



Disk Caches in Linux 2.6

Hao-Ran Liu



The purpose of Disk Caches

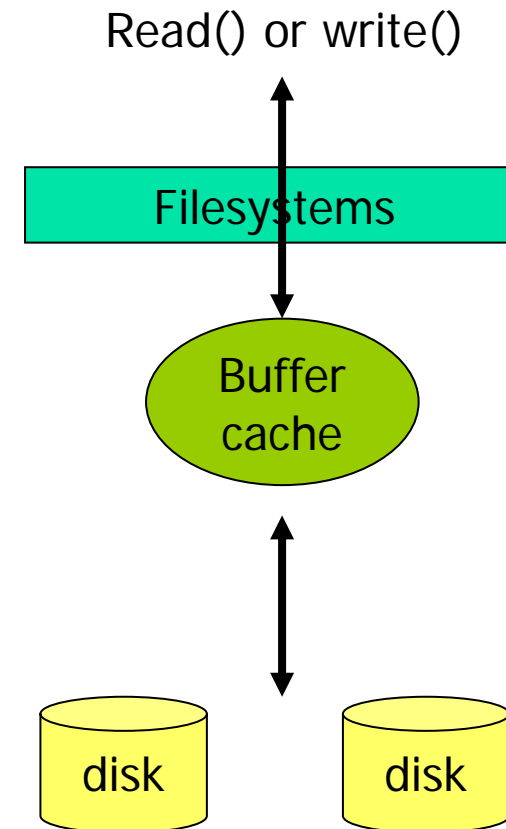
- To improve system performance by saving data in disk caches to reduce disk accesses
- Several kinds of targets can be cached in Linux:

Target	Property	Cache
Disk blocks	Physical blocks	Buffer cache
Inode / Directory entries	dentry objects	dentry cache
File blocks	Logical blocks	Page Cache

We will not cover dentry cache in this talk!

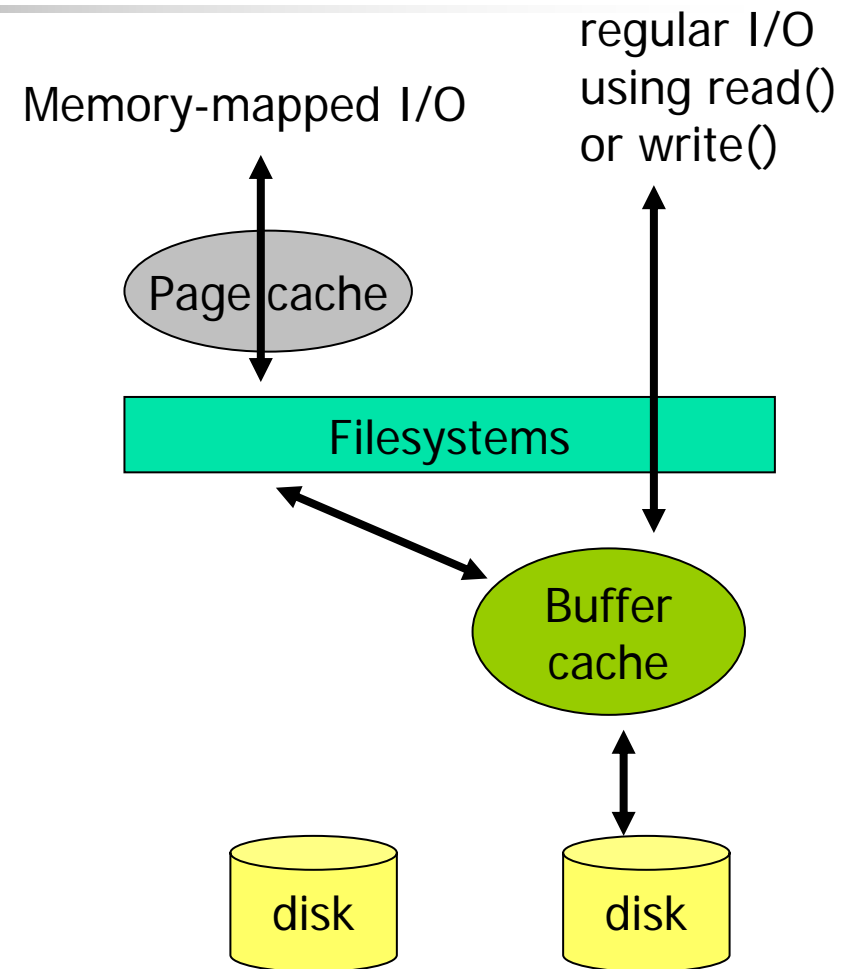
Traditional design of the disk cache

- Buffer Cache
 - Cache disk blocks
 - Index key: (dev_t, blk_no)
- Disadvantage
 - Cannot cache NFS
 - Disintegrate from memory paging



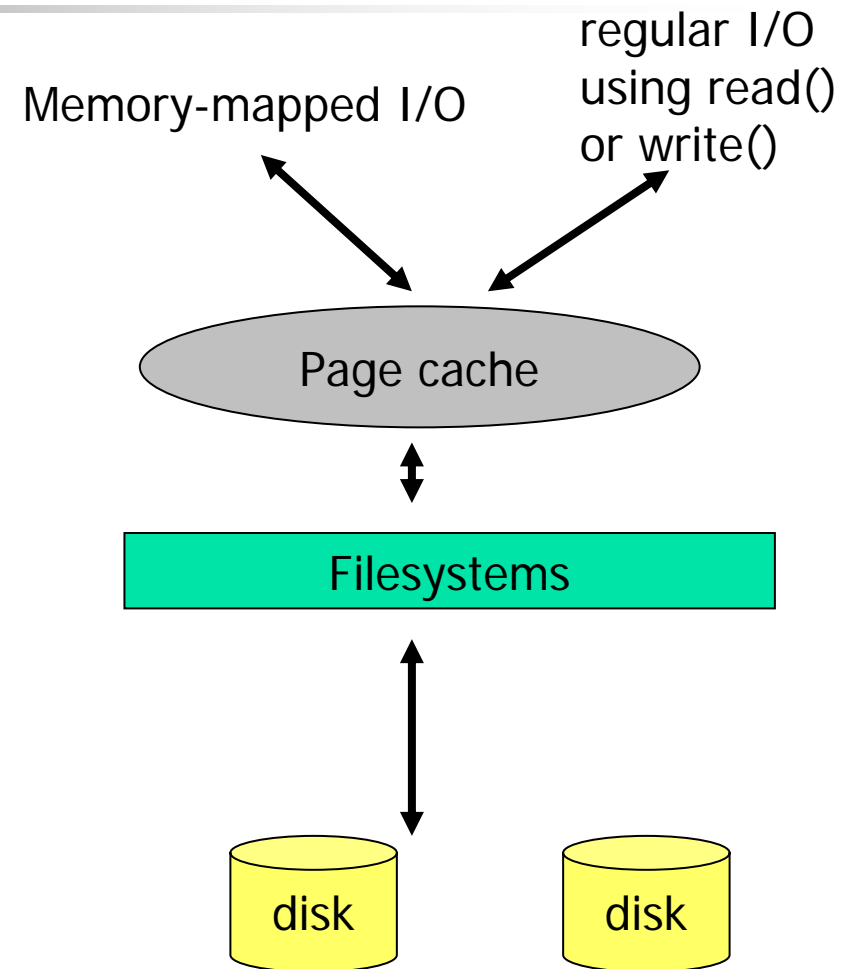
Hybrid design of the disk caches

- Page cache
 - For memory-mapped I/O
 - Cache regular file or block device file
 - Index key: (`address_space`, `page_offset`)
- Disadvantage
 - Data synchronization between the two caches
 - Waste memory



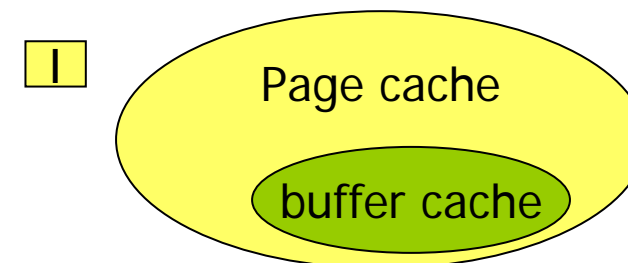
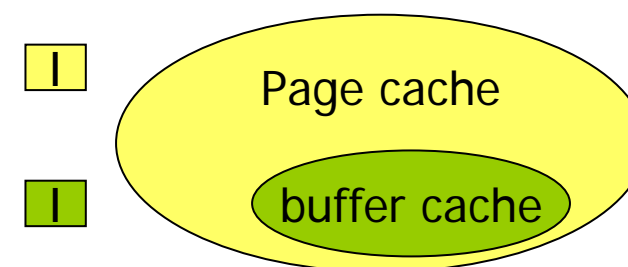
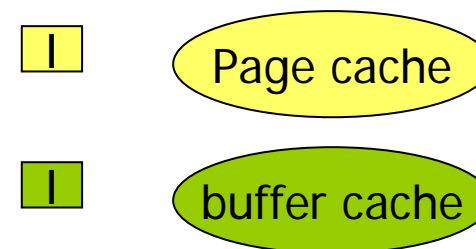
Modern design of the disk cache

- A unified page cache



Evolution of disk caches in Linux

- 2.0 – 2.2
 - Buffer cache and page cache
 - Data synchronization is needed
- 2.4
 - Buffer cache is a subset of page cache, but still maintains its own hash table
 - No data synchronization
- 2.6
 - Buffer cache hash table removed
 - Buffer cache API built on top of page cache





Use of the buffer cache and page cache

Kernel function	System call	Cache	I/O operation
__bread()	None	Buffer	Read an ext2 superblock
__bread()	None	Buffer	Read an ext2 inode
generic_file_read()	read()	Page	Read an ext2 regular file
generic_file_write()	write()	Page	Write an ext2 regular file
generic_file_read()	read()	Page	Read a block device file
generic_file_write()	write()	Page	Write a block device file
filemap_nopage()	None	Page	Access a memory-mapped file
swap_readpage()	None	Page	Read a swapped-out page
swap_writpage()	None	Page	Write a swapped-out page



How to integrate the two cache?

- Enabling keys are:
 - address_space
 - To identify buffers and pages in the page cache
 - Buffer pages
 - Buffers in a page are mapped to **contiguous** blocks on the device
 - Block device file systems
 - Pseudo i node and address_space for block device

Design Concepts of address_space

- To identify a page in the page cache:
 - `address_space + page_offset` within the file
- To establish the relationship between pages and methods that operate on the pages
- It can represent a regular file, a block device file, or a swap space in the page cache
- A separation between the "*control plane*" and the "*data plane*", to use a networking analogy*.
 - `address_space`: **data** related stuff
 - `i node`: **control**/metadata/security stuff

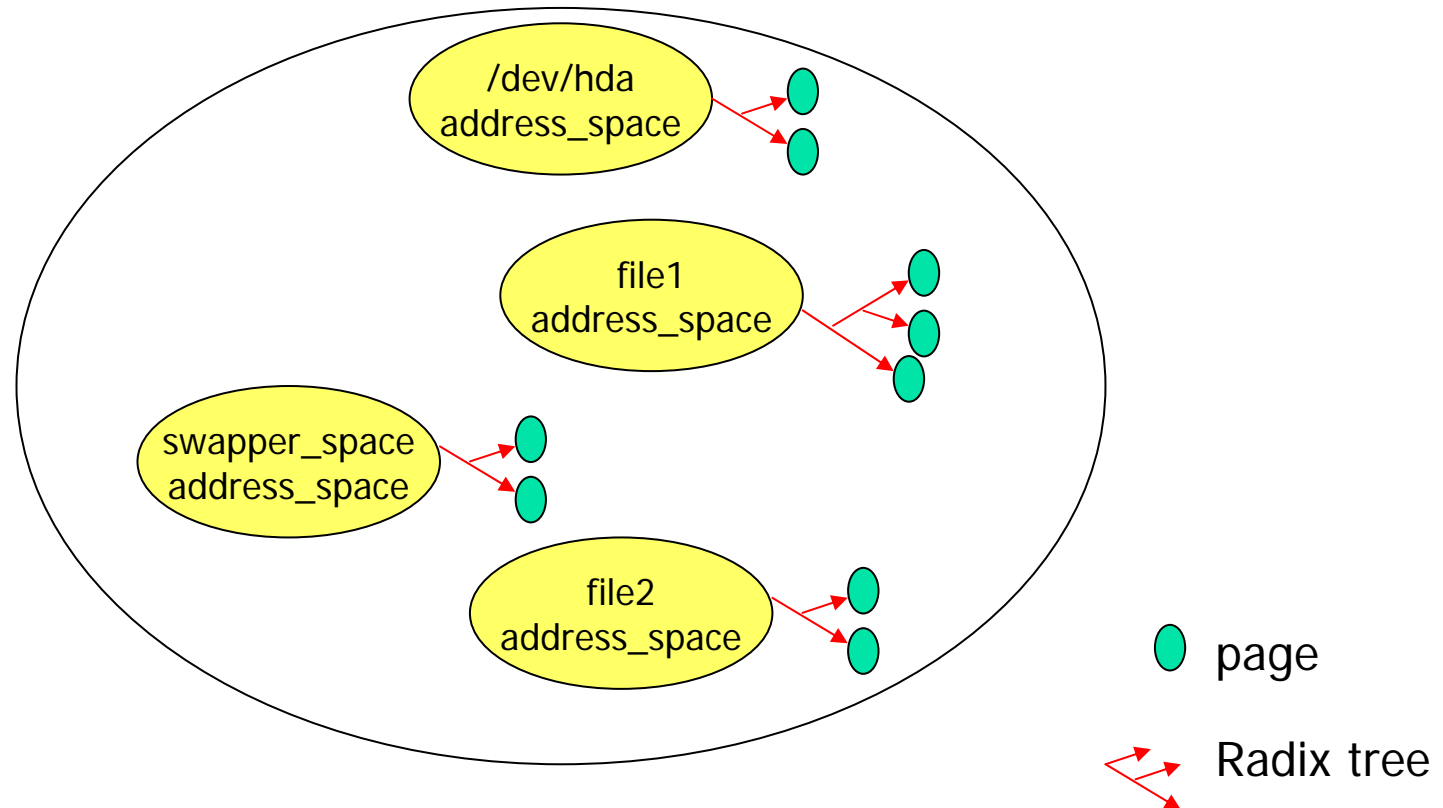
* Quoted from a mail by Andrew Morton in the Linux-MM mailing list

The fields of the address_space object

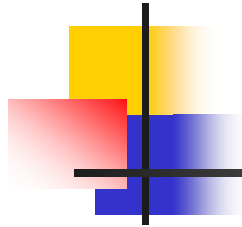
Type	Field	Description
struct inode *	host	owner: inode or block device's inode
struct radix_tree_root	page_tree	radix tree of all pages of the owner
spinlock_t	tree_lock	lock protecting page_tree
struct prio_tree_root	i_mmap	tree of private and shared mappings (VMA)
spinlock_t	i_mmap_lock	lock protecting i_mmap
unsigned long	nrpages	total number of owner's pages
struct address_space_operations *	a_ops	methods that operate on owner's pages
unsigned long	flags	memory allocator flags for owner's pages
struct backing_dev_info *	backing_dev_info	readahead information and congestion state of the backing device
spinlock_t	private_lock	lock protecting the private_list of other mappings which have listed buffers from this mapping onto themselves
struct list_head	private_list	a list of buffers for the owning inode
struct address_space *	assoc_mapping	address_space of the backing device

The scenario of the page cache

Page cache



Page cache related fields in struct page



Type	Name	Description
page_flag_t	flags	The status of the page. When the page is in swap cache, PG_swapcache flag is set
atomic_t	_count	The reference count to the page. When the page is added to the page cache, add 1 to this counter.
unsigned long	private	Mapping private opaque data: usually used for buffer_heads if PG_private; used for swp_entry_t if PG_swapcache
struct address_space *	mapping	If low bit clear, points to inode address_space, or NULL. If page mapped as anonymous memory, low bit is set, and it points to anon_vma object. <i>If the page does not belong to the page cache, page_mapping() return NULL.</i>
pgoff_t	index	Our offset within mapping, in page-size units.



Page cache handling functions

```
int add_to_page_cache(struct page *page, struct address_space *mapping,
                    pgoff_t offset, int gfp_mask)
```

Add newly allocated pages to page cache. It sets `mapping` and `index` fields of the page structure. The `gfp_mask` is for memory allocation of `radix_tree_node`

```
void remove_from_page_cache(struct page *page)
```

Remove the specified page from page cache.

```
struct page *find_get_page(struct address_space *mapping,
                          unsigned long offset)
```

Find a page in the page cache and increase the reference count of the page atomically.

```
struct page *find_or_create_page(struct address_space *mapping,
                                unsigned long index, unsigned int gfp_mask)
```

Find a page in the page cache. If the page is not present, a new page is allocated using `gfp_mask` and is added to the page cache and to the VM LRU list. The returned page is locked and has its reference count incremented.



Radix tree

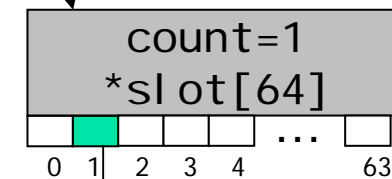
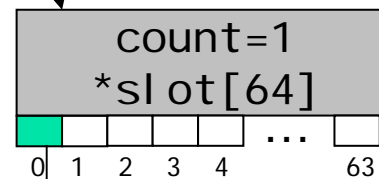
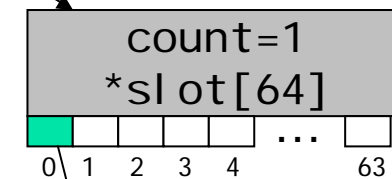
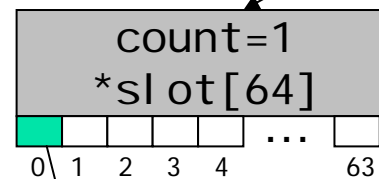
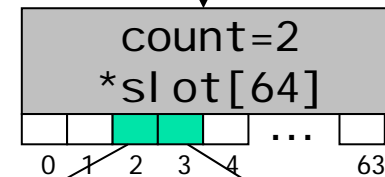
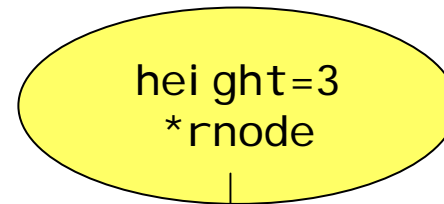
- Searching into page cache should be a very fast operation because it takes place for almost all disk I/O.
 - Hash table in Linux 2.4
 - Radix tree in Linux 2.6
- Radix tree is another dictionary structure with operations like:
 - Insert, delete, lookup
- Time complexity: $O(1)$

Radix tree example

RADIX_TREE_MAP_SHIFT = 6
RADIX_TREE_MAP_SIZE = 2**6 = 64

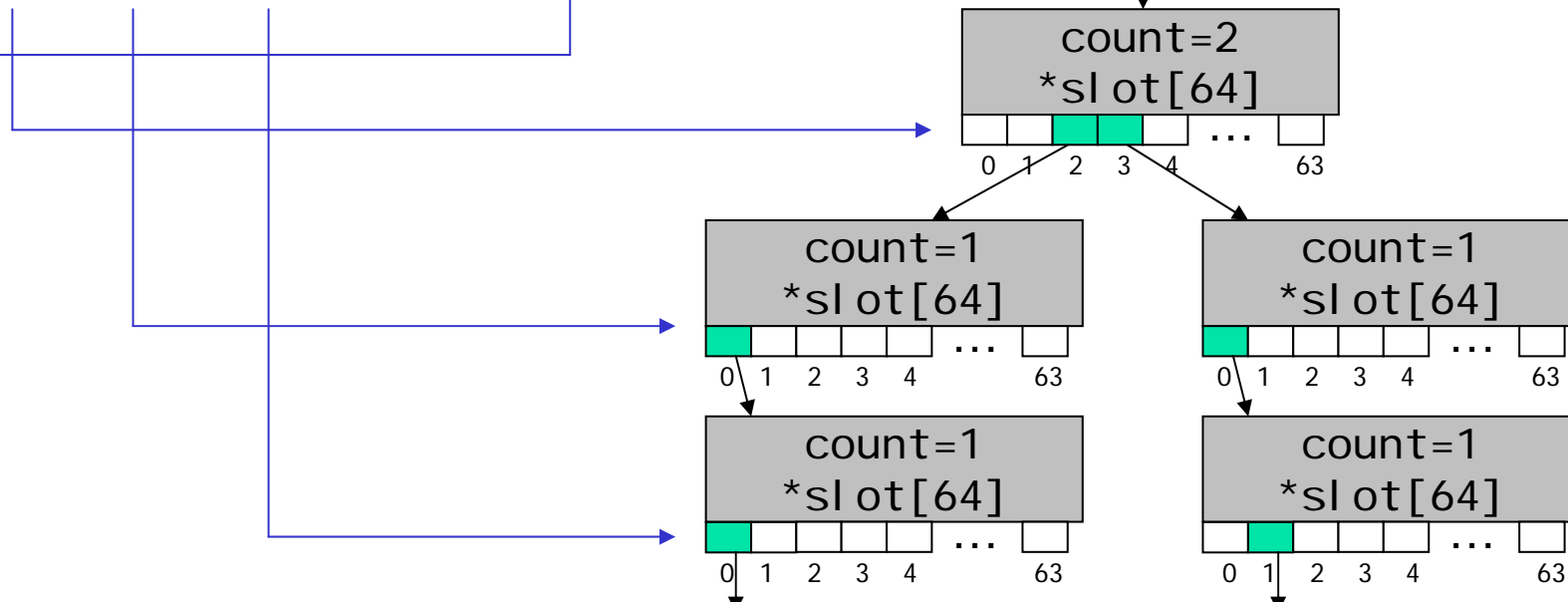
0x2000
= 0010 000000 000000b

radix_tree_root



Page
Index=0x2000

Page
Index=0x3001





Radix tree structure

```
struct radix_tree_root {
    unsigned int    height;
    int            gfp_mask;
    struct radix_tree_node *rnode;
};
```

```
struct radix_tree_node {
    unsigned int    count;
    void           *slots[RADIX_TREE_MAP_SIZE];
    unsigned long   tags[RADIX_TREE_TAGS][RADIX_TREE_TAG_LONGS];
};
```

```
#define RADIX_TREE_MAP_SHIFT 6
#define RADIX_TREE_TAGS      2
#define RADIX_TREE_MAP_SIZE (1UL << RADIX_TREE_MAP_SHIFT)
#define RADIX_TREE_TAG_LONGS ((RADIX_TREE_MAP_SIZE + BITS_PER_LONG - 1) /
                               BITS_PER_LONG)
#define PAGECACHE_TAG_DIRTY  0
#define PAGECACHE_TAG_WRITEBACK 1
```

1. A page in the page cache has two tags, indicating if it's dirty or under writeback.
2. A slot is tagged dirty or writeback if any pages of the slot are dirty or under writeback.



Radix tree operations

```
int radix_tree_insert(struct radix_tree_root *root, unsigned long index,  
                    void *item)
```

Insert an `item` into the radix tree `root` at position `index`. Return 0 on success.

```
void *radix_tree_delete(struct radix_tree_root *root, unsigned long index)
```

Remove the item at `index` from the radix tree rooted at `root`. Return the address of the deleted item, or NULL if it was not present.

```
void *radix_tree_lookup(struct radix_tree_root *root, unsigned long index)
```

Return the item at the position `index` in the radix tree `root`.

```
void *radix_tree_tag_set(struct radix_tree_root *root, unsigned long index,  
                        int tag)
```

Set the search `tag` corresponding to `index` in the radix tree. From the `root` all the way down to the leaf node. Return the address of the tagged item.



Radix tree operations

```
void *radix_tree_tag_clear(struct radix_tree_root *root,
                          unsigned long index, int tag)
```

Clear the search **tag** corresponding to **index** in the radix tree **root**. If this causes the leaf node to have no tags set then clear the tag in the next-to-leaf node, etc. Return the address of the tagged item on success, else **NULL**.

```
unsigned int radix_tree_gang_lookup(struct radix_tree_root *root,
                                   void **results, unsigned long first_index, unsigned int max_items)
```

Performs an index-ascending scan of the tree **root** for present items, starting from **first_index**. Places them at ***results** and returns the number of items which were placed at ***results**. **max_items** limits the number of items can be returned.

```
unsigned int radix_tree_gang_lookup_tag(struct radix_tree_root *root,
                                       void **results, unsigned long first_index, unsigned int max_items, int tag)
```

Performs an index-ascending scan of the tree **root** for present items which have the tag indexed by **tag** set, starting from **first_index**. Places the items at ***results** and returns the number of items which were placed at ***results**. **max_items** limits the number of items can be returned.



Buffer cache data structures

- Buffer_head to describe buffers
- Block device filesystem help identify buffers in the page cache
 - `bd_inode->address_space` to represent buffers in page cache

Data structure of the buffer_head

```

struct buffer_head {
    /* First cache line: */
    unsigned long b_state;           /* buffer state bitmap (see above) */
    struct buffer_head *b_this_page; /* circular list of page's buffers */
    struct page *b_page;             /* the page this bh is mapped to */
    atomic_t b_count;                /* users using this block */
    u32 b_size;                       /* block size */

    sector_t b_blocknr;              /* block number */
    char *b_data;                     /* pointer to data block */

    struct block_device *b_bdev;
    bh_end_io_t *b_end_io;           /* I/O completion */
    void *b_private;                  /* reserved for b_end_io */
    struct list_head b_assoc_buffers; /* associated with another mapping */
};

```

State of buffers

```
enum bh_state_bits {
    BH_Uptodate,          /* Contains valid data */
    BH_Dirty,             /* Is dirty */
    BH_Lock,              /* Is locked */
    BH_Req,               /* Has been submitted for I/O */

    BH_Mapped,           /* Has a disk mapping */
    BH_New,              /* Disk mapping was newly created by get_block */
    BH_Async_Read,      /* Is under end_buffer_async_read I/O */
    BH_Async_Write,     /* Is under end_buffer_async_write I/O */
    BH_Delay,           /* Buffer is not yet allocated on disk */
    BH_Boundary,        /* Block is followed by a discontiguity */
    BH_Write_EIO,       /* I/O error on write */
    BH_Ordered,         /* ordered write */
    BH_Eopnotsupp,      /* operation not supported (barrier) */

    BH_PrivateStart, /* not a state bit, but the first bit available
                    * for private allocation by other entities
                    */
};
```

Dirty buffer lists of inode

- Inode->i_data->private_list

```

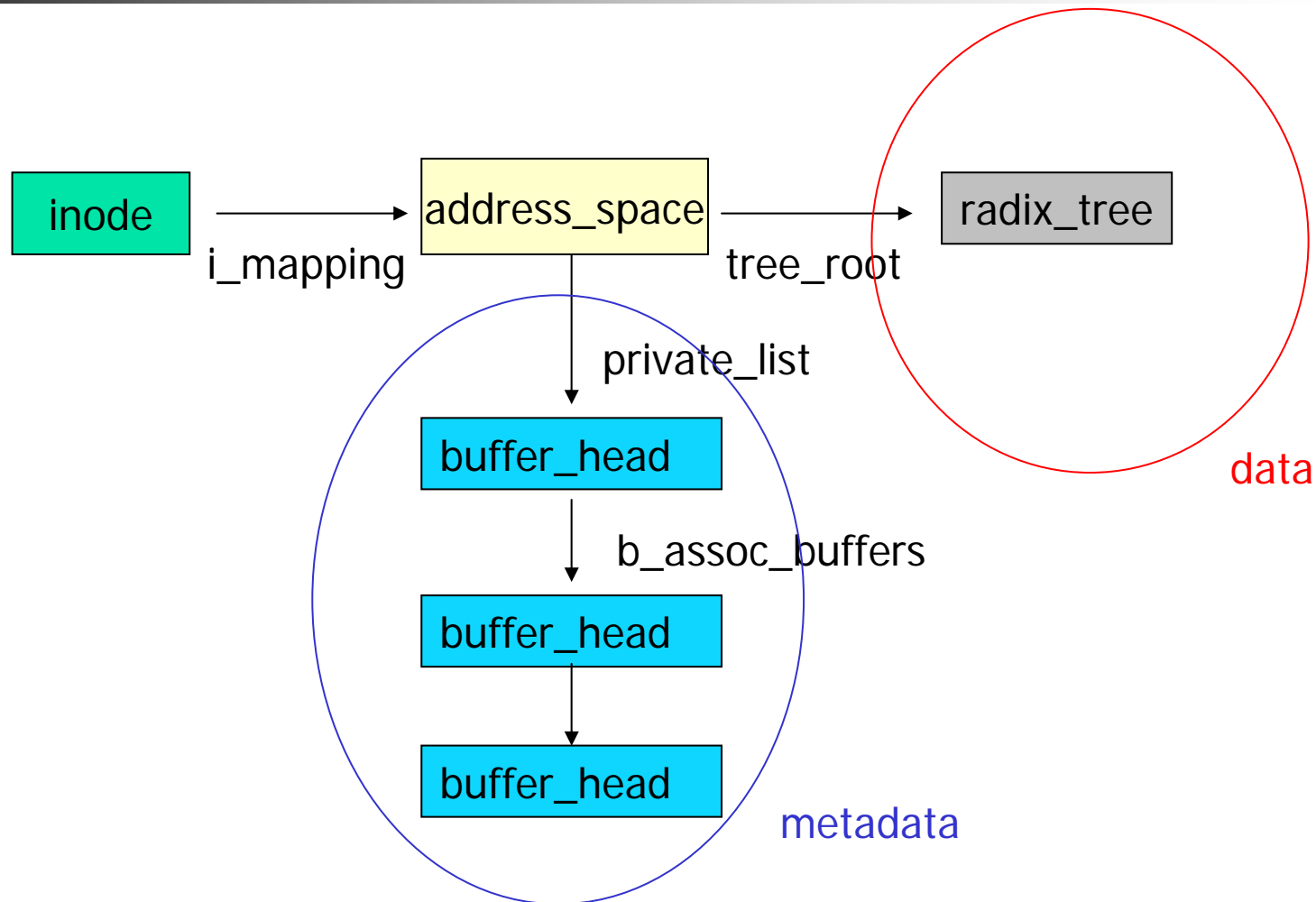
struct inode {
    ...
    struct address_space    *i_mapping;
    struct address_space    i_data;
    ...
}

```

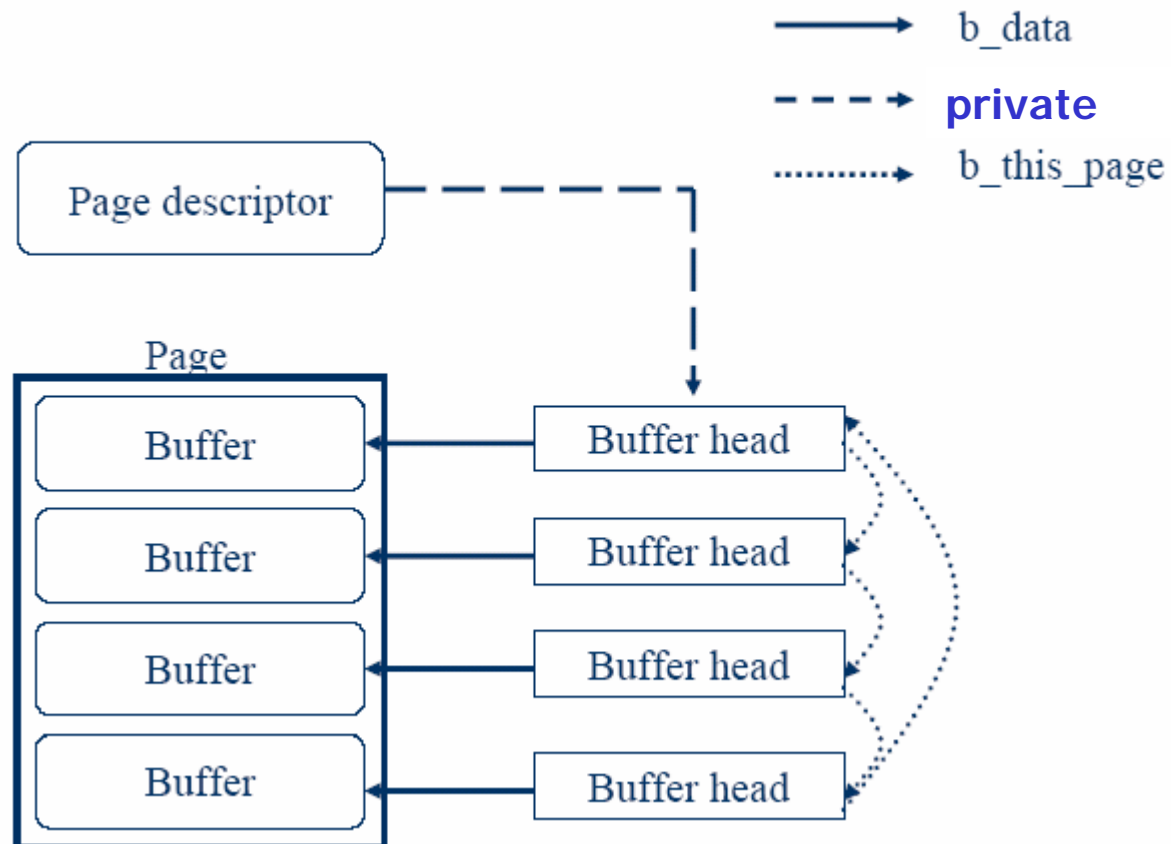
***i_data** is "pages read/written by this **inode**"*
***i_mapping** is "whom should I ask for pages?"*

*IOW, everything outside of individual filesystems should use the latter. They are same if (and only if) **inode** owns the data. CODA (or anything that caches data on a local fs) will have **i_mapping** pointing to the **i_data** of **inode** it caches into. Ditto for block devices if/when they go into pagecache - we should associate pagecache with struct block_device, since we can have many inodes with the same major:minor. IOW, ->**i_mapping** should be pointing to the same place for all of them.*

Dirty pages and buffers of a inode



Buffer pages





Buffer cache handling functions

- `grow_buffers(bdev,block,size)`
- `__getblk(bdev,block,size)`



Block I/O and Page I/O

- Block I/O
 - Read 1 block at a time
 - Example:
 - `__bread()`: for reading superblock and inode
- Page I/O
 - Read 1 or **several** pages at a time
 - Misnamed “Async I/O” in Linux
 - Example:
 - `generic_file_read()`, `generic_file_write()`: for regular file, block device file read/write
 - `filemap_nopage()`: access to memory-mapped file



__bread()

```
__bread(struct block_device *bdev, sector_t block, int size)
{
    struct buffer_head *bh = __getblk(bdev, block, size);

    if (!buffer_uptodate(bh))
        bh = __bread_slow(bh);
    return bh;
}
```



__bread()

```
static struct buffer_head *__bread_slow(struct buffer_head *bh)
{
    lock_buffer(bh);
    if (buffer_uptodate(bh)) {
        unlock_buffer(bh);
        return bh;
    } else {
        get_bh(bh);
        bh->b_end_io = end_buffer_read_sync;
        submit_bh(READ, bh);
        wait_on_buffer(bh);
        if (buffer_uptodate(bh))
            return bh;
    }
    brelse(bh);
    return NULL;
}
```



__bread()

```
void end_buffer_read_sync(struct buffer_head *bh, int uptodate)
{
    if (uptodate) {
        set_buffer_uptodate(bh);
    } else {
        /* This happens, due to failed READA attempts. */
        clear_buffer_uptodate(bh);
    }
    unlock_buffer(bh);
    put_bh(bh);
}
```



How do filesystems use page cache?

- Types of I/Os using page cache
 - Regular file read, write
 - Block device file read, write
 - Memory mapped read, write
- Types of I/Os without page cache
 - Direct I/O



generic_file_read()

- This function is used for both regular file and block device file read.
- `generic_file_read()`
 - `do_generic_mapping_read()`
 - `mapping->a_ops->readpage(filp, page)`
- Ext2: `readpage = ext2_readpage`
blkdev: `readpage = blkdev_readpage`
They are just wrapper functions which invokes `mpage_readpage()` or `block_read_full_page()`



Trace these functions

- `do_generic_mapping_read()`
- `block_read_full_page()`
- `mpage_readpage()`



Writing Dirty Pages to Disk

- When to write?
 - Too many dirty pages in memory
 - Periodic write back for safety
 - Memory not enough
 - User request (fsync(), sync(), msync())
- Who is responsible for the write out?
 - pdflush - background writeout operation
 - pdflush – kupdate operation
 - kswapd, try_to_free_pages



Writing Dirty Pages to Disk

- What to write?
 - Dirty process page (swap out)
 - Dirty file or directory data page
 - Dirty memory mapped pages
 - Dirty filesystem inode
 - Dirty filesystem indirect blocks (ex: ext2, ext3)
 - Dirty filesystem superblocks

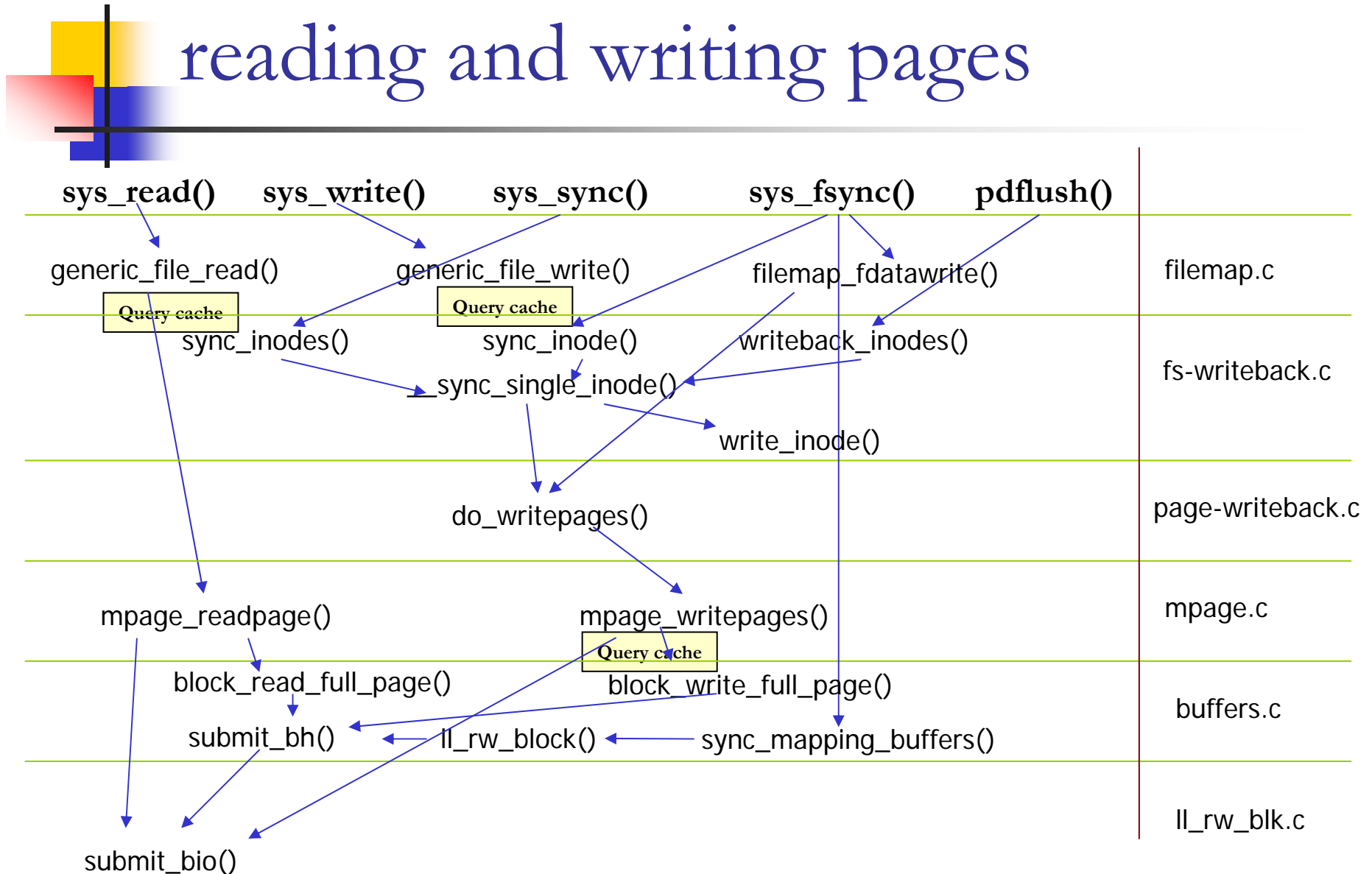
PS: for memory mapped file, Kernel knows whether a page is a dirty page if and only if user called `msync()` or `kswapd` try to swap out, which scan the page table and update corresponding page's dirty flag

Writing Dirty Pages to Disk

- How to keep track of dirty pages, dirty buffers and dirty metadata?

	Linux 2.4	Linux 2.6
regular block device files	per inode and systemwide dirty buffer lists	dirty page tag in every <code>address_space</code> 's radix tree
memory-mapped files(*)	per <code>address_space</code> dirty page list	
dirty filesystem inodes	dirty inode list at filesystem superblock structure	
dirty filesystem indirect blocks	per inode and systemwide dirty buffer lists	per <code>address_space</code> dirty buffer lists (<code>private_list</code>) why?
dirty filesystem superblocks	flag at superblock structure	
process dirty swap pages	dirty page list in <code>swapper_space</code>	dirty page tag in <code>swapper_space</code> 's radix tree

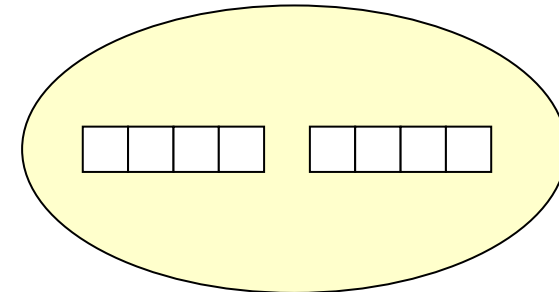
Layers of source code of reading and writing pages



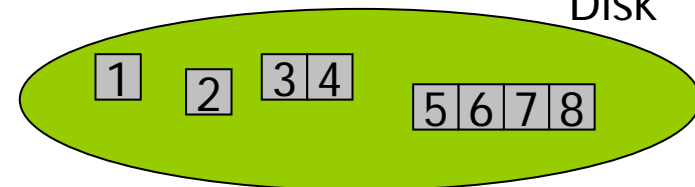
Scenario of reading a file from page cache – generic_file_read()

1. Check page cache and the corresponding pages do not exist, allocate free pages and add them to page cache
2. Invoke `mpage_readpages()` to read from disk
3. First page not contiguous on disk, invoke `block_read_full_page()` to read a block at a time(`submit_bh()`)
4. Second page is contiguous, read via `submit_bio()` to read 4 block at once

Page cache



Disk



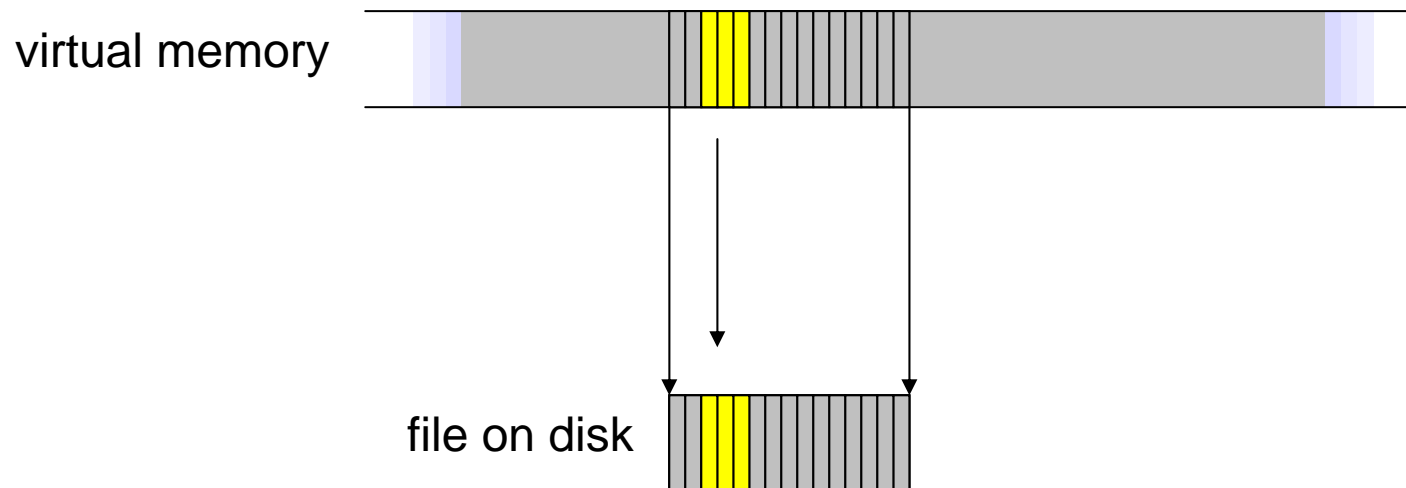


pdflush

- Thread pool
 - Dynamic add or remove threads in thread pool if thread is idle or pool is empty for more than 1 second
- 2 kinds of operations
 - background_writeout
 - write until dirty pages / total free pages < background_thresh
 - wb_kupdate
 - Write dirty pages older than 30 seconds.
- pdflush_operation()
 - Wakeup a pdflush thread and get it to do the work assigned
- wakeup_bdflush()
 - Use pdflush_operation() to wakeup a pdflush thread to do background_writeout

What are memory mapped files?

- A memory region that is associated with some portion of a regular file on a block device
 - Access to a byte within a page of the memory region is translated into an operation on the corresponding byte of the file

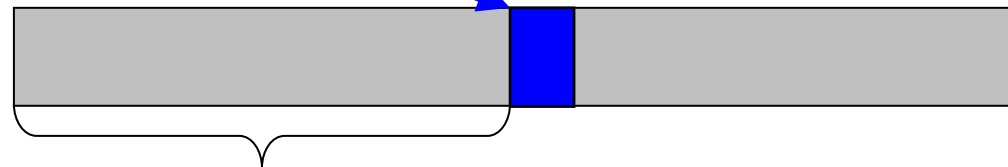


Example

```
int fd;  
char *data;  
fd = fopen(...);  
data = mmap(...);
```

data[x]

content of file:



offset x



Types of memory mapping

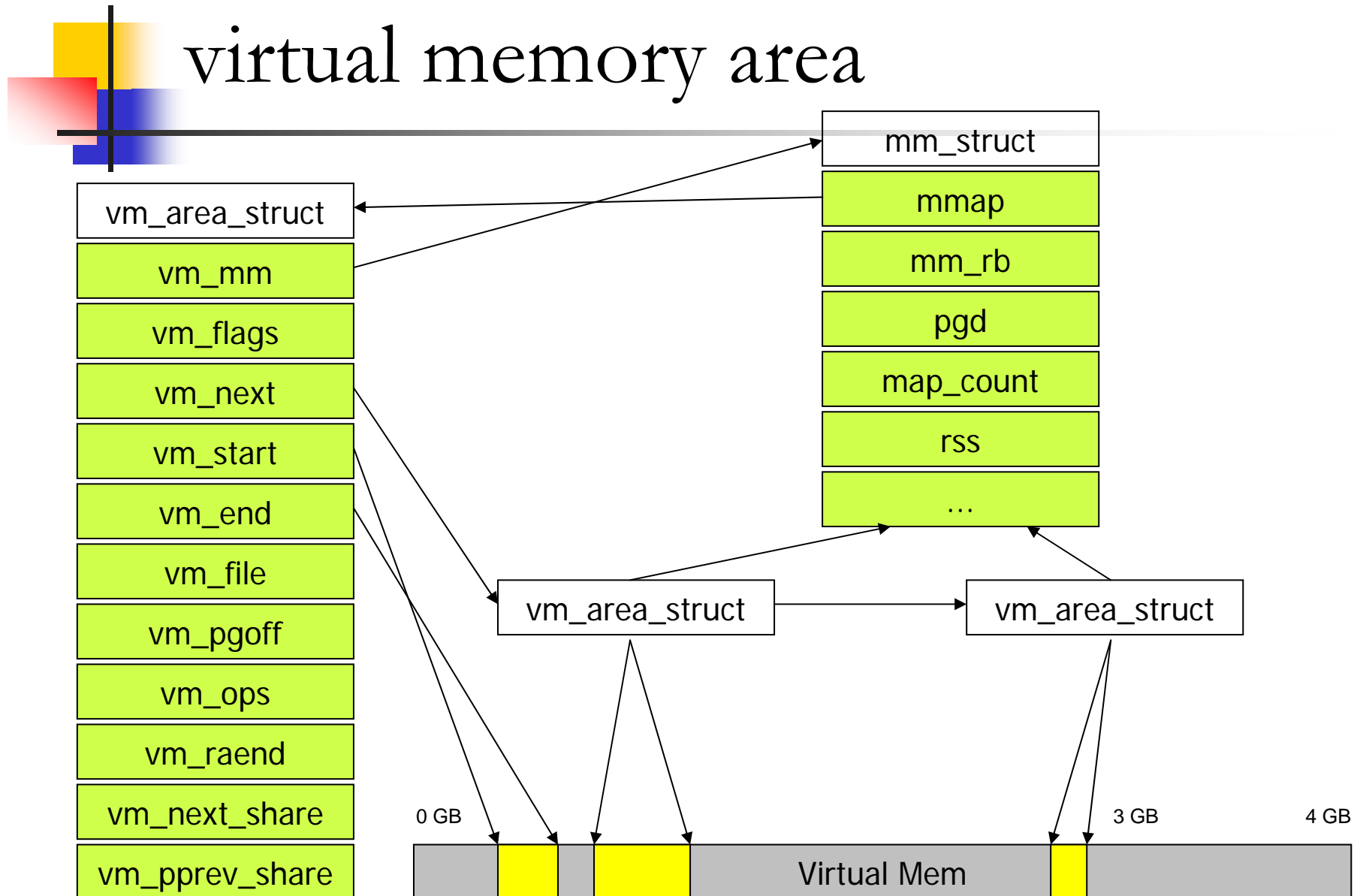
- Shared
 - write operation changes file on the disk
 - changes visible to all other processes that map the same file
- Private
 - meant to be used for read only
 - copy on write
 - writing stops mapping a page to the file
- a file can be mapped private and shared at the same time



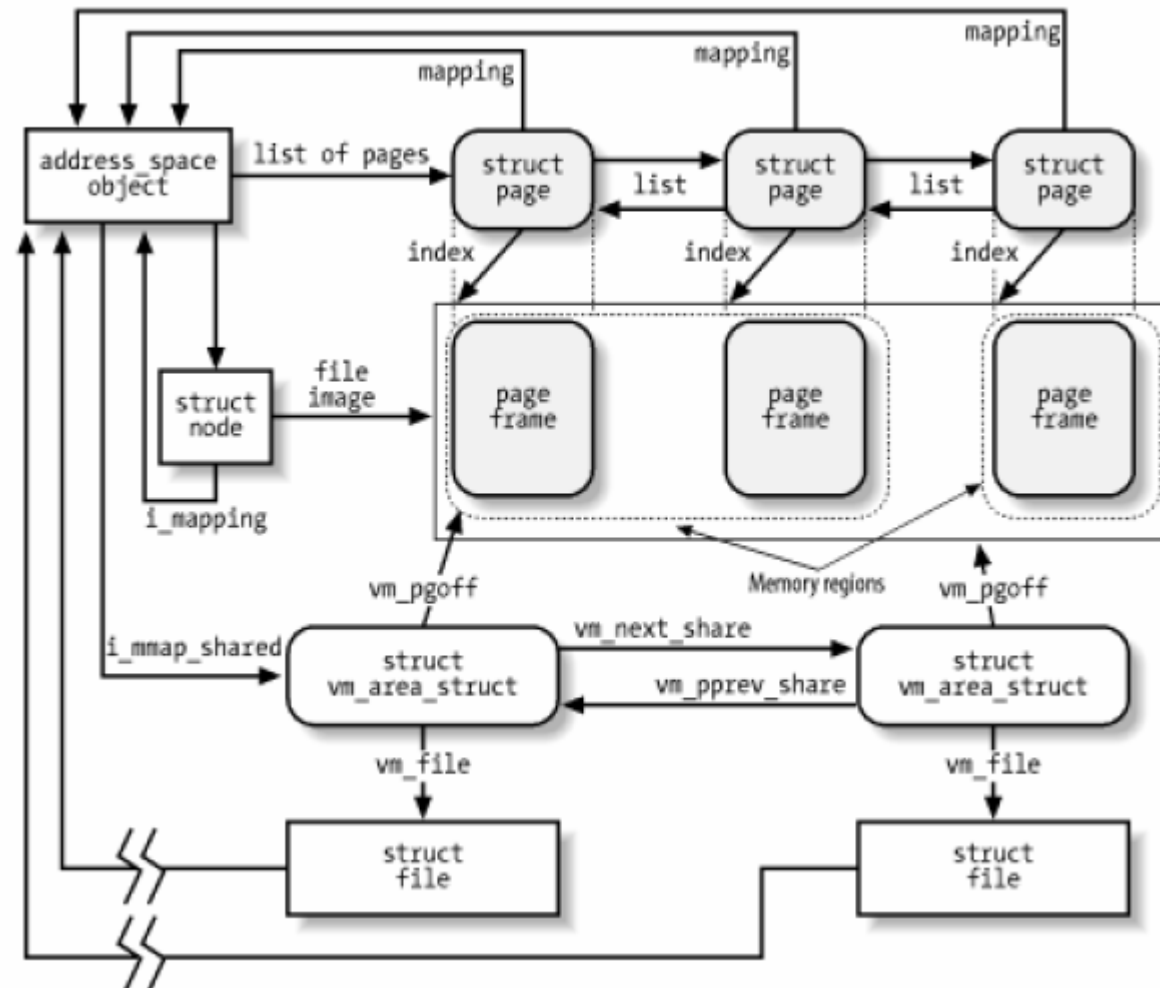
Memory-mapping data structures

- A memory mapping is represented by:
 - **inode** object of the file
 - **address_space** object of the file
 - **vm_area_struct** descriptor for each mapping of the file
 - **file** object for each mapping of the file
 - **page** descriptor for each frame assigned to the memory region

data structure representing a virtual memory area



Data structures for file memory mapping

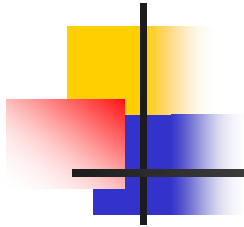




mmap()

- Parameters:
 - file descriptor
 - offset inside the file
 - length of the file portion
 - flags (either **MAP_SHARED** or **MAP_PRIVATE**)
 - set of permission
 - **PROT_READ**
 - **PROT_WRITE**
 - **PROT_EXEC**
 - optional linear address

- Returns: linear address of first location of the memory region



- **MAP_PRIVATE:**
 - copy on write – no changes

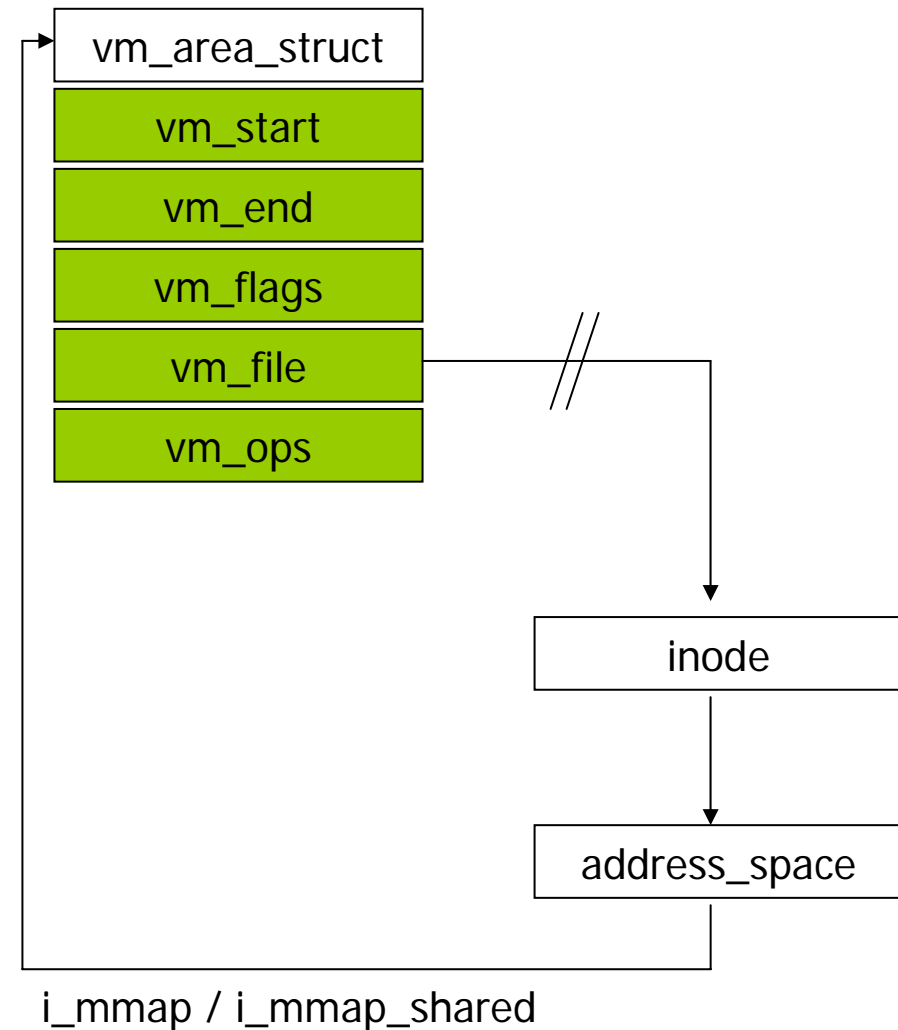
- **MAP_SHARED:**
 - changes will be saved

- **MAP_FIXED:**
 - forces to use the specified address

- **MAP_ANON or MAP_ANONYMOUS:**
 - mapping not connected to a file
 - memory region initialized with zeros

mmap() Details

- File mappable?
- vm_area_struct created
- File access rights?
- Set vm_flags and vm_file
- Init nopage()





Destroying a Memory Mapping

- System call to destroy a memory mapping: `munmap()`
- Parameters:
 - address
 - length
- It is possible to either remove or reduce the size of a memory region.
- Notes:
 - removes the memory region from the `address_space` object
 - hole inside a region => two smaller regions are created

Classifying the page faults - do_page_fault()

Fault address		CPU mode		user mode	kernel mode		
		in VMA	text/heap/stack		interrupt handler	kernel thread	system call
user space	in VMA	text/heap/stack	stack	H	O		
	not in VMA	other					
kernel space	vmalloc area		other	K	V		
	other				O		

K = send SIGSEGV signal to process
H = If permission is okay, called handle_mm_fault()
O = print Oops message and kill current process
V = handle the vmalloc fault by fixing page table

Classifying the page faults - handle_pte_fault()

- If present bit = 0 in the page table entry
 - If page table entry = NULL
 - do_no_page() Anonymous page or file-mapped page
 - Else
 - do_swap_page() The value in page table entry is the index of the **swapped-out page** In swap space
- Else
 - If this is a write access and page table entry write bit = 0
 - do_wp_page() Shared page – Copy on write



do_no_page()

- Anonymous page
 - vma->ops = NULL
 - do_anonymous_page()



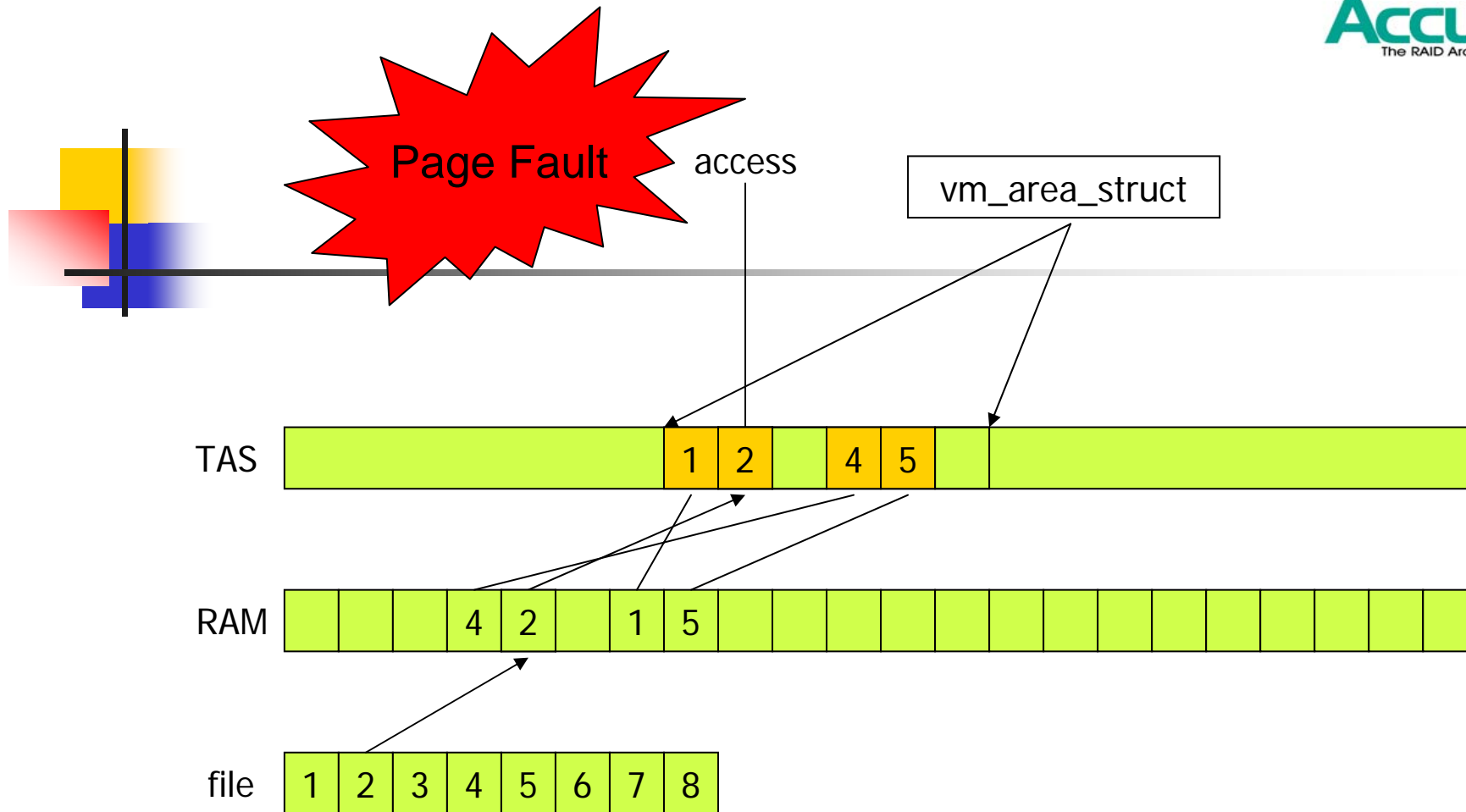
do_no_page()

- File-mapped page
 - filemap_nopage() to get the requested page
 - if this a write access and the vma's VM_SHARED flag off
 - do an early C-O-W break (allocate new page and copy the content of the requested page to the new page)
 - Set page table entry!



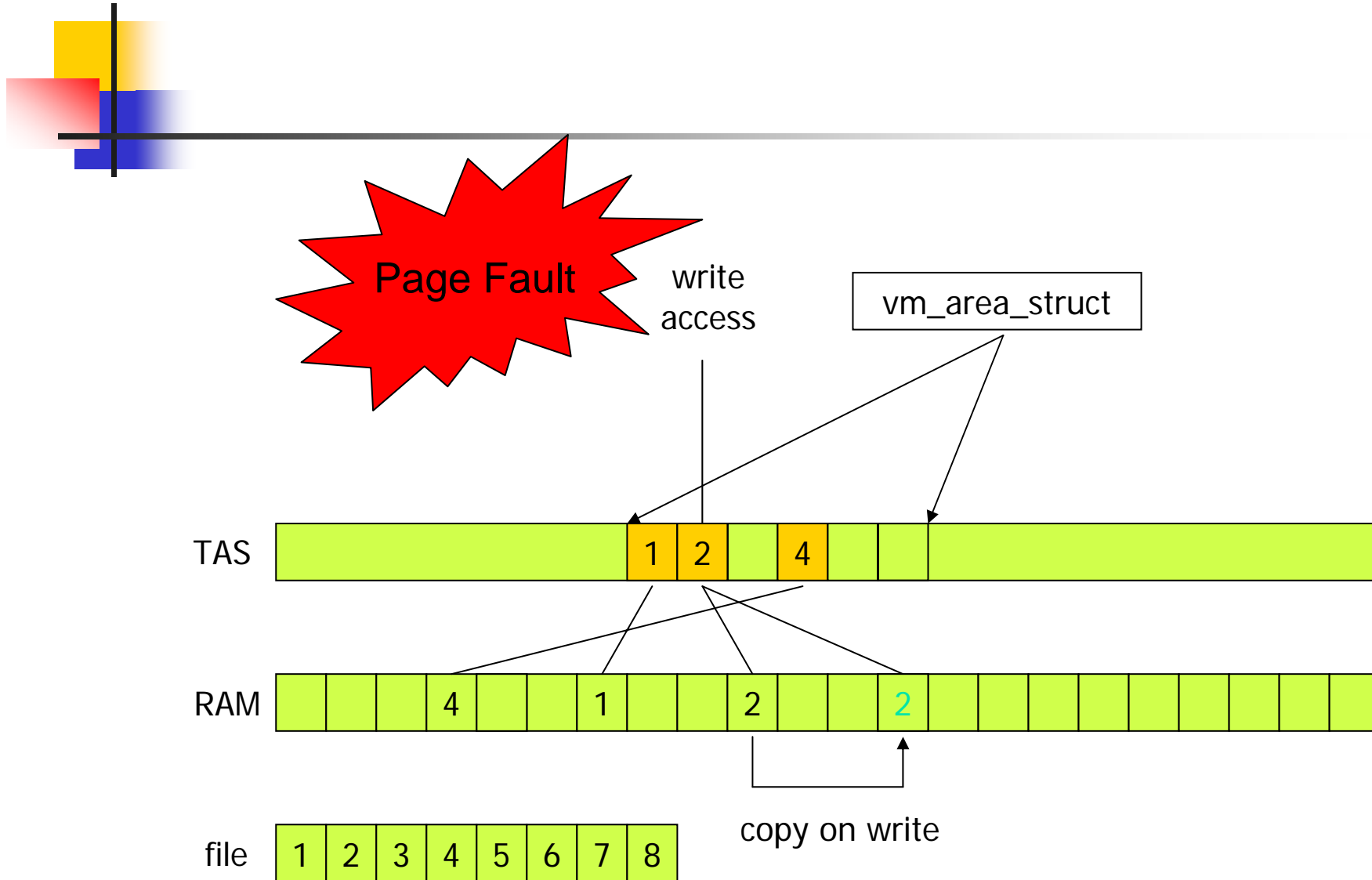
Filemap_nopage()

- Invokes `find_get_page()` to find the page in the page cache
- If the page is not in the page cache, use `page_cache_read()` to read it from disk
- The page is in the page cache. If the page is not up to date, invoke `a_ops->readpage()`
- The page is up to date. mark page accessed and return the page.



do_no_page: • vm_area_struct->vm_ops->nopage() loads page from disk or cache

• Page table entries updated





The purpose of swapping

- More memory
 - Expand the address space effectively usable by a process
 - Expand the amount of RAM to load more processes
- Better memory utilization
 - swap out unused pages and use the RAM for disk cache.

Swapping issues to be considered

- What kind of page can be swap out?

Type \ Action	What to do?			Note
	swap out	discard	writeback	
Anonymous (private or shared)	v			Heap, stack
File mmap'ed (private and modified)	v			in fact, this is anonymous page
File mmap'ed (shared and modified)			v	
File mmap'ed (clean)		v		
IPC shared	v			



Swapping issues to be considered

- How to distribute pages in the swap areas
 - Cluster swap pages by allocating them sequentially
 - Minimize disk seek time
 - Swap area priority
 - Faster swap area get a higher priority
 - Round-robin among swap areas with the same priority
- How to select the page to be swapped out
 - LRU approx. – second chance algorithm
 - Active and inactive lists
- When to perform page swap out
 - kswapd is activated whenever the number of free pages falls below a predefined threshold
 - `alloc_pages()` cannot be satisfied

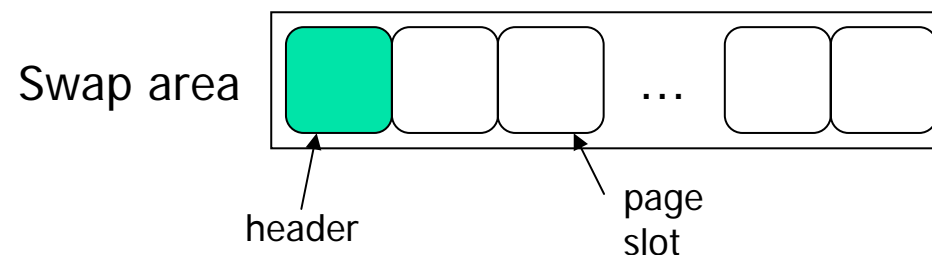


Swap Area

- May be implemented as a disk partition or as a file
- Several different swap areas may be defined
 - spread among several disks so as to use them concurrently
- Each swap area consists of a sequence of page slots. The first page slot is used to store some information (swap header) about the swap area

Swap header

Type	Name	Description
Char [1024]	bootbits	Not used. May store partition information, disk labels, etc.
unsigned int	Version	Swap algorithm version
unsigned int	Last_page	Last page slot that is effectively usable
unsigned int	nr_badpages	Number of defective page slots
unsigned int [125]	padding	Padding bytes
unsigned int [637]	badpages	Up to 637 numbers specifying the location of defective page slots
char [10]	magic	“SWAP-SPACE” or “SWAPSPACE2”





Swap Area Descriptor (1)

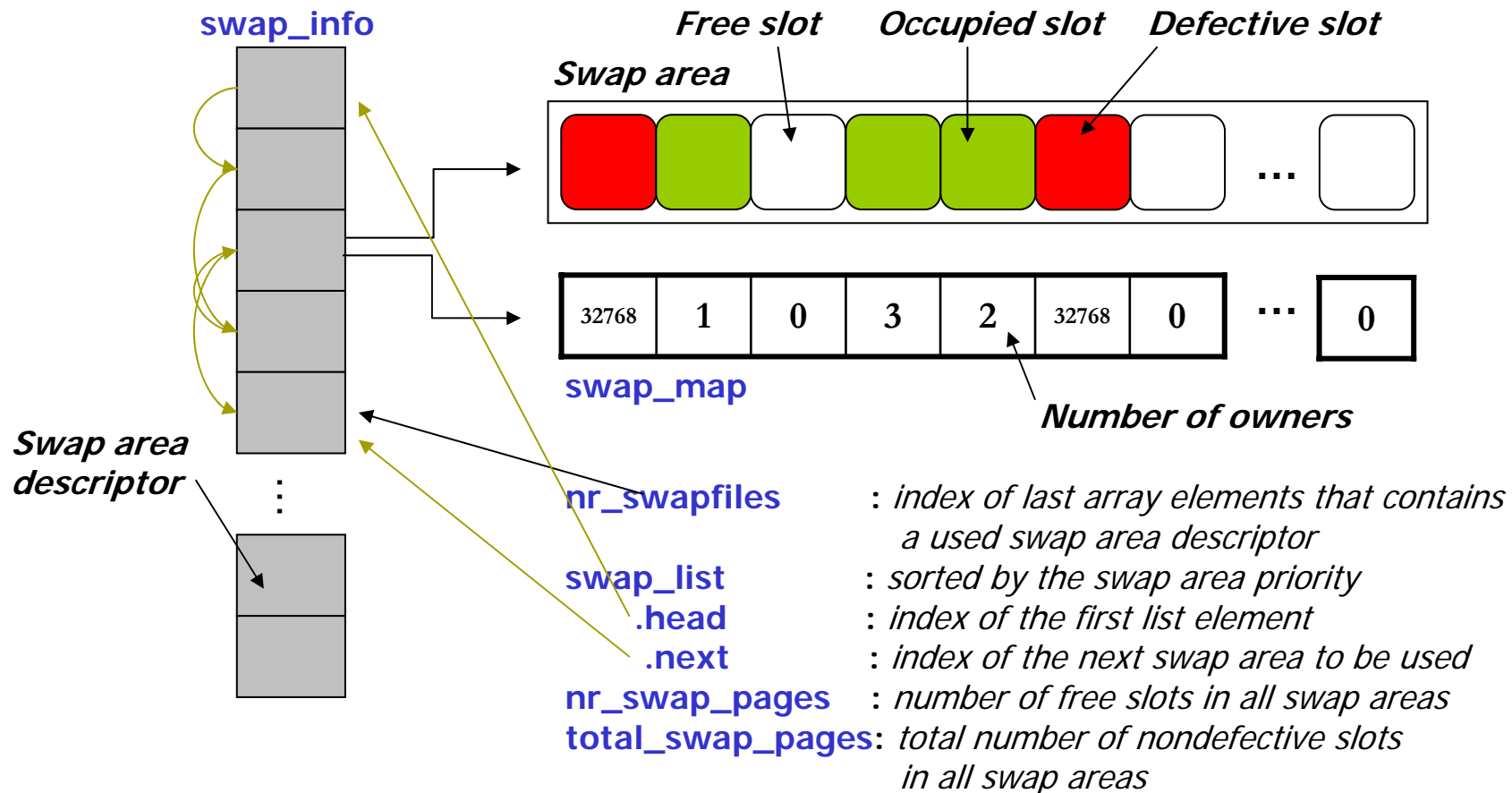
Type	Name	Description
unsigned int	flags	Swap area flags (SWP_USED, SWP_WRITEOK)
spinlock_t	sdev_lock	Swap area descriptor spinlock
struct file *	swap_file	File pointer to swap file (regular or block device)
struct block_device *	bdev	The block device the swap file resides on
unsigned short *	swap_map	Array of counters, one for each page slot
unsigned int	lowest_bit	The index of the lowest possibly-free page slot
unsigned int	highest_bit	The index of the highest possibly-free page slot
unsigned int	cluster_next	The starting index of next search in cluster
unsigned int	cluster_nr	Number of page slots left in this cluster
int	prio	Swap area priority
int	pages	Number of usable page slots
unsigned long	max	Size of swap area in pages
int	next	The index in swap_info array of next swap area



Swap Area Descriptor (2)

Type	Name	Description
<code>struct list_head</code>	<code>extent_list</code>	List head for extents in the swap area
<code>int</code>	<code>nr_extents</code>	Number of extents in the list
<code>struct swap_extent *</code>	<code>curr_swap_extent</code>	Last extent that was searched

Swap Area Data Structures

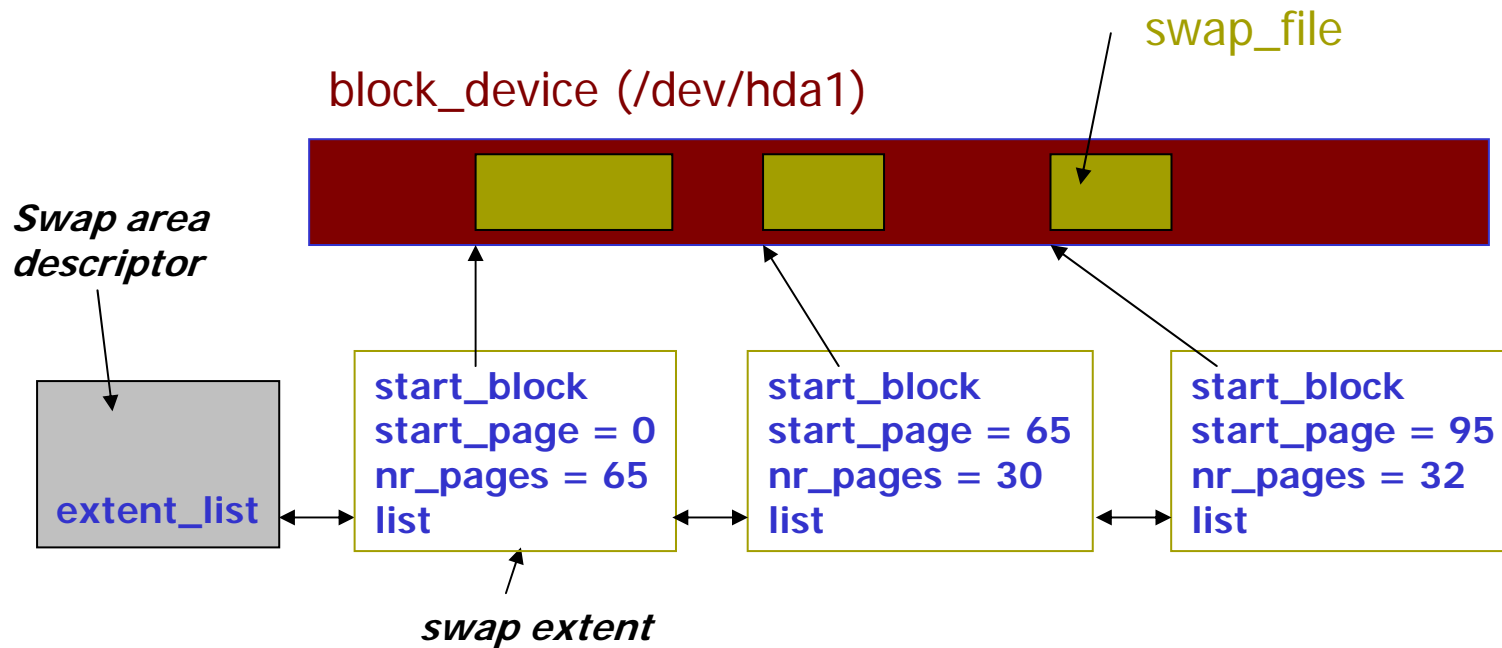




Swap extents

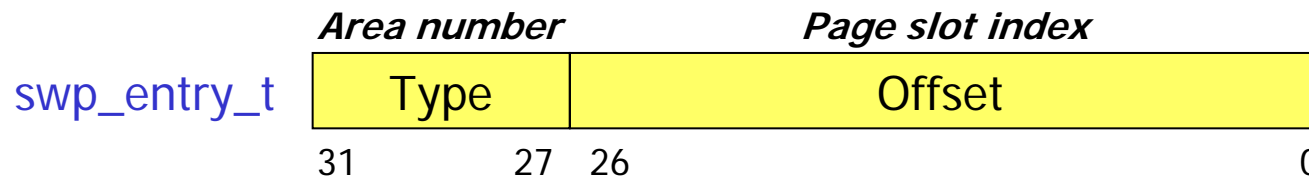
- Extents map a contiguous range of pages in the swap area into a contiguous range of disk blocks.
- An ordered list of swap extents is built at swapon time and is then used at swap_writepage / swap_readpage time for locating where on disk a page belongs.
- For block devices, there will only be one swap extent, and it will not improve performance. But it can make a large difference with swap files, which is not necessary contiguous on disks and will have multiple extents.

Swap extents example



Swapped-out page identifier

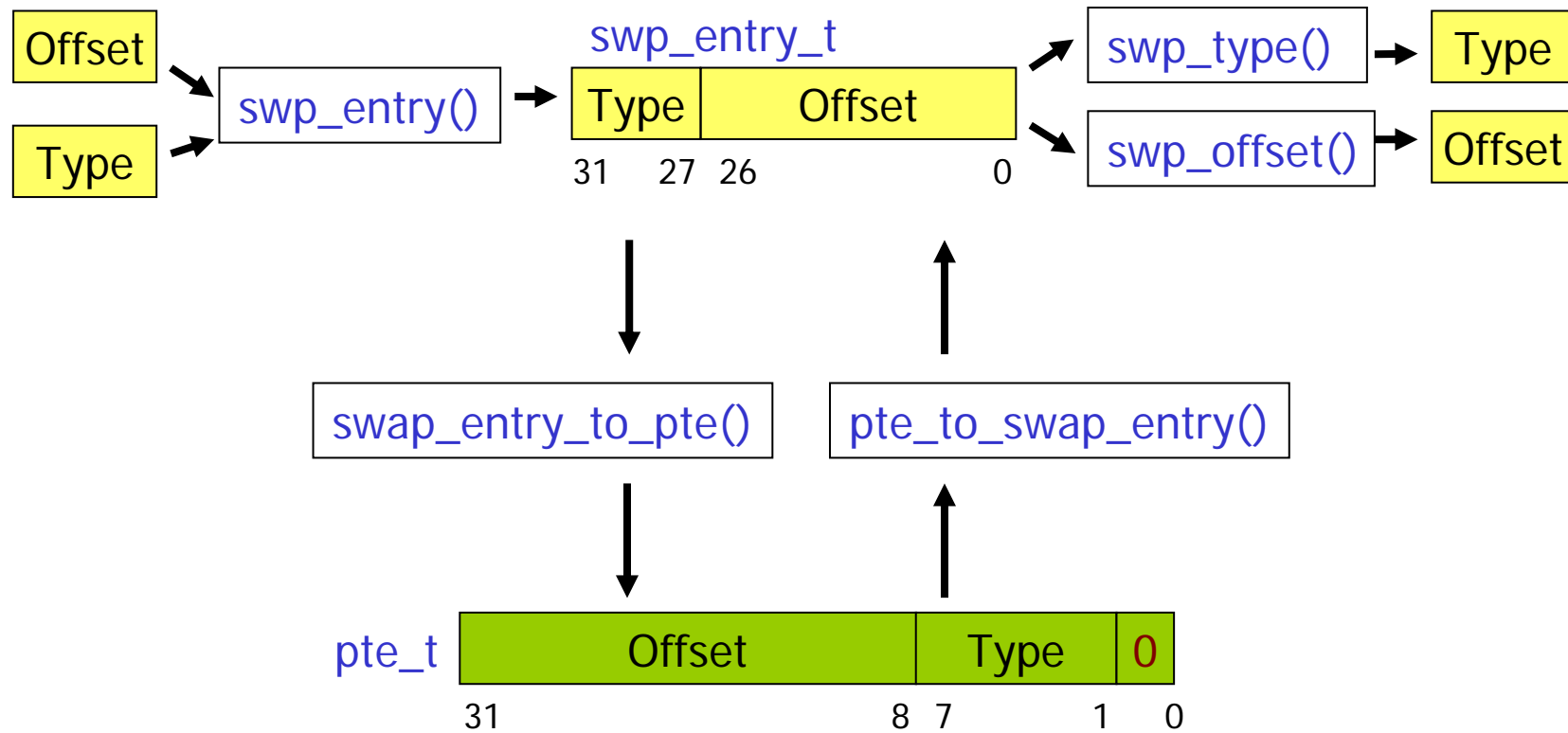
- A swapped-out page is identified by:
 - The index of swap area
 - Page slot index inside the swap area

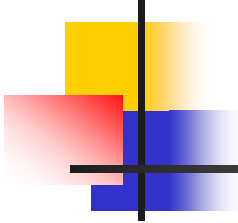


- When a page is swapped out, its identifier is inserted into the corresponding page table entries.



Encoding and decoding swap entry





Activating a swap area – `sys_swapon()`

1. Search `swap_info` array for empty slot (`SWP_USED`), update `nr_swapfiles`
2. Read in `swap_header`.
3. Initialize `lowest_bit`, `highest_bit`, `swap_map` according to the information in the `swap_header`
4. Setup swap extents (use `bmap()` to find block numbers in a file)
5. Update `nr_swap_pages`, `total_swap_pages`
6. Insert `swap_info` into `swap_list`



Deactivating a swap area – `sys_swapoff()`

- Partition being deactivated may still contain pages that belong to several processes
 - Scan the swap area and swap in all existing pages
1. Remove the swap area descriptor to be deactivated from `swap_list`
 2. Update `nr_swap_pages`, `total_swap_pages`
 3. Invoke `try_to_unuse()` to swap in pages
 4. Destroy swap extents, free up `swap_map` and unset `swap_info` flags (`SWP_USED`, `SWP_WRITEOK`)



try_to_unuse()

1. Scan `swap_map` for in-use page slot
2. For each in-use page slot,
 1. Invoke `read_swap_cache_async()` to read the page in, and put it in the swap cache.
 2. Scan the page table entries of all processes and replace each occurrence of the swapped-out page identifier with the physical address of the page just read in.
 3. `delete_from_swap_cache()` to remove the page from swap cache



Allocating a page slot

- Store pages in contiguous slots to minimize disk seek time
 - Always start from the beginning of the swap area
 - Bad for swap-out operations
 - Always start from the last allocated page slot
 - Bad for swap-in operations
- Linux's approach
 - Always start from the last allocated page slot, but if one of these conditions occurs, restart from the beginning
 - The end of the swap area is reached
 - `SWAPFILE_CLUSTER` free page slots were allocated after the last restart from the beginning of the swap area



Swap page Functions

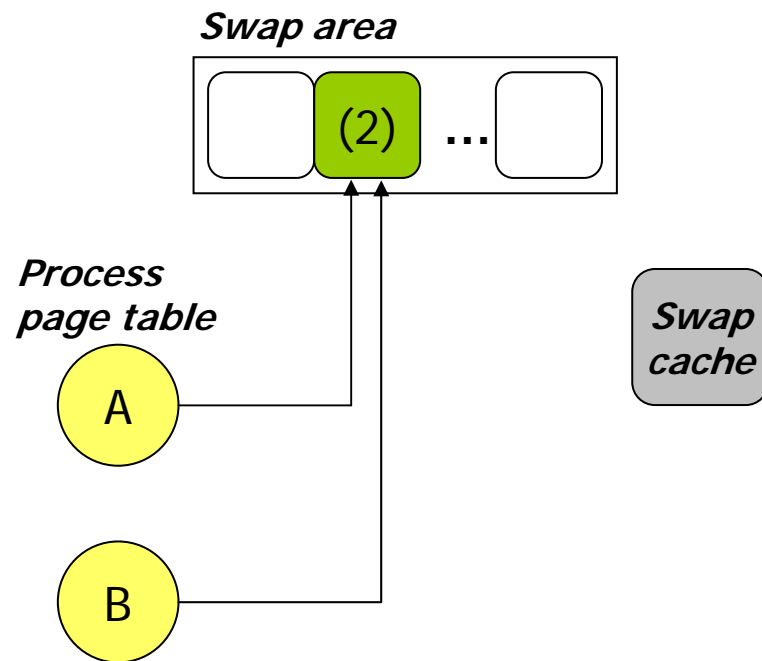
- `scan_swap_map()`
 - Given a swap area, scan its `swap_map` to find a free page slot
 - allocate from current cluster first
- `get_swap_page()`
 - Invoke `scan_swap_map()` to find a free slot among all swap areas
 - The first pass searches areas having the same priority in a round-robin way. If no free slot is found, the second pass searches from the beginning of the swap area list.
- `swap_free()`
 - Invoked when swapping in a page to decrement the corresponding `swap_map` counter
 - If the counter reaches 0, it means the page slot is free and its identifier is not saved in any page table entry



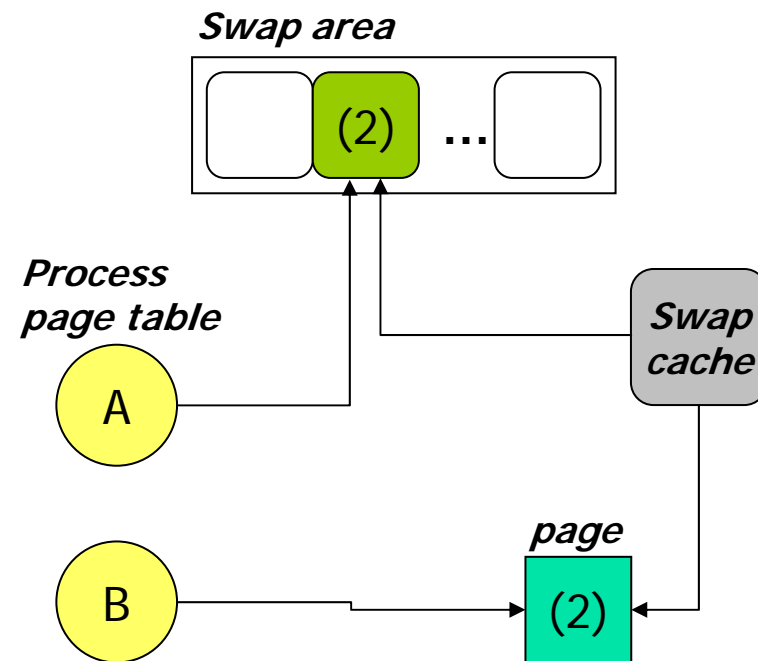
Why Swap Cache?

- Page frames may be shared among several processes if it belongs to:
 - Shared memory-mapped file
 - no swap, just writeback
 - C-O-W memory region
 - process fork or private memory mapping
 - Anonymous shared or IPC shared memory
- The same page may be swapped out for some processes and and present in memory for others
 - eg. A completely swapped-out page got swapped-in because of the memory access from one of the processes sharing the page
- Kernel needs a data structure to find partially swapped-out pages.

The Role of the Swap Cache



Page is completely swapped-out



Page is swapped-in for process B

Swap Cache Implementation

- Swap cache is implemented by the page cache data structures and functions.
- Conceptually
 - `address_space` object for swap space: `swapper_space`
 - Page in the swap cache are just like any other page in the page cache, except
 - `page.mapping = swapper_space`
 - `page.index = swp_entry_t`
- But in Linux 2.6, anonymous page's mapping points to `anon_vma` for object-based reverse mapping. The solution:
 - `page.private = swp_entry_t`
 - Set `page.flags = PG_swapcache`
 - Use `page_mapping()` to find page's actual mapping
 - ie. If `PG_swapcache` is set, it will return `swapper_space`

Swap cache insert, delete and lookup

- `lookup_swap_cache(swp_entry_t)`
 - Lookup a swap entry in the swap cache. A found page will be returned unlocked with its reference count incremented
- `add_to_swap_cache(page, swp_entry_t)`
 - Insert a page into the swap cache, increment page reference count and lock the page. It insert the page into `swapper_space`'s radix tree (also set `PG_swapcache` and `page.private = swp_entry`)
- `delete_from_swap_cache(page)`
 - Remove a page from the swap cache
- `free_page_and_swap_cache(page)`
 - Perform a `put_page()` and remove the page from the swap cache

Transferring swap pages

- `sector_t map_swap_page(swap_info, offset)`
 - Use this swapdev's extent information to locate the (PAGE_SIZE) block which corresponds to page offset 'offset'.
- `get_swap_bio(gfp_flags, index, page, end_io)`
 - Prepare the bio for writing the page to the swap area. It invokes `map_swap_page()` to find the corresponding block number for the page.

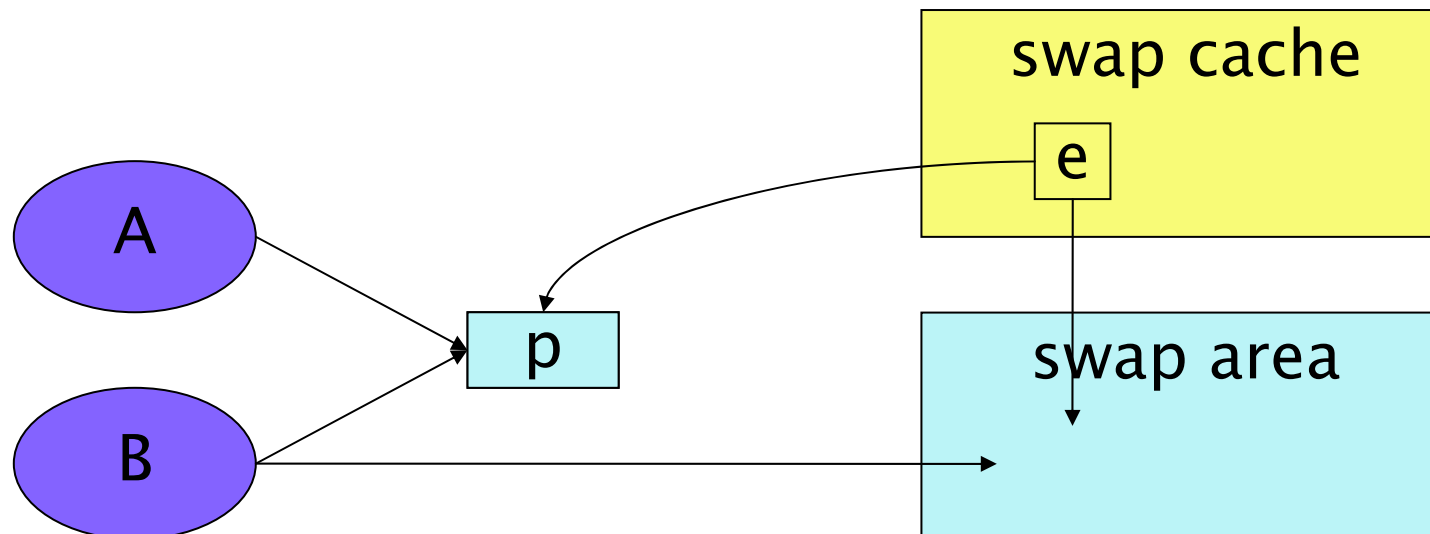
`swapper_space's readpage() and writepage()`

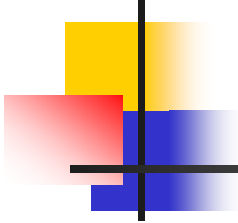
- `swap_readpage(page)`
 - Asynchronous read the page from the swap area by calling `get_swap_bio()` and `submit_bio()`
- `swap_writepage(page, writeback_control)`
 - Asynchronous write the page to the swap area by calling `get_swap_bio()` and `submit_bio()`
- `read_swap_cache_async(swp_entry)`
 - Locate a page in the swap cache. If no page is found, allocate a new page, insert it into swap cache and read it from the disk by `swap_readpage()`

Problem:

Before swapping out a page ..

- You need to find an inactive page and unmap all its references from processes' page table.
- **Linux 2.4** cannot
 - derive from a page the list of PTEs mapping it. It uses swap cache to keep track of partially swap-out pages and reclaim the page only when the page reference count drops to 1 (page cache)
 - unmap only the pages it really wants to evict.





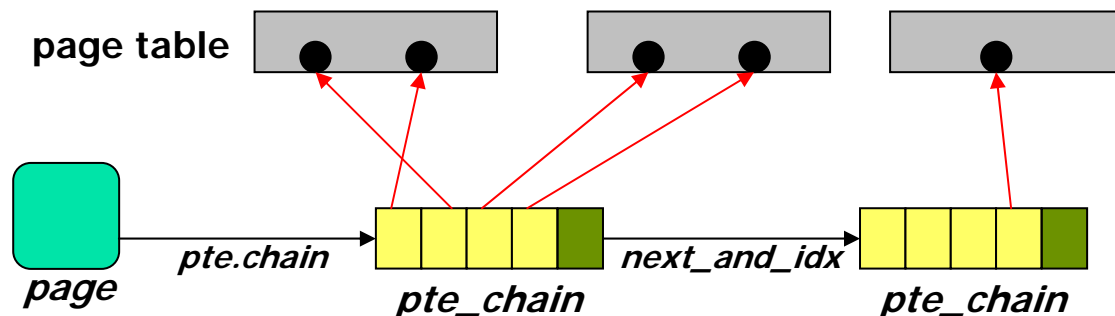
Linux 2.4's approach to swap out a page

- Because of this, in **Linux 2.4**, page reclaim routine `shrink_cache()` cannot check if a page returned from the inactive LRU list is really inactive by reading its PTEs nor can it swap the page out if the page is mapped. It invokes `swap_out()` when the number of mapped pages return from the inactive LRU list exceeds some threshold.
- `swap_out()` scans each process's page table for inactive pages and resets the PTE. It terminates when it successful releases `SWAP_CLUSTER_MAX` pages. A page frame is considered released when all references from the page tables of all processes are removed.
- Eventually, a inactive mapped page will become unmapped and thus eligible for page reclaim.

Solution 1:

Direct reverse mapping

- To help Linux find the page table entries associated with a given page
- Direct reverse mapping
 - Given a physical page, return a list of pointers to PTEs which point to that page.
 - Disadvantage
 - Memory exhaustion in low memory
 - `fork()` slow down significantly since it must add a new reverse mapping entry for every page in the process's address space

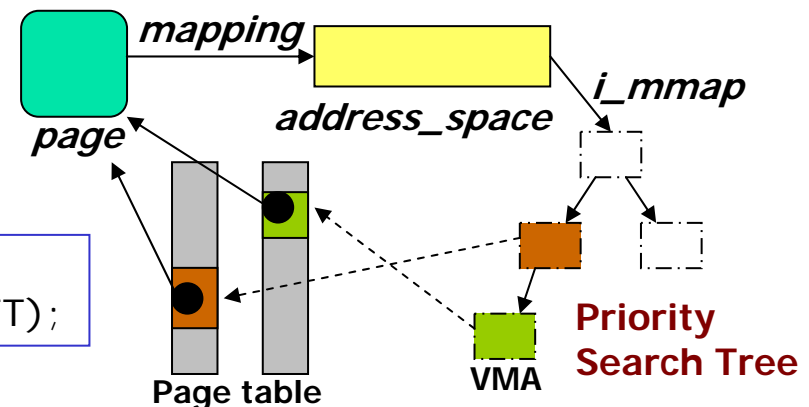


Solution 2:

Object-based reverse mapping

- Find a given page's page table entries indirectly
- For file-backed pages:
 - kernel can find all VMAs that maps the page at `page->mapping`
 - The VMA provides the information needed to find out what a given page's virtual address is in that process's address space, and that, in turn, can be used to find the correct page table entry.
 - Ref slide no. 44

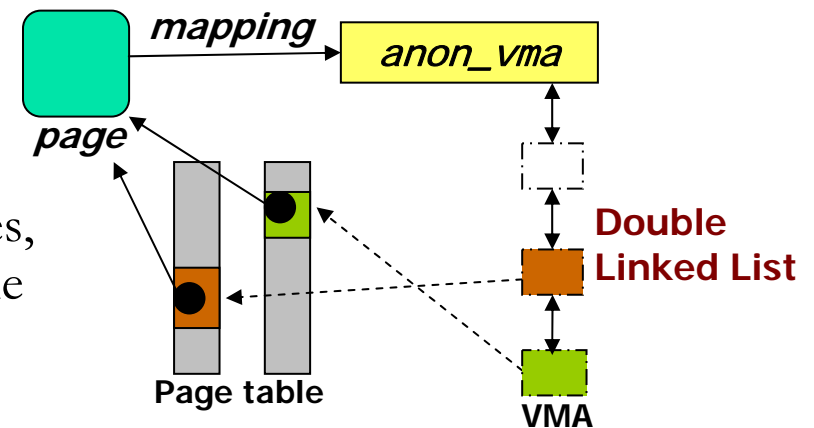
```
address = vma->vm_start +
((page->index - vma->vm_pgoff) << PAGE_SHIFT);
```



Solution 2:

Object-based reverse mapping

- For anonymous pages
 - Anonymous pages don't have mapping behind
 - Introduce new data structures to chain VMAs
- Advantage
 - One VMA can refer to thousands of pages, so a per-VMA cost will be far less than the per-page costs
- Disadvantage
 - Greater computational cost
 - Freeing a page requires scanning multiple VMAs which may or may not contain references to the page under consideration.



Priority search tree (prio_tree) – speeding up the search of VMAs

- Use to store and query intervals
 - A file-mapped `vm_area_struct` can be considered as an interval of file pages. We store all `vm`s that map a file in a `prio_tree`, then we can execute a query like: find a set of `vm`s that map a single or a set of contiguous file pages
- Radix tree + heap tree
- Time complexity: $O(\log n + m)$
 - "log n" indicates the height of the tree (maximum 64 in a 32 bit machine) and "m" represents the number of `vm`s that map the page(s).

Priority search tree (prio_tree) — indexes and rules

- Indexes

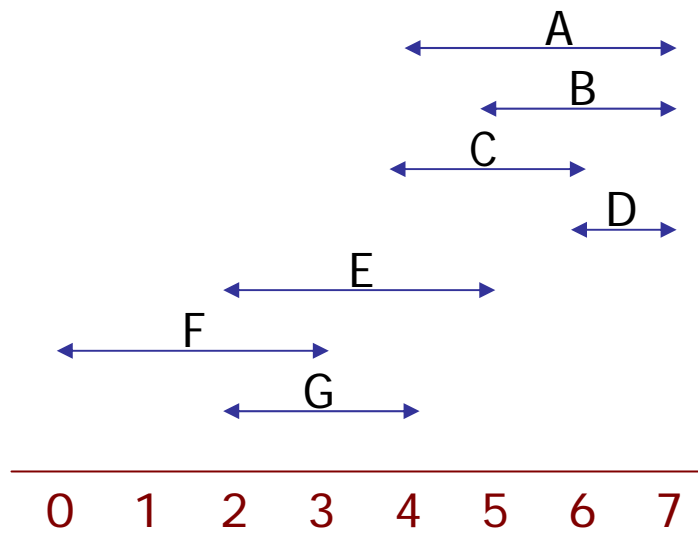
- heap_index: $vm_pgoff + vm_size_in_pages$ (end)
- radix_index: vm_pgoff (start)

- Rules

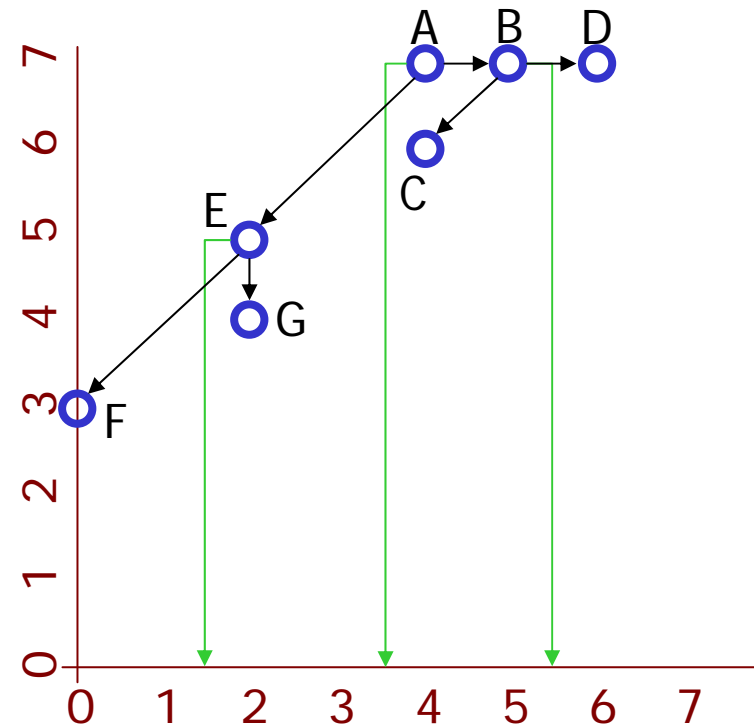
- heap_index of parent \geq heap_index of any direct child
- If heap_index of parent $==$ heap_index of any child, then
radix_index of parent $<$ radix_index of that child
- Nodes are hashed to left or right subtree using
radix_index similar to a pure binary radix tree

Priority search tree example

7 VMAs maps different intervals of a file
(radix_index, heap_index)



For easier understanding, we build the PST on a 2D coordinate. A VMA with an interval from 4 to 7 has a point (4,7) on the plane



Querying a priority search tree

- A query of all the intervals overlapping $[a,b]$?
 - If the tree is empty, we return NULL
 - Let R be the root of the tree, (x, y) be its coordinate, and $P(R)$ be the value separating the X -ranges of R 's child subtrees
 - Compare y to a . if $y < a$, we return without finding any intervals (all other nodes in the tree will have an even smaller Y -coordinate)
 - If $a \leq y$ and $b \geq x$, report the root interval
 - Recursively search the left subtree of R
 - If $P(R) < y$, recursively search the right subtree of R



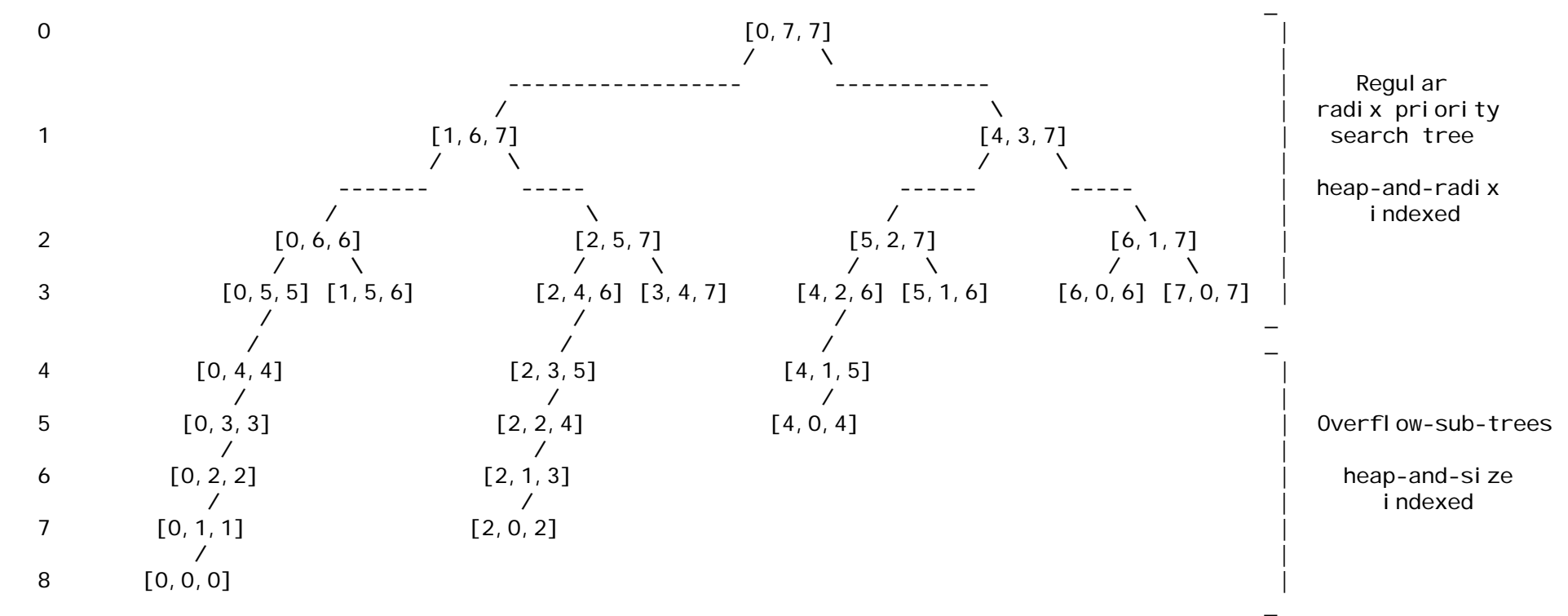
Extending priority search tree

- A regular priority search tree is only suitable for storing vmas with different radix indices (`vm_pgoff`)
- Solution
 - All vmas with the same radix and heap indices are linked `vm_set.list`
 - If there are many vmas with the same radix index, but different heap indices and if the regular priority search tree cannot index them all, we build an overflow subtree that indexes such vmas using heap and `size` (`vm_size_in_pages`) indices instead of heap and radix indices.

Overflowed Priority search tree

vmas are represented [radix_index, size_index, heap_index]
i.e., [start_vm_pgoff, vm_size_in_pages, end_vm_pgoff]

Level prio_tree_root->index_bits = 3





Unmapping a page frame

- `try_to_unmap(page)`
 - Try to remove all page table mappings to a page. According to the page type, it invokes `try_to_unmap_file()` or `try_to_unmap_anon()`
- `try_to_unmap_file(page)`

```

vma_prio_tree_foreach(vma, &iter,
    &mapping->i_mmap, pgoff, pgoff) {
    ret = try_to_unmap_one(page, vma);
    if (ret == SWAP_FAIL || !page_mapped(page))
        goto out;
}
    
```

- `try_to_unmap_anon(page)`

```

list_for_each_entry(vma, &anon_vma->head, anon_vma_node) {
    ret = try_to_unmap_one(page, vma);
    if (ret == SWAP_FAIL || !page_mapped(page))
        break;
}
    
```



Check if a page is referenced?

- `page_referenced()`
 - Test if a page was referenced by checking `PG_referenced` flag in `struct page` and associated page table entries.
- `page_referenced_anon()`
 - Go through `anon_vma` linked list and return the number of PTEs with accessed bit on
- `page_referenced_file()`
 - Go through `i_mmap` priority search tree and return the number of PTEs with accessed bit on

Swapping in pages – do_swap_page()

- Invoked when page is not present and page table entry is not null (recall slide no.50)
- It executes these steps:
 1. Get the swapped-out identifier and invoke `lookup_swap_cache()` to find the page in swap cache. If found, jump to step 3
 2. Invoke `read_swap_cache_async()` to swap in the page
 3. Invoke `mark_page_accessed()`, lock the page (will block if page I/O is in flight) and acquire `page_table_lock`
 4. Check if the page has been swapped in by other process. If true, release `page_table_lock`, unlock the page and return 1 (minor fault)
 5. Invoke `swap_free()` to decrement the usage counter of the page slot
 6. If swap cache is more than 50% full and the page is owned only by the process that caused the fault, remove the page from swap cache
 7. Increment `mm->rss` and unlock the page
 8. Update the page table entry with the physical address of the requested page and invoke `page_add_anon_rmap()` to add reverse mapping
 9. If this is a write access and the page is read-only shared among several processes, avoid another C-O-W fault by invoking `do_wp_page()`
 10. Unlock `page_table_lock`



Page Frame Reclaim

- When? (recall slide 33)
 - Reclaim memory when you cannot allocate memory from buddy system
 - Reclaim memory when kernel finds that memory is low
- From where?
 - Slab cache
 - Page cache, buffer cache
 - Pages belong to user processes

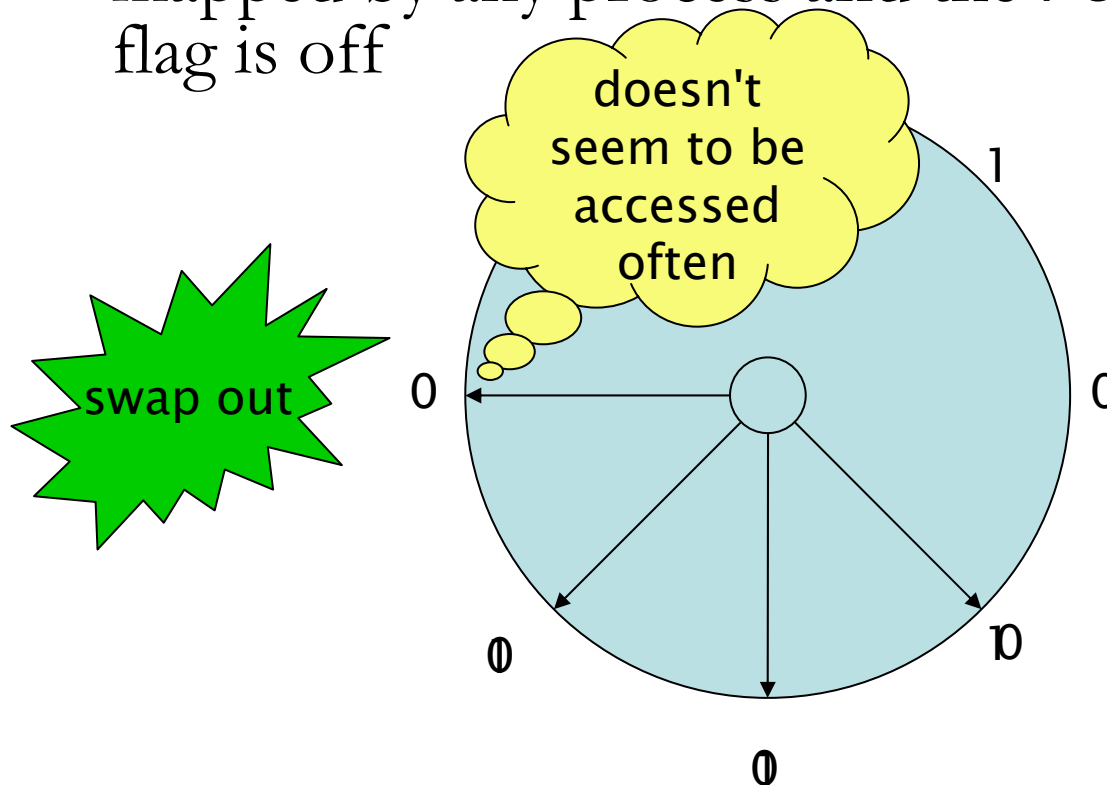
Page frame reclaiming algorithm consideration

- Ordering of pages based on ageing
 - Least recently used pages should be freed before pages accessed recently
- Distinction of pages based on the page state
 - Nondirty pages are better candidates than dirty pages for swapping out
- Don't write out tons of dirty pages just to reclaim a few pages (*)
 - Systems may be dealing mostly with dirty pages. Dirty pages with pageout I/O started should be reclaimed as soon as possible after the I/O on them are finished. It's nonsense to write out one gigabyte of data just for reclaiming 10 megabyte of memory.
- Balancing file cache vs. anonymous memory (*)
 - The amount of pages caching files tends to be several magnitudes larger than that those taken by processes. Systems can end up evicting frequently accessed pages from memory in favor of a mass of recently but far less frequently accessed pages

* Since Linux 2.6

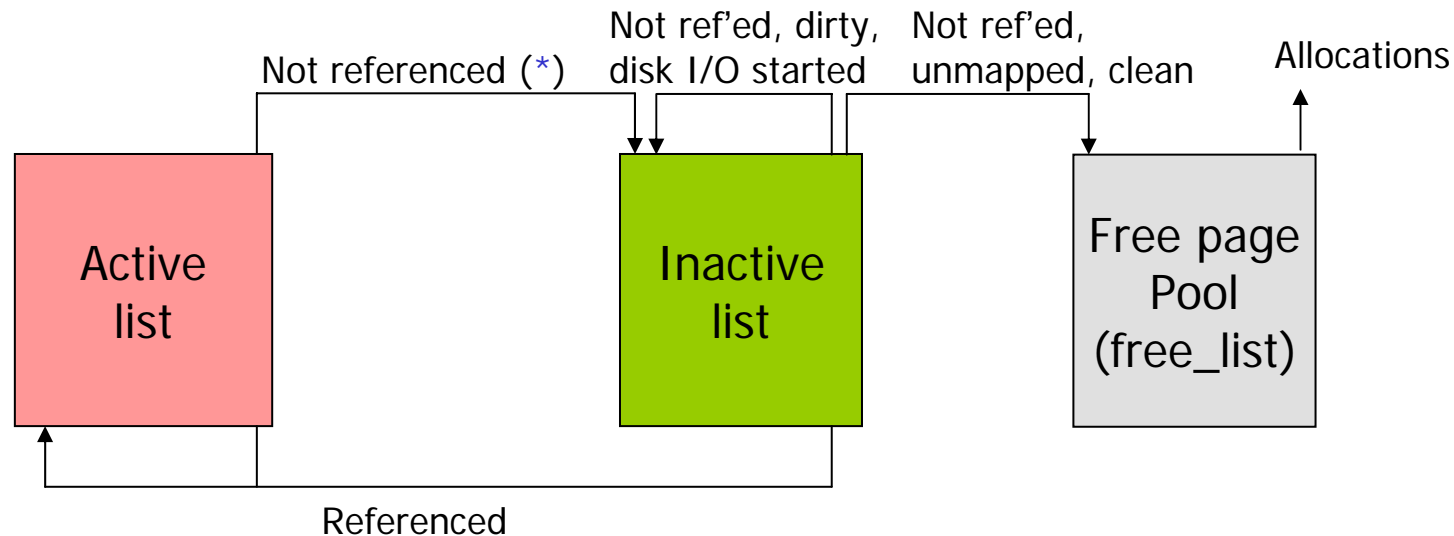
Second-chance (clock) page replacement algorithm

- Used in Linux 2.0, 2.2
- Scan `mem_map` array and reclaim a page if it is not mapped by any process and the `PG_referenced` flag is off



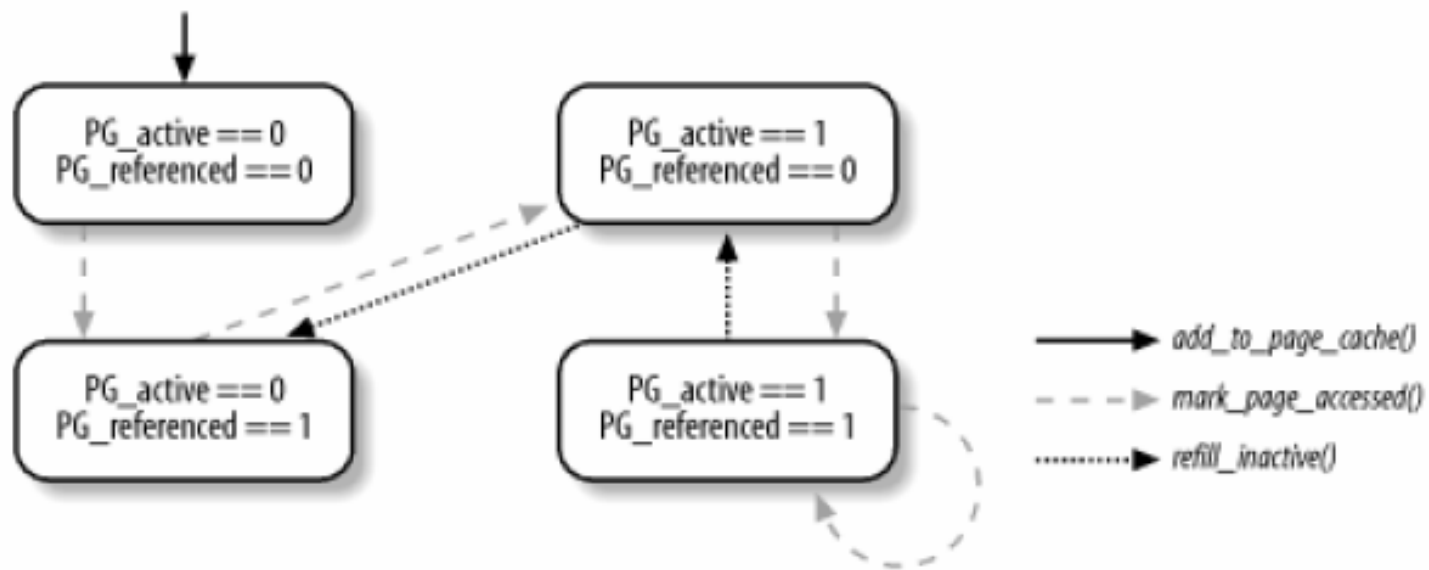
Mach-style page replacement algorithm

- Used in Linux 2.4, 2.6
- Maintain an active list and an inactive list per zone

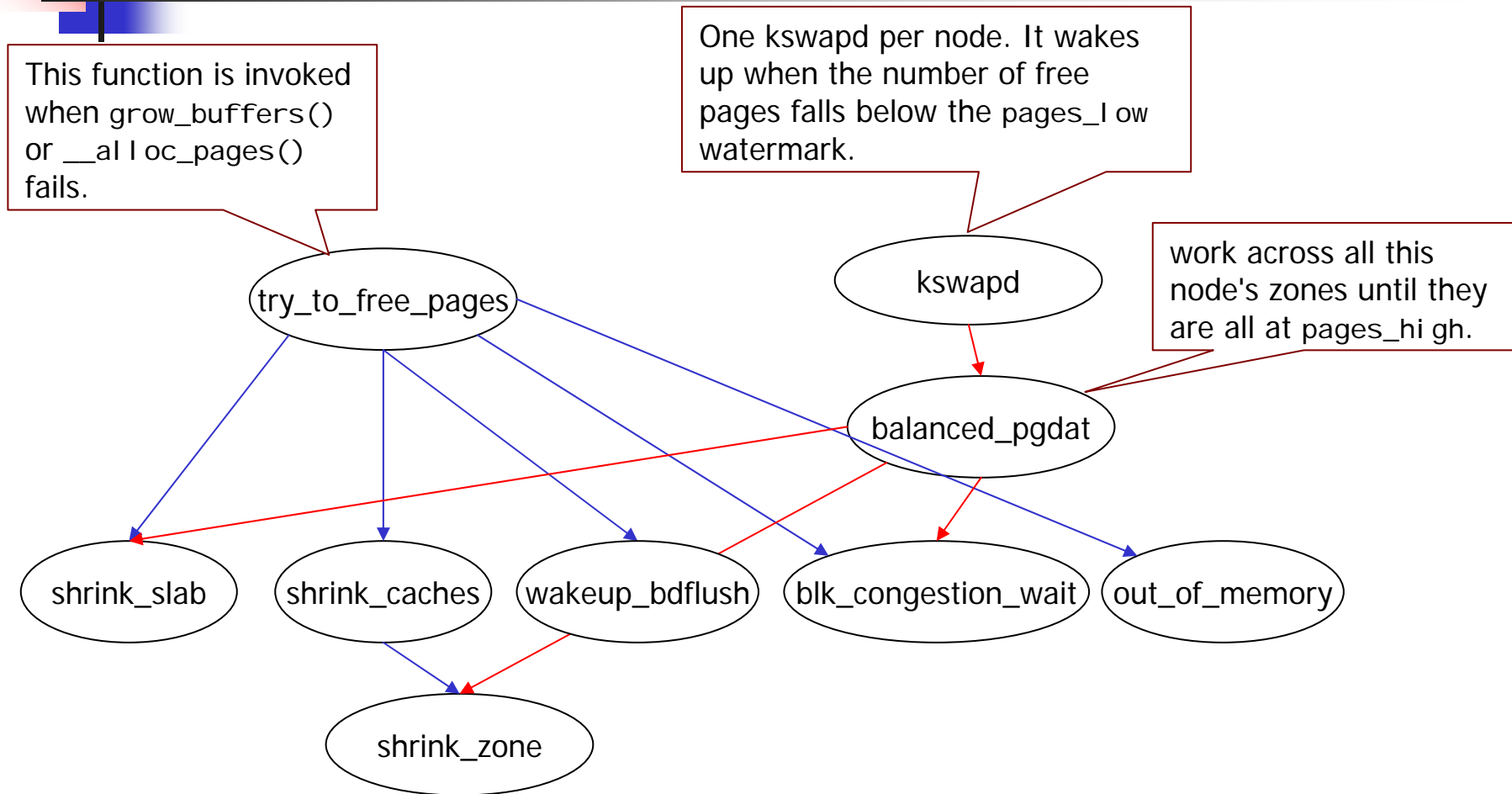


* Filter out anonymous pages when some conditions met
Also, limit the number of pages per move to slowly sift through the active list

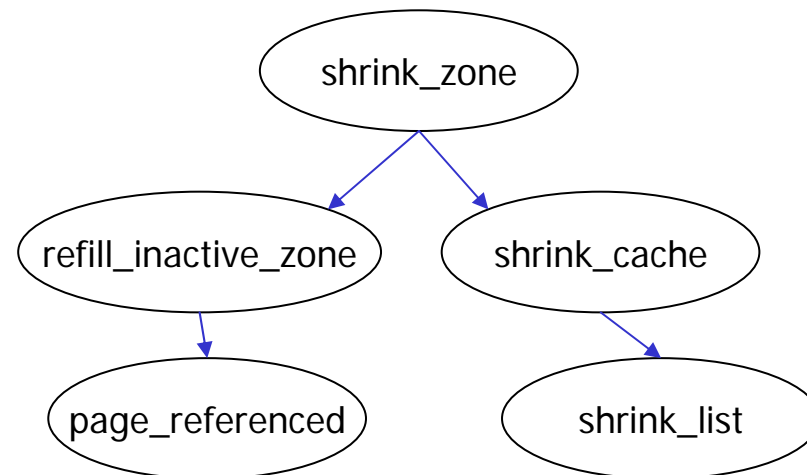
Moving pages across the LRU lists



Function overview



shrink_zone()



shrink_list()

