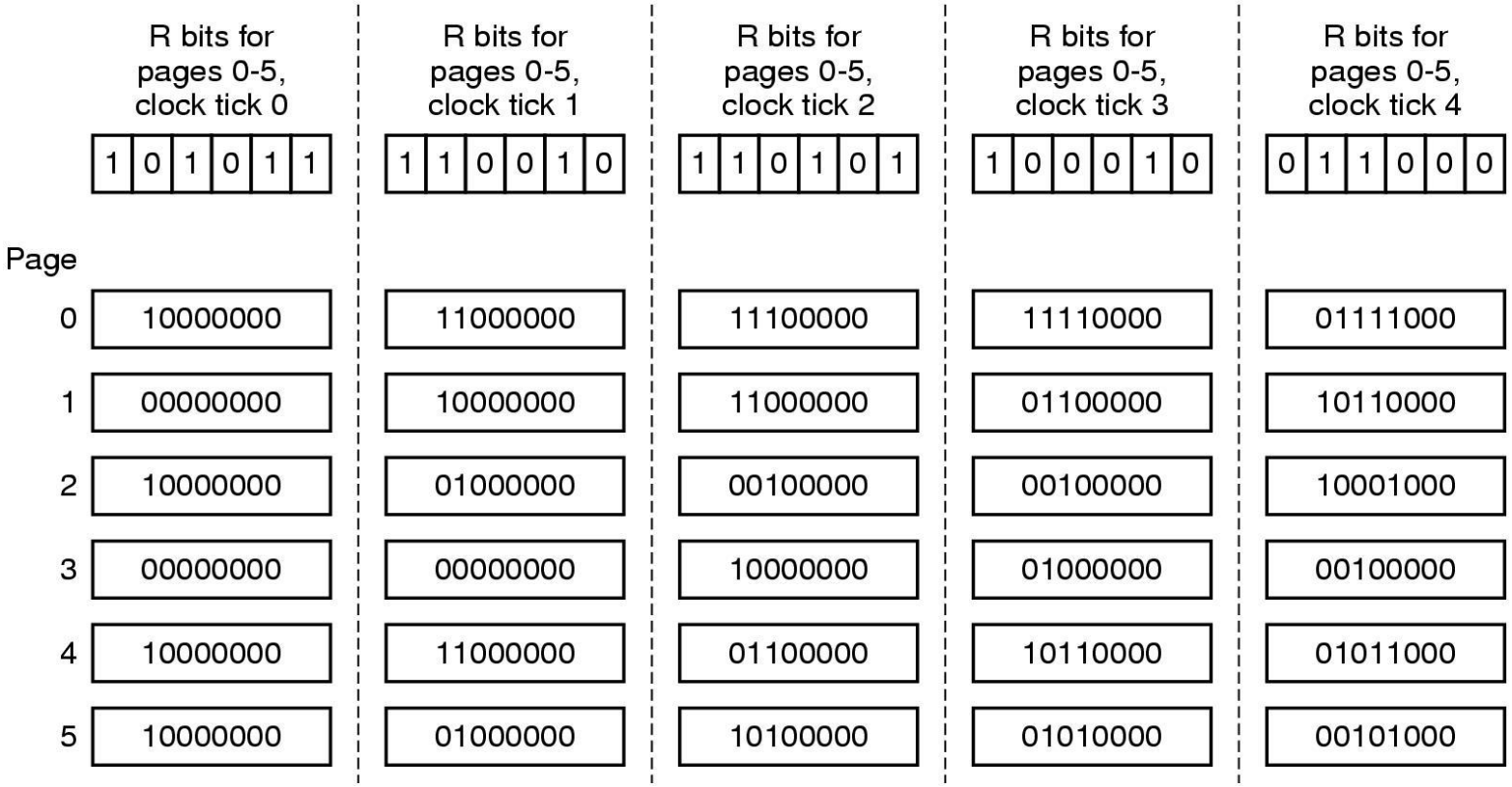
- NRU
- FIFO
- Second chance
- Clock
- LRU
- NFU - didn't talk about this yet, will do it now

# Not Frequently Used (NFU) Page Replacement Algorithm

- A counter is associated with each page
- At each clock interval, the counter is incremented if the page has been referenced (R=1)
- The page with the lowest counter is removed
- Problem:

  pages that have been heavily used in the past will always maintain high counter values
- Need for an aging mechanism
  - First shift the counter
  - Then set bit in most significant position if referenced

# Not Frequently Used (NFU) Page Replacement Algorithm

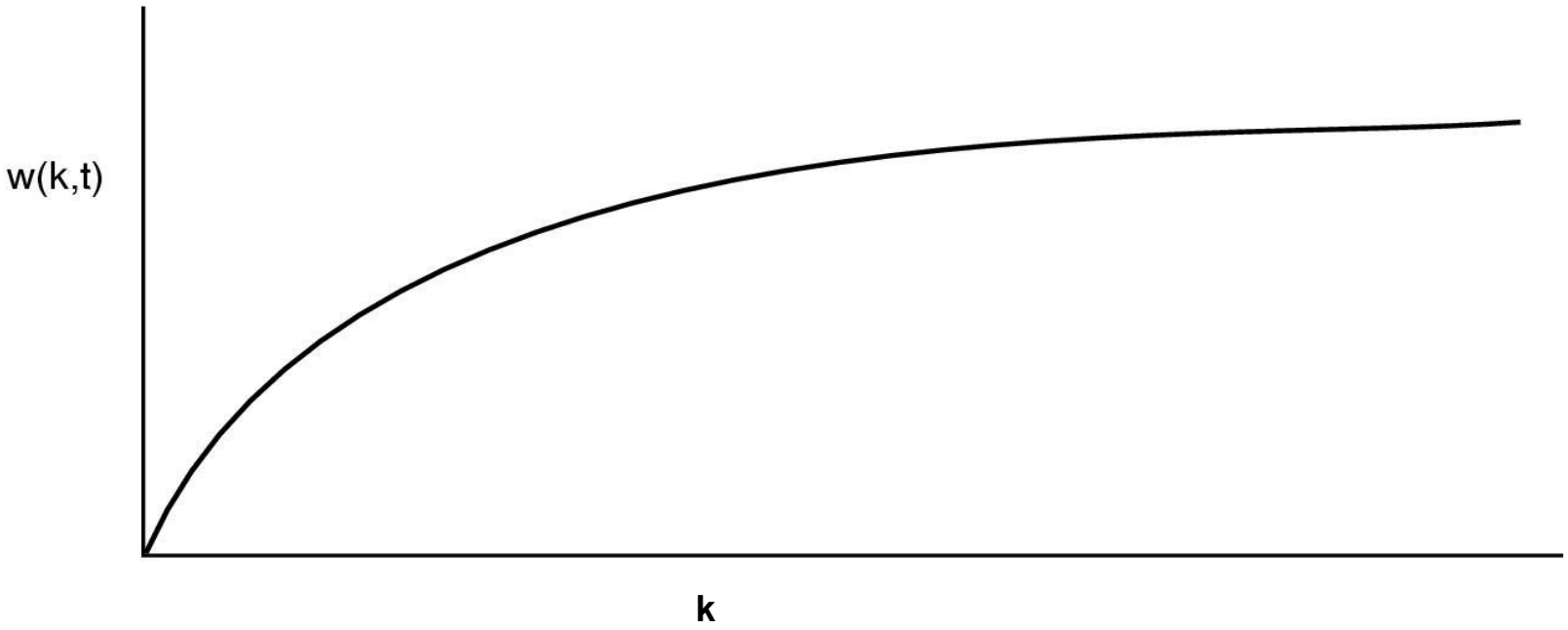


# Working Set Page Replacement Algorithm

- Locality of reference: Most programs use a subset of their memory pages (loops, sequential executions, updates to the same data structures, etc.) and the set changes slowly
- The working set is the set of pages used by the k most recent memory references
- For a reasonable value of k, the number of page faults is reduced and the process does not *thrash*
- If the working set can be determined it can be preloaded at context switch to minimize the initial demand of pages

# Working Set
# Page Replacement Algorithm

**w(k,t): the size of the working set at time, t**



# Working Set Page Replacement Algorithm

- Algorithm:

  when a page has to be evicted, find one that is not in the working set

- Use a shift register of size k
- At every reference
  - Right-shift the register
  - Insert page in left most position
- At replacement time
  - Remove duplicates and obtain working set
  - Remove page not in working set

# Working Set Page Replacement Algorithm

- Use execution time instead of references
- Working set composed of pages referenced in the last t msec of execution
- Each entry contains
  - The time the page was last used
  - The reference bit, R

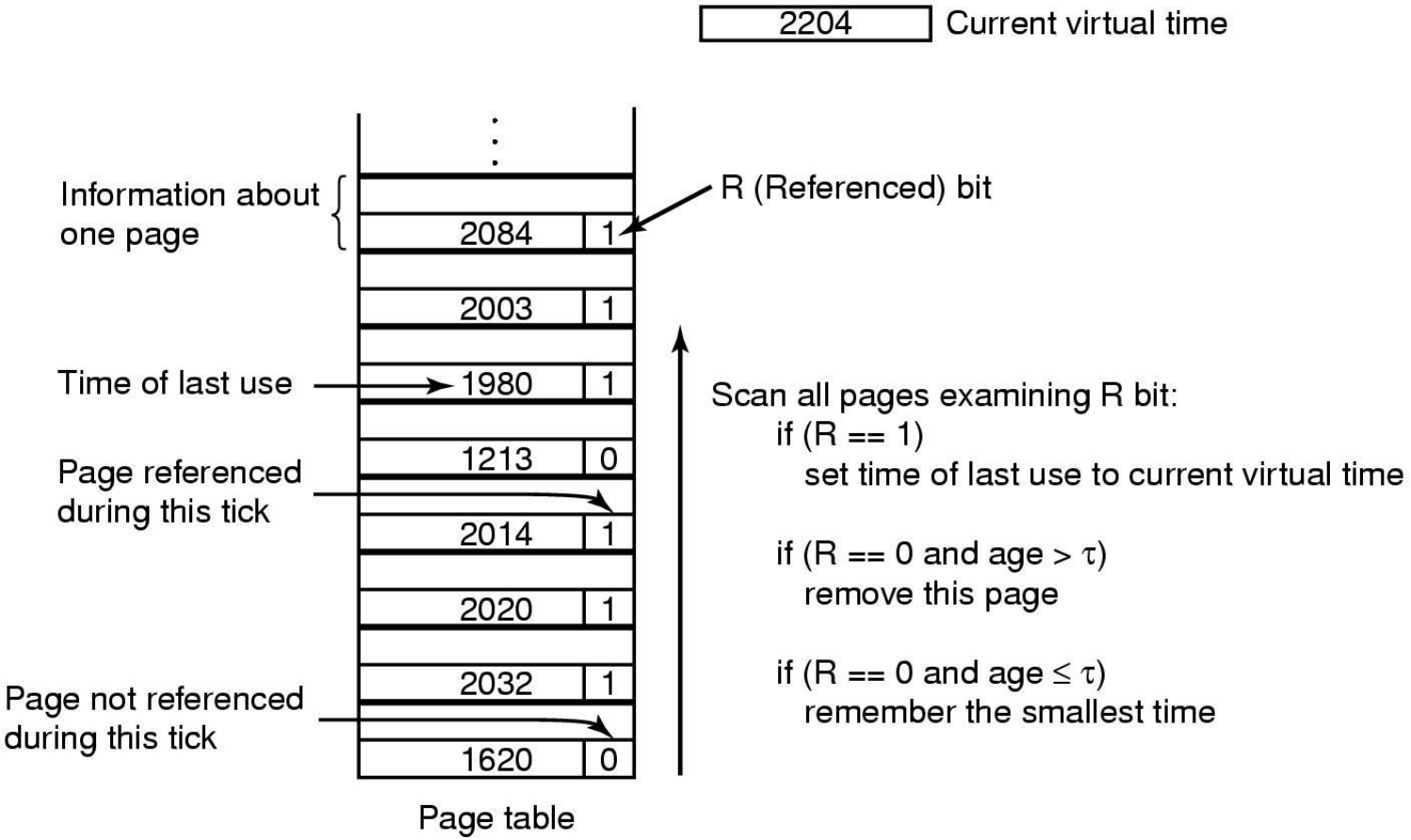
# Working Set Page Replacement Algorithm

- At every page fault scan page table
  - If R=1 the current time is written in the page entry
  - If R=0
    - If the “age” (current time - time of last reference) is smaller than t, the page is spared (but the page with the highest age/smallest time of last usage in the working set is recorded)
    - If the “age” is greater than t, the page is a candidate
  - If there is one or more candidates, the candidate with highest age is evicted
  - If there are no candidates the oldest page in the working set is evicted

- Problem:

  whole page table must be scanned every time you need find a candidate

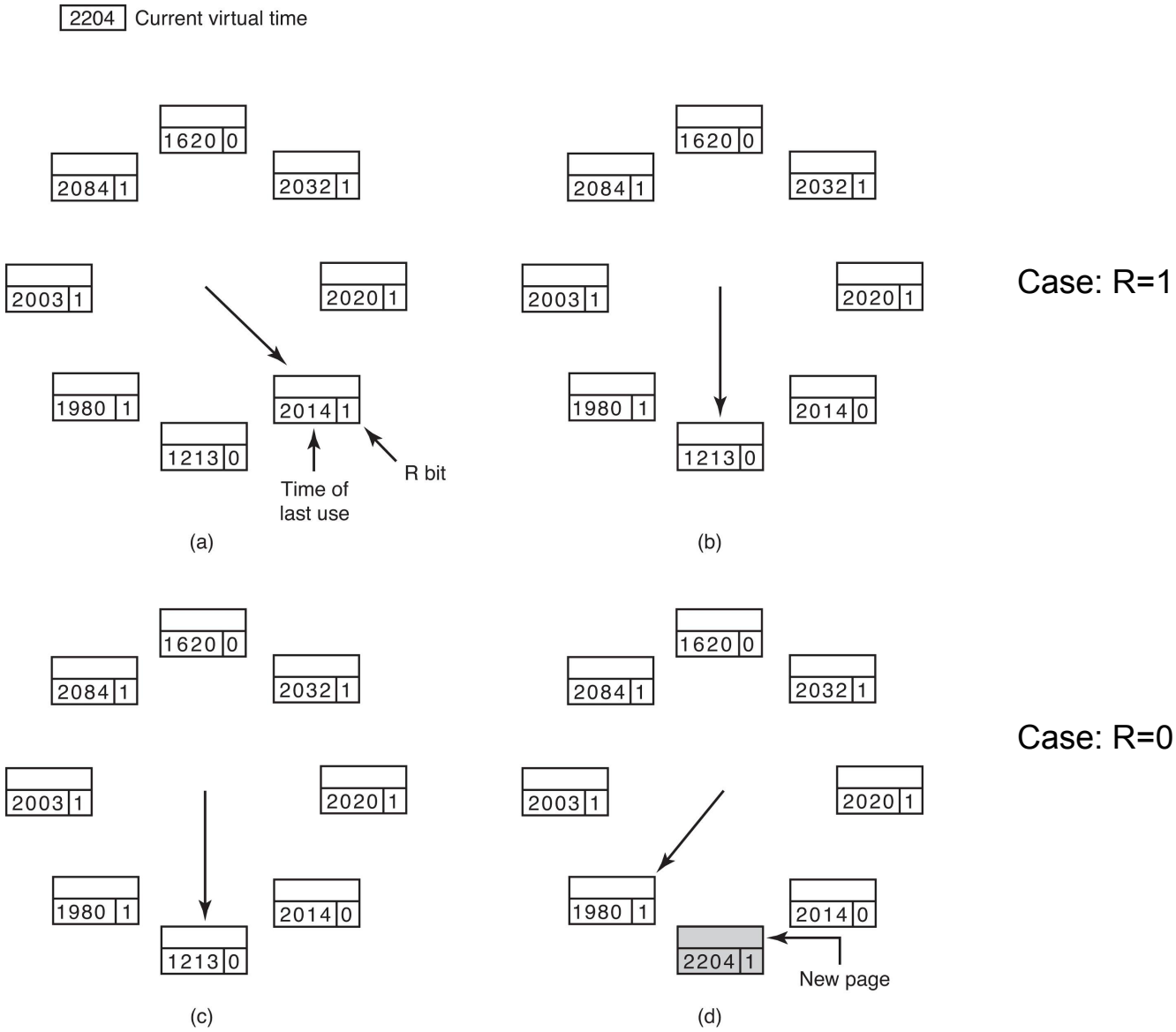
# Working Set
# Page Replacement Algorithm



# WSClock
# Page Replacement Algorithm

- Every time a page is loaded it is added to a circular list
- Each page is marked with the time of last use
- At page fault:
  - Examine page pointed by hand
    - If R=1: R is cleared, time is updated and hand advanced
    - If R=0:
      - If age is greater than t
        - » If page is clean (M=0) then evict
        - » If page is dirty (M=1) page is scheduled for writing to disk and hand advanced
      - If age is less than t the hand is advanced
  - If hand returns to initial position
    - If no writes are scheduled choose a random clean page
    - If writes have been scheduled continue until a clean, old page is found

# WSClock




Case: R=1

Case: R=0

# Review of Page Replacement Algorithms

| Algorithm | Comment |
|---|---|
| Optimal | Not implementable, but useful as a benchmark |
| NRU (Not Recently Used) | Very crude |
| FIFO (First-In, First-Out) | Might throw out important pages |
| Second chance | Big improvement over FIFO |
| Clock | Realistic |
| LRU (Least Recently Used) | Excellent, but difficult to implement exactly |
| NFU (Not Frequently Used) | Fairly crude approximation to LRU |
| Aging | Efficient algorithm that approximates LRU well |
| Working set | Somewhat expensive to implement |
| WSClock | Good efficient algorithm |

# Modeling Page Replacement Algorithms

- Program memory access is characterized as a string of referenced page numbers, called *reference string*
- Memory has $n$ virtual pages and $m < n$ page frames (what if $m \geq n$?)
- Memory is modeled as an array M divided in two portions:
  - Top m rows are the actual mapping
  - Bottom $n$ - $m$ rows represent swapped pages
- As the reference string is examined
  - the top portion is checked to see if the reference page is present
    - If not, a page fault is generated
  - In any case the chosen algorithm is used to determine the configuration of the next column

# Example with LRU

- State of memory array, M, after each item in reference string is processed (n =8, m=4)

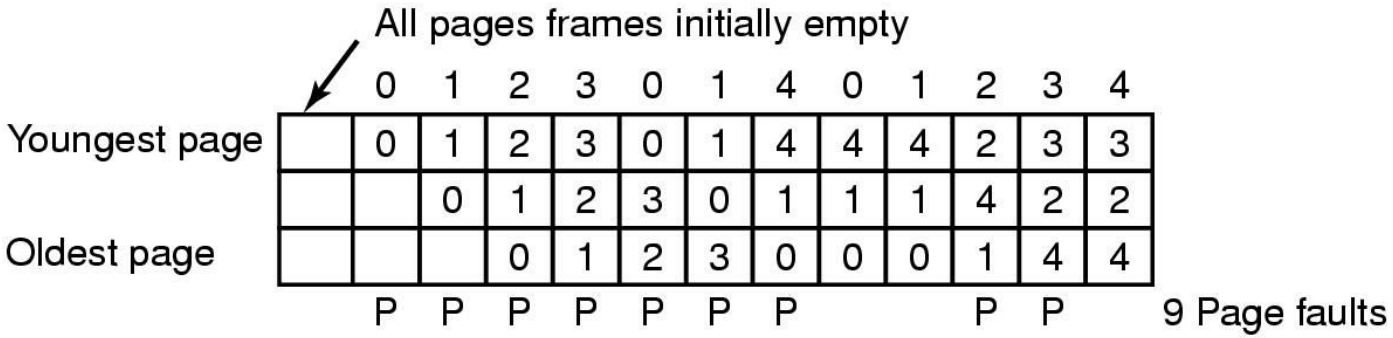
| Reference string | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 7 | 4 | 7 | 3 | 3 | 5 | 5 | 3 | 1 | 1 | 1 | 7 | 1 | 3 | 4 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 7 | 4 | 7 | 3 | 3 | 5 | 5 | 3 | 1 | 1 | 1 | 7 | 1 | 3 | 4 | 1 |
| | | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 7 | 4 | 7 | 7 | 3 | 3 | 5 | 3 | 3 | 3 | 1 | 7 | 1 | 3 | 4 |
| | | | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 3 | 4 | 4 | 7 | 7 | 7 | 5 | 5 | 5 | 3 | 3 | 7 | 1 | 3 |
| | | | | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 6 | 6 | 6 | 4 | 4 | 4 | 7 | 7 | 7 | 5 | 5 | 5 | 7 | 7 |
| | | | | | 0 | 2 | 1 | 1 | 5 | 5 | 5 | 5 | 5 | 6 | 6 | 6 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 5 |
| | | | | | | 0 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| | | | | | | | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| | | | | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Page faults | P | P | P | P | P | P | P | | P | | | | | P | | | P | | | | | | P | |

# Belady's Anomaly

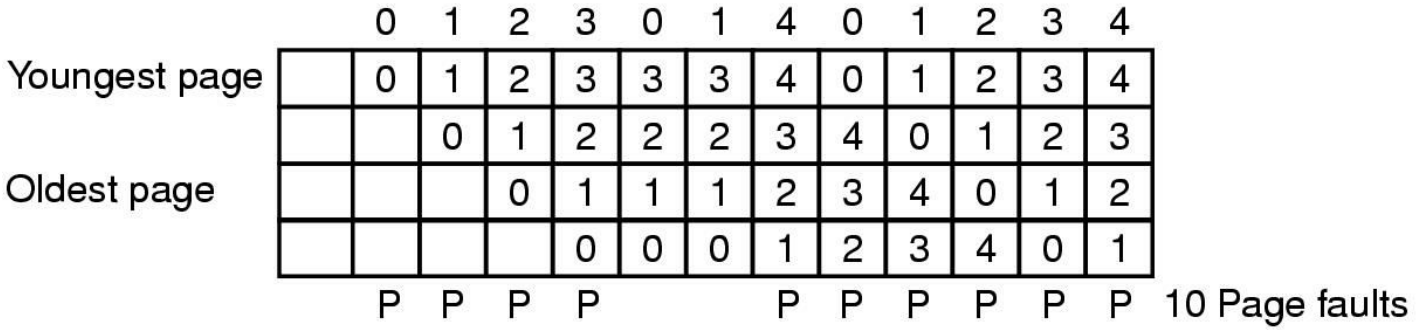
If you have more page frames
(i.e., more physical memory)
…
you'll have fewer page faults, right?!

# Belady's Anomaly (e.g., in FIFO)

FIFO with 3 page frames = 9 page faults!




FIFO with 4 page frames = 10 page faults!



# Stack Algorithms

- Algorithms that satisfy

  *M(m,r) is in M(m+1,r)*

  are called stack algorithms

- Stack algorithms do not suffer from the Belady's anomaly

- This means: if the same reference string is run on two memories with frame pages *m* and *m+1* respectively, the set of pages loaded in memory in corresponding points in the reference string are one a subset of the other

- Violated at the seventh reference in previous example

# Design Considerations

- Local vs. global policies
- Load control
- Page size and internal fragmentation
- Sharing pages
- Locking pages

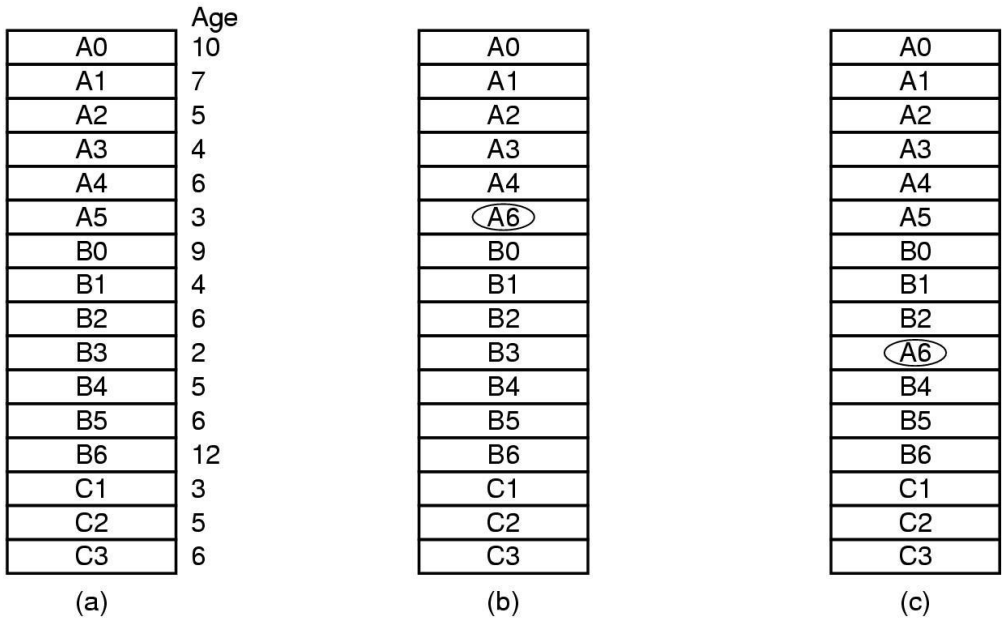
# Local versus Global Allocation Policies

(a) Original configuration

(b) Local page replacement

(c) Global page replacement

Scenario: Bring in page 6 for process A (i.e., A6)



# Load Control

- Despite good designs, system may still thrash
- The Page Fault Frequency algorithm uses the frequency of page faults to determine
  - Which processes need more memory
  - Which processes need less
- Reduce number of processes competing for memory
  - Swap one or more to disk, divide up pages they held
  - Reconsider degree of multiprogramming

# Different Page Sizes

# Small Page Sizes

- Advantages
  - Less internal fragmentation
  - Better fit for various data structures, code sections
  - Less unused program in memory

- Disadvantages
  - Programs need many pages, larger page tables
  - More TLB misses
  - Higher memory management overhead

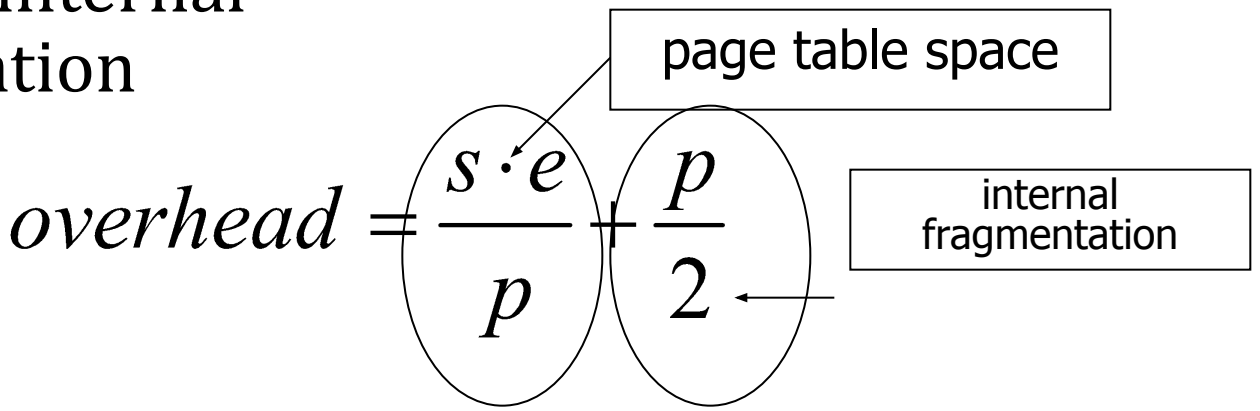
# Large Page Sizes

- Advantages
  - Smaller page tables
  - fewer TLB misses
  - Lower memory management overhead

- Disadvantages
  - More internal fragmentation
  - More unused program memory
  - Worse fit for various data structures, code sections

# Page Size

- Overhead due to page table and internal fragmentation




- Where
  - s = average process size in bytes
  - p = page size in bytes
  - e = page entry

Optimized when

$$p = \sqrt{2se}$$

With s= 1MB and e=8 bytes, optimize size is 4KB

# Page Table Overhead

- Worst case: 64-bit virtual address space, 4KB pages, 8byte PTEs
  - $2^{64}$/4KB*8byte PTEs = $2^{55}$ bytes = 32 petabytes of memory used for page tables!
  - And that's just for one process!!
  - You can clearly the page size(4KB in this case) is in the denominator.
  - This is one reason we need larger pages!!!
    - (or inverted page tables)

# Supporting multiple page sizes simultaneously

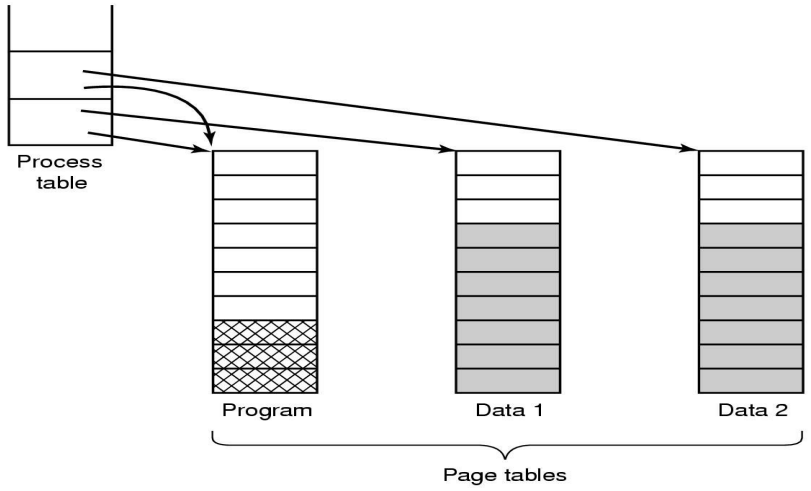
- Best of all worlds
- x86_64 supports 4KB, 2MB and 1GB page sizes
- Advantages
  - page tables sizes scale with page sizes
  - lowest possible memory management overhead
  - TLB misses scale with page sizes
  - Internal fragmentation can be minimized.
- Disadvantages
  - Much more complicated memory management algorithms and data structures.
  - Careful about external fragmentation

# Shared Memory

2 approaches one could take...

- Shared data pages
  - shared page is mapped by multiple PTEs
  - used for shared libs and on every fork()
  - This is what Linux does.

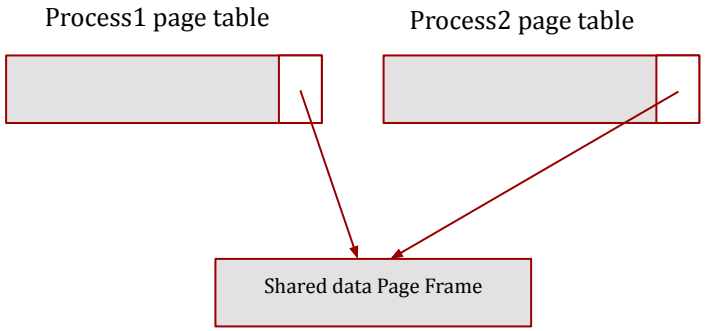
- Share page table pages
  - could be used for code...
  - shared data must be mapped at same VA



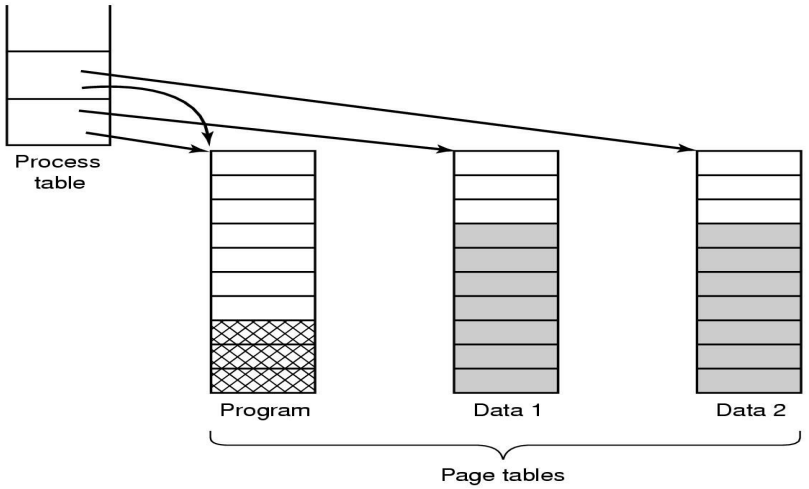
# Shared Memory

2 approaches one could take...

- Shared data pages
  - shared page is mapped by multiple PTEs
  - used for shared libs and on every fork()
  - This is what Linux does.




- Share page table pages
  - could be used for code...
  - shared data must be mapped at same VA

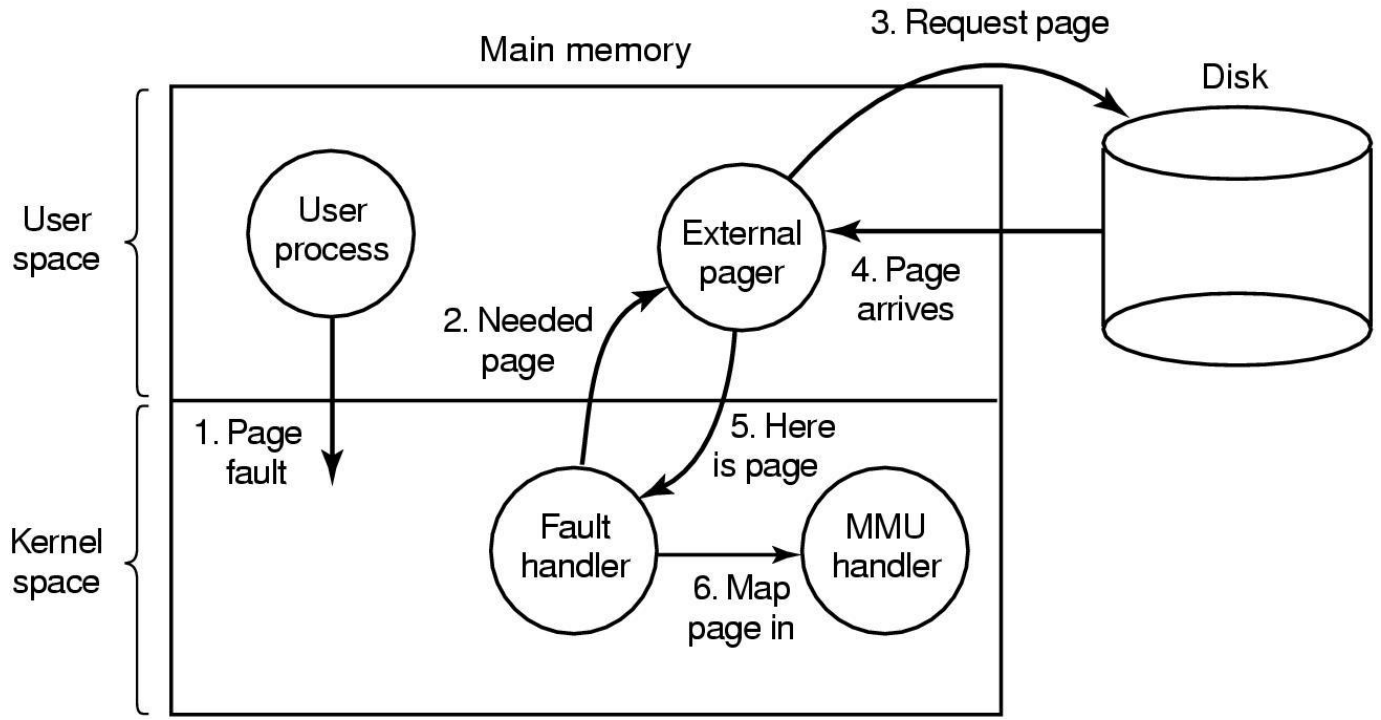


# Locking Pages in Memory

- Virtual memory and I/O occasionally interact
- Process issues call for read from device into buffer
  - While process 1 waits for I/O
  - Process 2 incurs a page fault
  - Buffer for process 1 may be chosen to be paged out
- Need to specify some pages locked
  - Exempted from being target pages
  - This can be accomplished by:
    - removing page from list of pages considered for reclaim while I/O is outstanding.
    - setting a “locked” flag in the page struct while I/O is outstanding and skipping it

# Separation of Policy and Mechanism

Page fault handling with an external pager



# What we talked about so far

- Physical memory management
- Virtual address spaces
- Segmentation
- Page tables, paging and page faults
- File backed virtual memory
- Anonymous virtual memory
- Page reclaiming
- Shared memory

# What's left?

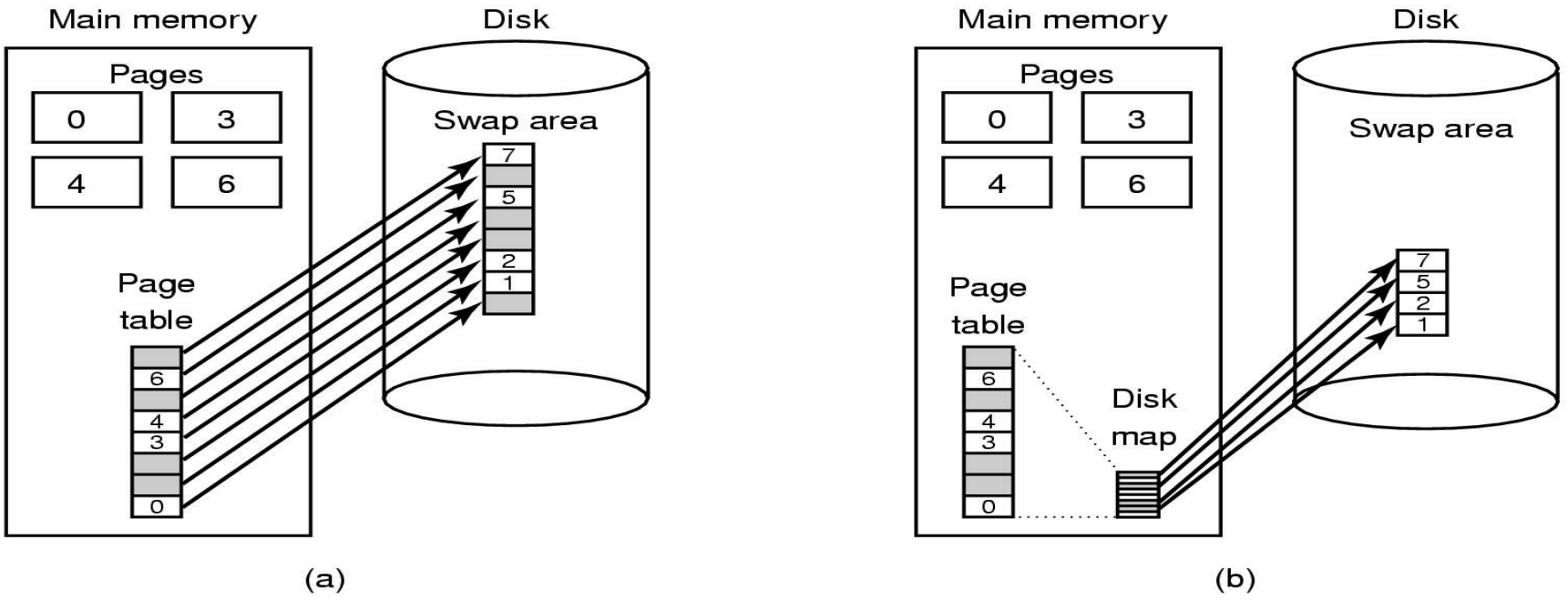
- Paging/reclaiming anonymous virtual memory
- Paging/reclaiming file backed memory
- The Buffer-Cache: file system data caching
- Page Fault details
- Anonymous memory page faults
- Mapped file page faults
- Copy On Write(COW) page faults

# Anonymous Virtual Memory

- Any virtual memory not backed by a file(stack, heap, uninitialized data, etc.)
- Allocated via mmap(fd=NULL), sbrk(), brk(), malloc()
- Freed via munmap(), sbrk(), brk(), free(), exit()
- Reclaiming anonymous memory requires it to be written-to/stored-on a paging/swap device or file before it can be freed.
  - The kernel must keep track of where it is stored.
  - Allows it to be paged back in when its referenced again.

# Backing Store

- (a) Paging to static swap area
- (b) Backing up pages dynamically



# File Backed Memory

- Any virtual memory backed by a file (code, mapped files)
  - mapping is allocated via mmap(fd=FD) system call.
  - mapping is freed via munmap() system call.
- Reclaiming file backed memory results in writing modified pages back to the file they are caching and if they are mapped, unmapping them.
- The kernel allocates and uses RAM/page frames to cache file system data in a subsystem known as the Buffer Cache.

# The Buffer Cache

- Used both for mapped files and for avoiding I/O on read/write requests...
  - Uses physical memory/Page Frames to cache file system data.
  - Improves file system performance by orders of magnitude by eliminating **most** file system IO and associated blocking.
- Integrates memory management with file systems.
- The buffer cache complicates physical memory management, especially reclaiming but well worth the effort!

# Buffer Cache implementation and use

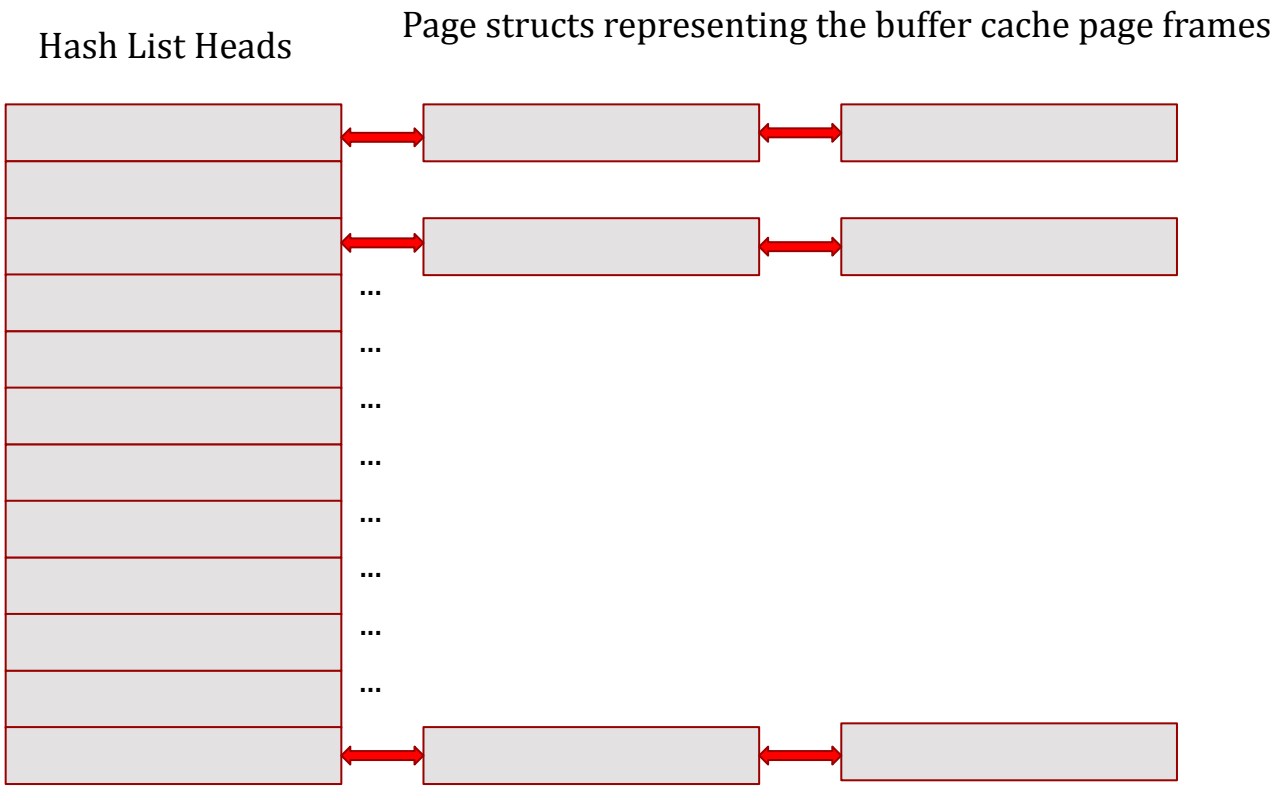
- The simplest example consists of array of hash lists.
- The page structures representing page frames containing the filesystem data are inserted into a hash list based on File/Offset tuples

1. When file system data is needed the buffer cache is searched for a page containing it:
   a. Hit: a page frame containing data is found
   b. Miss: no page frame is found
      i. Allocate a free page frame
      ii. read data from disk into the page frame
      iii. insert the page frame into page cache hash.
2. On write/read copy data to/from page frame.

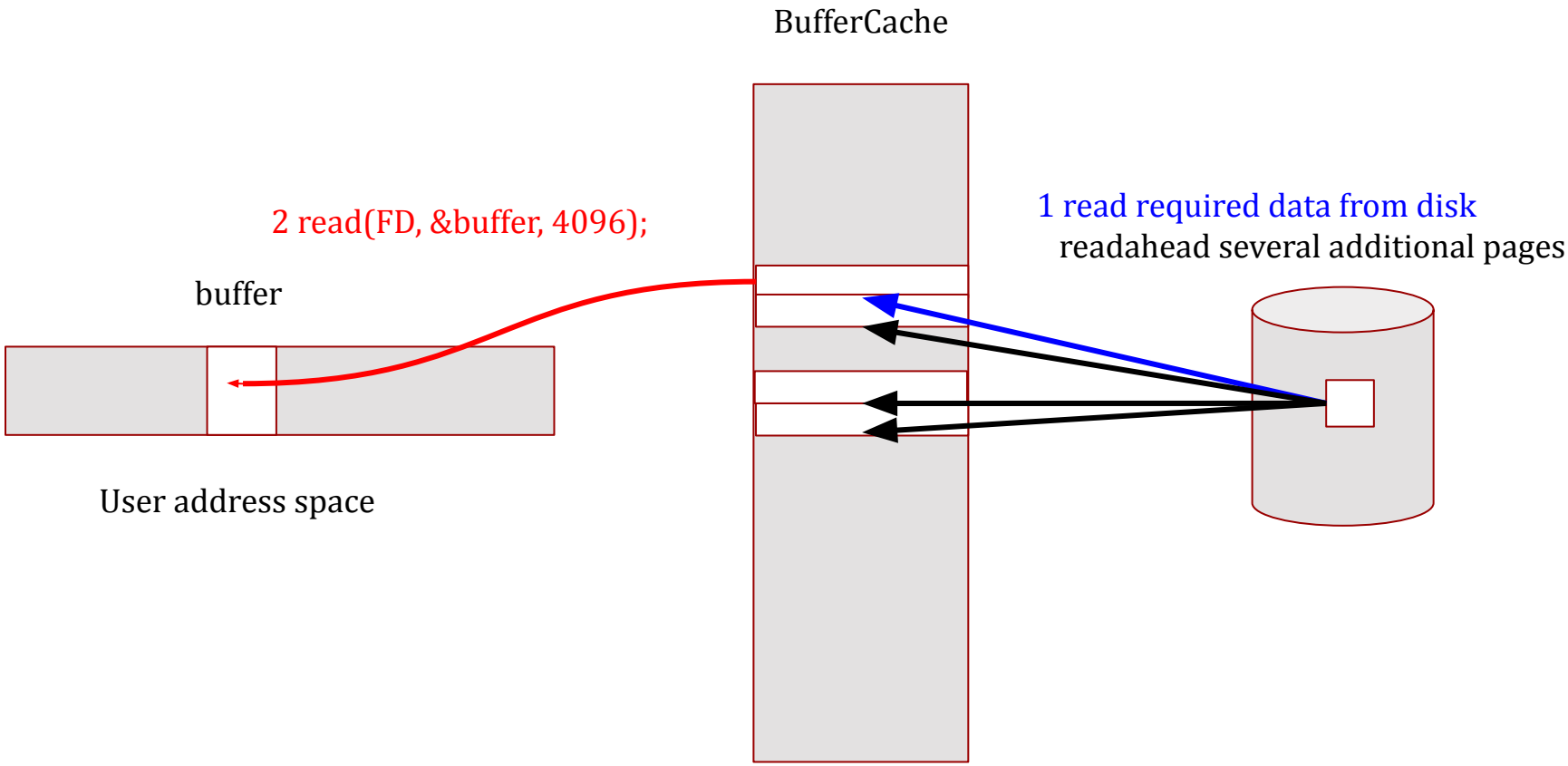
# The Buffer Cache hash

1. Use the file/offset tuple to locate the hash list to search
2. Search a short list of page structs looking for that page
3. Each page struct contains the file & offset information
4. This is a good example of where RCU should be used.

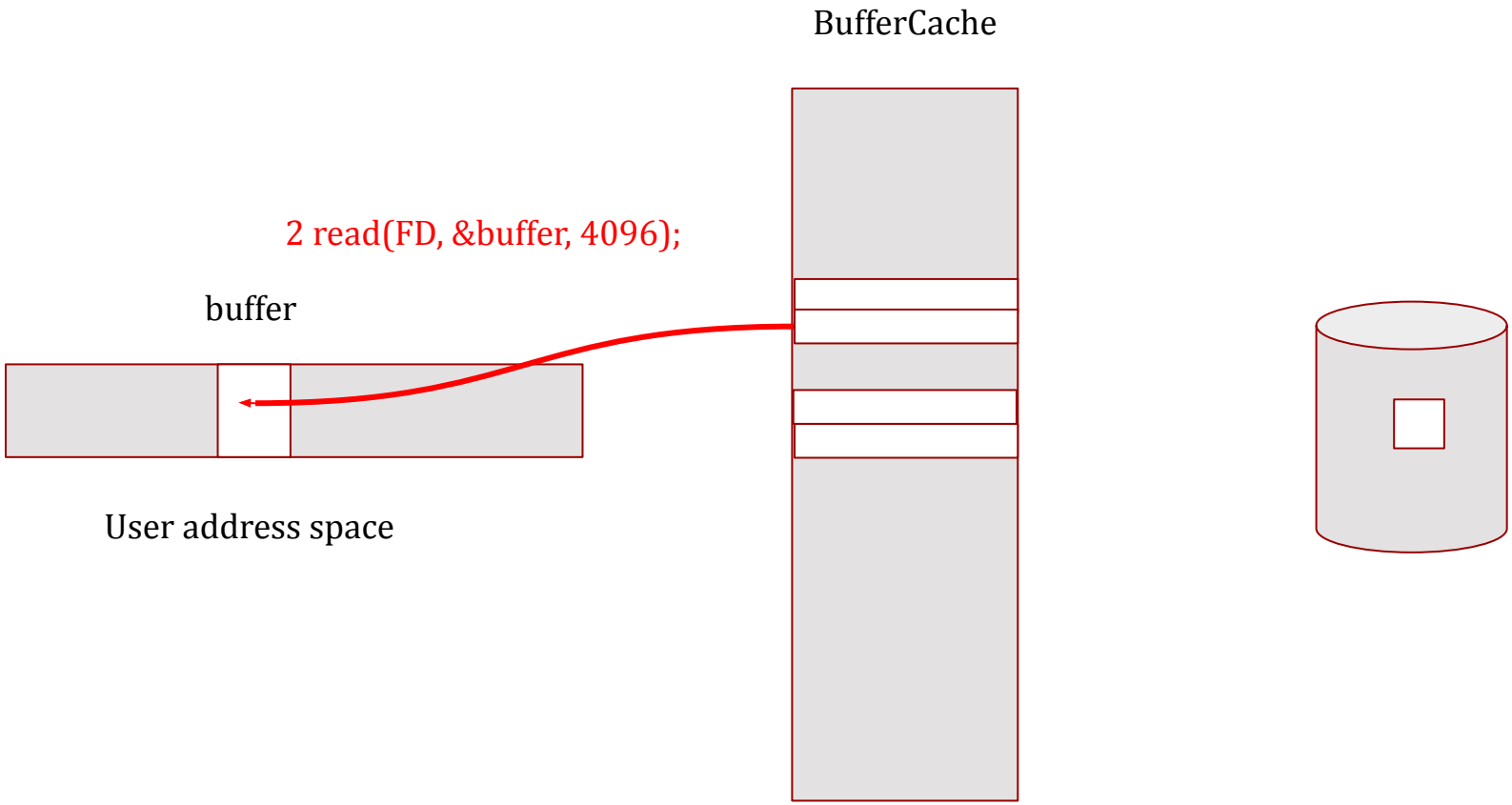
Anyone see a problem with using a hash here???



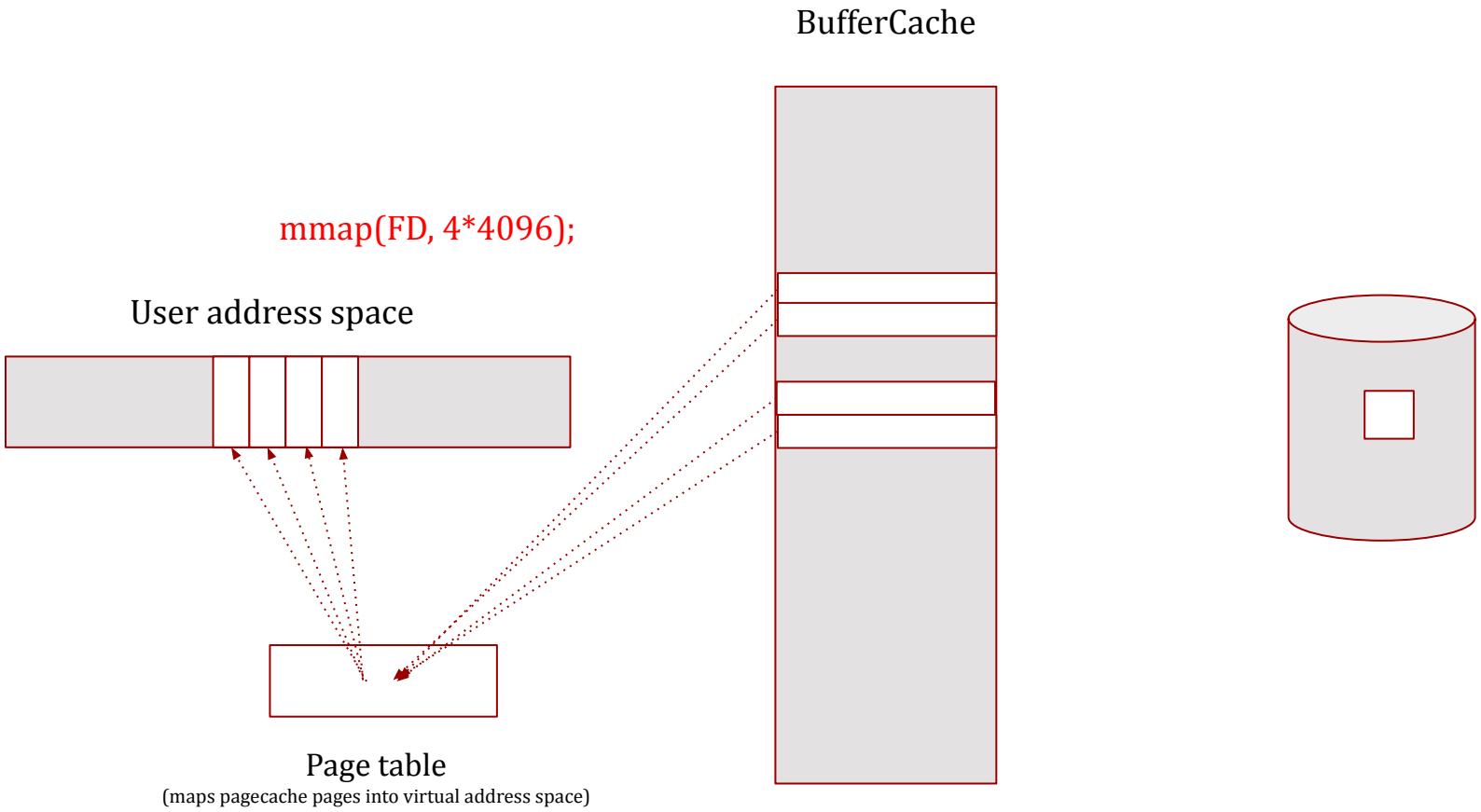
# Read() buffer cache miss



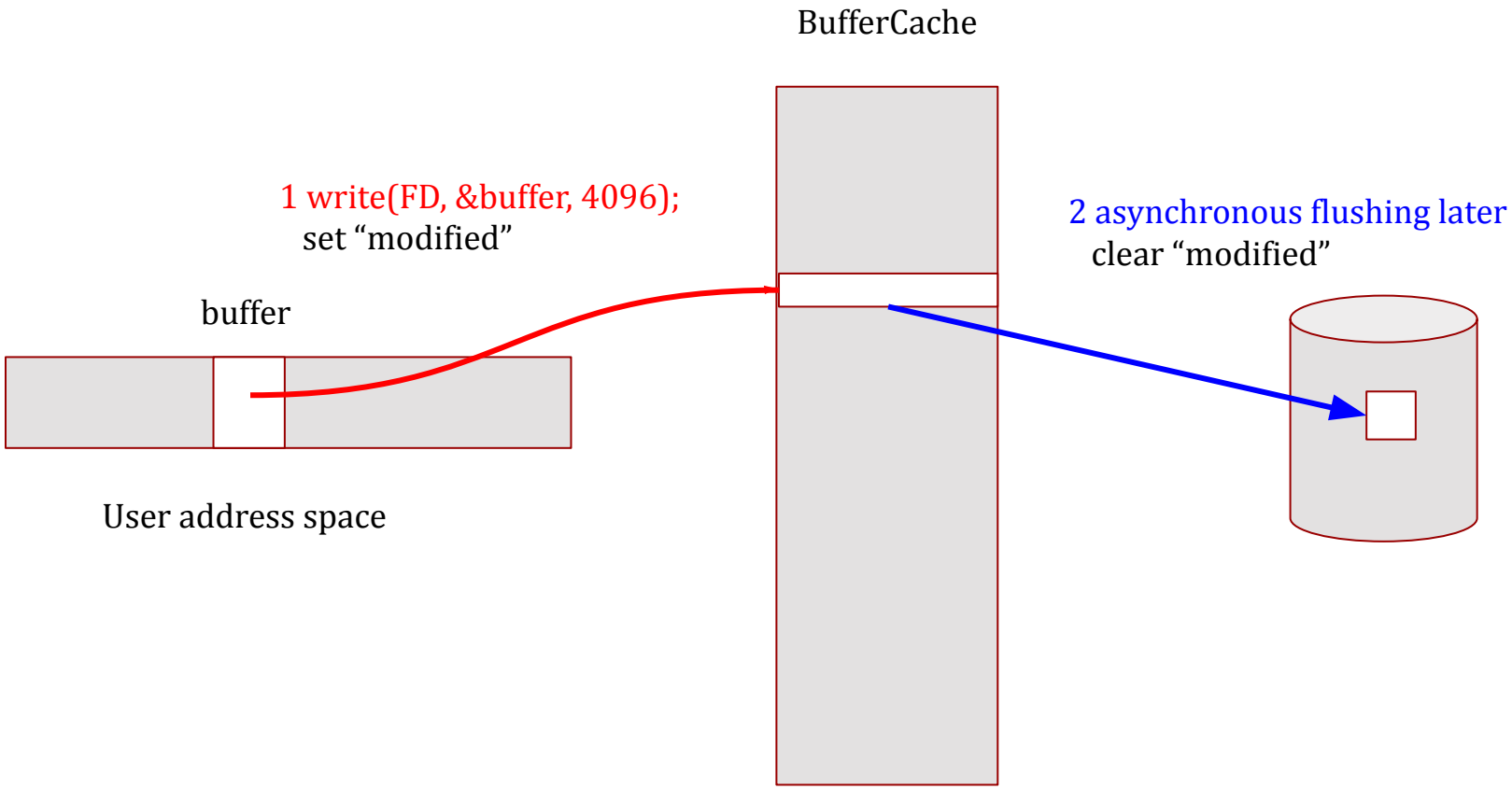
# Read() buffer cache hit



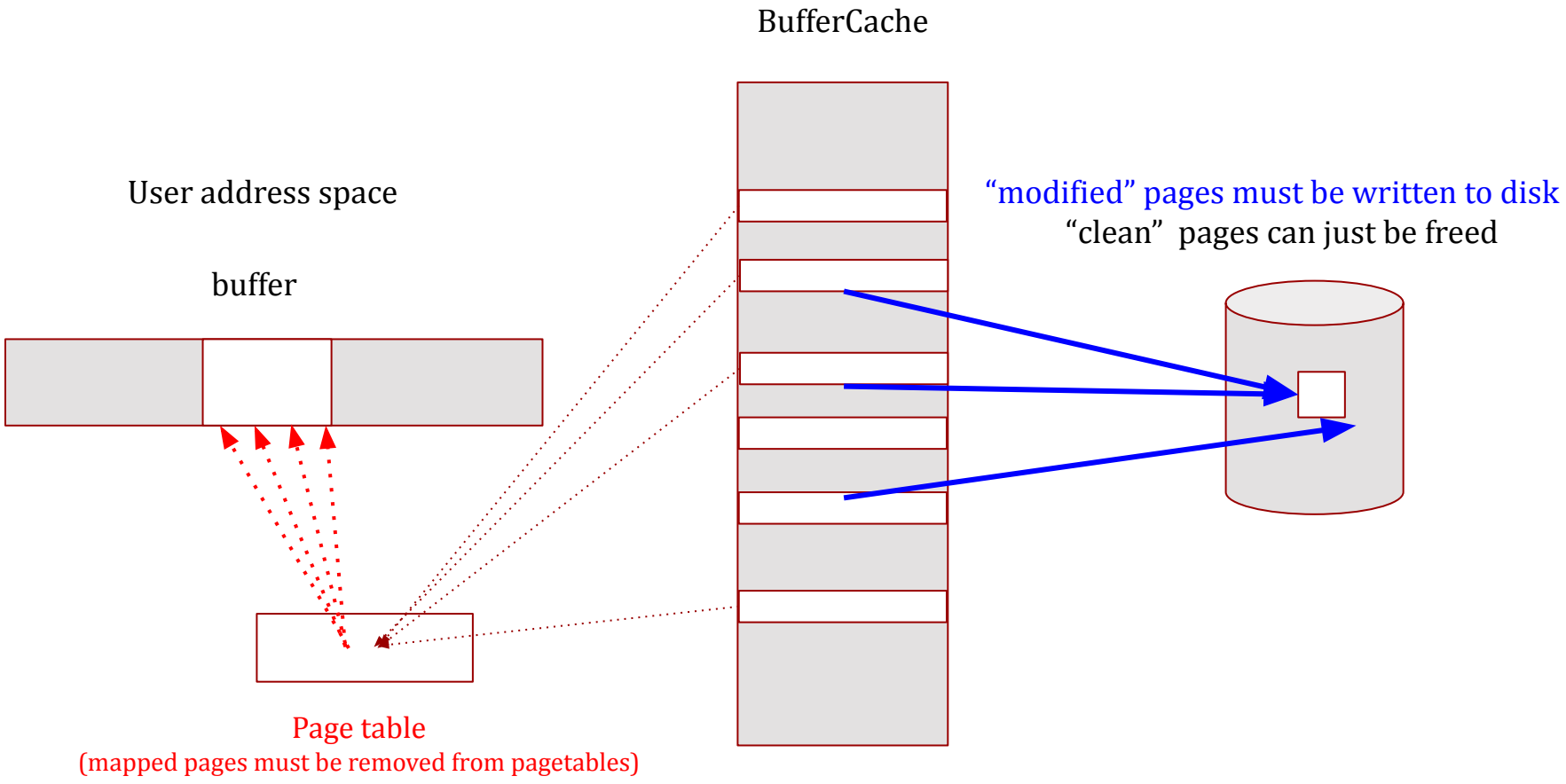
# mmap() file


BufferCache
mmap(FD, 4*4096);
User address space
Page table
(maps pagecache pages into virtual address space)

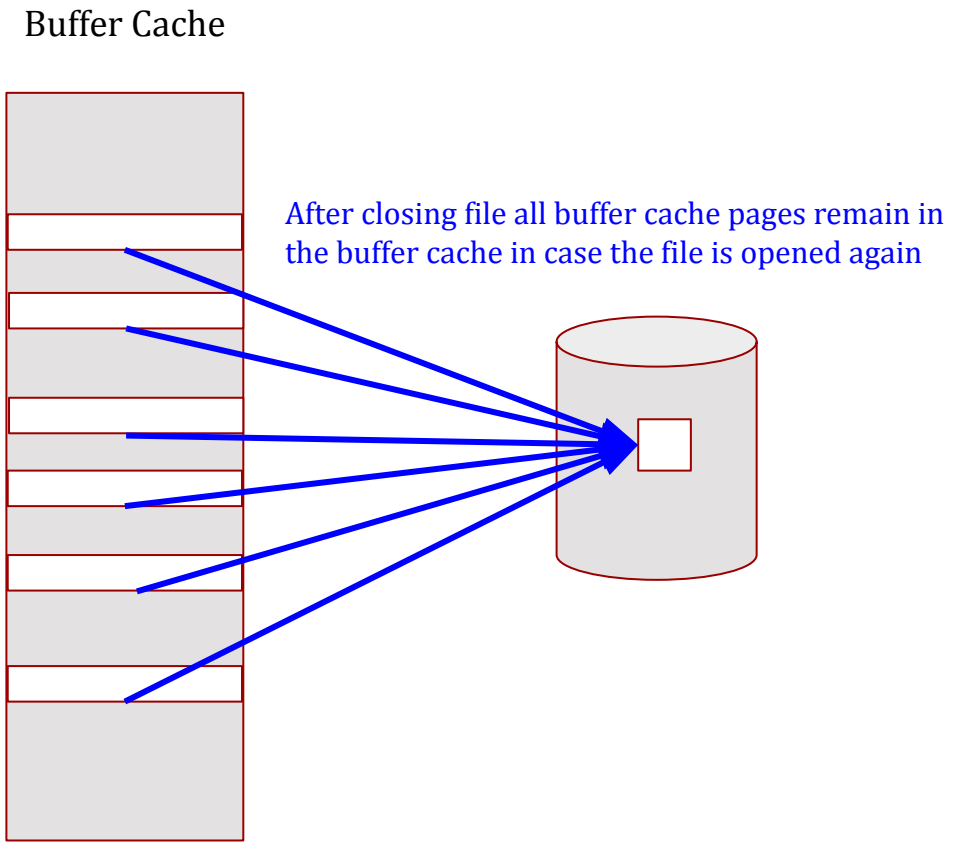
# write() into the buffer cache



# reclaiming buffer cache pages



# Closing a file: buffer cache pages remain



# Now to the details

# Operating System Involvement

- Process creation
  - Determine program size
  - Create page table
  - Allocate/reserve swap
- Process execution
  - Switching page table the for new process
  - TLB flushed
  - Make page table current
- Page fault time
  - Determine virtual address causing fault
  - Swap target page out, needed page in
- Memory allocation
  - handing out pages of RAM
  - Initiate page reclaiming if necessary
- Process termination time
  - Release page table and anonymous pages

# Page Fault details

1. User code touches VA with PTE.valid not set.
2. HW traps into kernel page fault handler
   a. CPU enters kernel mode
   b. switches to the per-thread kernel stack
3. PF handler verifies VA using MM structures
4. PF handler calls User VA specific handler:
   a. Anonymous fault handler
   b. Mapped-File fault handler
5. PF handler switches to user stack & REI to faulting user instruction.

# Anonymous Faults

- Initial anonymous page faults are Zero Filled on Demand(ZFOD)
- subsequent anonymous faults are:
  - swapped-in(if they were reclaimed)
  - Copy-On-Write(if the page is not shared)

# Mapped File Page Faults

- As we said earlier, all file data is cached in memory pag frames via the Buffer Cache.
- Mapped file page faults simply map Buffer Cache page frames into the faulting VA via the process's PTEs.
  a. locate page in the Buffer Cache
     i. (reading it into the pagecache on a miss).
  b. map Buffer Cache page frame via the PTE.

# COW Faults

- Copy On Write fault gives private copy of page
- Fork() duplicates page tables in parent/child.
  - All load/read instructions use shared pages.
  - store/write instruction creates private page.
- Files mapped “shared” always share page.
  - This is “initialized data”.
  - writes/stores get written back to disk
- Files mapped “private” use COW.
  - This is “uninitialized data” or BSS.
  - All load/read instructions use shared pages.
  - store/write instruction creates private page.