Page Frame Management

Nima Honarmand

(Based on slides by Don Porter and Mike Ferdman)

Recap and background

- Page tables: translate virtual addresses to physical addresses
- VM Areas (Linux): track *what* should be mapped at in the virtual address space of a process
  - What does *mmap()* do?
- Linux represents physical memory with an array of page structs
  - Similar to JOS

Lecture goals

- Part 1: How does kernel manage and allocate physical memory?
- Part 2: How does kernel reclaim physical memory?
  - **Reverse Mapping**: given a physical page, how do I figure out, which *VMA or file inode* map to it?

Part 1: How does kernel manage physical pages?

Buddy algorithm

- Kernel tries to allocate consecutive physical pages whenever possible
  - Why? In a bit!
- 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

Why consecutive physical pages?

- DMA buffers larger than a page
- To support 2MB page-table entries
- To simplify kernel portion of the page table

Part 2: How does kernel reclaim physical pages?

Motivation: Swapping

- 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., cache)
    - Worst case, page some stuff out to disk

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
- Today's lecture:
  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?

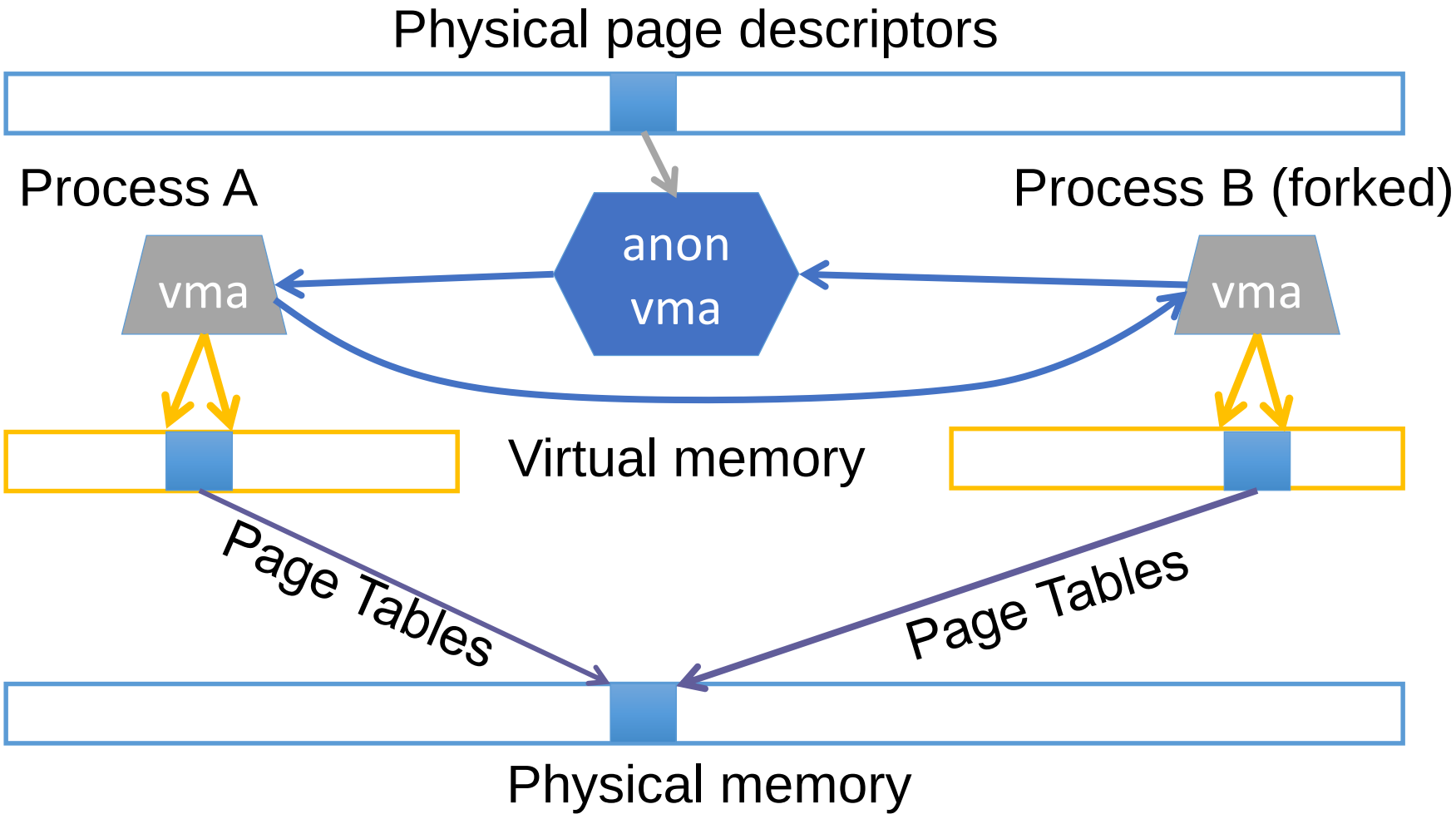
Shared pages

- 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
  - The pages caching libc in memory
  - Even anonymous application data pages can be shared, after a copy-on-write fork()
- So far, we have elided this issue. No longer!

Tracking anonymous memory

- Mapping anonymous memory creates VMA
  - Physical pages are allocated on demand (laziness rules!)
- When the first 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

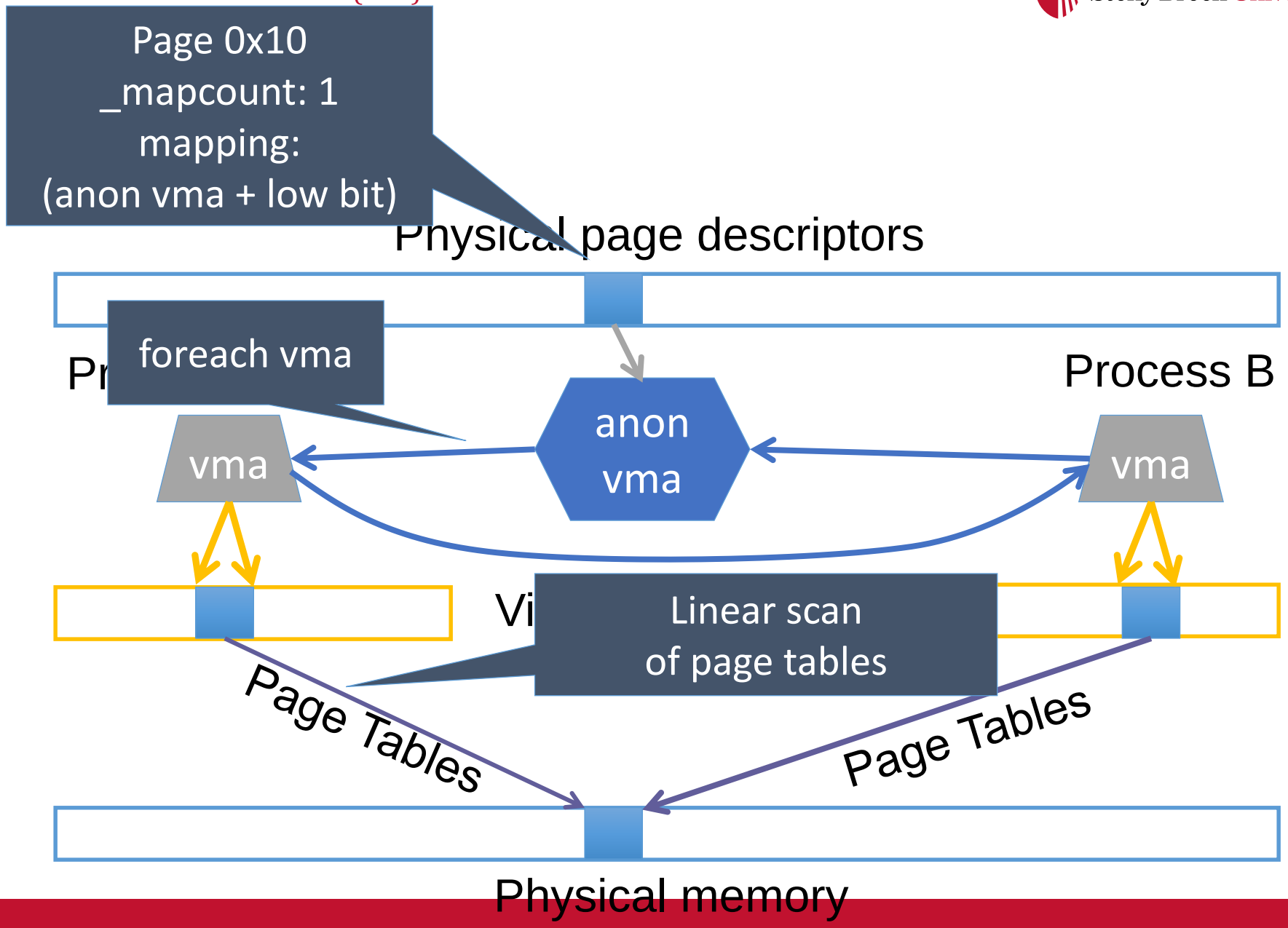

Physical page descriptors
Process A
Process B (forked)
anon
vma
vma
vma
Virtual memory
Page Tables
Page Tables
Physical memory

Reverse mapping

- Pick a physical page X, what is it being used for?
- Linux example
  - Add 2 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)

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
  - Linear scan of page table entries for each vma
    - vma-> mm -> pgdir


Page 0x10
_mapcount: 1
mapping:
(anon vma + low bit)
Physical page descriptors
foreach vma
Process B
anon
vma
vma
vma
Linear scan
of page tables
Page Tables
Page Tables
Physical memory

Choosing pages to swap

- 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
    - Should work on phone, supercomputer, and everything in between

Types of pages

- Unreclaimable:
  - Free pages (obviously)
  - Pinned/wired pages
  - Locked pages
- Swappable: anonymous pages
- Dirty file pages: data waiting to be written to disk
- 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

- 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
- 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 those hardware access bits in the page table?
  - 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 an opportunity?