CSE 506: Operating Systems

Physical Page Reclamation

VA to PA, PA to VA

- Saw how virtual addresses translate to physical
  - Where is address 0x1000 in process 100?
- How to do the reverse?
  - Given physical page X, what has a reference to it?
  - Why would you want to know?

Motivation: Paging / 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

Paging Memory

- 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)

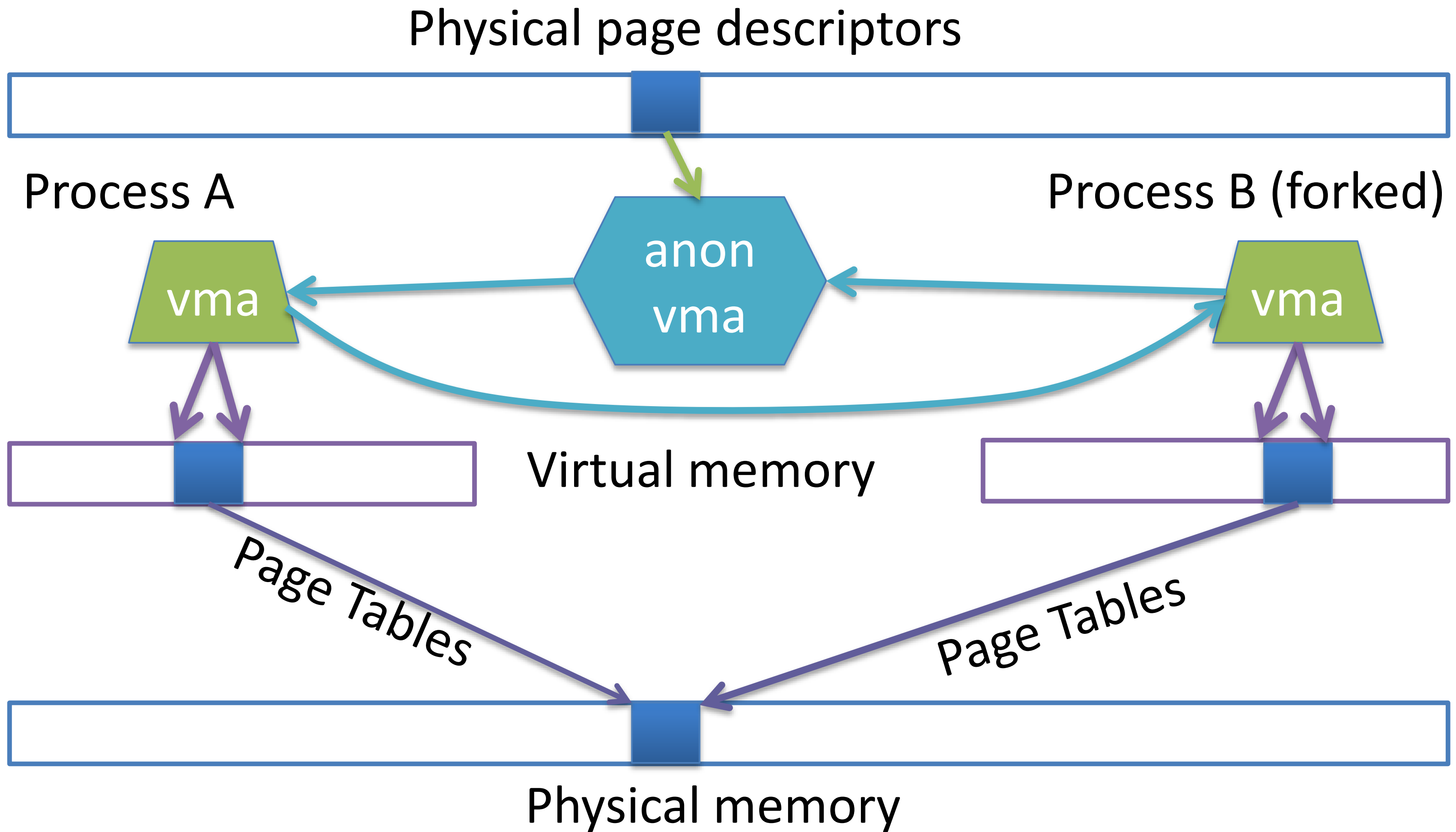
Shared Pages

- VMA represents a region of virtual address space
  - VMA is private to a process
- Physical pages can be shared
  - Files loaded from disk, opened by multiple processes
    - Most commonly, shared libraries (libc.so)
  - Anonymous pages are shared too
    - Remember COW fork()?

Tracking Anonymous Memory

- Mapping anonymous memory creates VMA
  - Pages are allocated on demand
- Keep track of VMAs that reference this page in list
  - List linked from page descriptor
    - Along-side the ->next pointer used when page was free
- Linux creates extra structure ***anon_vma***
  - Newer versions create ***anon_vma_chain***
  - Tradeoff between space/complexity and performance

Linux Example



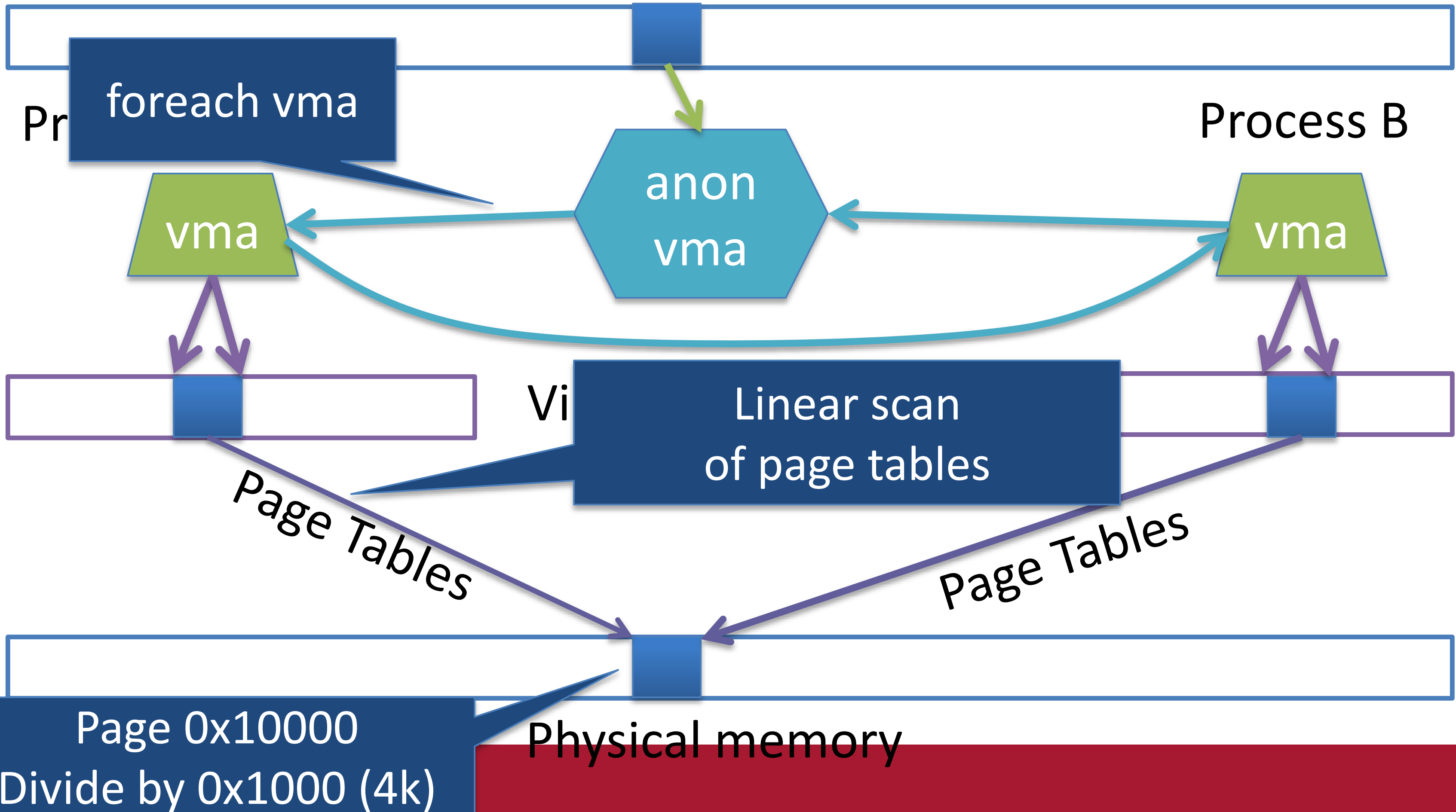
Reverse Mapping

- 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
- Iterate over VMAs on the anon_vma list
  - Linear scan of page table entries for each VMA
    - VMA -> mm -> pgdir

Example



Reclaiming Pages

- 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 Cache: data waiting to be written to disk
- Clean Cache: 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
- When reclaiming page, remove all references at once
  - Removing one reference is a waste of time
  - Consider removing entire object (needs extra linked list)

Finding Candidates to Reclaim

- Try reclaiming pages not used for 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 active page
- How to know when page isn’t accessed for a while?
  - Would page fault too often on false candidates
  - Use PTE_Accessed bit (e.g., clock algorithm)

Big Picture

- Kernel keeps a heuristic "target" of page ratios
  - Free+Zeroed, Free, Cached, Active, Inactive, …
  - Makes a best effort to maintain that target
    - Can fail
      - Miserably
- Kernel gets worried when allocations start failing
  - Some allocations simply can't fail
    - Kernel panic if they do
  - Some allocations can fail
    - User process called malloc()
- If things get bad, OOM kill processes?