



Memory management

Page cache

Page frame reclaiming

Swapping

Swap cache



Table of contents

- Why caching?
- Page cache
 - Structure `address_space`
 - Radix tree (XArray)
 - Buffering blocks in page cache
- Writing dirty pages to disk
- Page frame reclaiming
 - LRU lists of active and inactive pages
 - Checking memory demand and *kswapd*
 - Unmapping process pages
- Swapping
 - Swap cache



Why caching?

Latency numbers

L1 cache reference	0.5 ns	
Branch mispredict	5 ns	
L2 cache reference	7 ns	
Mutex lock/unlock	25 ns	
Main memory reference	100 ns	
Compress 1K bytes with Zippy	3,000 ns	= 3 μ s
Send 2K bytes over 1 Gbps network	20,000 ns	= 20 μ s
SSD random read	150,000 ns	= 150 μ s
Read 1 MB sequentially from memory	250,000 ns	= 250 μ s
Round trip within same datacenter	500,000 ns	= 0.5 ms
Read 1 MB sequentially from SSD*	1,000,000 ns	= 1 ms
Disk seek	10,000,000 ns	= 10 ms
Read 1 MB sequentially from disk	20,000,000 ns	= 20 ms
Send packet CA->Netherlands->CA	150,000,000 ns	= 150 ms

[Latency numbers every programmer should know](#)



Why caching?

Humanized version (x 1,000,000,000)

L1 cache reference	0.5 s	One heart beat (0.5 s)
Branch mispredict	5 s	Yawn
L2 cache reference	7 s	Long yawn
Mutex lock/unlock	25 s	Making a coffee
Main memory reference	100 s	Brushing your teeth
Compress 1K bytes with Zippy	50 min	One episode of a TV show (including ad breaks)
Send 2K bytes over 1 Gbps network	5.5 hr	From lunch to end of work day
SSD random read	1.7 days	A normal weekend
Read 1 MB sequentially from memory	2.9 days	A long weekend
Round trip within same datacenter	5.8 days	A medium vacation
Read 1 MB sequentially from SSD	11.6 days	Waiting for almost 2 weeks for a delivery
Disk seek	16.5 weeks	A semester in university
Read 1 MB sequentially from disk	7.8 months	Almost producing a new human being
The above 2 together	1 year	
Send packet CA->Netherlands->CA	4.8 years	Average time it takes to complete a bachelor's degree

[Latency numbers every programmer should know](#)



Mapping file to memory and page cache

In Linux **read()** and **write()** operations are implemented via **mapping files to memory**.

The content of the **mapped file** is loaded to memory on-demand when handling **page faults** and cached in so-called **page cache**.

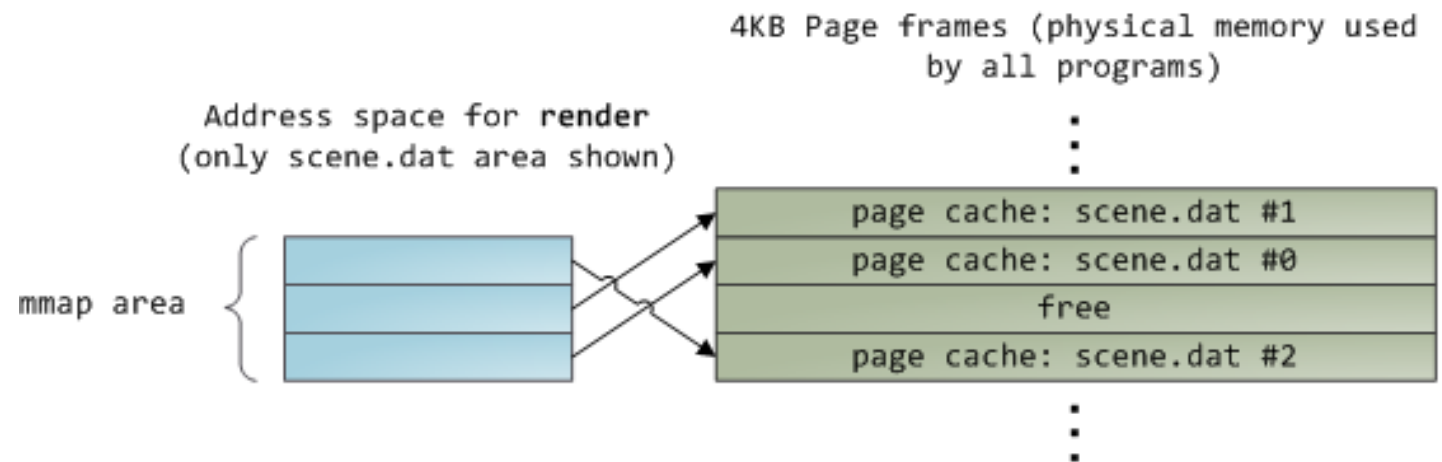
The kernel maps the **process virtual address space** pages directly to the **page cache** after loading the file contents into the **page frame**.

The pages are cached even after the program has finished. As long as there is enough free memory, the **cache size will increase**.

When writing with **write()**, the bytes are cached and the page is marked as **dirty**. The actual write to disk takes place later.

The program does not have to wait for the **write()** to finish (**fsync()** forces immediate write).

Reading is **blocking**, which is why the kernel tries to **read ahead**





Page cache – private or shared

A file mapping may be **private** (the updates are not committed to disk or made visible to other processes) or **shared** (the updates are visible to other processes).

The **read-only page table entries** mean, that a kernel tries to ensure **file sharing** as long as possible.

A virtual page that maps a file **privately** sees changes done to the file by other programs *as long as the page has only been read from*.

Once **copy-on-write** is done, changes by others are no longer seen.

A **shared mapping** is simply mapped onto the page cache. Updates are visible to other processes and end up in the disk.

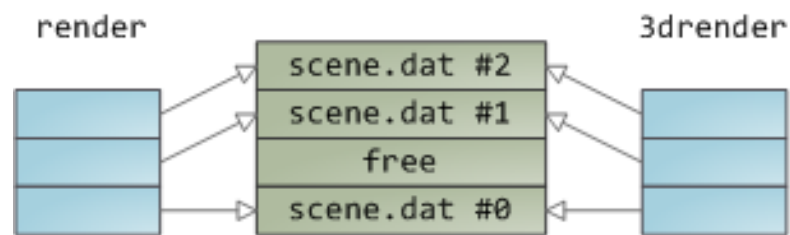
If the mapping were **read-only**, page faults would trigger a **segmentation fault** instead of **copy on write**.

Dynamically loaded libraries are brought into your program's address space via file **private mapping**.

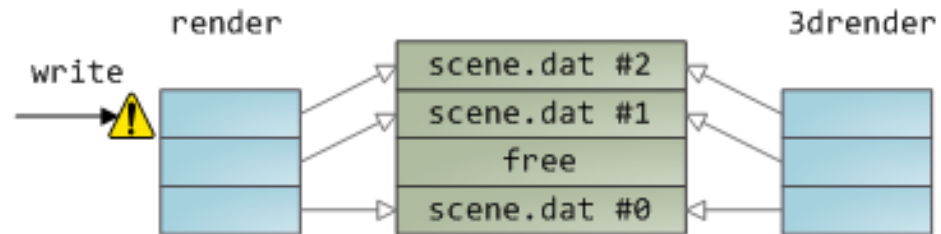
In the example both **render** and another program **render3d** map the file privately . **Render** then writes to its virtual memory area that maps the file.

————▷ Page table entry marked read-only
————▶ Page table entry marked read/write

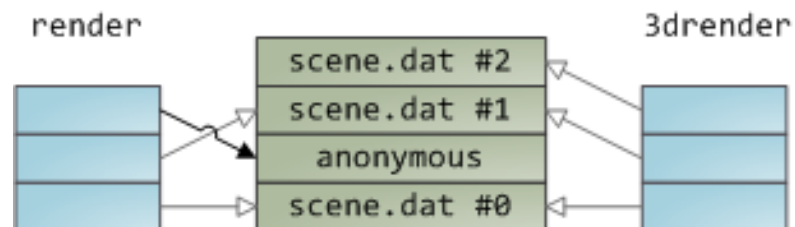
1. Two programs map scene.dat privately. Kernel deceives them and maps them both onto the page cache, but makes the PTEs read only.



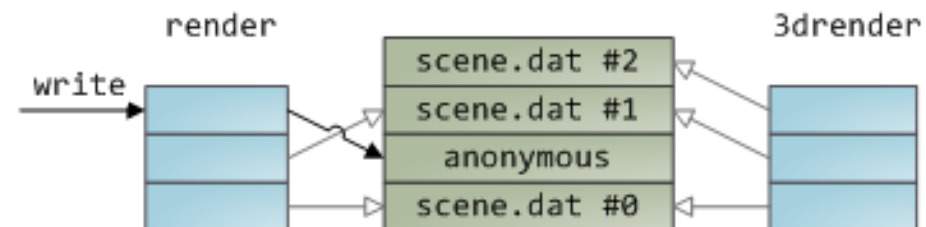
2. Render tries to write to a virtual page mapping scene.dat. Processor page faults.



3. Kernel allocates page frame, copies contents of scene.dat #2 into it, and maps the faulted page onto the new page frame.

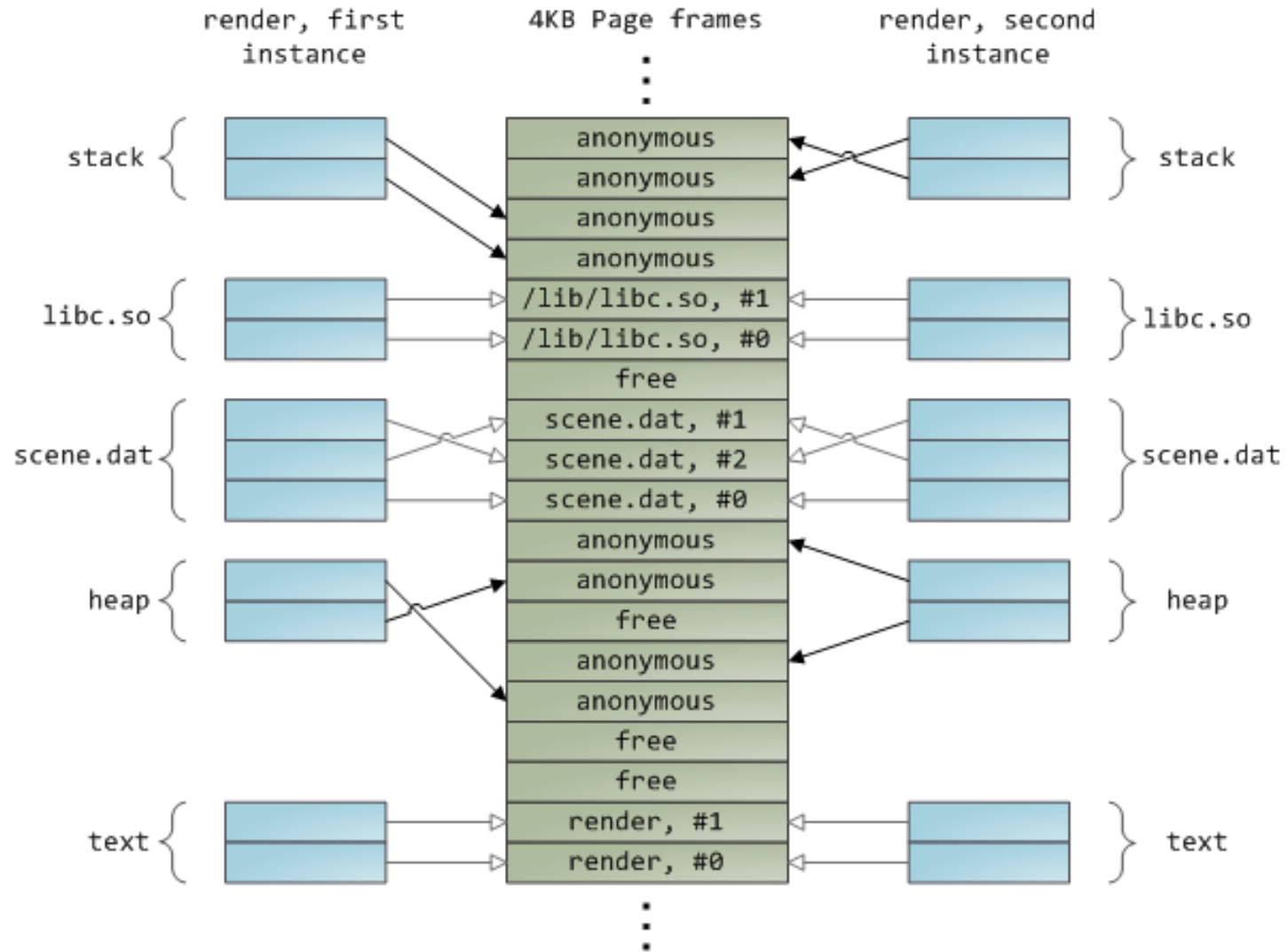


4. Execution resumes. Neither program is aware anything happened.





Page cache



Mapping a file in two instances of the same program (source: Duarte, [Software Illustrated](#))



Page cache

The **kernel's** code and data structures are **never swapped**, so they do not need to be read from or written to disk.

The following types of pages are stored in the **page cache**:

- pages containing **regular file** data;
- pages containing **directories**;
- pages containing data **read directly** from **block devices** (bypassing the VFS layer);
- pages belonging to files from **special file systems** (e.g. SHM used to support *shared memory segments* – IPC mechanism).

In any case, this data comes from a **file**. This file, or more precisely its descriptor (i.e. inode) is called the **page owner**.

Anonymous memory is not handled by the page cache.

If a system provides swap and if anonymous memory is swapped – it enters the **swap cache**, not the page cache.



Page cache

Almost all **read()** and **write()** operations go through the **page cache** (except for files open with the **O_DIRECT** flag – in this case buffers in the process address space are used; this mechanism is used by database applications that want to implement their own caching algorithms).

Page cache is designed in such a way that:

1. you can quickly find in the **cache pages** coming from a **specific file**,
2. to take into account the need to perform **different read** and **write** operations, depending on the location of the file.

The unit of information stored in the page cache is a page that **does not need** to include **physically adjacent disk blocks**, so it cannot be identified by providing the logical number of the device and the block number.

The page identifier will be: the **page owner** and the **index** within the owner's data (usually just the **inode** of the file and **offset** within the file).



Structure address_space

The basic data structure of the **page cache** is **address_space**, contained in the **inode** of the file. The **page cache** can contain **multiple pages** from the **same file** that point to the **same inode**, all of them will be handled by the **same set of methods**.

```
struct address_space { Simplified version
    struct inode      *host;      /* owner: inode or block_device */
    struct xarray      i_pages;    /* cached pages */
        // struct radix_tree_root page_tree;    /* radix tree of all pages */
        // spinlock_t      tree_lock;    /* and lock protecting it */
    atomic_t          i_mmap_writable; /* number of VM_SHARED mappings */
    struct rb_root_cached i_mmap;    /* tree of private and shared mappings */
    struct rw_semaphore i_mmap_rwsem; /* protect tree, count, list */
    unsigned long      nrpages;    /* number of page entries */
    struct address_space_operations *a_ops; /* methods */
    struct list_head    private_list; /* for use by the owner of the address_space */
    void               *private_data; /* for use by the owner of the address_space */
};
```

Comment in the [code](#) (Matthew Wilcox, March 13, 2018)

*Remove the **address_space ->tree_lock** and use the **xa_lock** newly added to the **radix_tree_root**.*

*Rename the **address_space ->page_tree** to **->i_pages**, since we don't really care that it's a tree.*



Linux radix tree and XArray

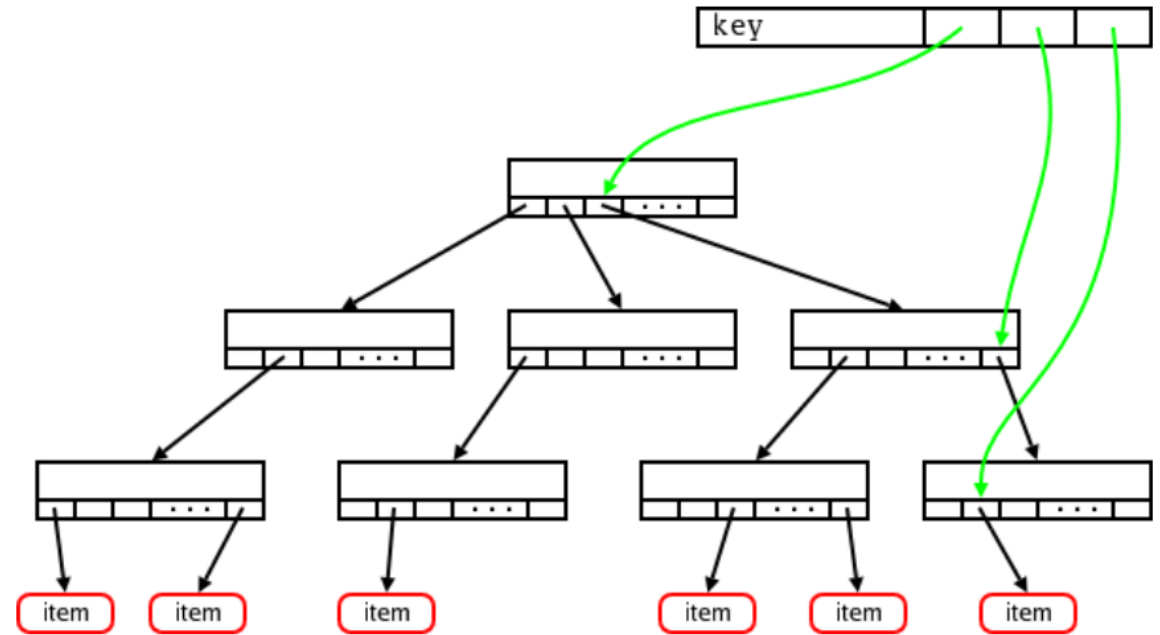
[Radix trees](#) (Jonathan Corbet, March 2006)

Each node has **64** slots

Slots are indexed by a **6-bit** ($2^6=64$) portion of the key.

At leaves, a slot points to an address of data.

At non-leaf nodes, a slot points to another node in a lower layer.



Xarray is a nicer **API** for **radix tree**.

[The XArray data structure](#), J. Corbet, January 2018

An automatically resizing array of pointers. Indexed by an unsigned long. Embeds a spinlock. Loads are store-free using RCU.

[4.20 Merge window part 2](#) (November 5, 2018)

*The **XArray** data structure, a reworking of the **radix tree structure**, has been merged at last and the **page cache** has been converted to use it.*



XArray



Xarray – referenced in 124 files (v6.3.2)

Additional reading:

- [XArray](#) (in /Documentation/core-api/xarray.rst, Matthew Wilcox, 2018).

*The **XArray** is an **abstract data type** which behaves like a **very large array of pointers**. It meets many of the same needs as a **hash** or a **conventional resizable array**. Unlike a hash, it allows you to sensibly go to the next or previous entry in a **cache-efficient manner**. In contrast to a resizable array, there is **no need to copy data** or **change MMU mappings** in order to grow the array. It is more memory-efficient, parallelisable and cache friendly than a **doubly-linked list**. It takes advantage of **RCU** to perform lookups without locking.*

- [The XArray data structure](#), Jonathan Corbet, January 2018.
- [Willy's memory-management to-do list](#), Jonathan Corbet, April 2018.
- [XArray and the mainline](#), Jake Edge, June 2018.
- [The design and implementation of the Xarray](#), Matthew Wilcox, LCA, January 2018.
- [Xarray: one data structure to rule them all](#), Matthew Wilcox, LCA, January 2019.



Structure address_space

The **private_list** field is the head of a generic list that can be freely used by the filesystem for its specific purposes. For example, the Ext2 filesystem makes use of this list to collect the **dirty buffers** of **indirect blocks** associated with the inode. When a flush operation forces the inode to be written to disk, the kernel flushes also all the buffers in this list.

Each **page descriptor** contains **mapping** and **index** fields that link the **page** with the **page cache**. The first points to the **address_space** object, and the second to the **offset** (page **index**) within the address space of the **page owner**.

The page cache may contain **multiple copies of the same disk data**. For example, the same 4 KB block of data of a regular file can be accessed as follows:

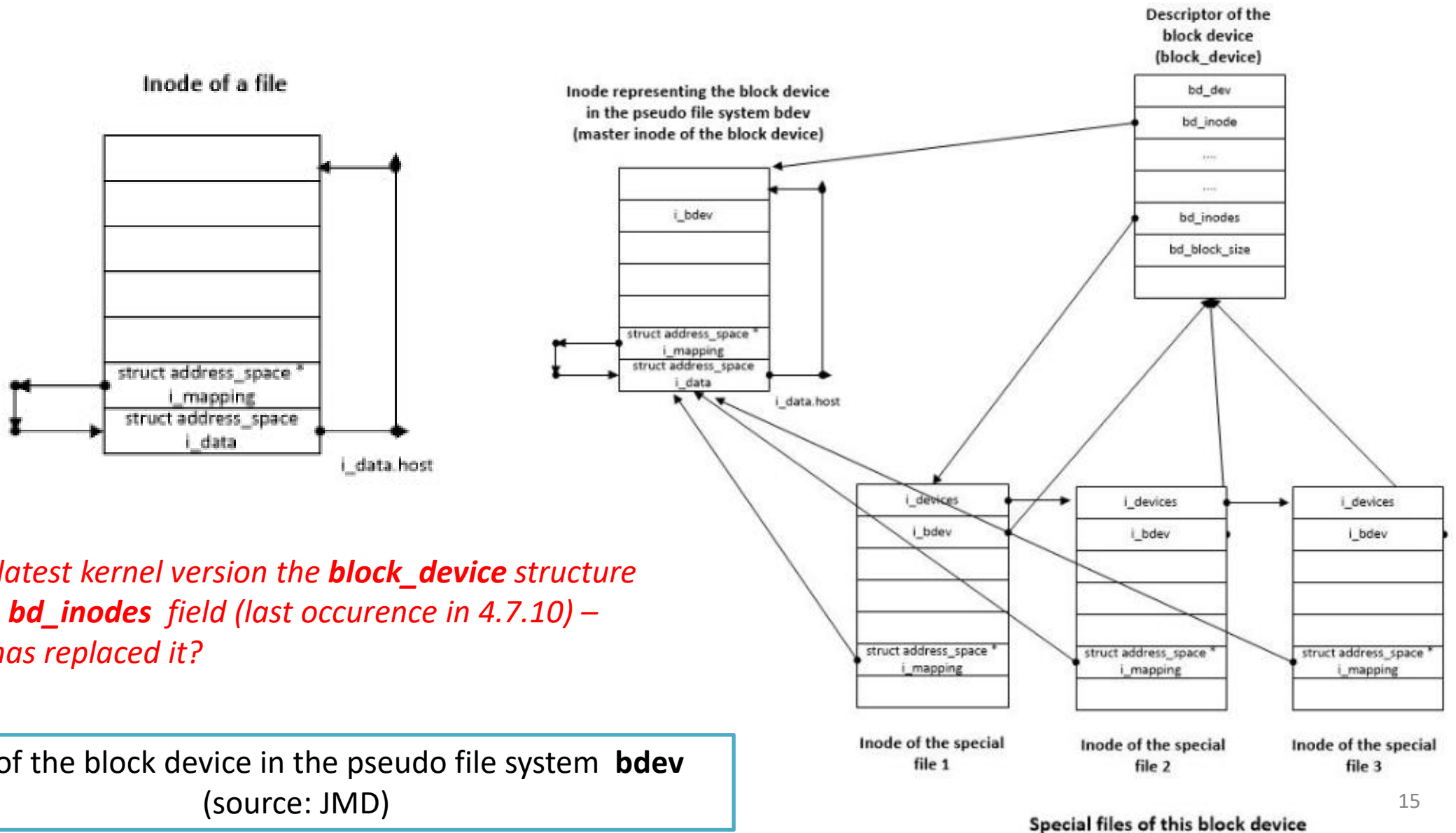
- by **reading** the **file** – the data is included in a page owned by the **regular file's inode**;
- by **reading** the **block** from the **device file** (disk partition) that hosts the file – the data is included in a page owned by the **master inode** of the **block device file**.

Thus, the same data appears in two different pages referenced by two different **address_space** objects.



Inode of a file, inode of a block device file, and the address_space object

SKIP

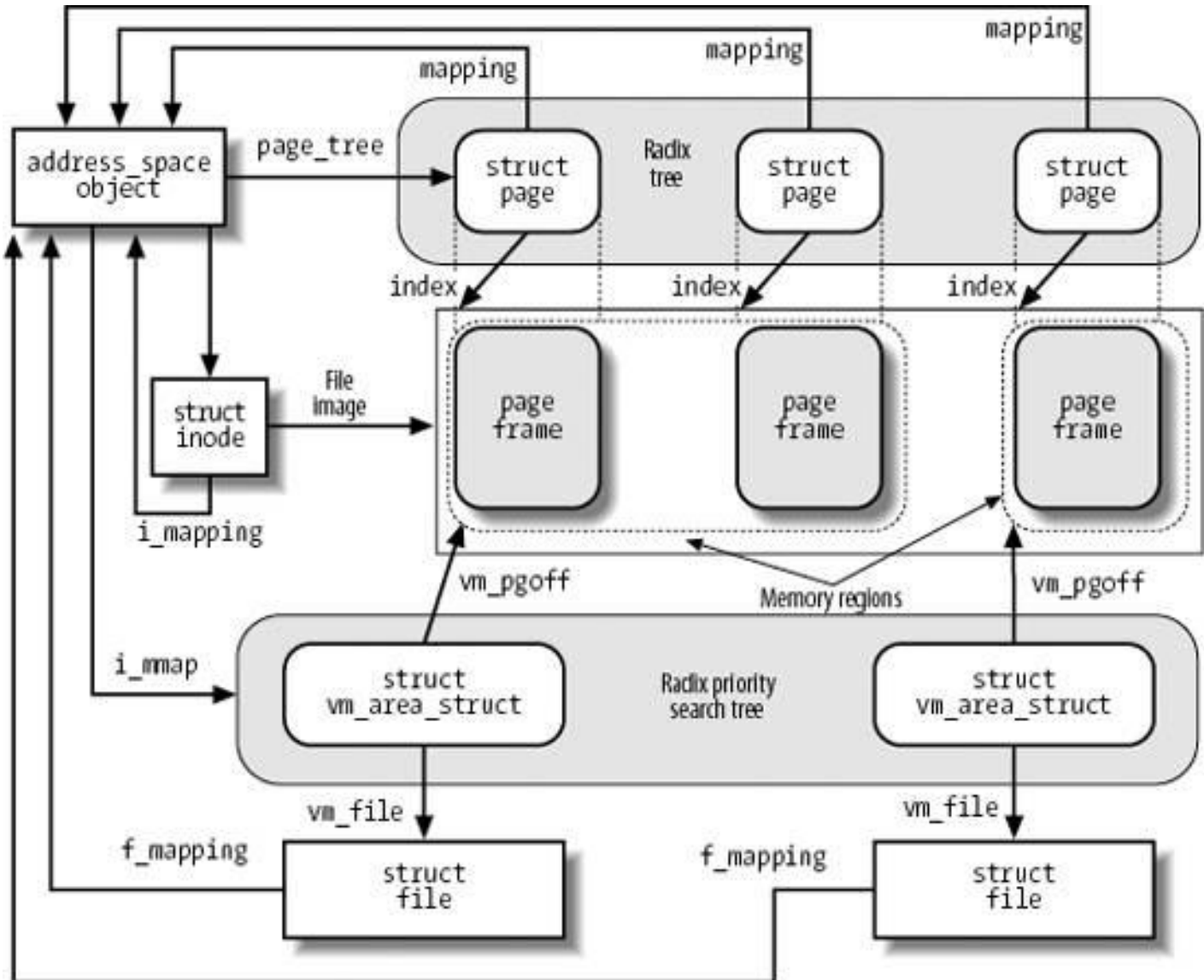




Data structures for file memory mapping

Radix priority search tree has been replaced (2012) by the [interval tree](#), built on top of the augmented rbtree API.

It contains **memory areas** that refer to **the same frame in memory** – that is, it allows you to quickly locate **shared pages**, those from **files**.

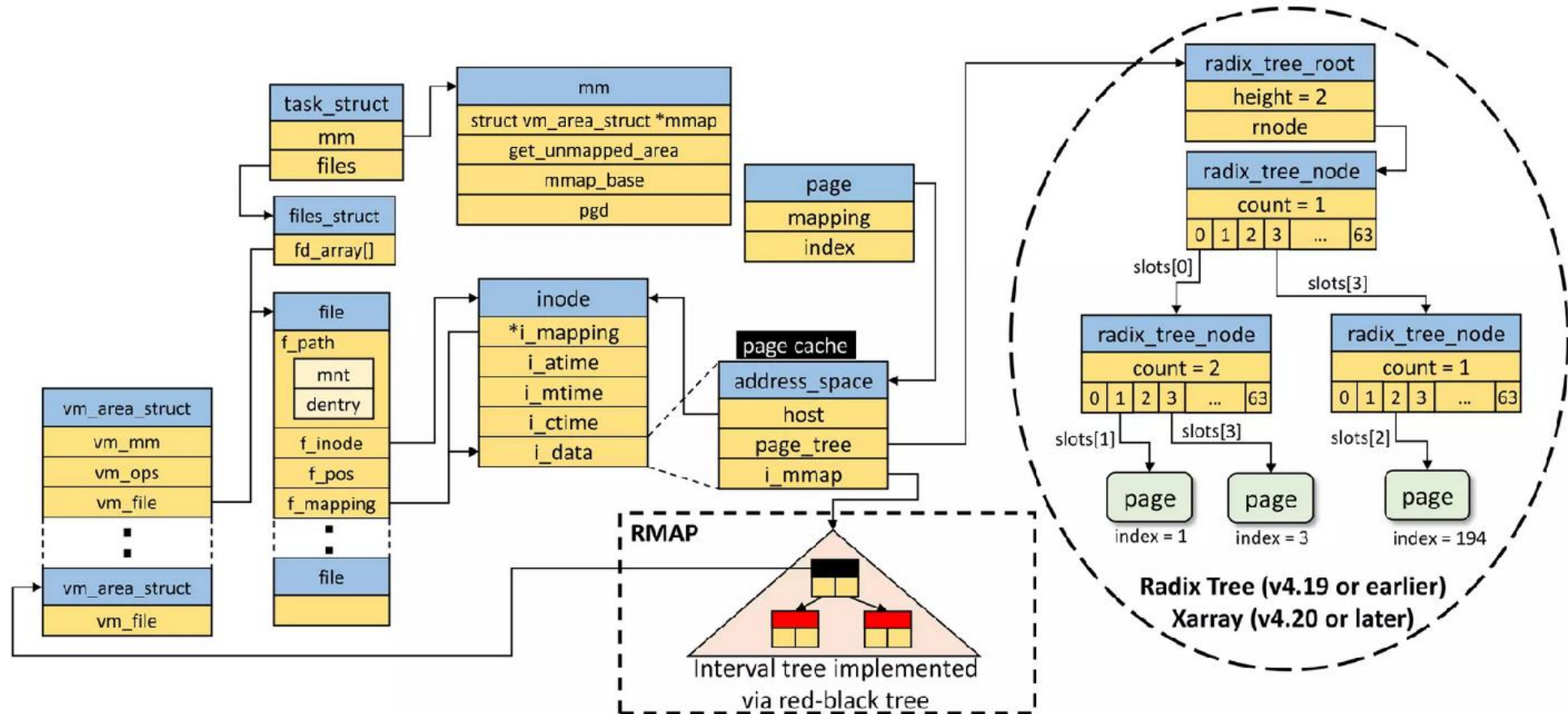


Source:
Bovet, Cesati, Understanding the
Linux Kernel)

Partly obsolete



Radix tree (or Xarray) – how to find a page i_mmap for recording all memory mappings (vma) of the memory-mapped file



1. **i_mmap**: Reverse mapping (RMAP) for the memory-mapped file (page cache) → check `__vma_link_file()`
2. **anon_vma**: Reverse mapping for anonymous pages → `anon_vma_prepare()` invoked during page fault



Additional reading

- [Lockless Page Cache](#), Kornilios Kourtis, 2013.
- [A multi-order radix tree](#), Ross Zwisler, May 2016.
- [Two transparent huge page cache implementations](#), Jonathan Corbet, April 2016.
- [The future of the page cache](#), Jonathan Corbet, January 2017.
- [The future of the Linux page cache](#), Matthew Wilcox, LCA January 2017.
- [Direct Access for files](#). DAX is the mechanism that enables direct access to files stored in persistent memory arrays without the need to copy the data through the page cache.
- [The future of DAX](#), Jonathan Corbet, March 2017.
- [Defending against page-cache attacks](#), Jonathan Corbet, January, 2019.
- [Large Pages in Linux](#), Matthew Wilcox, LCA 2020.



Storing blocks in the page cache

Starting from version 2.4.10, Linux does not support a separate **buffer cache**. **Block buffers** are no longer allocated individually; they are stored in dedicated pages called “**buffer pages**,” which are kept in the **page cache**.

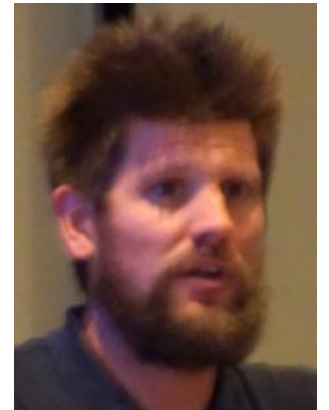
A **buffer page** is a page of data associated with additional descriptors called “**buffer heads**,” whose main purpose is to quickly locate the **disk address** of each individual block in the page.

The chunks of data stored in a page belonging to the page cache are **not** necessarily **adjacent** on disk.

Each block buffer has a **buffer head descriptor** of type **buffer_head**.

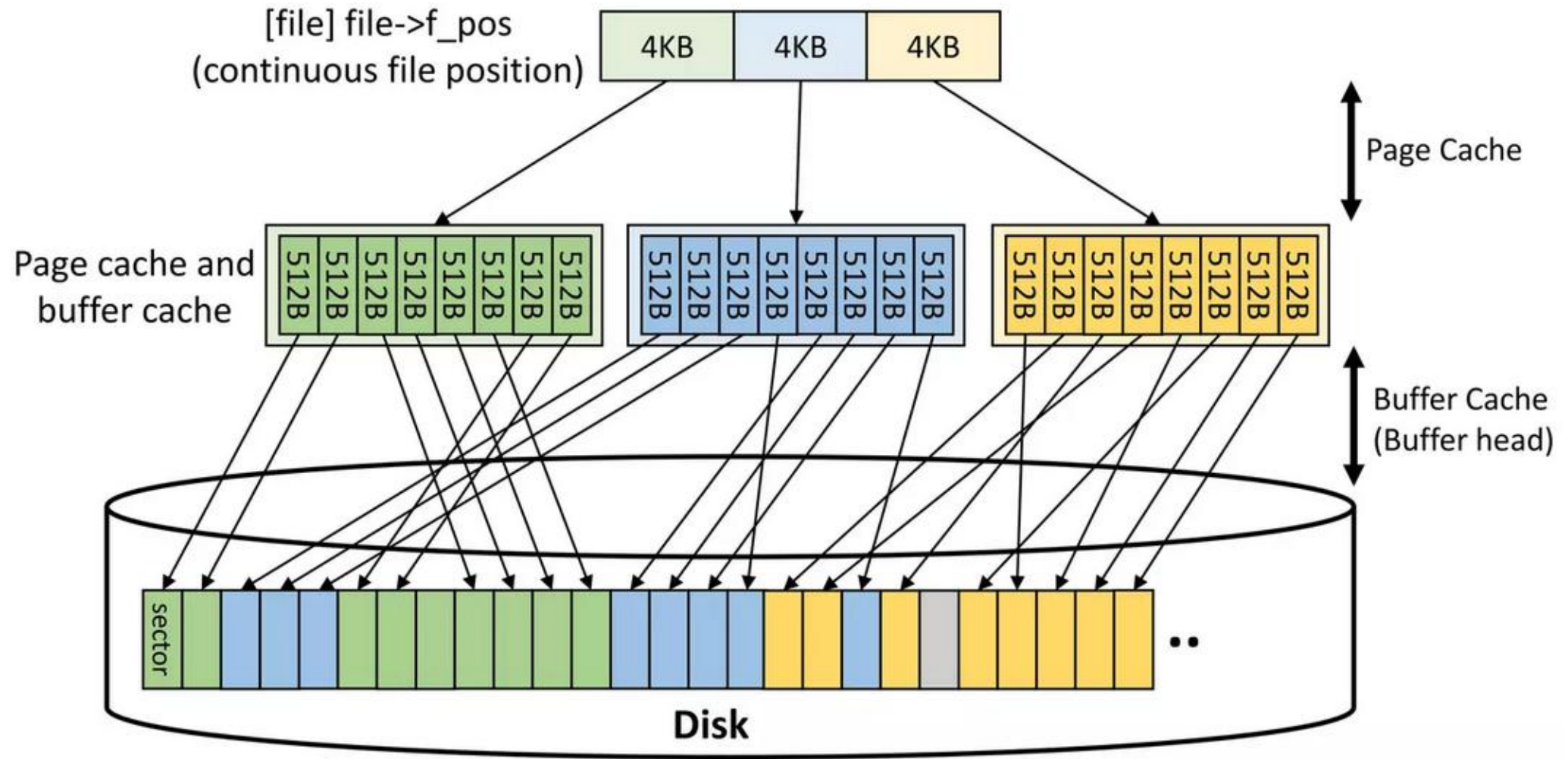
From the code: *Historically, a **buffer_head** was used to map a single block within a page, and of course as the unit of I/O through the filesystem and block layers. Nowadays the basic I/O unit is the **bio**, and **buffer_heads** are used for **extracting block mappings**, for **tracking state within a page** and for **wrapping bio submission for backward compatibility reasons**.*

[A kernel without buffer heads](#) (J. Corbet, May 2023) – Christoph Hellwig has posted [a patch series](#) that enables the building of a kernel without buffer heads — but the cost of doing so at this point will be more than most want to pay.





Page cache & buffer cache

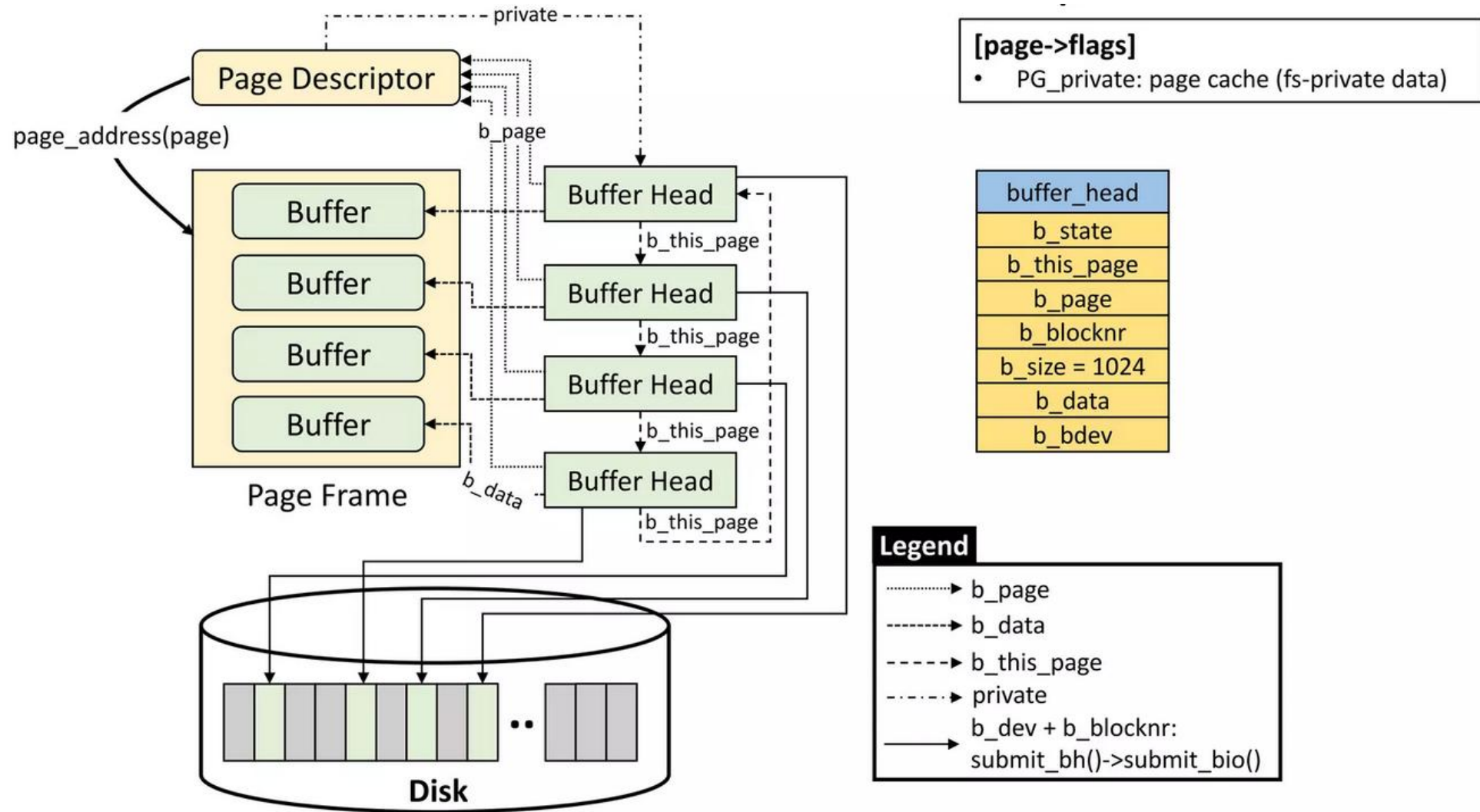


Page cache: interaction with VFS. Buffer cache: interaction with the disk.

(source: Adrian Huang, [Page cache](#), 2022)



Page cache & buffer cache – relationship

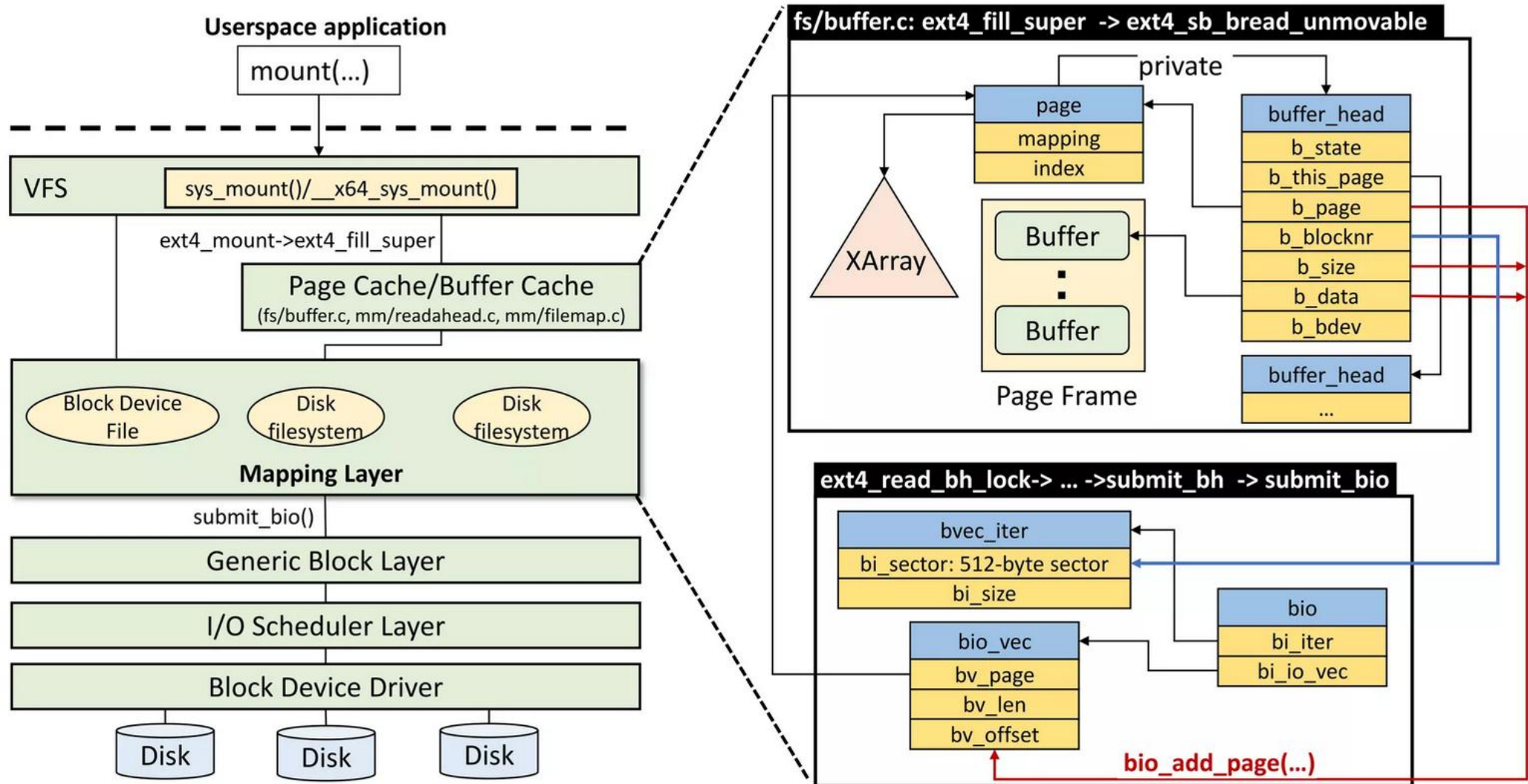


Block size = file system-based unit. Page cache might *NOT* include buffer_head struct.

(source: Adrian Huang, [Page cache](#), 2022)



Interaction with generic block layer: bio based on buffer_head





Storing blocks in the page cache

The **disk address of the block** is encoded in two fields: **b_bdev** (which identifies the block device) and **b_blocknr**, which stores the logical block number (index of the block inside the disk).

The **b_data** field specifies the position of the block buffer inside the buffer page.

The kernel **creates buffer pages** in two common cases:

1. **When reading or writing pages of a file that are not stored in contiguous disk blocks.**

The buffer page's descriptor is inserted in the **radix tree** of a **regular file**. The buffer heads are preserved because they store information that specify the position of the data in the disk.

If the blocks in the file are written in adjacent disk blocks, then a buffer page is not needed (the **PG_private** flag of the page frame descriptor indicates what situation we are dealing with).



Storing blocks in the page cache

2. When accessing a single disk block (for instance, when reading a superblock or an inode block).

The buffer page's descriptor is inserted in the radix tree rooted at the `address_space` object of the inode in the **bdev** special filesystem associated with the **block device**.

This kind of buffer pages must satisfy a constraint that all the block buffers must refer to **adjacent blocks** of the underlying **block device**.

An instance of where this is useful is when the VFS wants to read the 1 KB inode block containing the inode of a given file. Instead of allocating a single buffer, the kernel must allocate a whole page storing four buffers; these buffers will contain the data of a group of four adjacent blocks on the block device, including the requested inode block (in ext2fs inode has a size of 128 bytes, one 1 KB block contains 8 inodes, one 4 KB page contains 32 inodes).

Such buffer pages are called **block device buffer pages** or **blockdev pages**.

*All buffers within one page must be the same size; in 80x86 architecture, the buffer page can contain from **1 to 8 buffers**, depending on the block size.*



Writing dirty pages to disk

The kernel **keeps filling** the **page cache** with pages containing data of block devices. Whenever a process modifies some data, the corresponding page is marked as dirty – its **PG_dirty** flag is set.

Linux allows the **deferred writes** of **dirty pages** into **block devices**. Dirty pages are flushed to disk under the following conditions:

- The **page cache** gets too **full** and more pages are needed, or the number of dirty pages becomes too large.
- Too much **time** has elapsed since a page has **stayed dirty**.
- A process requests all pending changes of a block device or of a particular file to be flushed by invoking a **sync()**, **fsync()** or **fdatasync()**.

The **PG_dirty** flag of the **buffer page** should be set if **at least one** of the associated buffer heads has the **BH_Dirty** flag set. When the kernel selects a dirty buffer page for flushing, it scans the associated buffer heads and effectively writes to disk only the contents of the **dirty blocks**. As soon as the kernel flushes all dirty blocks in a buffer page to disk, it clears the **PG_dirty** flag of the page.



Writing dirty pages to disk – per-BDI flusher

Historically different kernel threads have been responsible for flushing dirty pages to disk:

bdflush, kupdate, pdflush.

In the 2.6.32 kernel version, the **pdflush** thread has been replaced by the **per-BDI writeback** (**Backing Device Info**) mechanism.

Writeback can be defined as the process of writing **dirty memory** from the page cache to the disk.

The amount of data that needs to be written can be huge – hundreds of MB, or even GB, and the work is done by the **pdflush** kernel threads when the amount of dirty memory surpasses the limits set in **/proc/sys/vm**.

The current **pdflush** system has disadvantages, specially in systems with multiple storage devices that need to write large chunks of data to the disk. This design has some deficiencies that cause poor performance and seekiness in some situations.

Jens Axboe in his patch set proposes a new idea of using **flusher threads per backing device info (BDI)**, as a **replacement for pdflush threads**. Unlike **pdflush** threads, **per-BDI flusher threads focus on a single disk spindle**. With per-BDI flushing, when the **request_queue** is congested, blocking happens on request allocation, avoiding request starvation and providing better fairness.

(We may come back to this topic later)



Additional reading



- [Linux Page Cache Basics](#), Thomas Krenn, 2009.
- [Per backing device writeback](#), Jens Axboe, 2009.
- [Flushing out pdflush](#), Goldwyn Rodrigues, April 2009.
- [Toward less-annoying background writeback](#), Jonathan Corbet, April 2016.
- [Background writeback](#), Jake Edge, May 2016.

How to get info:

- The number of megabytes of main memory currently used for the page cache is indicated in the **Cached column** of the report produced by the **free -m** command.

```
jmd@students:~$ free -m
```

	total	used	free	shared	buff/cache	available
Mem:	336356	80939	24458	4129	230958	248625
Swap:	4045	3698	347			

- The amount of dirty memory is listed in **/proc/meminfo**.
- [Documentation for /proc/sys/vm/](#).

The **sysctl** file in **/proc/sys/vm** can be used to tune the operation of the virtual memory subsystem and the writeout of dirty data to disk.



Page frame reclaiming

Linux does not impose a **limit** on the **total amount** of RAM assigned to the processes created by a **single user**.

Similarly, no **limit** is placed on the size of the many **disk caches** and **memory caches** used by the kernel.

When the system load is **low**, the RAM is filled mostly by the disk caches and the few running processes can benefit from the information stored in them. When the system load **increases**, the RAM is filled mostly by pages of processes and the caches are shrunk to make room for additional processes.

When there is not enough free memory in the system, the kernel tries to find a page frame containing unnecessary data (**page frame reclaiming**), then removes its content from memory.

The recovered page frames go to the **buddy system**. The kernel should not wait with recovery until the last possible moment, so it tries to maintain a **minimum level of free page frames** in the system.



Page frame reclaiming – types of pages

The **page frame reclaiming algorithm (PFRA)** distinguishes the following types of pages:

- **Unreclaimable** – no reclaiming allowed or needed:
 - Free pages included in the buddy system lists,
 - Reserved pages (with PG_reserved flag set),
 - Pages dynamically allocated by the kernel,
 - Pages in the kernel mode stacks of the processes,
 - Temporarily locked pages (with PG_locked flag set),
 - Memory locked pages (with VM_LOCKED flag set);
- **Swappable** – Save the page contents in a swap area:
 - Anonymous pages in user mode address spaces (e.g. stack or heap pages),
 - Mapped pages of **tmpfs** filesystem (e.g. pages of IPC shared memory);



Page frame reclaiming – types of pages cont.

- **Syncable** – Synchronize the page with its image on disk, if necessary:
 - Mapped pages in user mode address spaces,
 - Pages included in the page cache and containing data of disk files,
 - Block device buffer pages,
 - Pages of some disk caches (e.g. the inode cache);
- **Discardable** – nothing to be done:
 - Unused pages included in memory caches (e.g. slab allocator caches),
 - Unused pages of the dentry cache.

When reclaiming page frames, the kernel must also consider whether the pages are **shared** or are used by only one process (because COW or processes are mapping the same file). The kernel maintains special data structures for quickly identifying shared pages.



Page frame reclaiming – heuristics

Heuristics used to free frames:

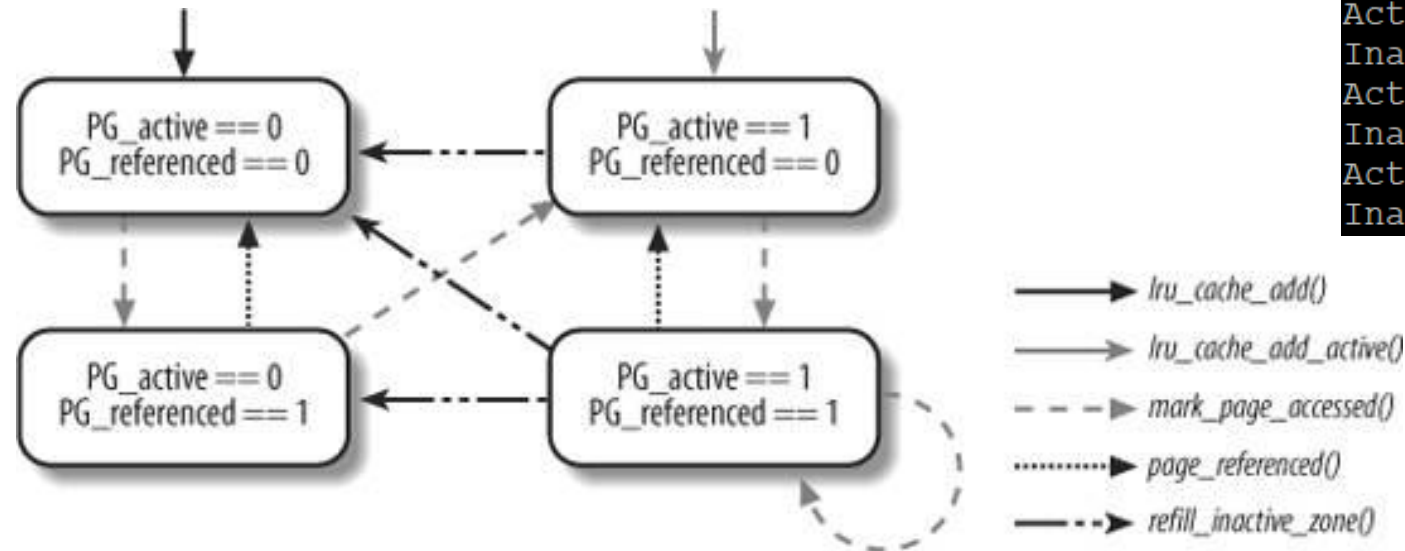
- Pages included in **disk** and **memory caches not referenced** by any process should be reclaimed before pages belonging to the **user mode address spaces** of the processes;
- With the **exception** of **locked pages**, the PFRA must be able to steal **any page** of a **user mode process**, including the anonymous pages. Processes that have been **sleeping** for a long period of time will progressively **lose** all their page frames.
- Reclaim a **shared** page frame by **unmapping** all page table entries that reference it.
- PFRA uses an approximation of the **LRU** replacement algorithm to classify pages as **in-use** and **unused**. If a page has not been accessed for a long time, it can be considered unused, otherwise as used. The **PFRA reclaims only unused pages**.
- Each page in RAM has a counter storing the **age** of the page. The 80x86 architecture do not offer such a hardware feature, thus Linux uses the **Accessed** bit included in each page table entry. The bit is **automatically set** by the **hardware** when the page is **accessed**. Moreover, the **age** of a **page** is represented by the **position** of the page descriptor in **LRU lists**.



LRU lists of active and inactive pages – overview

The **PG_referenced** flag in the page descriptor is used to **double** the number of accesses required to move a page from the **inactive** list to the **active** list; it is also used to **double** the number of “missing accesses” required to move a page from the **active** list to the **inactive** list.

If a page in the **inactive** list has the **PG_referenced** flag set to 0, the **first** page access sets the value of the **flag to 1**, but the page remains in the **inactive** list. The **second** page access finds the flag set and causes the page to be moved to the **active** list. Similarly when moving the page in the opposite direction.



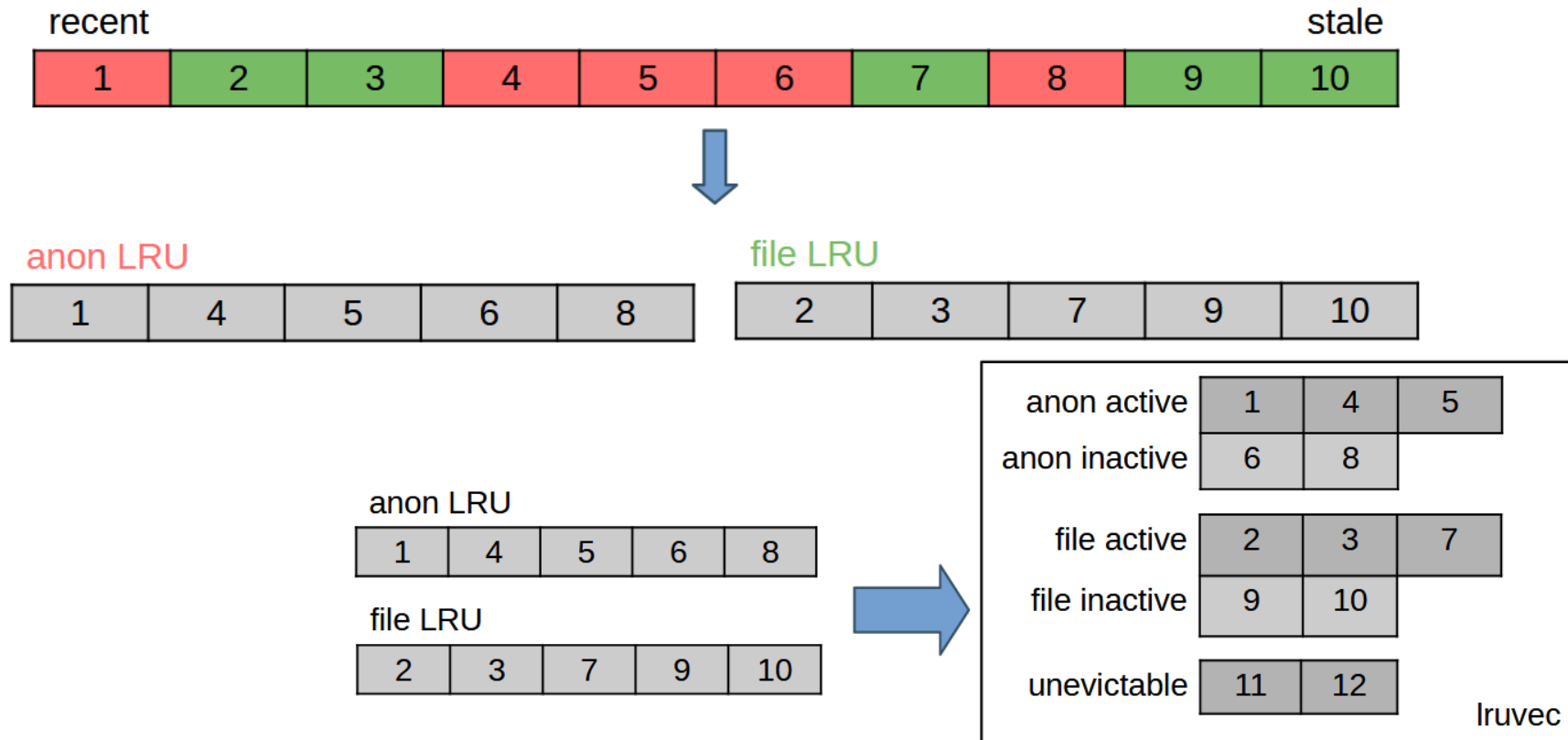
```
jmd@students:~$ grep -i active /proc/meminfo
Active:          57102528 kB
Inactive:        160420332 kB
Active(anon):    18520324 kB
Inactive(anon):  10496748 kB
Active(file):    38582204 kB
Inactive(file):  149923584 kB
```

`grep -i active /proc/meminfo`

Moving pages across the LRU lists
(source: Bovet, Cesati, Understanding the Linux Kernel)



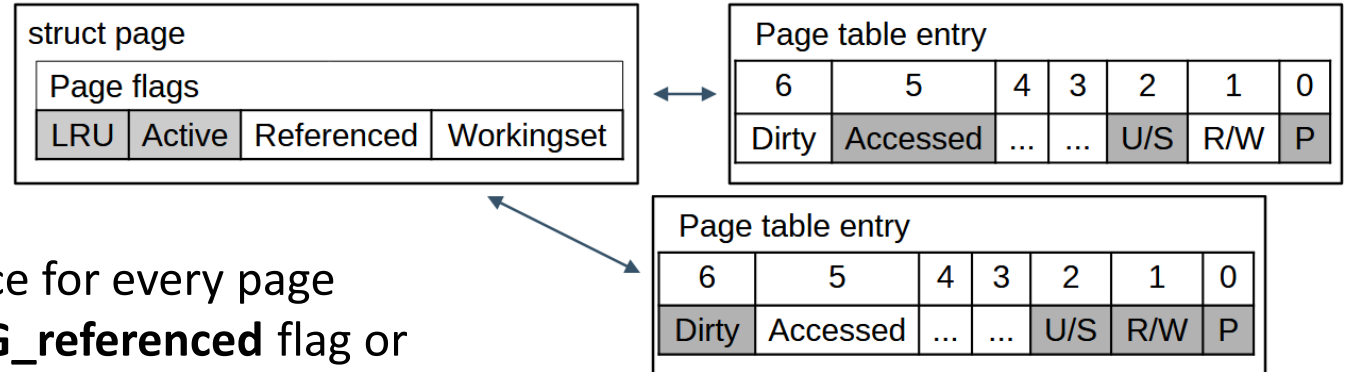
Overview of Memory Reclaim – LRU lists



	Root memcg	Memcg1	Memcg2	Memcg3	Memcg4	Memcg5
Node 0	lruvec	lruvec	lruvec	lruvec	lruvec	lruvec
Node 1	lruvec	lruvec	lruvec	lruvec	lruvec	lruvec



Overview of Memory Reclaim



The **page_referenced()** function is invoked once for every page scanned by the PFRA, returns **1** if either the **PG_referenced** flag or some of the **Accessed** bits in the **page table entries** was **set**; it returns **0** otherwise.

- Complex system, results of years of evolution, including big recent changes
 - No overall documentation (perhaps getting there? :)
- Many moving parts, hard to predict behavior, hard to evaluate patches!
 - Elaborate cost models applied only to 1/3 of decision space
 - OTOH, major decisions made by looking if a number has changed since last time
 - Explicit corner case heuristics against undesired feedback loops
 - Lots of suspicious details to look at in my TODO
 - We've seen issues (in older kernel) e.g. with file pages thrashing and anon not reclaimed
- How to get better insight? A simulation model?



LRU-list manipulation with DAMON (2022)

- **DAMON** (**D**ata **A**ccess **MON**itor) is a data access monitoring framework subsystem for the Linux kernel.
- Uses various heuristics to determine which **pages** of memory are in **active use**.
- Tries to create a clearer picture of **actual memory usage** while, at the same time, **limiting** its own **CPU usage**.
- It is designed to be **efficient enough** to use on production systems while being **accurate enough** to improve memory-management decisions.
- Merged into kernel **6.0** – DAMON can **actively reorder pages on the LRU lists**.
- SeongJae Park (author, **AWS**) calls this mechanism „**proactive LRU-list sorting**”.
- [DAMON and DAMOS: Writing a fine-grained access pattern oriented lightweight kernel module using DAMON/DAMOS in 10 minutes](#), SeongJae Park, Linux Plumbers Conference, 2021.

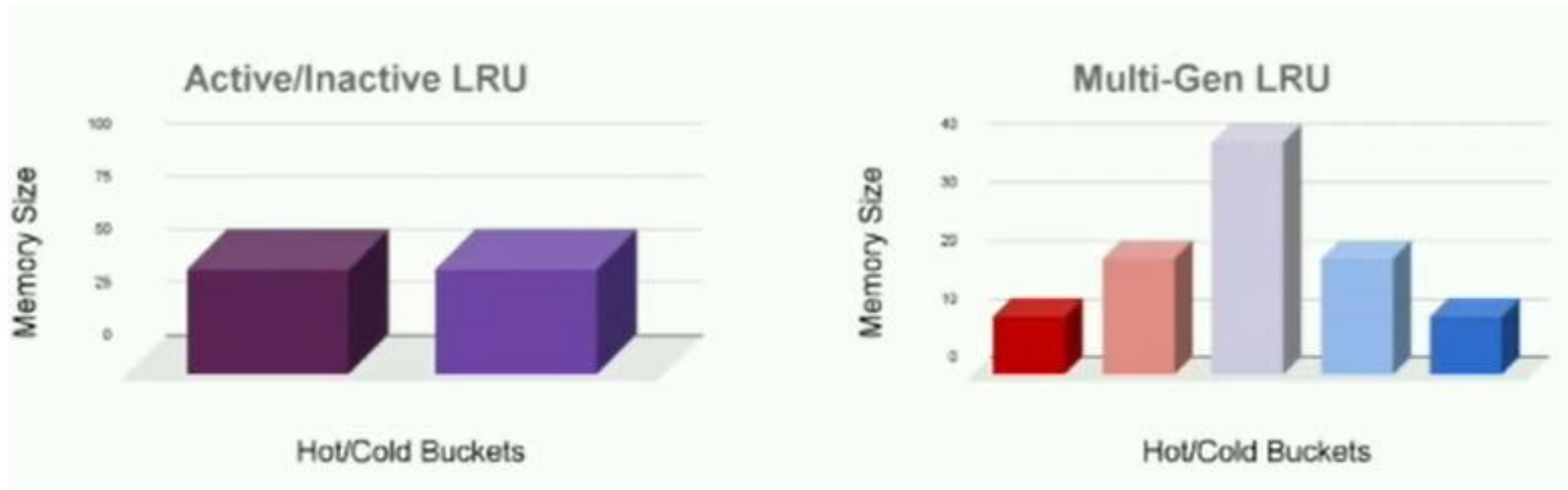


Multi-generational LRU (2021)

- [The multi-generational LRU](#), Jonathan Corbet, April 2021.
- Problems with the current solution:
 - The active/inactive sorting is too coarse for accurate decision making.
 - The use of independent lists in control groups makes it hard for the kernel to compare the relative age of pages across groups.
 - The kernel has a longstanding bias toward evicting file-backed pages, which can cause useful file-backed pages to be tossed while idle anonymous pages remain in memory. This problem has gotten worse in cloud-computing environments, where clients have relatively little local storage and, thus, relatively few file-backed pages in the first place.
 - The scanning of anonymous pages is expensive, partly because it uses a complex [reverse-mapping mechanism](#) that does not perform well when a lot of scanning must be done.
- Solution (by **Yu Zhao**, read Ju Džao, **Google**)
 - Add more LRU lists to cover a range of page ages between the current active and inactive lists; these lists are called "generations".
 - Change the way page scanning is done to reduce its overhead.
- Reaction of the kernel developers
 - [Google is working on a new and possibly better LRU memory management framework for the Linux kernel](#), April 2021.



Multi-generational LRU (2022)



[Multi-generational LRU](#), Yu Zhao (from Google), Linux Storage, Filesystem, MMU & BPF Summit, May 2022.

- MG LRU goals
 - Simplicity
 - Flexibility
 - Performance

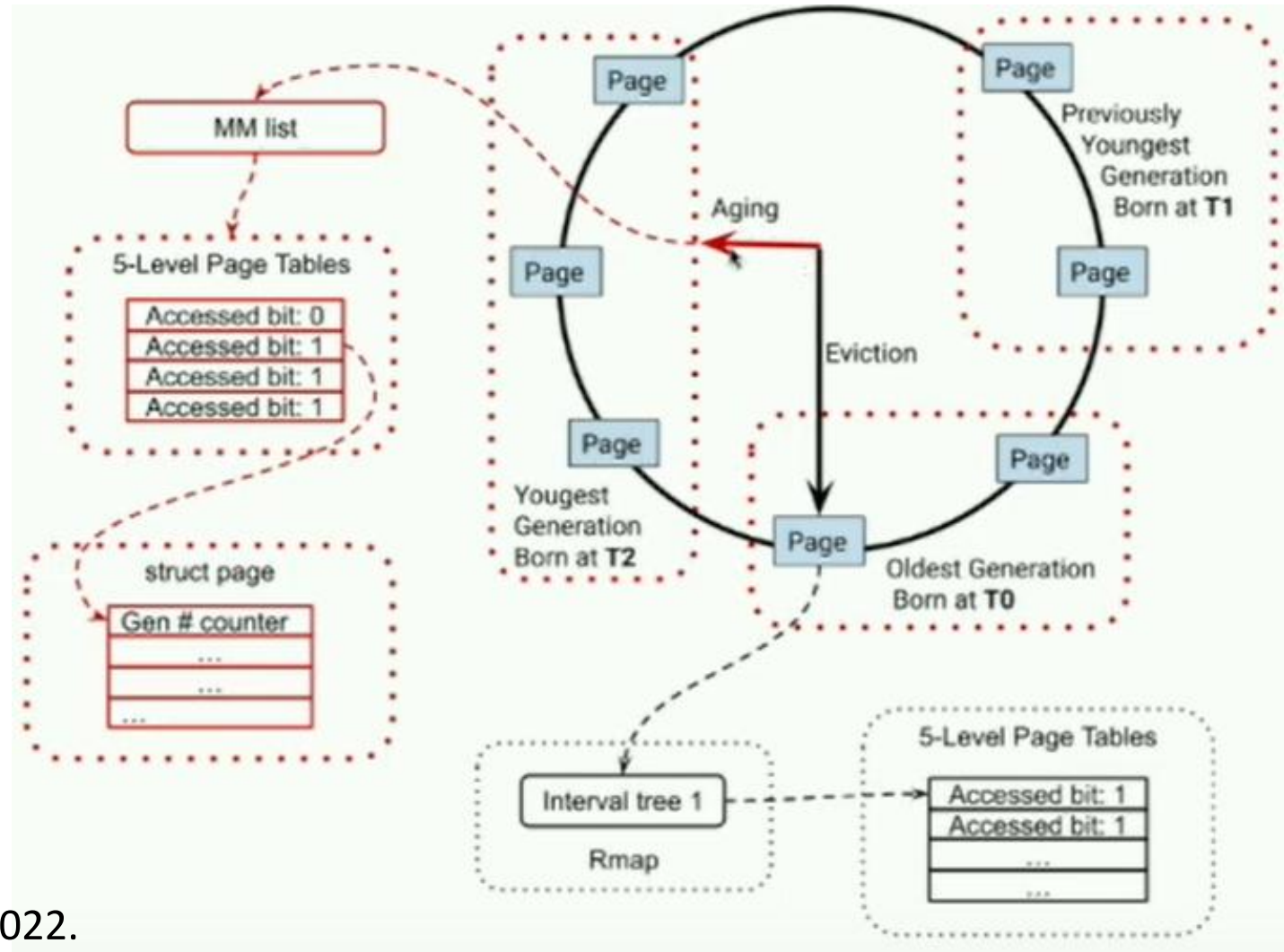




Multi-generational LRU (2022)

[Multi-generational LRU](#), Yu Zhao, Linux Storage, Filesystem, MMU & BPF Summit, May 2022.

- Generations
- Page table walks
- Feedback loops
 - Bloom filters
 - PID controller
- Merged into **6.1**, December 2022.
- **Does not replace** the current memory management scheme, can be configured at compilation time.





Page frame reclaiming – *Summary*

The **PFRA** works **cyclically** using two mechanisms: **kswapd** kernel threads that recover pages from **LRU** lists and a **function** that tries to recover **unused slabs** from the slab allocator.

The **kswapd()** function executes as a **kernel thread**. Its task is to **reclaim** pages when the number of **free pages** in the **zone** falls **below** a certain level.

The system first attempts to **free memory** by **reclaiming** it from the **slab allocator**.

If the memory cannot be reclaimed in this way, the system tries to reclaim it from the **page cache**.

First, the PFRA browses pages in the **active** list, moving those less used to the **inactive** list. It then browses the **inactive** list, **synchronizing pages** with **buffers** and trying to **free** pages that **nobody uses**. If it finds such pages and they are **dirty**, it initiates their **writing**.

If multiple mapped pages are found in the inactive list, the function is called to **remove** the **mapping**.

If the system cannot free pages from the cache, it tries to shrink the file system cache: **inode** cache, **dentry** cache and **quota** cache.

If the system is still **unable** to **recover** memory, it selects one **active process** and attempts to **kill** it to regain its memory.



Swapping

Disk caches enhance **system performance** at the expense of **free RAM**, while **swapping extends** the **amount** of addressable **memory** at the expense of **access speed**.

Swapping applies only to the following kinds of pages :

- Pages belonging to an **anonymous** memory region of a process (e.g. a **user mode stack** or **heap**),
- **Modified pages** belonging to a **private** memory **mapping** of a process (mapping with the **MAP_PRIVATE** flag) – modifications will not go to the file and will not be seen by other processes,
- Pages belonging to an **IPC shared memory** region.

Reclaiming **anonymous pages** (swapping) is seen as being **considerably more expensive** than reclaiming **file-backed pages**. One of the key reasons is that **file-backed pages** can be read from (and written to) persistent storage in **large, contiguous chunks**, while **anonymous pages** tend to be **scattered randomly** on the swap device.

The pages swapped out from memory are stored in a **swap area**, which may be implemented either as a **disk partition** of its own or as a **file** included in a larger partition.

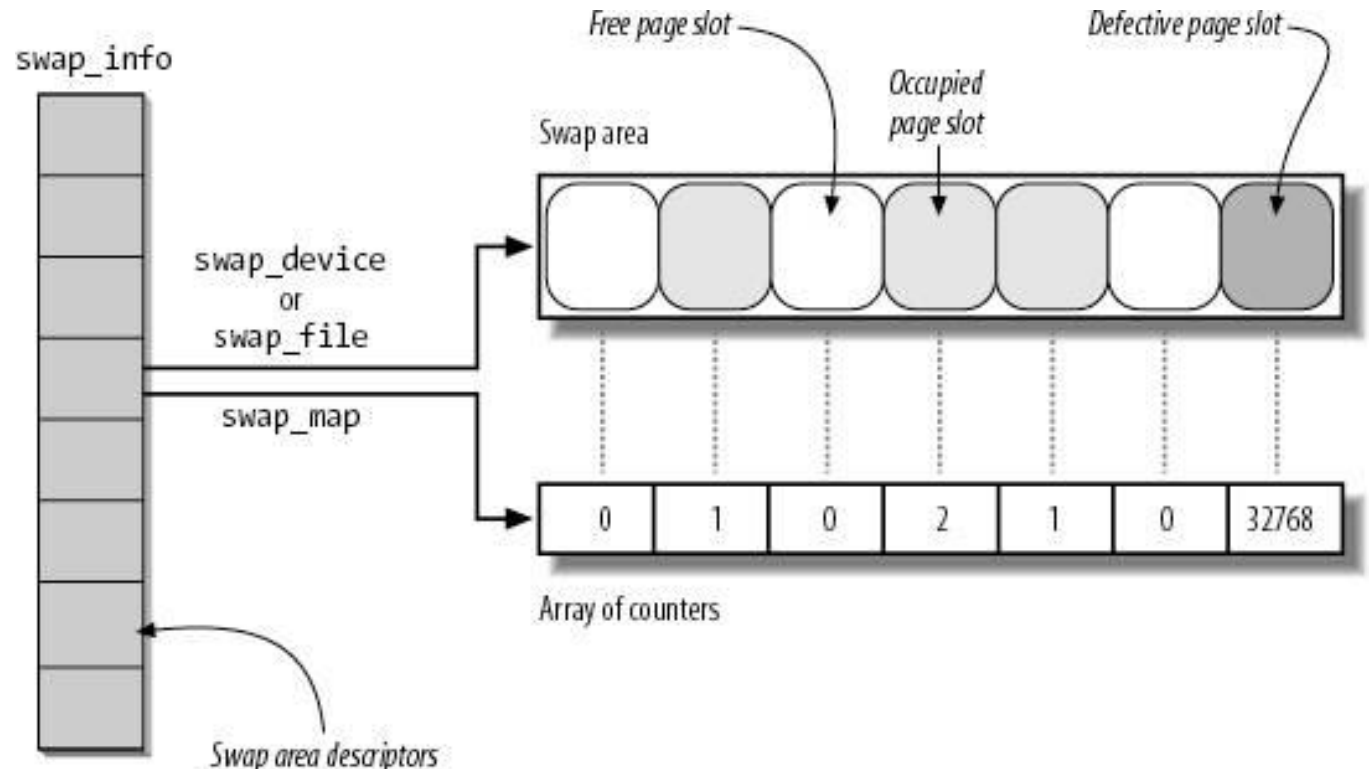
Each swap area consists of a sequence of **page slots**. The first page slot of a swap area is used to store information about the swap area.



Swapping

The **swap_map** field points to an **array of counters**, one for each swap area page slot. If the counter is equal to 0, the page slot is **free**; if it is positive, the page slot is filled with a **swapped-out** page (and indicates the number of processes sharing this page). If the counter has the value **SWAP_MAP_BAD** (equal to 32,768) the page slot is considered defective (unusable).

The **lowest_bit** and **highest_bit** specify the first and the last page slots that could be free – there are no free slots below or above these indexes.

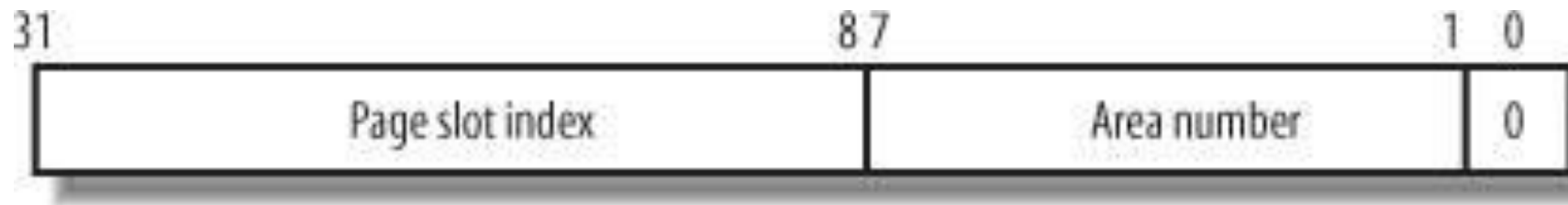




Swapping

The **swapped-out page identifier** is stored in the **page table entry**. All zeros mean that the page does not belong to the process address space or that the appropriate frame has not yet been allocated to the process.

If the last bit (the **Present** flag) is zero and the remaining 31 are not all zero, then the page is in the **swap area**. Otherwise it is in RAM. The swapped page is uniquely identified by the **swap area index** in the **swap_info** table and the **page slot index** in the swap area.



Swapped-out page identifier (source: Bovet, Cesati, Understanding the Linux Kernel)

The maximum size of the swap area is determined by the number of bits allocated for the page slot number. In 80x86 architecture, these are 24 bits, i.e. the area can have 2^{24} slots for 4 KB pages, i.e. 64 GB.



Swapping

Since a page may belong to the address spaces of several processes, it may be **swapped out from the address space of one process and still remain in main memory**. A page is physically swapped out and stored just once, but each subsequent attempt to swap it out increments the **swap_map counter**.

Each **swap area** consists of one or more **swap extents**. Each extent corresponds to a **group of page slots** that **are physically adjacent** on disk. An **ordered list of** the extents that compose a swap area is created when activating the swap area itself.

When **swapping out**, the kernel tries to store pages in **contiguous page slots** to **minimize disk seek** time when accessing the swap area.

Swap areas are linked in a **priority list**. When looking for a free slot, the search starts in the swap area that has the highest priority.

Information about swap areas is available in the **/proc/swaps** file:

```
[jmd@duch ~]$ cat /proc/swaps
```

Filename	Type	Size	Used	Priority
/dev/vda5	partition	5380092	534868	-1



Swap cache – synchronization

The **swap cache** does not exist as a **data structure** on its own, but the pages in the regular **page cache** are considered to be in the swap cache if **certain fields** are **set**.

The **swap cache** has been introduced to solve **synchronization problems** (*multiple swap-ins, concurrent swap-ins and swap-outs*) when transferring pages to and from a swap area.

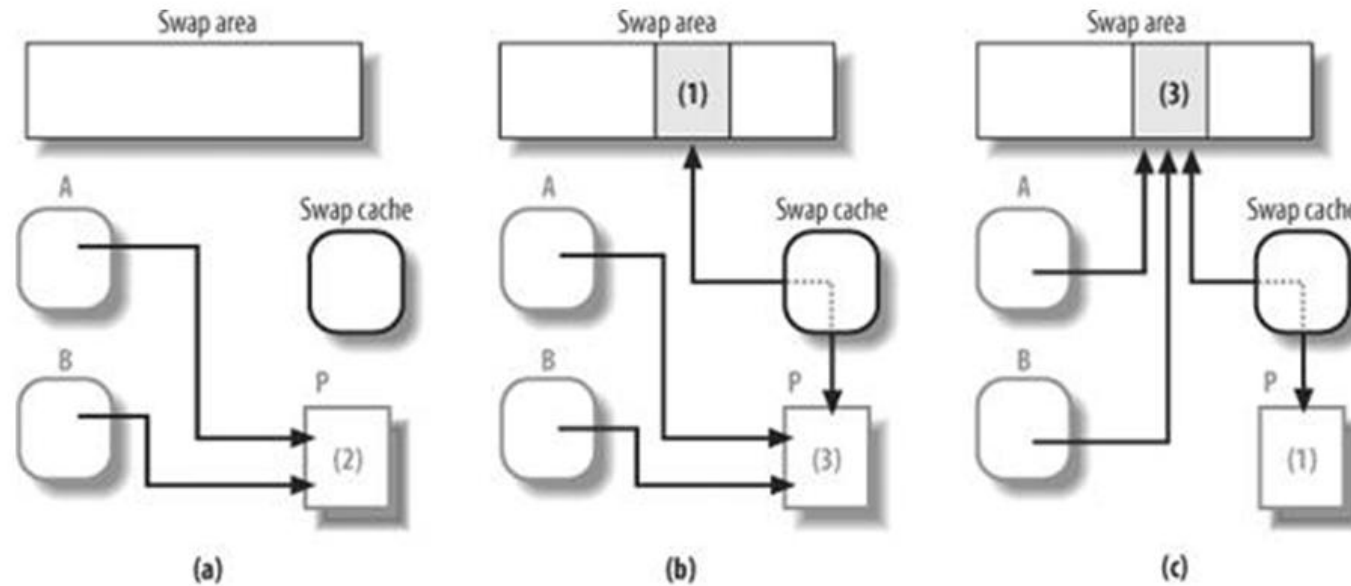
The key assumption is that nobody can start a swap-in or swap-out without checking whether the **swap cache** already **includes** the affected page.

Thanks to the page cache, **concurrent** swap operations affecting the same page always act on the same **page frame**; therefore, the kernel may safely rely on the **PG_locked** flag of the page descriptor to avoid any race condition.

*Consider two processes that **share** the same **swapped-out** page. When the **first process** tries to **access** the page, the kernel starts swap-in operation. It checks whether the page frame is already **included** in the **swap cache**. If it isn't, the kernel **allocates** a new **page frame** and inserts it into the swap cache; next, it starts the I/O operation. Meanwhile, the **second process** **accesses** the **shared** anonymous page. The kernel starts a **swap-in** operation but now it finds the page frame in the swap cache. It puts the current process to sleep until the **PG_locked** flag is cleared.*



Swap cache – synchronization



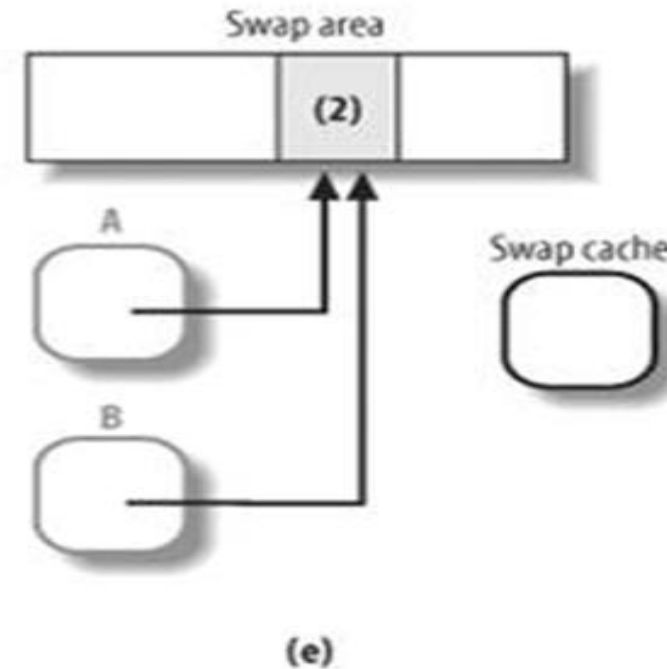
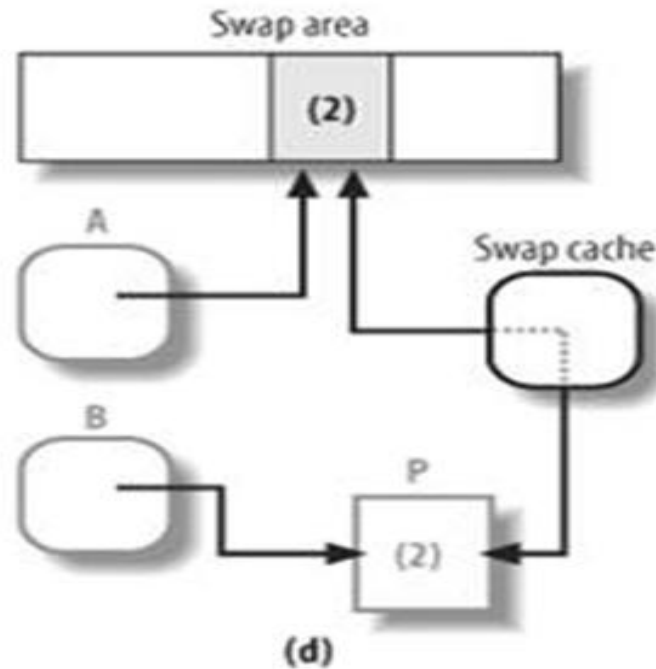
Consider a page **P** shared among two processes, **A** and **B**. Initially, the page tables of both processes point to the same **page frame**, so the page has **two owners** (see (a)).

When the PFRA select the page for reclaiming, it inserts the page frame in the **swap cache** – now the page frame has **three owners**, while the page slot in the swap area is referenced only by the swap cache (see (b)).

Next the PFRA **removes** the references to the page frame from the **page tables** of the processes; once finished the page frame is referenced **only** by the **swap cache**, while the **page slot** is referenced by the **two processes** and the **swap cache** (see (c)).



Swap cache – synchronization



Let's suppose that while the page's contents are being written to disk, process **B** accesses the page. The page fault handler finds the page frame in the swap cache and **puts back its physical address** in the page table entry of process B (**see (d)**).

Conversely, if the swap-out operation terminates **without concurrent** swap-in operations, the PFRA **removes the page frame** from the swap cache and releases the page frame to the **buddy system** (**see (e)**).



Swap cache – implementation

The swap cache is used during swap-ins and swap-outs, but it's mainly useful while **swapping-in pages shared** by a number of processes.

This asymmetry comes from the fact that when **swapping-out** the page, you can reach all processes **sharing the page** using **rmap** (ang. *reverse mapping*) structure, which connects the **vm_area_struct** structures including the shared page.

However when **swapping-in** a page, we only know how many users there are, but we do not know them.

The swap cache is implemented by the **page cache data structures and procedures**. Pages in the **swap cache** are stored as every other page in the **page cache**, with the following special treatment:

- The **mapping** field of the **page descriptor** is set to NULL.
- The **PG_swapcache** flag of the **page descriptor** is set.
- The **private** field stores the **swapped-out page identifier** associated with the page.



Swap cache – implementation

(see `/include/linux/swap.h` and `/mm/swap_state.c`)

A single **swapper_space** **address space** object is used for all pages in the swap cache (this is the entry in the **swapper_spaces** table). The Xarray structure pointed to from this address space object contains **all pages in the swap cache**.

This structure is used by the **lookup_swap_cache()** function which searches for the page in the swap cache.

```
struct address_space *swapper_spaces[MAX_SWAPFILES];
```



Swapping to solid-state devices

Some of the swap code is quite old. In the early days, the kernel would attempt to **concentrate swap-file** usage toward the beginning of the device – the **left end** of the **swap_map** array.

When one is swapping to **rotating storage**, this approach makes sense; keeping data in the swap device together should minimize the amount of seeking required to access it. **It works less well on solid-state devices**, for a couple of reasons:

- there is no **seek delay** on such devices,
- the **wear-leveling** requirements of SSDs are better met by spreading the traffic across the device.

In an attempt to perform better on **SSDs**, the swap code was changed in **2013**. When the **swap subsystem** knows that it is working with an **SSD**, it divides the device into **clusters**.

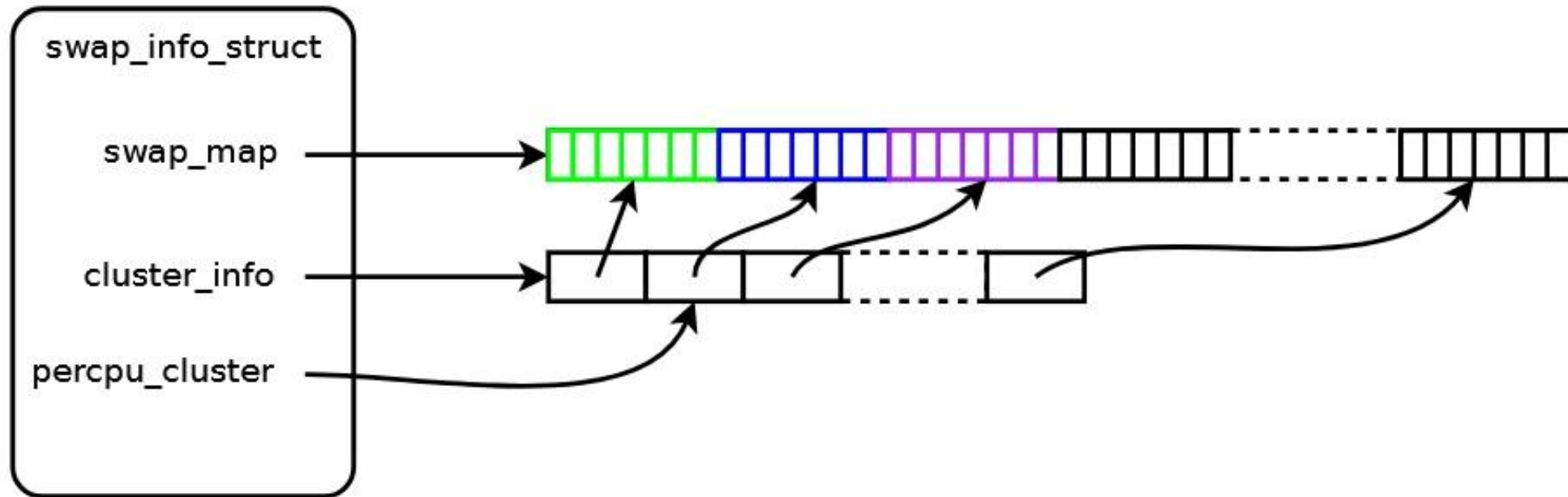
Swap cluster (source: [Making swapping scalable](#), Jonathan Corbet, 2016)



Swapping to solid-state devices

The **percpu_cluster pointer** points to a different cluster for each CPU on the system. With this arrangement, each **CPU** can allocate pages from the swap device from within its **own cluster**, with the result that those allocations are **spread across the device**.

Locking is done on a cluster level.



Swap cluster (source: [Making swapping scalable](#), Jonathan Corbet, 2016)



Additional reading

- [The zswap compressed swap cache](#), Seth Jennings, February 2013.
- [Compressed swap](#), Jonathan Corbet, March 2014.
*There are a number of projects oriented around improving memory utilization through the compression of memory contents. Two of these, **zswap** and **zram**, have found their way into the mainline kernel; they both aim to replace swapping with compressed, in-memory storage of data.*
- [Reconsidering swapping](#), Jonathan Corbet, June 2016.
Johannes Weiner changes the mechanism that decides whether to reclaim pages from the anonymous LRU list or the file-backed LRU. For each list, he introduces the concept of the "cost" of reclaiming a page from that list; the reclaim code then directs its efforts toward the list that costs the least to reclaim pages from.
- [The next steps for swap](#), Jonathan Corbet, March 2017.
Swapping in and out huge pages.