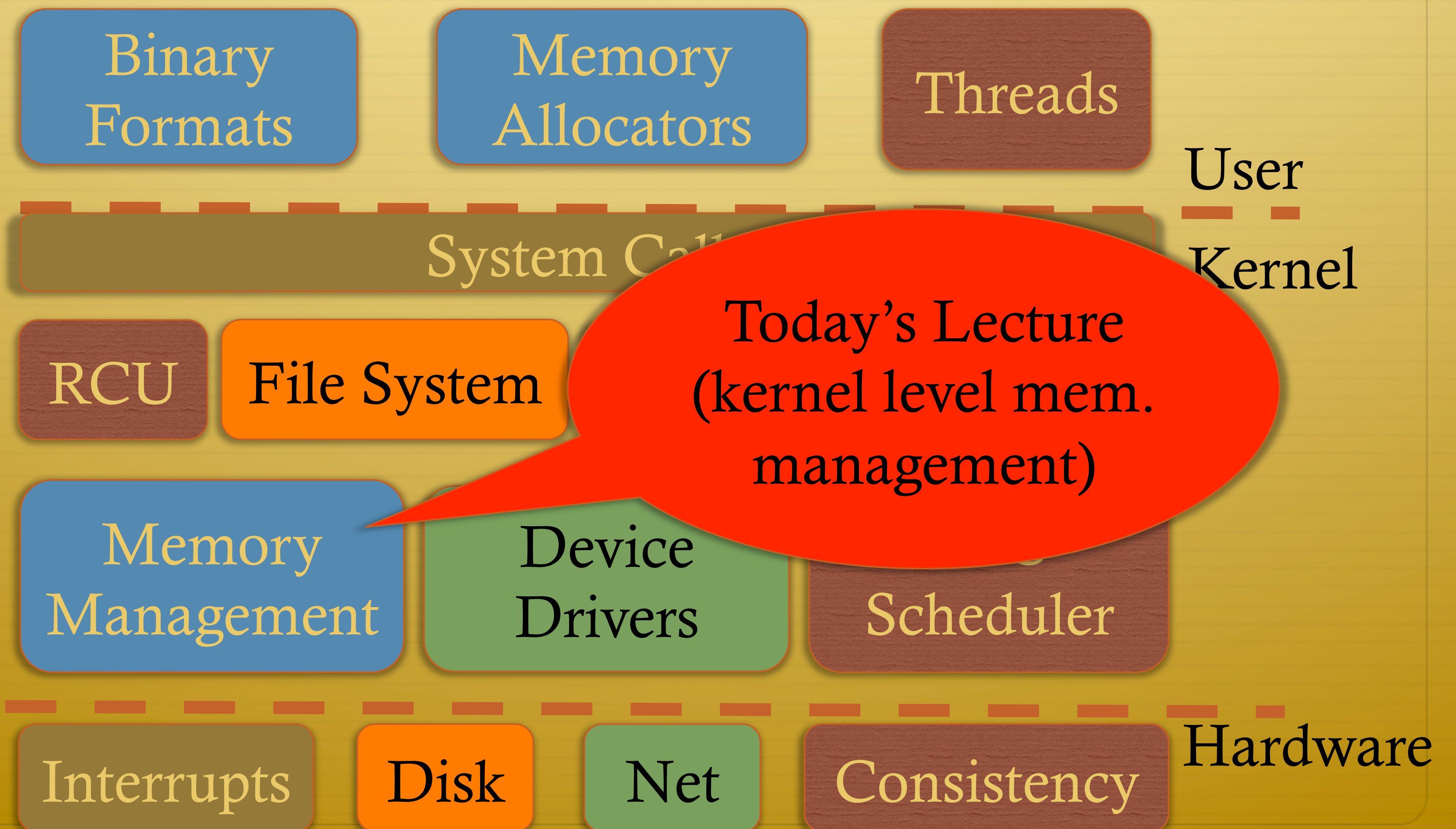
Page Frame Reclaiming

Don Porter
CSE 506

Logical Diagram



Last time…

- We saw how you go from a file or process to the constituent memory pages making it up
  - Where in memory is page 2 of file “foo”?
  - Or, where is address 0x1000 in process 100?
- Today, we look at reverse mapping:
  - Given physical page X, what has a reference to it?
- Then we will look at page reclamation:
  - Which page is the best candidate to reuse?

Physical page management

- Reminder: Similar to JOS, Linux stores physical page descriptors in an array
  - Contents are somewhat different, but same idea

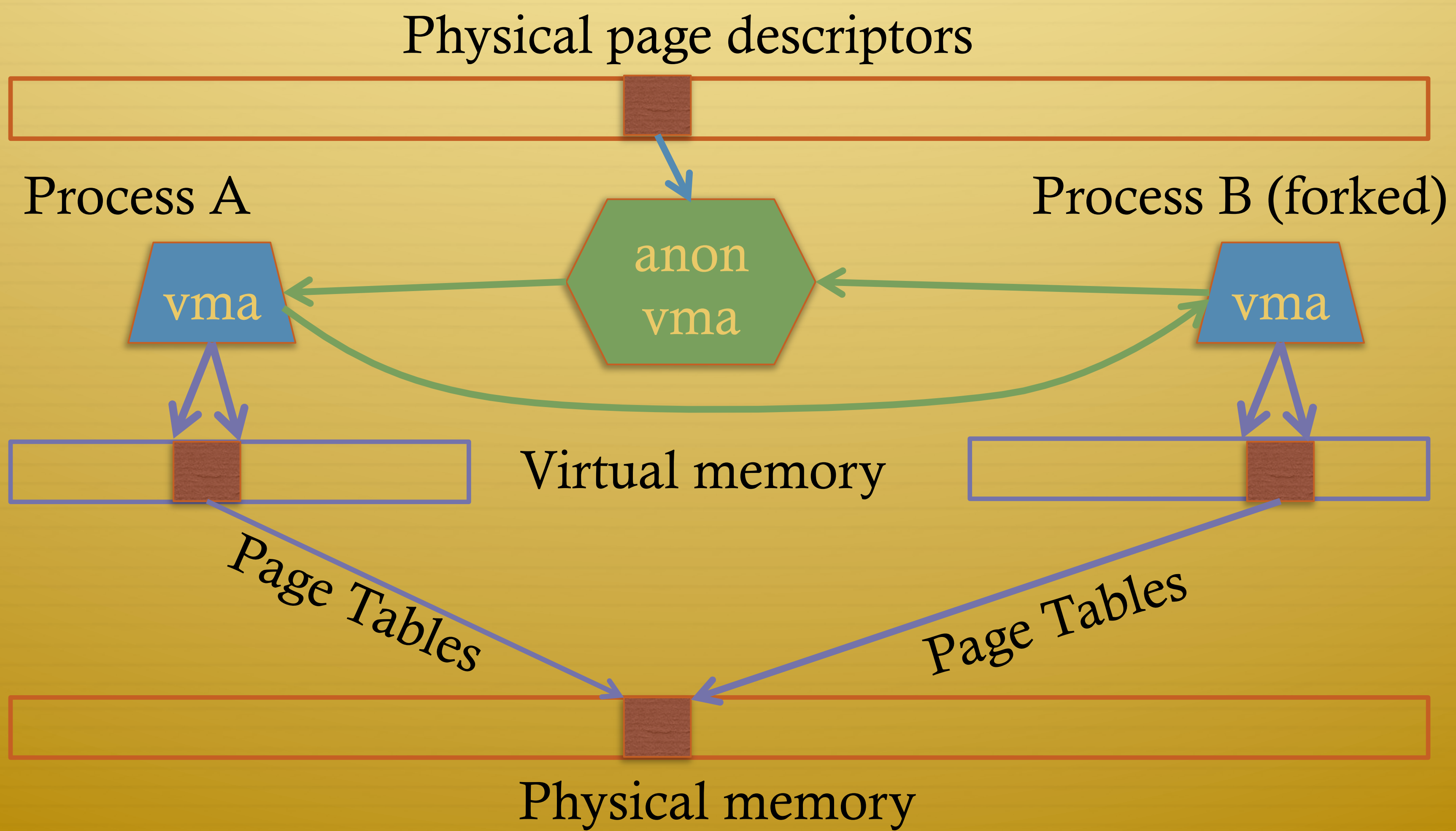
Shared memory

- 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!

Anonymous memory

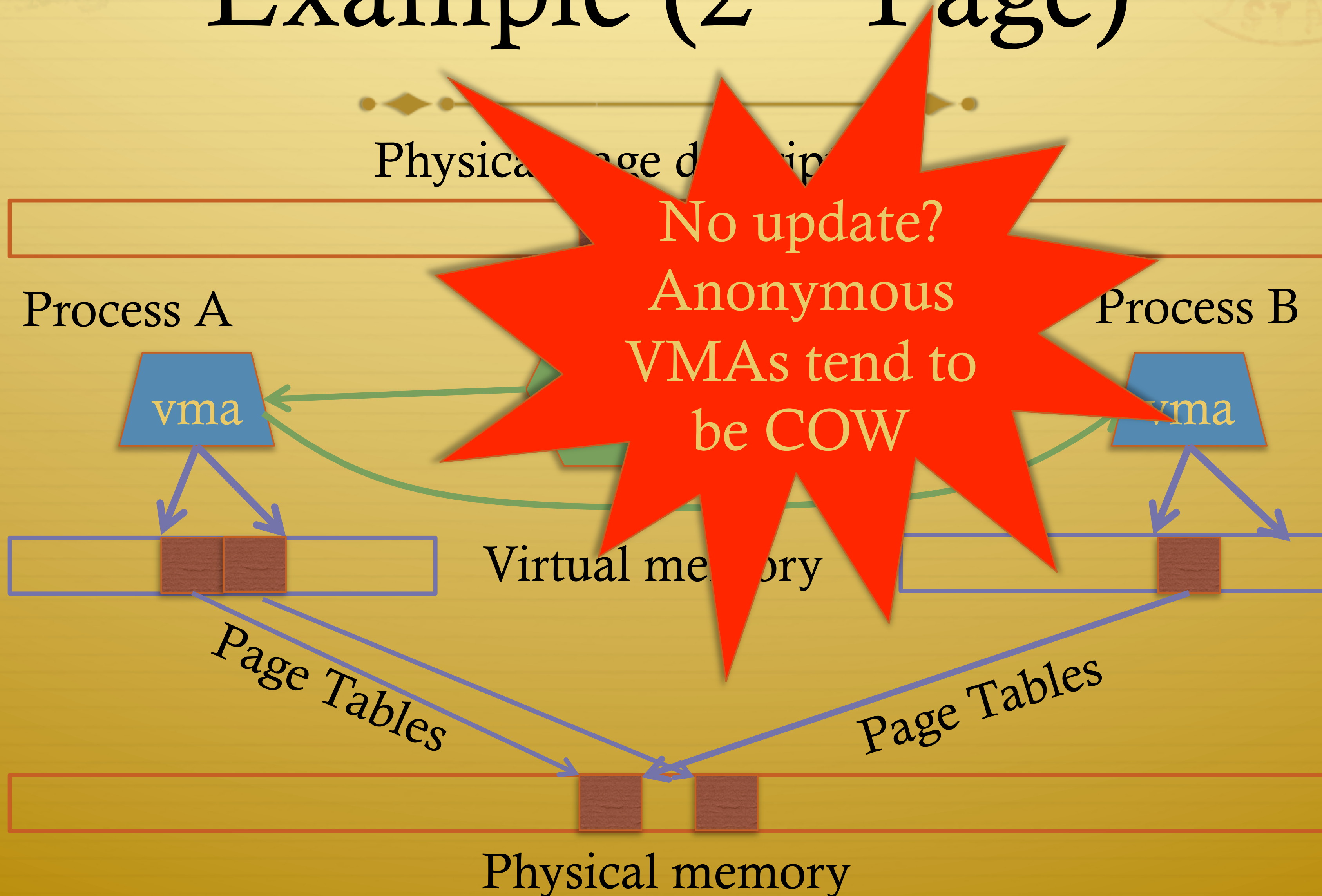
- When anonymous memory is mapped, a vma is created
  - Pages are added on demand (laziness rules!)
- When the first page is added, an anon_vma structure is also created
  - vma and page descriptor point to anon_vma
  - anon_vma stores all mapping vmas in a circular linked list
- When a mapping becomes shared (e.g., COW fork), create a new VMA, link it on the anon_vma list

Example



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

Example (2nd Page)



Reverse mapping

- Suppose I pick a physical page X, what is it being used for?
- Many ways you could represent this
- Remember, some systems have a lot of physical memory
  - So we want to keep fixed, per-page overheads low
  - Can dynamically allocate some extra bookkeeping

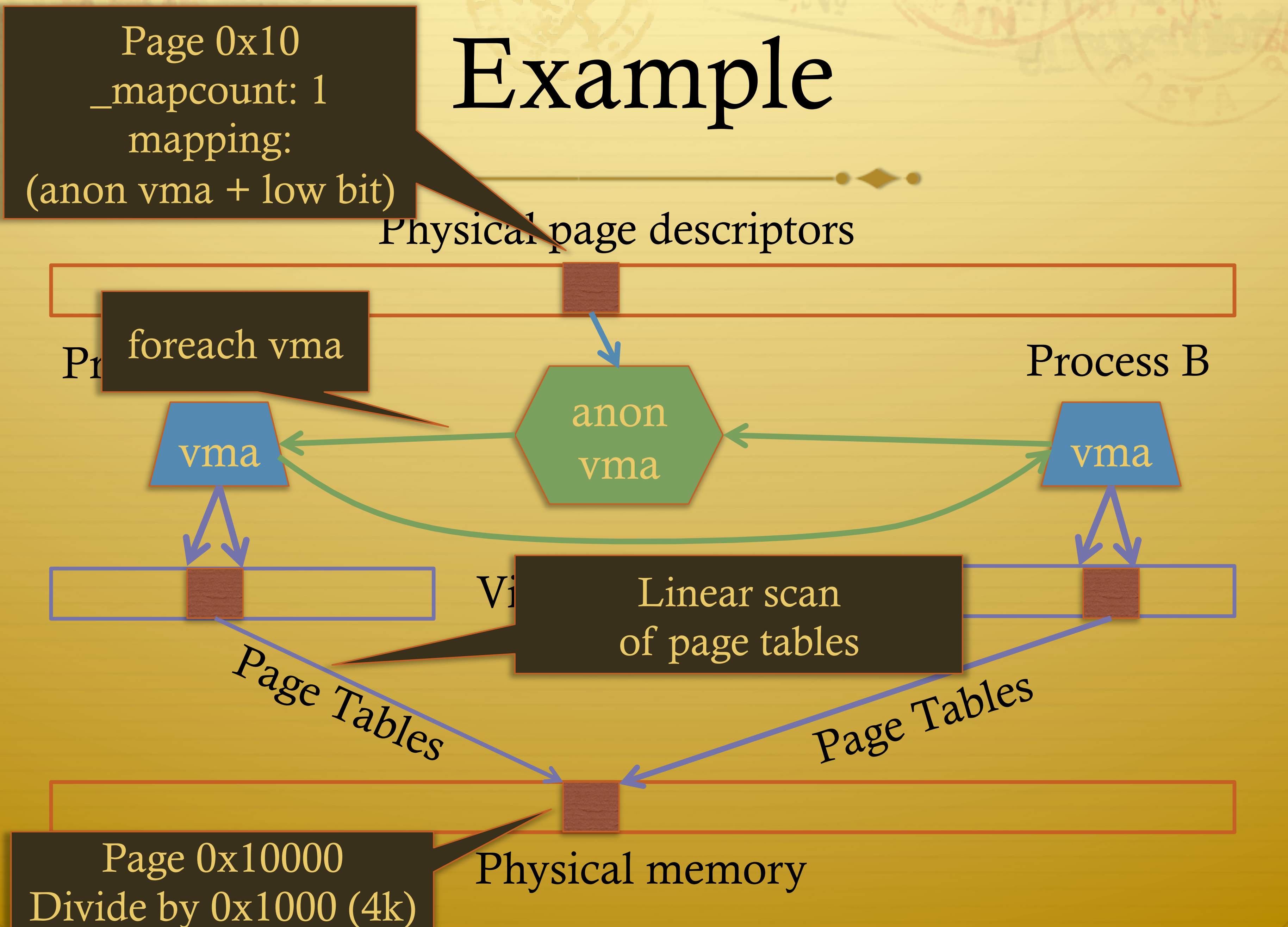
Linux strategy

- 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 physical address, page descriptor index is just simple division by page size
- 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

Example



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
Page 0x10000
Divide by 0x1000 (4k)
Physical memory

File vs. anon mappings

- Given a page mapping a file, we store a pointer in its page descriptor to the inode address space
  - Linear scan of the radix tree to figure out what offset in the file is being mapped
- 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.
- 2 complications:

Complication 1

- Not all file mappings map the entire file
  - Many map only a region of the file
- So, if I am looking for all mappings of page 4 of a file a linear scan of each mapping may have to filter vmas that don’t include page 4

Complication 2

- Intuition: anonymous mappings won't be shared much
  - How many children won't exec a new executable?
- In contrast, (some) mapped files will be shared a lot
  - Example: libc
- Problem: Lots of entries on the list + many that might not overlap
- Solution: Need some sort of filter

Priority Search Tree

- Idea: binary search tree that uses overlapping ranges as node keys
  - Bigger, enclosing ranges are the parents, smaller ranges are children
  - Not balanced (in Linux, some uses balance them)
- Use case: Search for all ranges that include page N
- Most of that logarithmic lookup goodness you love from tree-structured data!

Figure 17-2

(from Understanding the Linux Kernel)

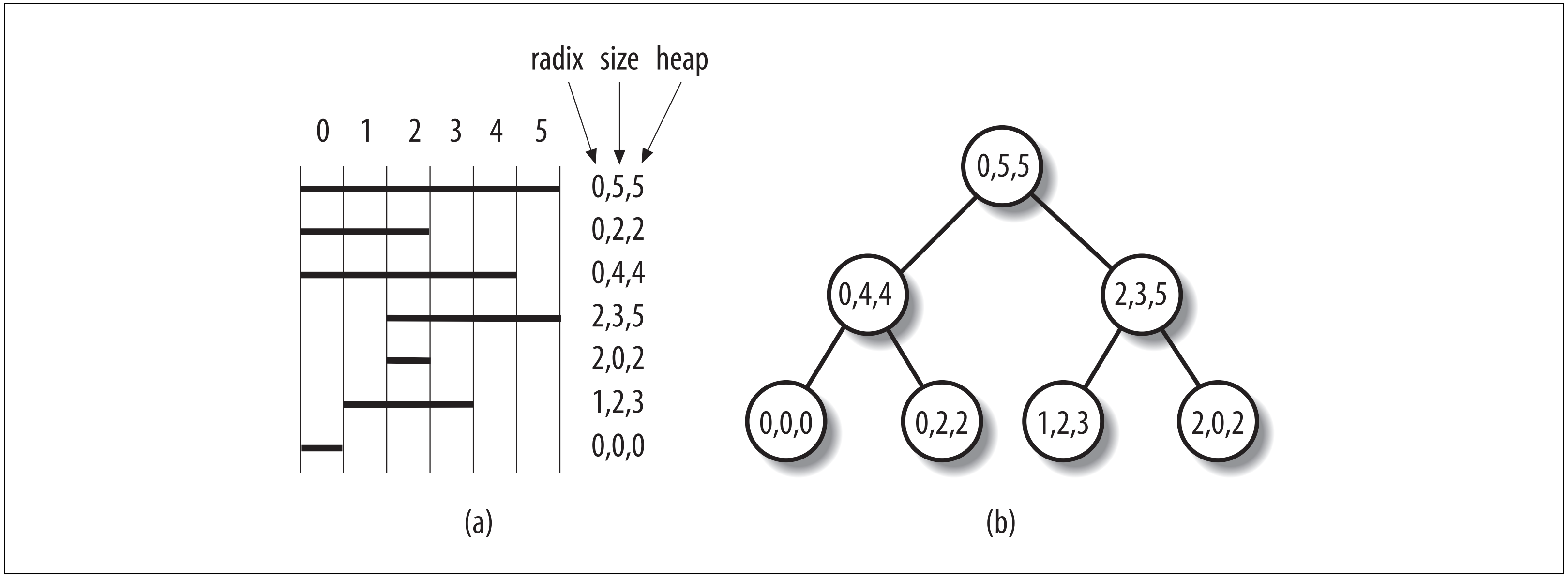


Figure 17-2. A simple example of priority search tree

- Radix – start of interval, heap = last page
- Range is exclusive, e.g., [0, 5)

How to find page 1?

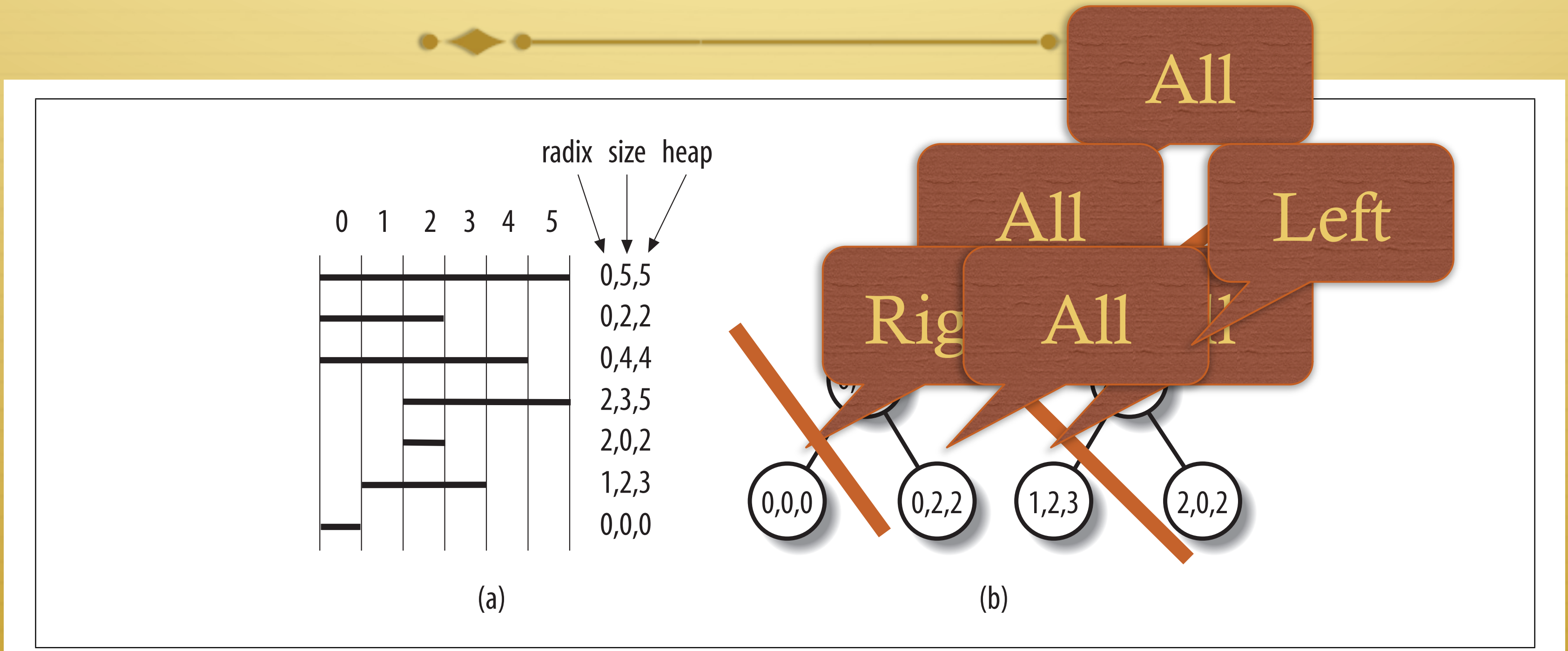


Figure 17-2. A simple example of priority search tree

- If in range: search both children
- If out of range: search only right or left child

PST + vmas

- Each node in the PST contains a list of vmas mapping that interval
  - Only one vma for unusual mappings
- So what about duplicates (ex: all programs using libc)?
  - A very long list on the (0, filesz, filesz) node
    - I.e., the root of the tree

Reverse lookup, review

- Given a page, how do I find all mappings?

Problem 2: Reclaiming

- Until there is a problem, kernel caches and processes can go wild allocating memory
- Sometimes there is a problem, and the kernel needs to reclaim physical pages for other uses
  - Low memory, hibernation, free memory below a "goal"
- Which ones to pick?
  - Goal: Minimal performance disruption on a wide range of systems (from phones to supercomputers)

Types of pages

- Unreclaimable – free pages (obviously), pages pinned in memory by a process, temporarily locked pages, pages used for certain purposes by the kernel
- Swappable – anonymous pages, tmpfs, shared IPC memory
- Syncable – cached disk data
- Discardable – unused pages in cache allocators

General principles

- Free harmless pages first
- Steal pages from user programs, especially those that haven't been used recently
- When a page is reclaimed, remove all references at once
  - Removing one reference is a waste of time
- Temporal locality: get pages that haven't been used in a while
- Laziness: Favor pages that are "cheaper" to free
  - Ex: Waiting on write back of dirty data takes time

Another view

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

LRU lists

- All pages are on one of 2 LRU lists: active or inactive
- Intuition: a page access causes it to be switched to the active list
  - A page that hasn't been accessed in a while moves to the inactive list

How to detect use?

- Tag pages with "last access" time
- Obviously, explicit kernel operations (mmap, mprotect, read, etc.) can update this
- What about when a page is mapped?
  - 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"
    - If they do get set, it is "hot"

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?

Summary

- Reverse mappings for shared:
  - Anonymous pages
  - File-mapping pages
- Basic tricks of page frame reclaiming
  - LRU lists
  - Free cheapest pages first
  - Unmap all at once
  - Etc.