Page Frame Management

Nima Honarmand

Recap and Background

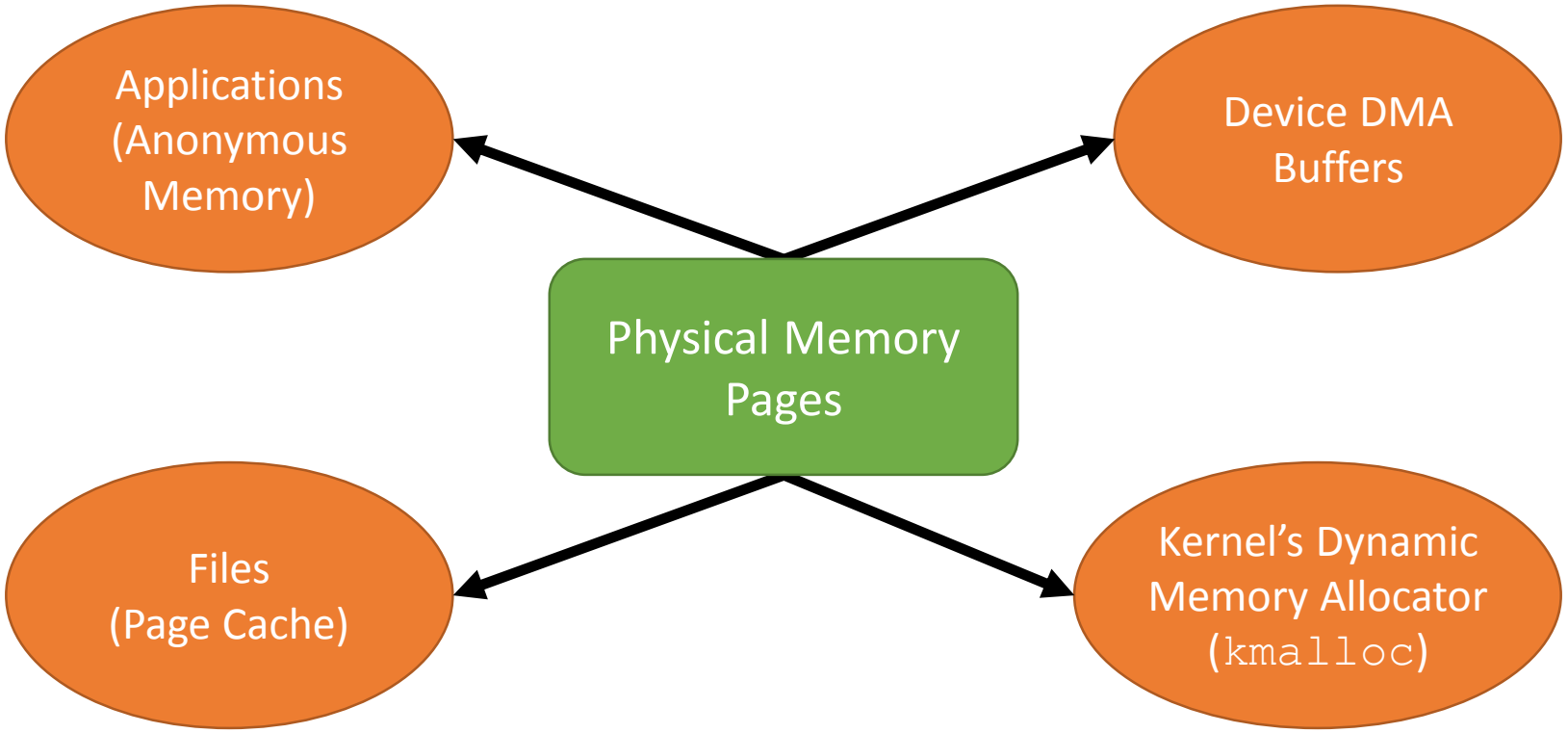
- Page tables: translate virtual addresses to physical addresses
- VM Areas (Linux): track ***what*** should be mapped in the virtual address space of a process
  - What does *mmap()* do?
- New: Linux represents physical memory with an *array* of `struct page` objects
  - Think of it as metadata for each physical page
  - Can easily find the descriptor given the physical address
  - Similar to JOS

Lecture Goals

- Part 1: How does kernel manage and allocate physical memory?
- Part 2: How does kernel reclaim physical memory?
  - **Replacement Policy:** which page to reclaim?
  - **Reverse Mapping**: given a physical page, how do I figure out which *address spaces* include it?

Part 1: How does kernel manage physical pages?

Physical Memory Users in OS



Buddy Algorithm

- Kernel tries to allocate consecutive physical pages whenever possible
  - Why?
  - DMA buffers larger than a page
  - To support 2MB and 1GB page-table entries
- Request size always a power of 2 (i.e., 2^{order}) number of pages
- Free page frames grouped into lists
  - One list for blocks of 1 PF
  - Another for blocks of 2 PFs
  - Another for blocks of 4 PFs, …
  - Last one for blocks of 1024 PFs (i.e. 4MB)

Buddy Algorithm

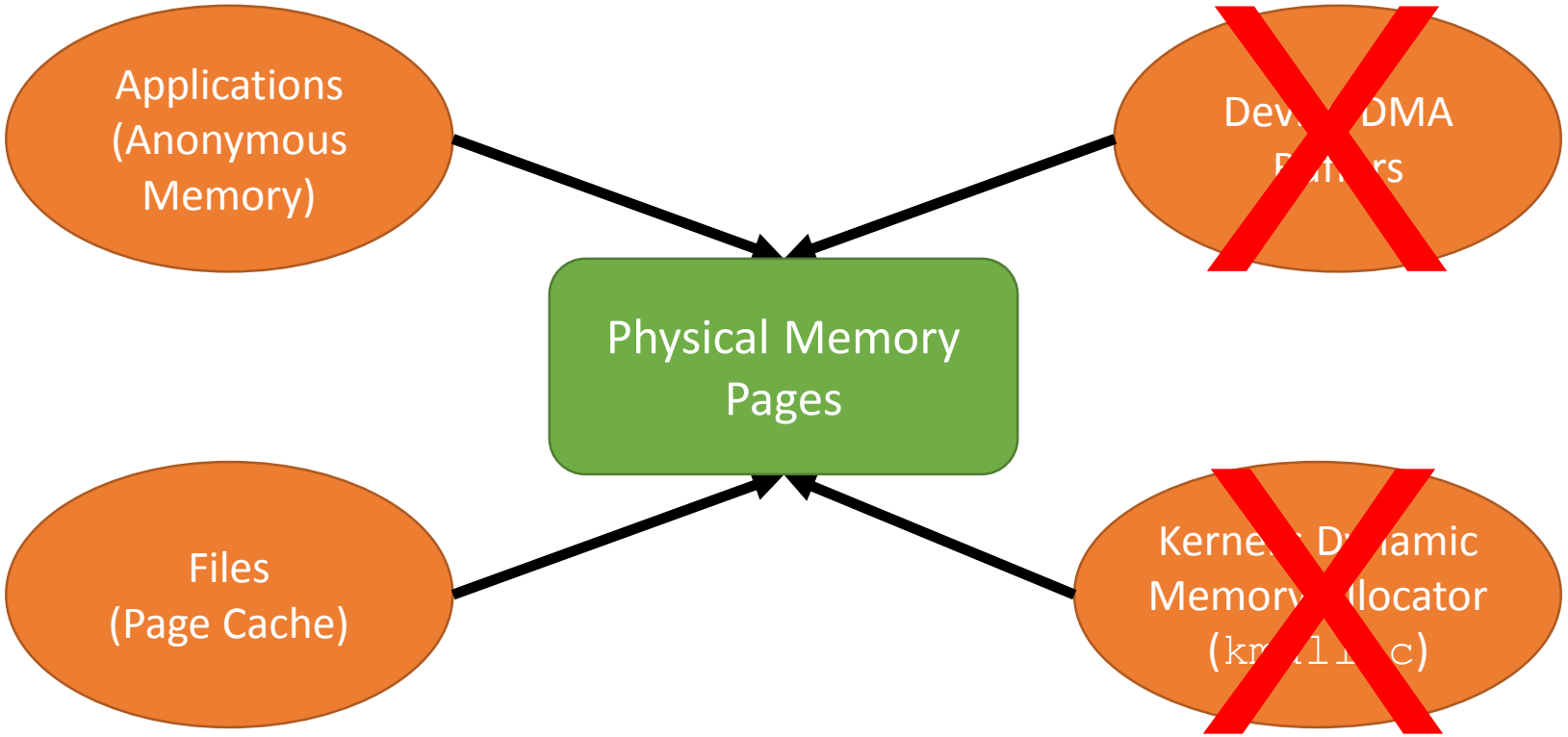
- On allocation, first check the list holding the blocks of requested size
  - If empty, check the next larger list
    - Pick a block, break it into two blocks; return one to the requester; add the other one to the smaller list
  - If also empty, continue with the next larger list
- On deallocation, check if the next block of memory is also free
  - try to *merge* buddy blocks of size *B* and create a larger buddy block of size *2B*
  - Iteratively repeat this

Part 2: How does kernel reclaim physical pages?

Motivation: Memory Overcommit

- Not every address space (process or file) uses all the memory it requests
- Most OSes allow ***memory overcommit***
  - Allocate more virtual memory than physical memory
- How does this work?
  - Physical pages allocated on demand only
  - If free space is low…
    - OS frees some pages non-critical pages (e.g., page cache)
    - Worst case, page some stuff out to disk

Whom to Reclaim From?



Swapping Pages In and Out

- To swap a page out…
  - Save contents of page to disk
  - What to do with page table entries pointing to it?
    - Clear the PTE_P bit
- If we get a page fault for a swapped page…
  - Allocate a new physical page
    - Read contents of page from disk
  - Re-map the new page (with old contents)

Choices, Choices…

- The Linux kernel decides what to swap based on scanning the page descriptor table
  - Similar to the Pages array in JOS
  - I.e., primarily by looking at physical pages
- Two questions:
  1) Given a physical page descriptor, how do I find all of the mappings? Remember, pages can be shared.
  2) What strategies should we follow when selecting a page to swap?

Question 1: Reverse Mapping

Reverse Mapping

- Given a physical page descriptor, how do I find all of the mappings?
  - First of all, where are those mappings?
    - Anonymous: just the page tables of containing process
    - Page-cache: inode's address space + page tables (if mmapped)
- Would be easy if there were no sharing
  - For anonymous pages: keep a pointer to the VMA containing the page + offset within the VMA
  - For page-cache pages: keep a pointer to the VMA (if mapped) and the inode's address space + offset within the file
  - Where to keep this data?
    - In the `struct page` descriptor of the physical page

But There is Sharing

- Recall: A VMA represents a region of a process's virtual address space
  - A VMA is private to a process
- Yet physical pages can be shared
  - E.g., the pages caching libc in memory
  - Even anonymous application data pages can be shared, after a copy-on-write fork()

→ Given a page, we need to know if it is shared, and find all VMAs and inode address space containing it

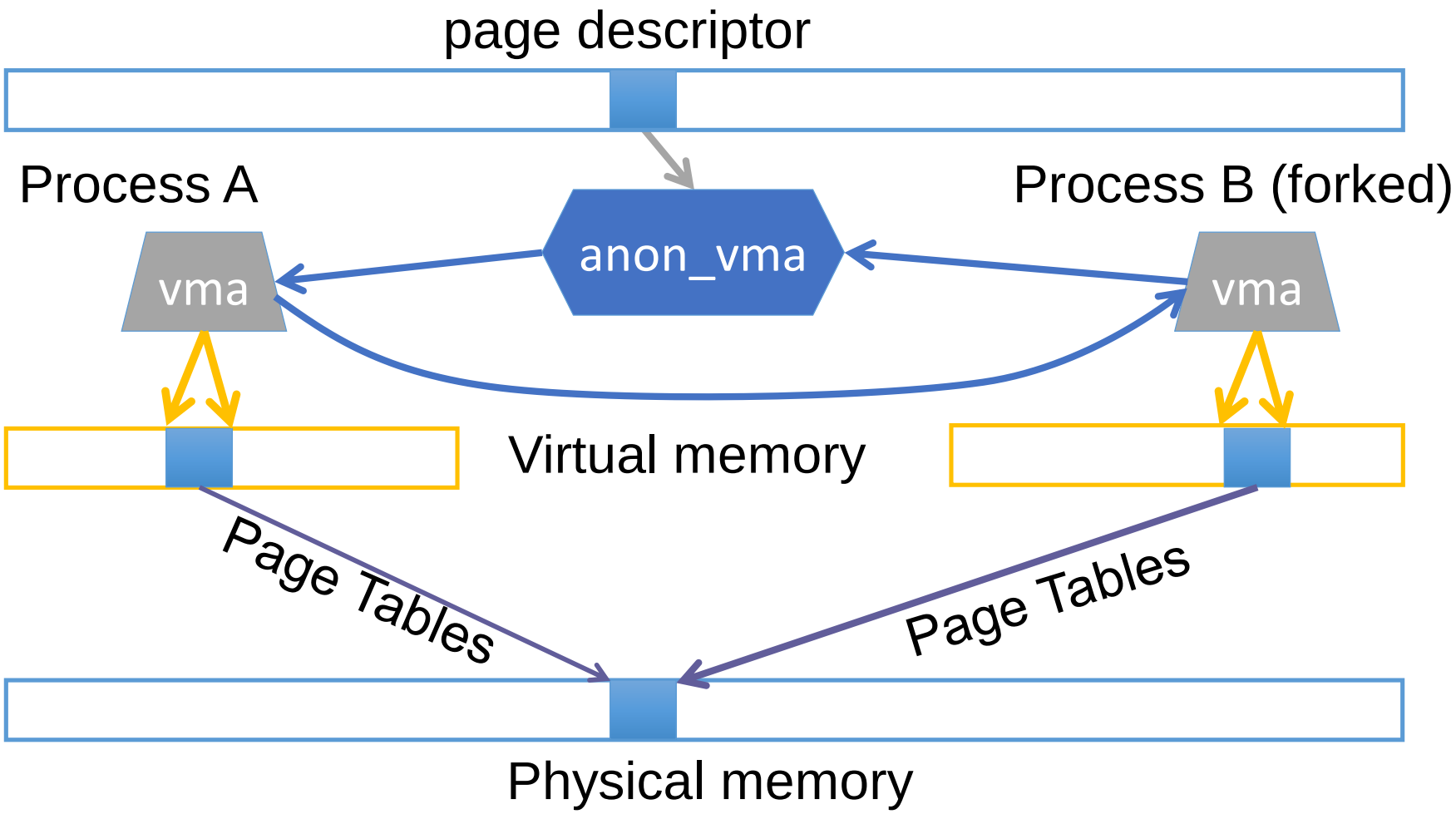
Reverse Mapping

- Pick a physical page X, what is it being used for?
- Linux example
  - Add 3 fields to each page descriptor
  - `_mapcount`: Tracks the number of active mappings
    - -1 == unmapped
    - 0 == single mapping (unshared)
    - 1+ == shared
  - `mapping`: Pointer to the owning object
    - Address space (file/device) or anon_vma (process)
    - Least Significant Bit encodes the type (1 == anon_vma)
  - `index`: offset within the VMA (for anonymous) or file (page-cache)

Tracking Anonymous Memory

- Mapping anonymous memory creates VMA
  - Physical pages are allocated on demand (laziness rules!)
- When the first physical page is added, an **`anon_vma`** structure is also created
  - VMA and page descriptor point to `anon_vma`
  - `anon_vma` stores all mapping VMAs in a circular linked list
- When a mapping becomes shared (e.g., COW fork), create a new VMA, link it on the `anon_vma` list

Example


page descriptor
Process A
Process B (forked)
anon_vma
vma
vma
Virtual memory
Page Tables
Page Tables
Physical memory

Anonymous Page Lookup

- Given a page descriptor:
  - Look at `_mapcount` to see how many mappings. If 0+:
  - Read mapping to get pointer to the `anon_vma`
    - Be sure to check, mask out low bit
- Iterate over VMAs on the `anon_vma` list
  - `index` field of `struct page` tells us which entry of the page table to check

File vs. Anonymous Pages

- Given a page mapping a file, we store a pointer in its page descriptor to the inode's address space
  - And index tells us the offset
  - → Easy to find the address space entry
- Now to find all processes mapping the file…
- So, let's just do the same thing for files as anonymous mappings, no?
  - Could just link all VMAs mapping a file into a linked list on the inode's address_space.

But There Are Complications

1. Not all file mappings map the entire file
   - Many map only a region of the file
   - Unnecessarily searching all the mappings to find a VMA
2. There can be Many mappings of a file
   - Example: libc
3. There can be different but overlapping mappings of a file

→ **Problem:** lots of entries on the list + many that might not overlap

- Need a smarter data structure

Linux Solution for File Pages (1)

- Linux uses a data structure called a Priority Search Tree to store all the VMAs mapping a file
  - `radix` index: start offset of the region
  - `heap` index: end offset of the region (exclusive)

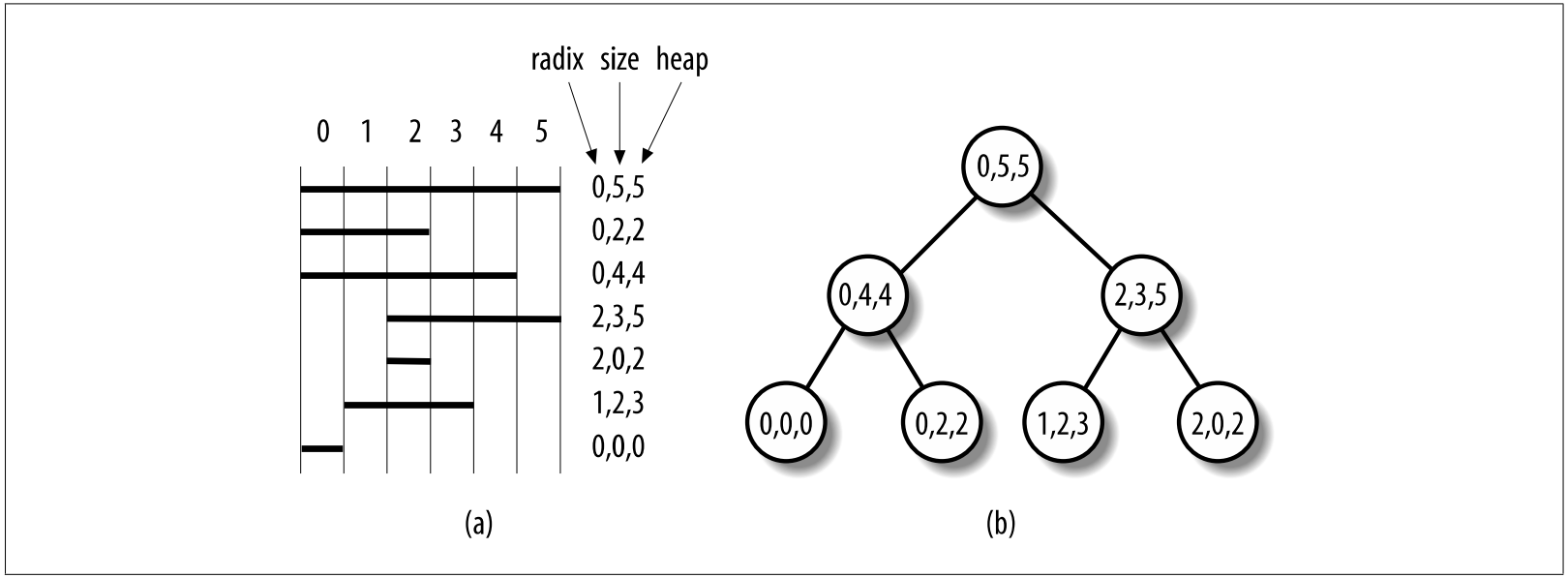


Figure 17-2. A simple example of priority search tree

Linux Solution for File Pages (2)

- Pointer to PST stored in inode's address space
- Given a file offset can easily find all the VMAs mapping it
  - Each node in PST stores a list of all VMAs corresponding to that range
- Using `index` field of `struct page` can find the linear address in the page table to invalidate
  - Recall: each VMA internally stores its own beginning offset and size

Editorial

- The data structures explained here are a bit old
  - Circa Linux 2.6
  - Especially, the linked-list-based anon_vma
- New Linux uses a more complex data structure
- **Project for extra grade (up to 5 points of course grade)**

  Investigate and write a detailed report of the *data structures and algorithms* used for reverse mapping in Linux 4.19 (latest version as of the time of this writing)

Question 2: Choosing Pages to Reclaim

Choosing Pages to Reclaim

- Until we run out of memory…
  - Kernel caches and processes go wild allocating memory
- When we run out of memory…
  - Kernel needs to reclaim physical pages for other uses
  - Doesn't necessarily mean we have zero free memory
    - Maybe just below a "comfortable" level
- Where to get free pages?
  - Goal: Minimal performance disruption

Types of Pages

1. Unreclaimable:
   - Free pages (obviously)
   - Pinned pages
   - Locked pages
2. Swappable: anonymous pages
3. Dirty file pages: data waiting to be written to disk
4. Clean file pages: contents of disk reads

General Principles

- Free harmless pages first
  - Consider dropping clean disk cache (can read it again)
  - Steal pages from user programs
    - Especially those that haven't been used recently
    - Must save them to disk in case they are needed again
  - Consider dropping dirty disk cache
    - But have to write it out to disk first
    - Doable, but not preferable
- Temporal locality: get pages that haven't been used in a while

Another View

- Suppose the system is bogging down because memory is scarce
- The problem only goes away permanently if a process can get enough memory to finish
  - Then it will free memory permanently!
- Avoid harming progress by taking away memory a process really needs
- If possible, avoid this with educated guesses

Finding Candidates to Reclaim

- Optimal technique: reclaim page that will be used farthest in the future
  - Called ***Belady*** algorithm
- But we are not oracles so we can't implement the optimal algorithm
- Approximation: use past history as indicator of future
- Try reclaiming pages not used in a while
  - All pages are on one of 2 LRU lists: *active* or *inactive*
  - Access causes page to move to the active list
  - If page not accessed for a while, moves to the inactive list

Finding Candidates to Reclaim

- How to know when an inactive page is accessed?
  - Remove PTE_P bit
    - Page fault is cheap compared to paging out bad candidate
- How to know when page isn't accessed for a while?
  - Remember the Accessed bits in PTEs?
  - Periodically clear them; if they don't get re-set by the hardware, you can assume the page is "cold"

Big Picture

- Kernel keeps a heuristic "target" of free pages
  - Makes a best effort to maintain that target
  - Can fail
- Kernel gets really worried when allocations start failing
  - In the worst case, starts out-of-memory (OOM) killing processes until memory can be reclaimed

Editorial

- Choosing the "right" pages to free is a problem without a lot of good science behind it
  - Many systems don't cope well with low-memory conditions
  - But they need to get better
    - (Think phones and other small devices)
- Important problem – perhaps a research opportunity?