CS5460/6460: Operating Systems

Lecture 23: Buffer cache

Anton Burtsev

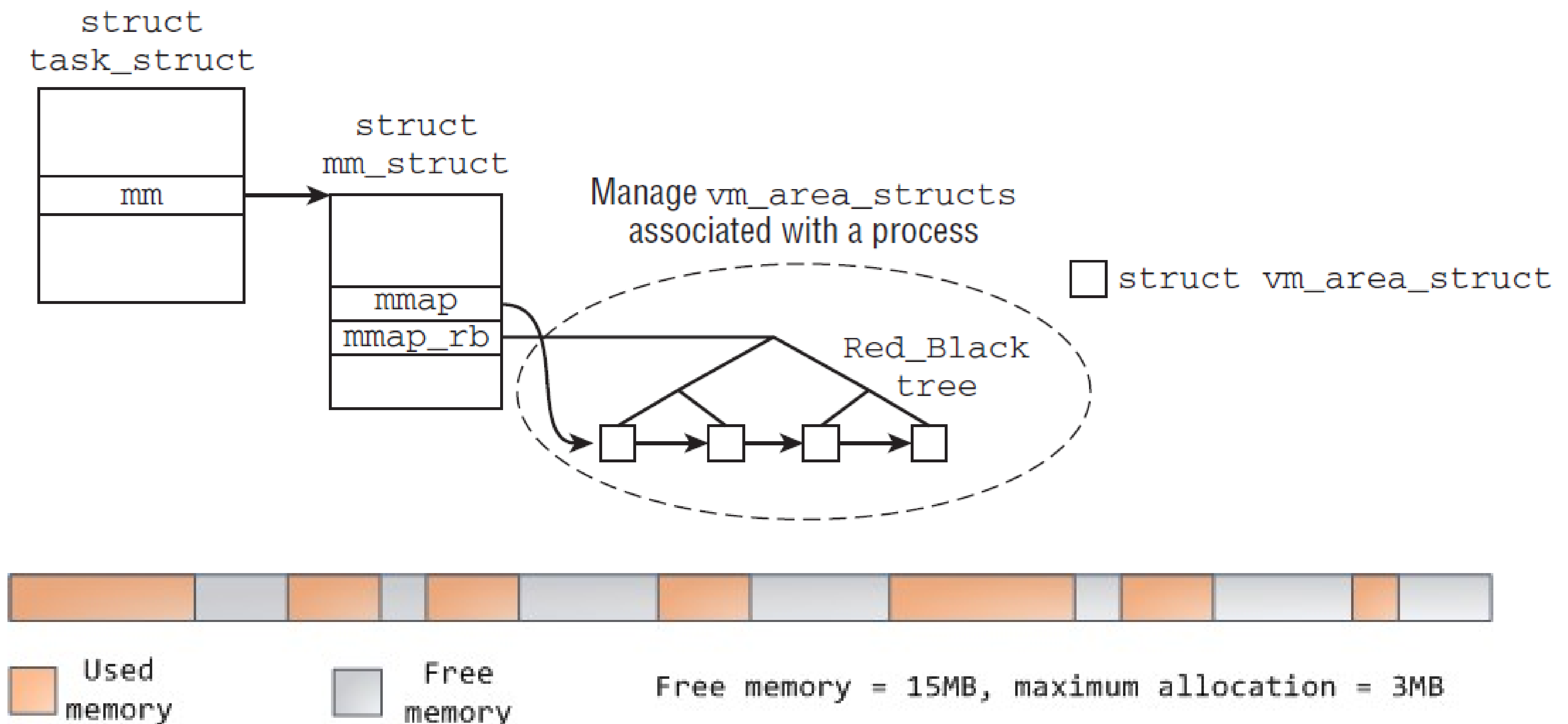
April, 2014

Recap: known mappings

- Virtual to physical mapping
  - Page tables

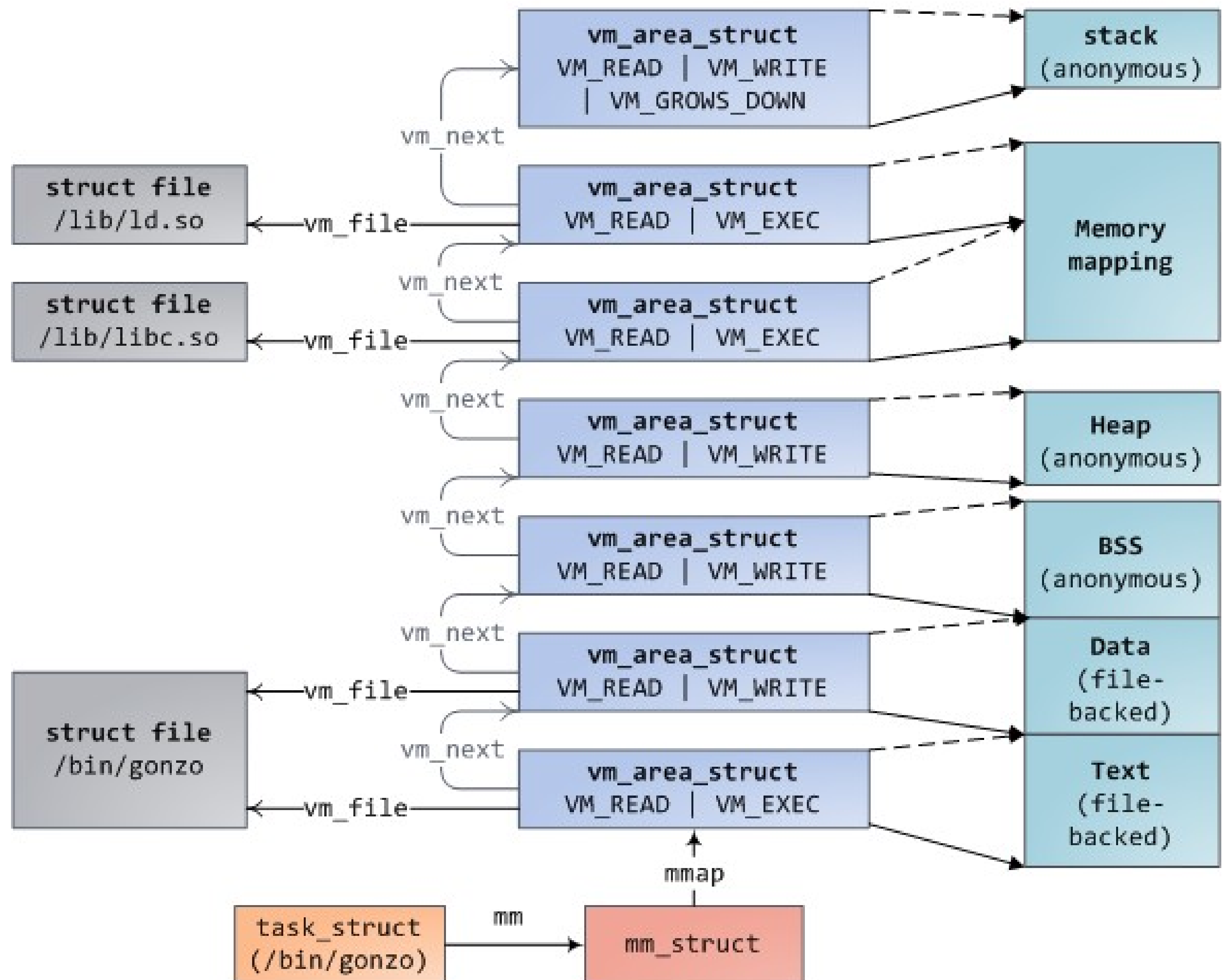
Recap: known mappings

- Virtual to memory regions mapping
  - struct mm_struct (memory map)



Two kinds of memory regions

- Anonymous
  - Not backed or associated with any data source
    - Heap, BSS, stack
  - Often shared across multiple processes
    - E.g., after fork()
- Mapped
  - Backed by a file



vm_end: first address outside virtual memory area
vm_start: first address within virtual memory area
vm_area_struct
VM_READ | VM_WRITE
| VM_GROWS_DOWN
stack
(anonymous)
vm_next
struct file
/lib/ld.so
vm_file
vm_area_struct
VM_READ | VM_EXEC
Memory
mapping
vm_next
struct file
/lib/libc.so
vm_file
vm_area_struct
VM_READ | VM_EXEC
vm_next
vm_area_struct
VM_READ | VM_WRITE
Heap
(anonymous)
vm_next
vm_area_struct
VM_READ | VM_WRITE
BSS
(anonymous)
vm_next
vm_area_struct
VM_READ | VM_WRITE
Data
(file-
backed)
struct file
/bin/gonzo
vm_file
vm_next
vm_area_struct
VM_READ | VM_EXEC
Text
(file-
backed)
vm_file
mmap
task_struct
(/bin/gonzo)
mm
mm_struct

Missing part: reverse mapping

- Connection between a page and all address spaces it is mapped into
- Motivation: ability to reclaim the page
  - Unmap it from memory
  - Save to the swap file
  - Keep track of a number of references (accesses to a page) in every address space
  - Find idle pages, and page them out

Reverse mapping

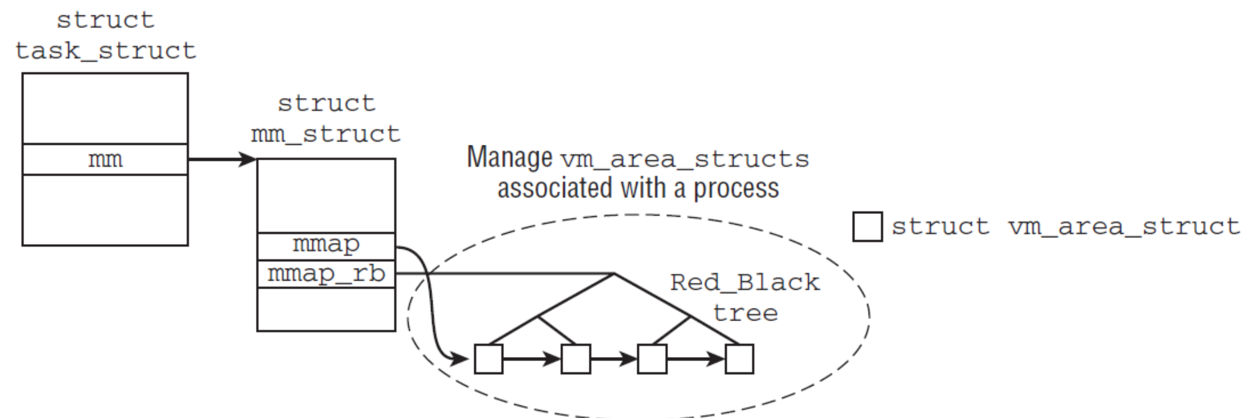
- Trivial solution:
  - For every page descriptor (struct page)
    - Keep a list of all address spaces in which it is mapped
  - High bookkeeping overhead
    - To many pages

Reverse mapping

- Insight: pages are always grouped in regions
  - Parts of memory mapped file
  - Regions of shared memory
- Instead of individual pages
  - Keep reverse mapping of those regions
  - **Remember what these are?**

Reverse mapping

- Insight: pages are always grouped in regions
  - Parts of memory mapped file
  - Regions of shared memory
- Instead of individual pages
  - Keep reverse mapping of those regions
  - **Remember what these are?**



Two kinds of regions

- Anonymous
- Mapped
  - Backed by a file

- Reverse mapping is implemented differently for them

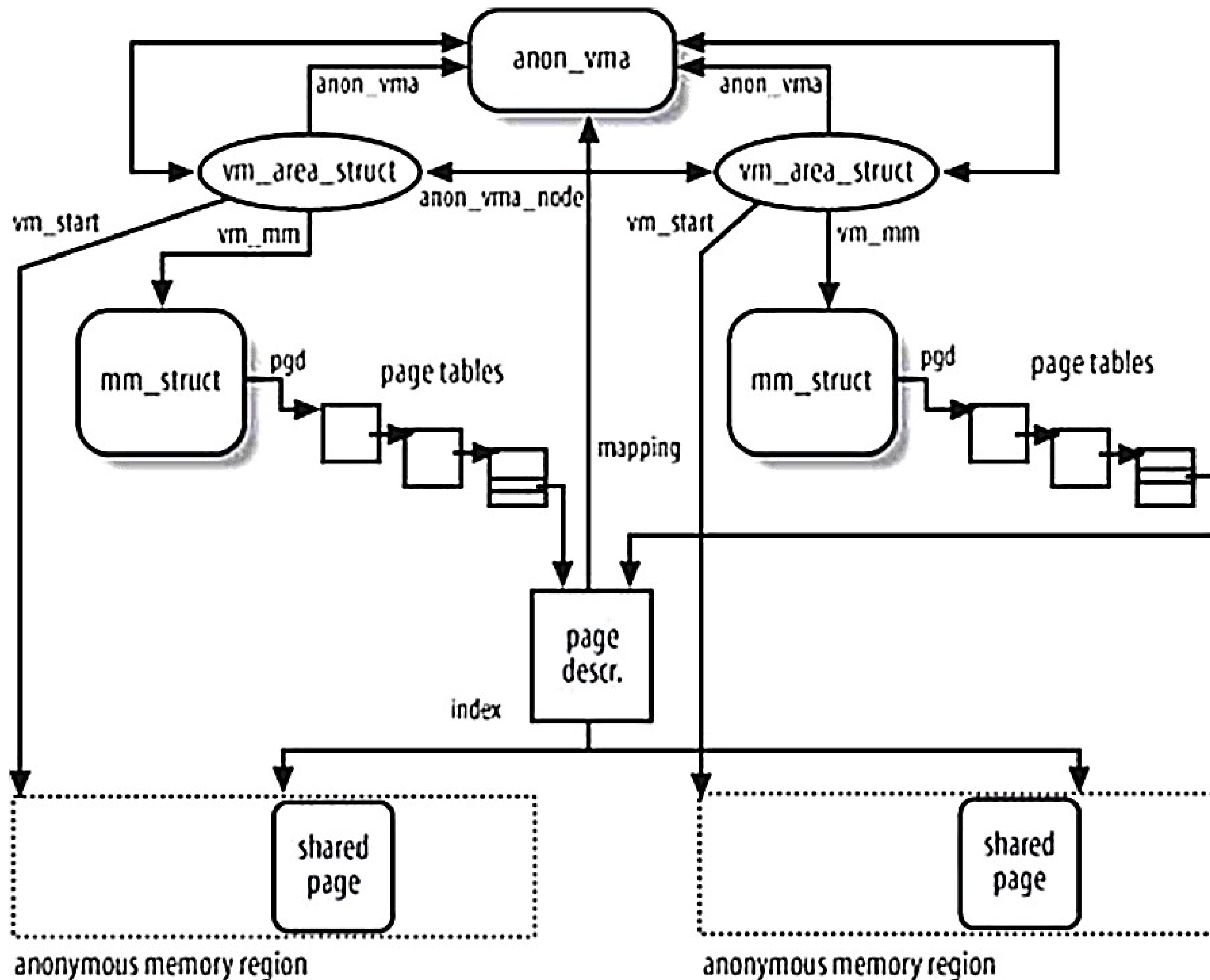
How do we know which page we deal with?

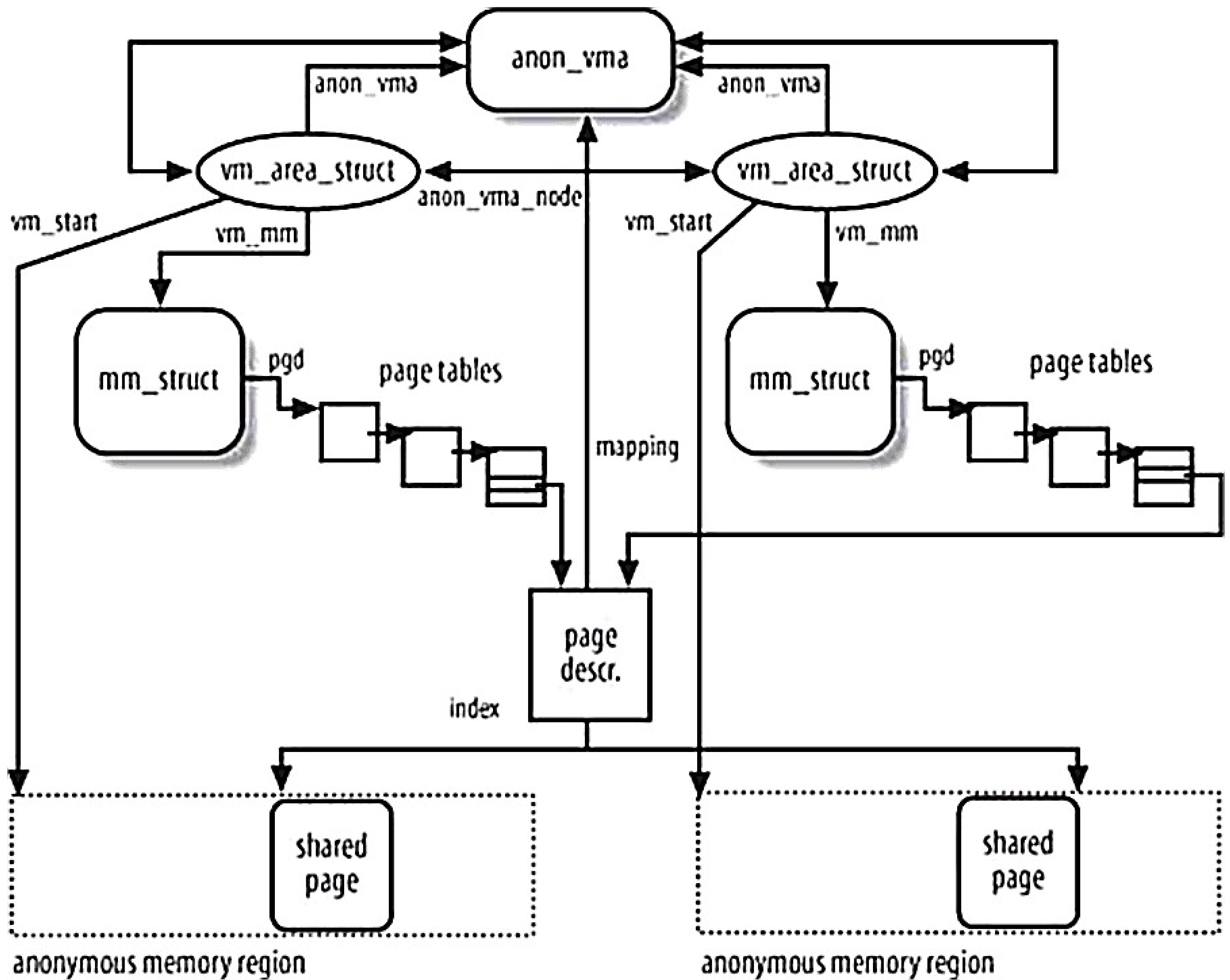
- A field in the struct page indicates type of the pointer
  - struct page.mapping
  - NULL than page belongs to the swap cache
  - Not NULL and
    - Lowest bit 1 – page is anonymous
    - Lowest bit 0 – page is mapped

A programming trick

- struct page.mapping can point to either
  - struct anon_vma
  - struct address_space
- Both structs are _always_ 4 bytes aligned
  - Lower 4 bits are always 0
  - These bits can carry some information, e.g., anonymous vs mapped

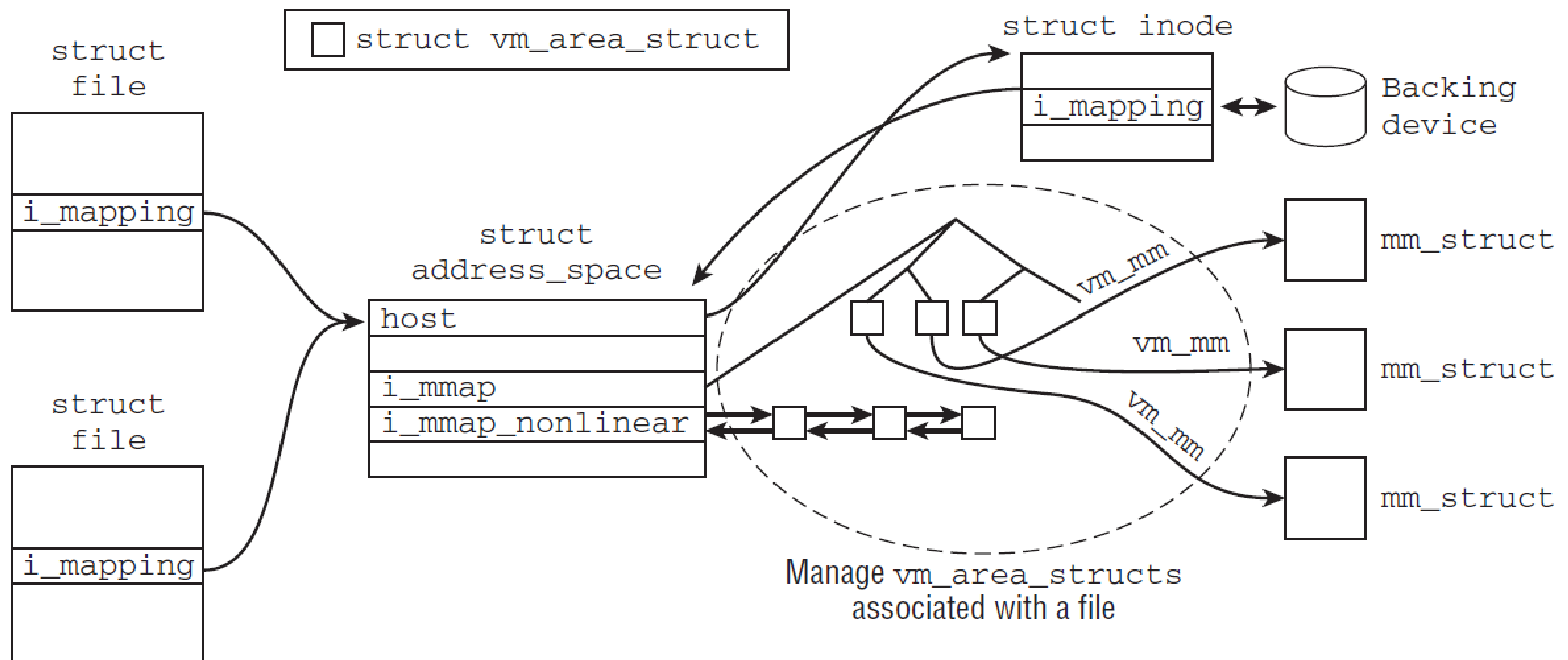
Reverse mapping of anonymous pages




anon_vma
anon_vma
anon_vma
vm_area_struct
anon_vma_node
vm_area_struct
vm_start
vm_mm
vm_start
vm_mm
mm_struct
pgd
page tables
mm_struct
pgd
page tables
mapping
page
descr.
index
shared
page
shared
page
anonymous memory region
anonymous memory region

Reverse mapping of mapped pages

- Priority search tree
  - struct page.mapping points to struct address_space



Page Cache

Buffering and caching

- Modern kernels rely on sophisticated buffering and caching mechanisms to boost I/O performance
  - Perform multiple file operations on a memory-cached copy of data
  - Cache data for subsequent accesses
  - Tolerate bursts of write I/O
- Caching is transparent to applications

User read and write requests

- **All user requests go through the cache**
- User read request
  - Check if read destination is in the cache
  - If not, new page is added to the cache
  - The data is read from disk
  - Kept in the cache until evicted
- User write request
  - Check if the page is in the cache
  - A new entry is added and filled with data to be written on disk
  - Actual I/O transfer to disk doesn't start immediately
    - Disk update is delayed for several seconds – waiting for subsequent updates

Heart of buffer cache: address space

- The owner of a page in the page cache is a file
  - struct address_space pointer is embedded in struct inode.i_mapping object
  - Each inode points to a set of pages caching its data

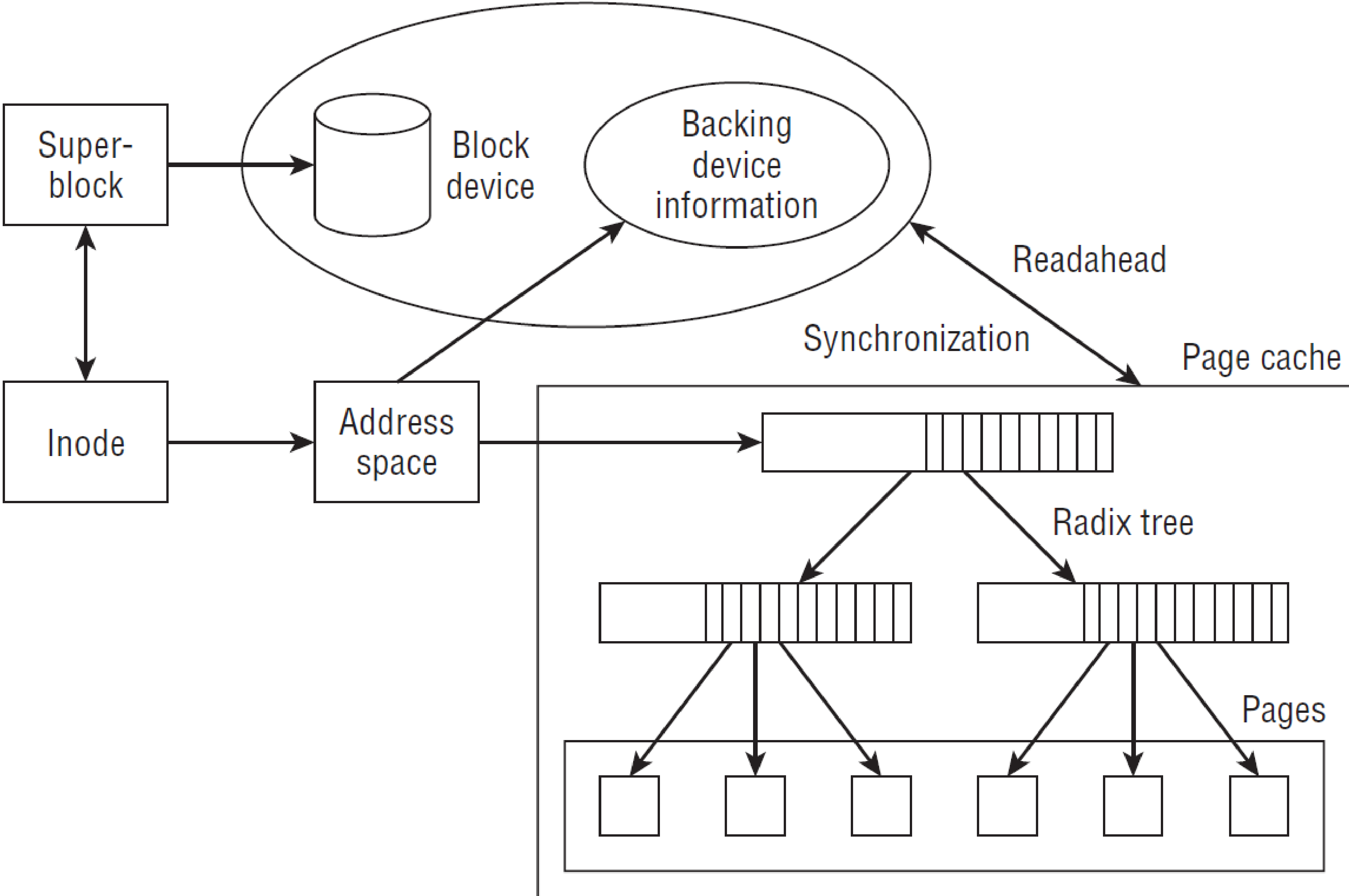
How to locate a page in buffer cache

- E.g., trying to perform a user read, how to check that the page is already in memory?

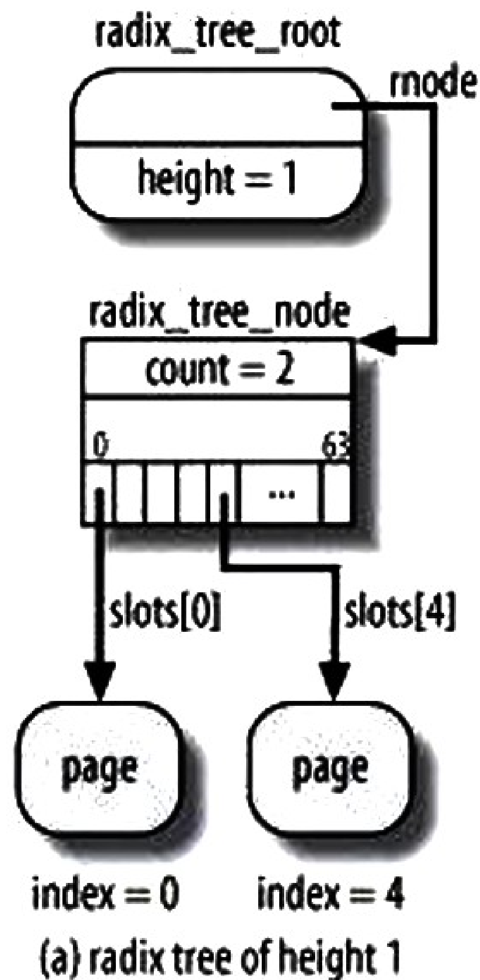
Radix tree

- Files can be large
  - Given a position in a file, we want to quickly find whether a corresponding page is in memory
  - Linear scan can take too long
  - Radix tree is a good option

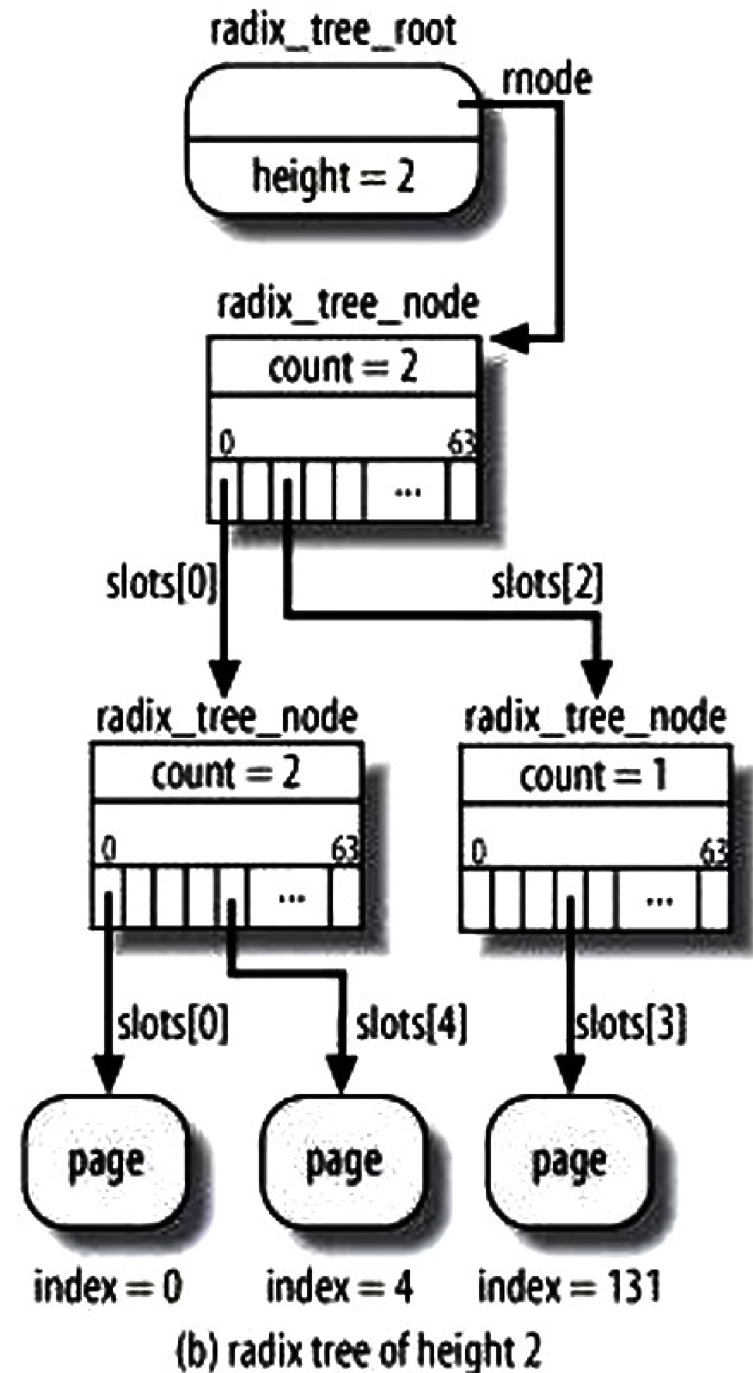
Page cache



Organization of the radix tree



(a) radix tree of height 1



(b) radix tree of height 2

Variable height

- 1 – max index (64) – max file size 256 KB
- 2 – max index (4095) – max file size 16MB
- 3 – max index (262 143) – max file size 1GB
- 4 – … – max file size 64GB
- 5 – … – max file size 4TB
- 6 – … – max file size 16TB

Radix tree lookup

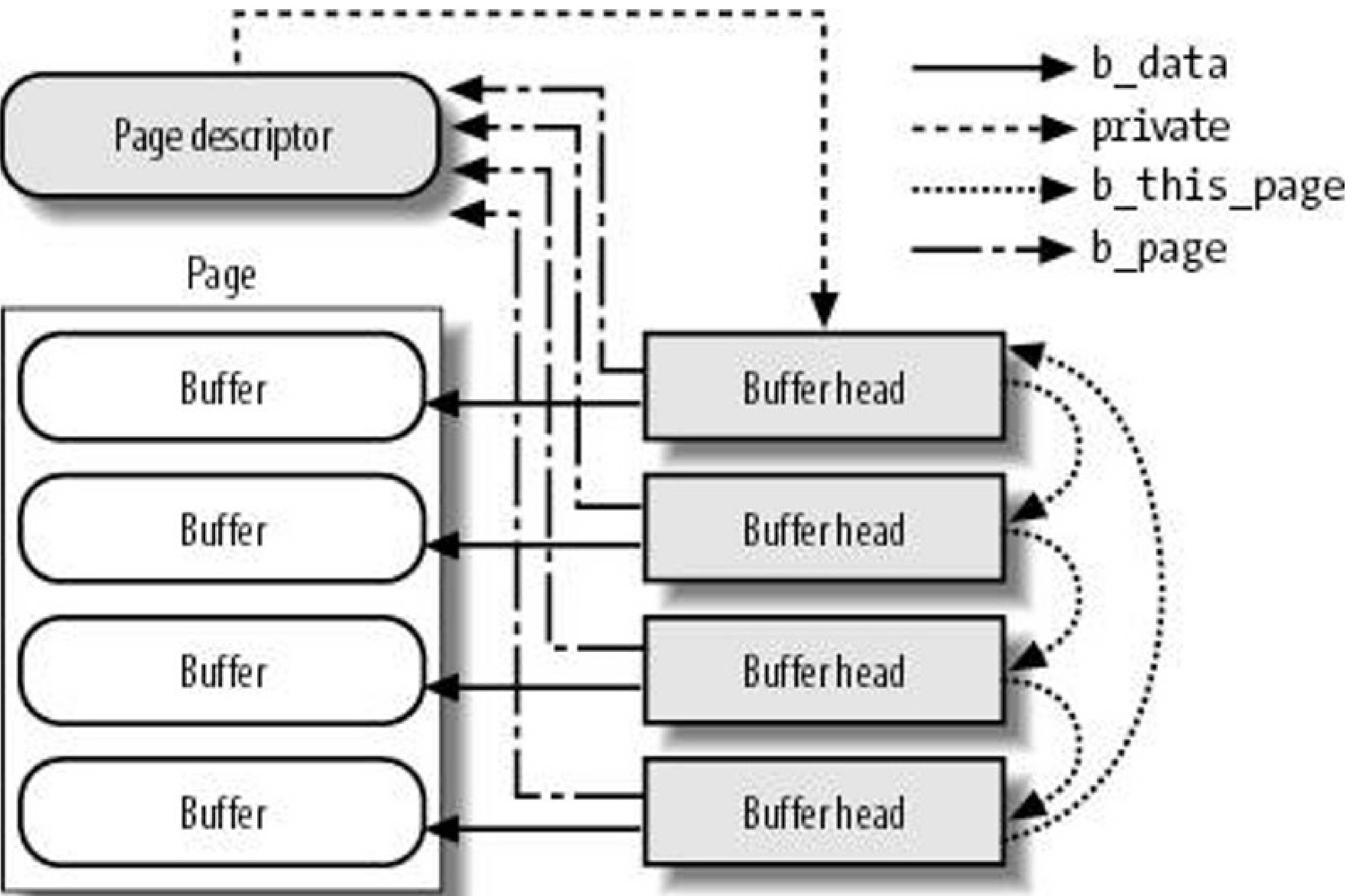
- Height 1:
  - 6 bits of index encode one of 64 pages
- Height 2:
  - 12 bits of index are meaningful
  - Top 6 bits choose the node on the second level
  - Lower 6 bits choose the page

Buffer Cache

Buffer cache

- Historically block devices performed I/O in blocks
  - The kernel had a separate buffer cache for I/O blocks
  - Today I/O blocks are kept in the page cache
- Buffer cache is a cache of objects which are not handled in pages, but in blocks
  - Different block devices can have different block sizes
  - Cached blocks are kept in the page cache

Block buffers and buffer heads



Searching blocks in the page cache

- Input: block number
- Idea: convert from block numbers to pages
  - Remember you can lookup pages in the page cache
- Each page contains n blocks
  - Conversion is trivial
  - The page number is block number / n
  - struct page.private keeps a pointer to buffer head

Cache synchronization

pdflush

- Set of background kernel threads which find dirty pages and flush them to disk
  - Invoked periodically
  - Number of threads changes dynamically
    - Adapts to current cache pressure

pdflush

- Each thread starts with a task to find X number of dirty pages and flush them to disk
  - Radix tree keeps a dirty flag for each subtree that has at least one dirty page inside
    - Helps to reduce unneeded radix tree scanning

Swapping and page reclamation

What pages can be swapped to disk?

Swappable pages

- Anonymous pages
  - No backing store, need to be swapped
- Private mappings of file section (PRIVATE flag)
  - I guess these are data sections of a file (can be wrong)
- Heap pages
  - Malloc'ed on top of brk() or mmap(ANONYMOUS)
- Shared memory pages
  - Shared across multiple processes for interprocess communication

What to swap?

- Kernel needs to detect the working set of a program
  - Pages which are frequently accessed
  - Maintain some sort of a LRU list

Naive LRU implementation

- For example, a FIFO queue of pages
  - A new page is added to the front
  - If the queue is full, the page from the back is swapped out
- It's ok, at least a page has a window of time to remain in memory
  - But of course no information about how many times the page was accessed

Second chance LRU

- Same FIFO queue of pages
  - A new page is added to the front
  - If the queue is full, the page from the back becomes a victim
  - If the accessed bit is set it survives – gets re-added to the front
  - If not, it is swapped out

Swap area

- Dedicated partition or a fixed-size file
  - Each divided in slots (size of a page frame)
  - Essentially an array on disk
- Kernel uses a bitmap to track free/used slots
- When the page is swapped out its PTE is updated to keep its position in the swap area
  - So later it can be retrieved from the swap area during the page fault

Conclusion

- We know how the kernel manages
  - Process memory
  - Memory mapped files
  - Page and buffer caches
  - Swap

Thank you!