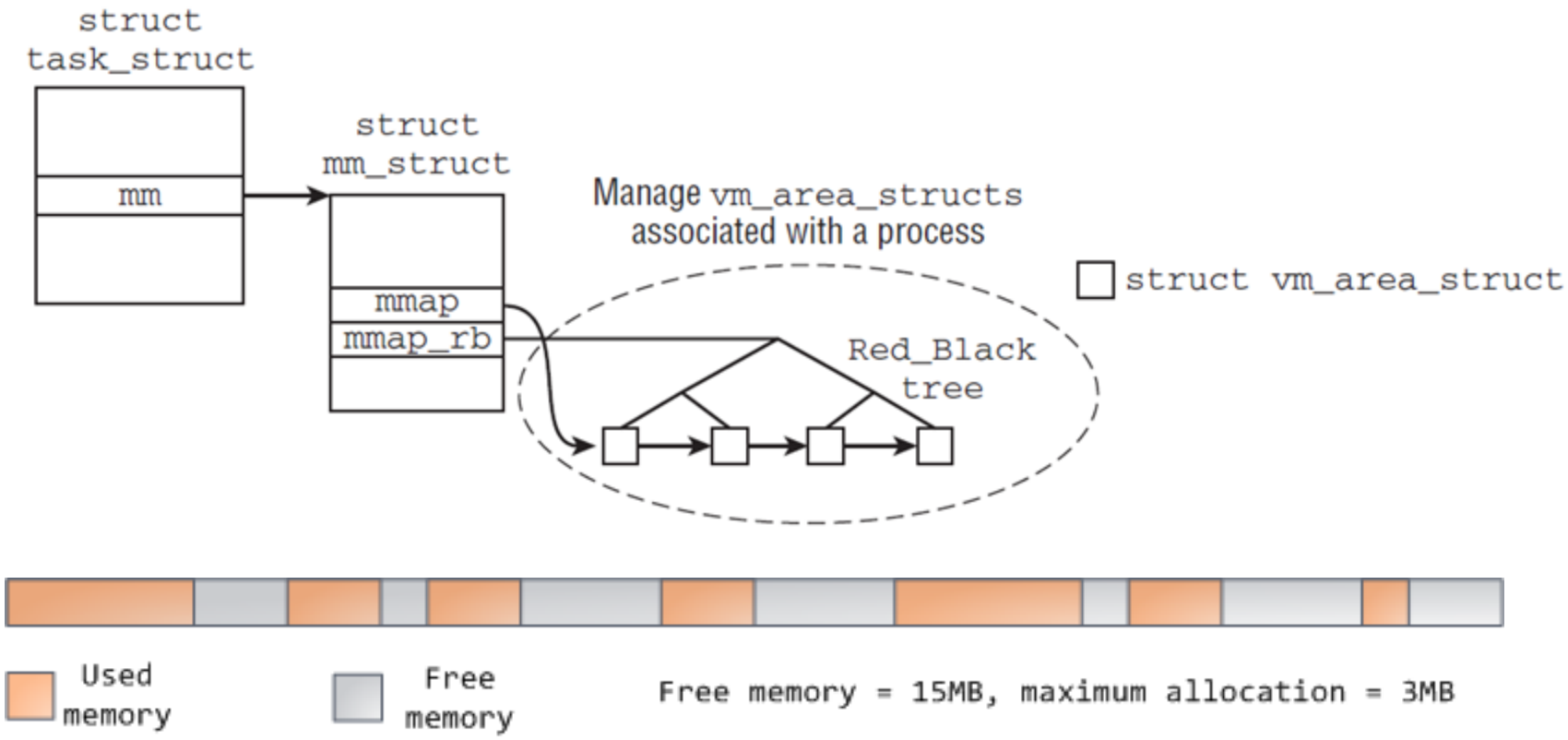
Cs6465: Advanced Operating System Implementation Lecture 03 – Buffer Cache

Anton Burtsev

September, 2024

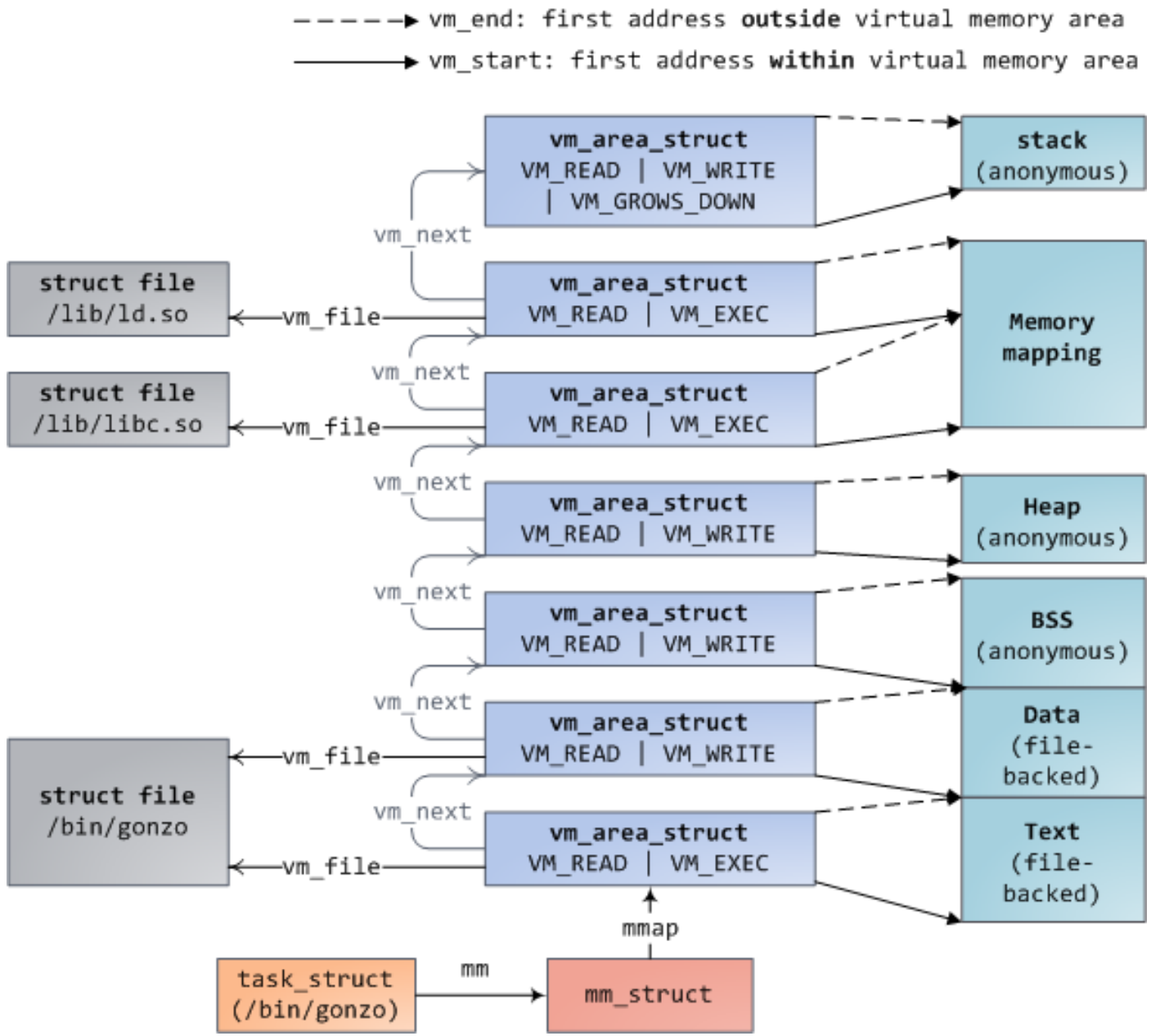
Recap

- 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

Buffering and caching

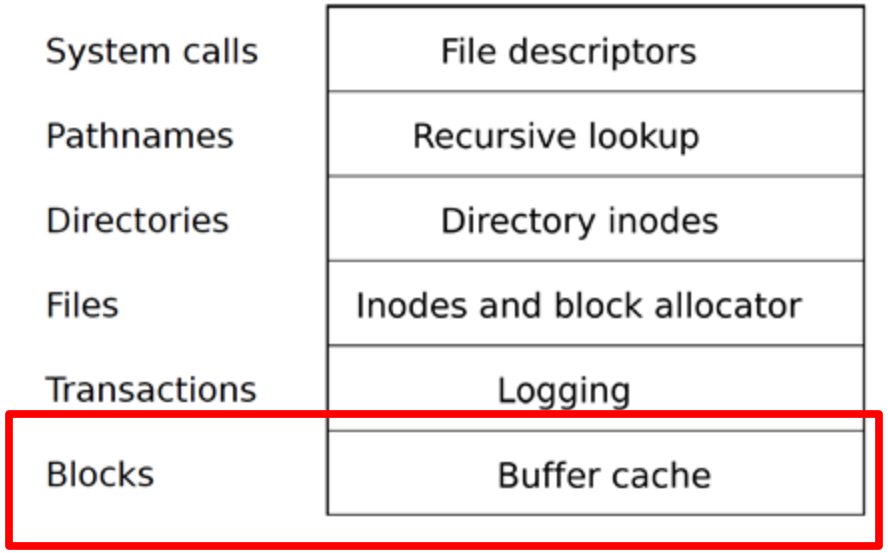
- Modern kernels rely on sophisticated buffering and caching mechanisms to boost I/O performance
- Perform multiple file operations on a memory-cached copy of data
- Cache data for subsequent accesses
- Tolerate bursts of write I/O
- Caching is transparent to applications

Buffering and caching

- 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

Recap: buffer cache in xv6

Block layer



- Read and write data
  - From a block device
  - Into a buffer cache
- Synchronize across multiple readers and writers

Buffer cache

- Two goals:
- Synchronization:
  - Only one copy of a data block exist in the kernel
  - Only one writer updates this copy at a time
- Caching
  - Frequently used copies are cached for efficient reads and writes

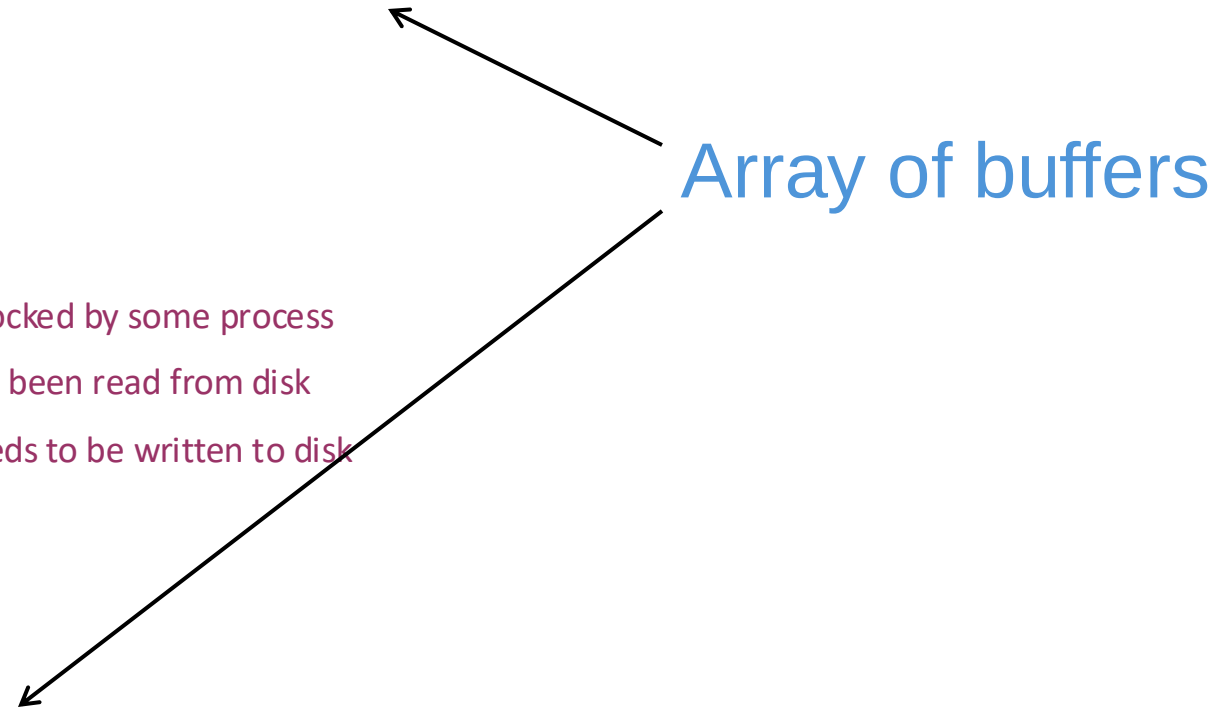
Buffer cache

```
3750 struct buf {
3751   int flags;
3752   uint dev;
3753   uint blockno;
3754   struct buf *prev; // LRU cache list
3755   struct buf *next;
3756   struct buf *qnext; // disk queue
3757   uchar data[BSIZE];
3758 };
3759 #define B_BUSY  0x1  // buffer is locked by some process
3760 #define B_VALID 0x2  // buffer has been read from disk
3761 #define B_DIRTY 0x4  // buffer needs to be written to disk
```

```
4329 struct {
4330   struct spinlock lock;
4331   struct buf buf[NBUF];
4332
4333   // Linked list of all buffers, through prev/next.
4334   // head.next is most recently used.
4335   struct buf head;
4336 } bcache;
```

Buffer cache

```
3750 struct buf {
3751   int flags;
3752   uint dev;
3753   uint blockno;
3754   struct buf *prev; // LRU cache list
3755   struct buf *next;
3756   struct buf *qnext; // disk queue
3757   uchar data[BSIZE];
3758 };
3759 #define B_BUSY  0x1  // buffer is locked by some process
3760 #define B_VALID 0x2  // buffer has been read from disk
3761 #define B_DIRTY 0x4  // buffer needs to be written to disk
```



Array of buffers

```
4329 struct {
4330   struct spinlock lock;
4331   struct buf buf[NBUF];
4332
4333   // Linked list of all buffers, through prev/next.
4334   // head.next is most recently used.
4335   struct buf head;
4336 } bcache;
```

Buffer cache

```
3750 struct buf {
3751   int flags;
3752   uint dev;
3753   uint blockno;
3754   struct buf *prev; // LRU cache list
3755   struct buf *next;
3756   struct buf *qnext; // disk queue
3757   uchar data[BSIZE];
3758 };
3759 #define B_BUSY  0x1  // buffer is locked by some process
3760 #define B_VALID 0x2  // buffer has been read from disk
3761 #define B_DIRTY 0x4  // buffer needs to be written to disk
```

Cached data (512 bytes)

```
4329 struct {
4330   struct spinlock lock;
4331   struct buf buf[NBUF];
4332
4333   // Linked list of all buffers, through prev/next.
4334   // head.next is most recently used.
4335   struct buf head;
4336 } bcache;
```

Buffer cache

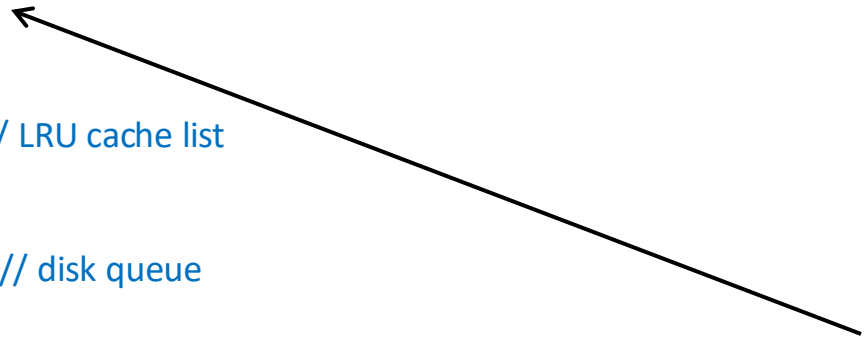
```
3750 struct buf {
3751   int flags;
3752   uint dev;
3753   uint blockno;
3754   struct buf *prev; // LRU cache list
3755   struct buf *next;
3756   struct buf *qnext; // disk queue
3757   uchar data[BSIZE];
3758 };
3759 #define B_BUSY  0x1  // buffer is locked by some process
3760 #define B_VALID 0x2  // buffer has been read from disk
3761 #define B_DIRTY 0x4  // buffer needs to be written to disk
```

Flags

```
4329 struct {
4330   struct spinlock lock;
4331   struct buf buf[NBUF];
4332
4333   // Linked list of all buffers, through prev/next.
4334   // head.next is most recently used.
4335   struct buf head;
4336 } bcache;
```

```
3750 struct buf {
3751   int flags;
3752   uint dev;
3753   uint blockno;
3754   struct buf *prev; // LRU cache list
3755   struct buf *next;
3756   struct buf *qnext; // disk queue
3757   uchar data[BSIZE];
3758 };
3759 #define B_BUSY  0x1  // buffer is locked by some process
3760 #define B_VALID 0x2  // buffer has been read from disk
3761 #define B_DIRTY 0x4  // buffer needs to be written to disk


4329 struct {
4330   struct spinlock lock;
4331   struct buf buf[NBUF];
4332
4333   // Linked list of all buffers, through prev/next.
4334   // head.next is most recently used.
4335   struct buf head;
4336 } bcache;
```



Buffer cache

- Device
- We might have multiple disks

- bread() and bwrite() - obtain a copy for reading or writing
  - Owned until brelse()
  - Locking with a flag (B_BUSY)
- Other threads will be blocked and wait until brelse()

```
4570 // Copy committed blocks from log to their home location
4571 static void
4572 install_trans(void)
4573 {
4574   int tail;
4575
4576   for (tail = 0; tail < log.lh.n; tail++) {
4577     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
4578     struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
4579     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
4580     bwrite(dbuf);  // write dst to disk
4581     brelse(lbuf);
4582     brelse(dbuf);
4583   }
4584 }
```

Example

```
4401 struct buf*
4402 bread(uint dev, uint sector)
4403 {
4404   struct buf *b;
4405
4406   b = bget(dev, sector);
4407   if(!(b->flags & B_VALID)) {
4408     iderw(b);
4409   }
4410   return b;
4411 }
4415 bwrite(struct buf *b)
4416 {
4417   if((b->flags & B_BUSY) == 0)
4418     panic("bwrite");
4419   b->flags |= B_DIRTY;
4420   iderw(b);
4421 }
```

Block read and write operations

```
4365 static struct buf*
4366 bget(uint dev, uint blockno)
4367 {
4368   struct buf *b;
4369
4370   acquire(&bcache.lock);
4371
4372  loop:
4373   // Is the block already cached?
4374   for(b = bcache.head.next; b != &bcache.head; b = b->next){
4375     if(b->dev == dev && b->blockno == blockno){
4376       if(!(b->flags & B_BUSY)){
4377         b->flags |= B_BUSY;
4378         release(&bcache.lock);
4379         return b;
4380       }
4381       sleep(b, &bcache.lock);
4382       goto loop;
4383     }
4384
```

Getting a block from a buffer cache (part 1)

```
4385
4386  // Not cached; recycle some non−busy and clean buffer.
4387  // "clean" because B_DIRTY and !B_BUSY means log.c
4388  // hasn’t yet committed the changes to the buffer.
4389  for(b = bcache.head.prev; b != &bcache.head; b = b−>prev){
4390    if((b−>flags & B_BUSY)== 0 && (b−>flags & B_DIRTY)== 0){
4391      b−>dev = dev;
4392      b−>blockno = blockno;
4393      b−>flags = B_BUSY;
4394      release(&bcache.lock);
4395      return b;
4396    }
4397  }
4398  panic("bget: no buffers");
4399 }
```

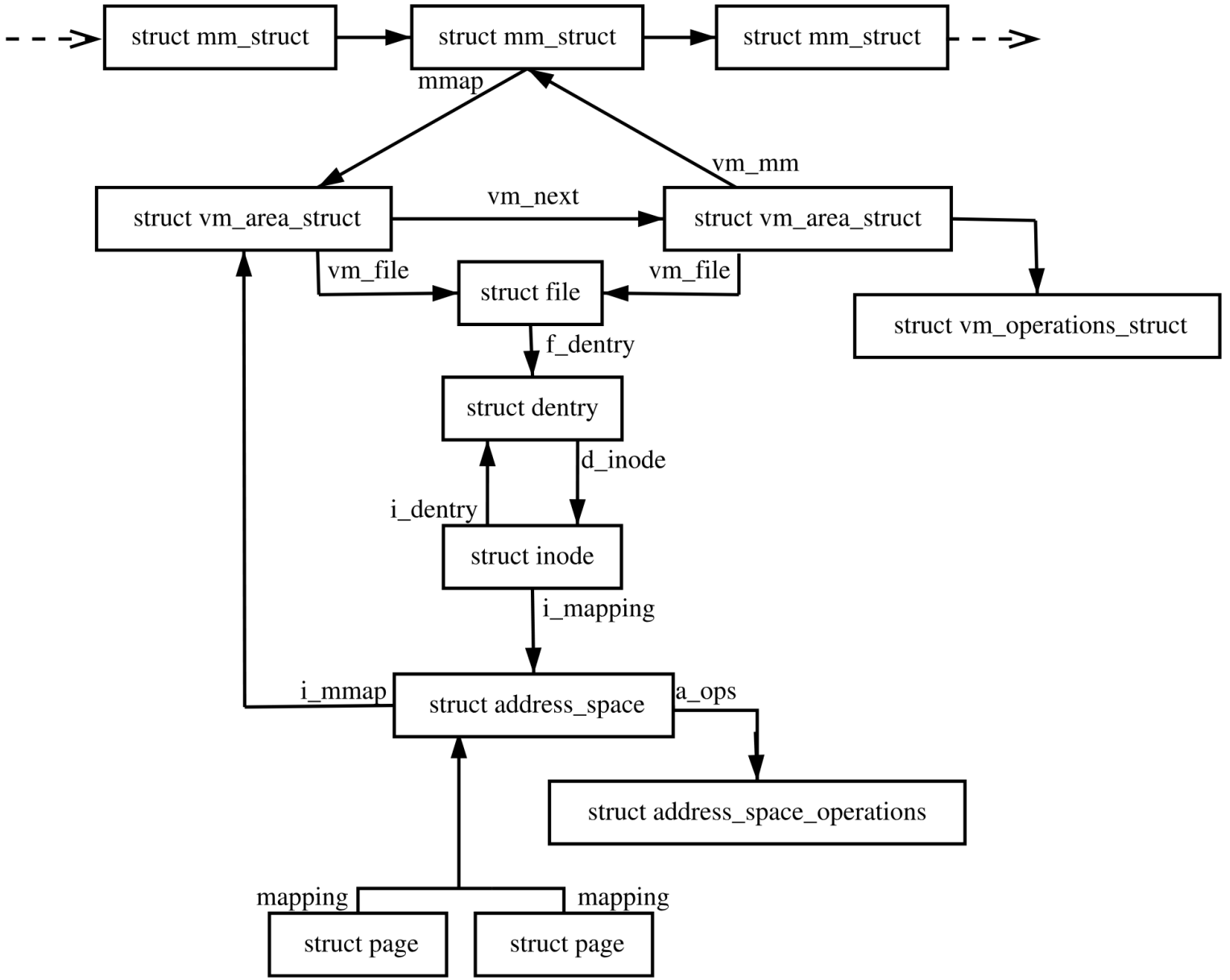
Getting a block from a buffer cache (part 2)

Page cache (Linux)

Buffer cache in Linux

- 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

Core of the buffer cache: address space



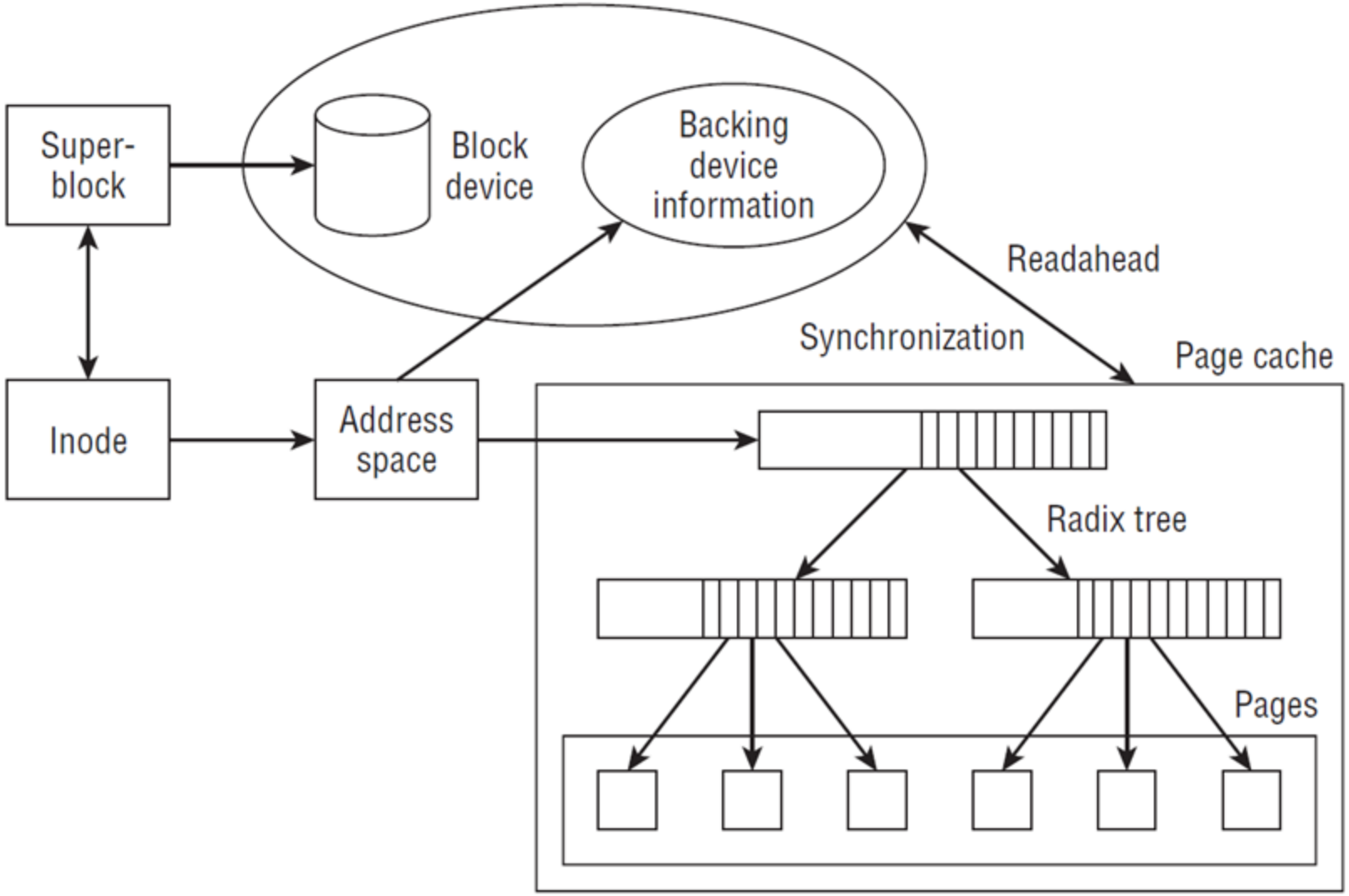
How to locate a page in buffer cache

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

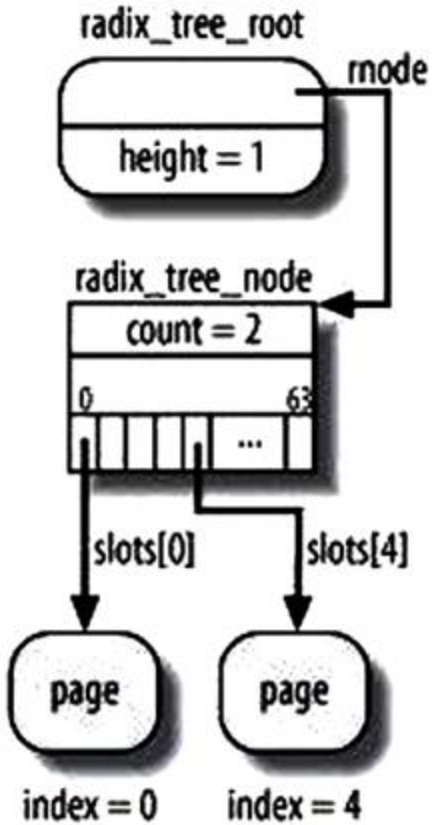
Radix tree

- 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

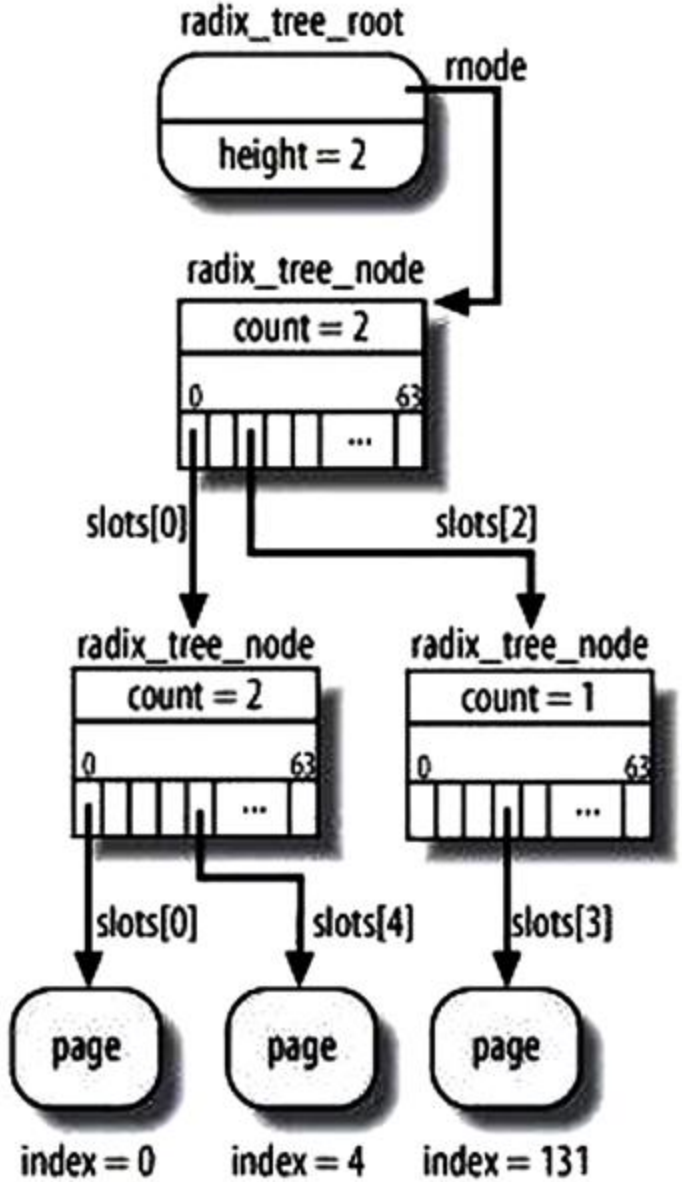
Page cache


Super-
block
Block
device
Backing
device
information
Readahead
Synchronization
Page cache
Inode
Address
space
Radix tree
Pages

Organization of the radix tree



(a) radix tree of height 1



(b) radix tree of height 2

Radix tree lookup

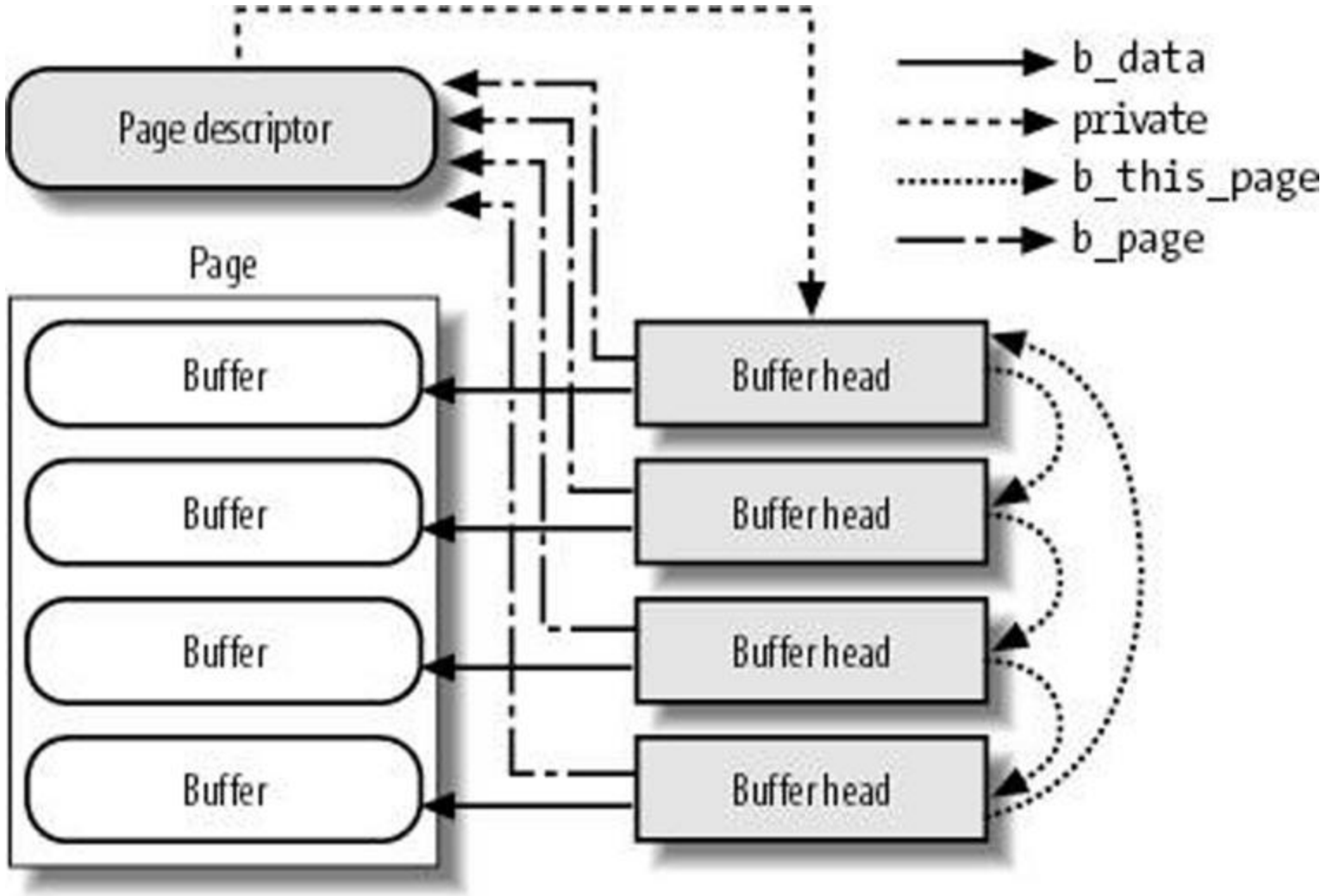
- Height 1:
  - 6 bits of index encode one of 64 pages
- Height 2:
  - 12 bits of index are meaningful
- Top 6 bits choose the node on the second level
- Lower 6 bits choose the page

Buffer Cache

- 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 buffer 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