# Linux Kernel Programming
# The Page Cache and Page Writeback

Pierre Olivier

Systems Software Research Group @ Virginia Tech

April 27, 2017

# Outline

# Outline

# General notions about caching

## Page cache: general presentation

- The **page cache** buffers disk I/O in RAM
  - RAM access is several orders of magnitude faster than disk

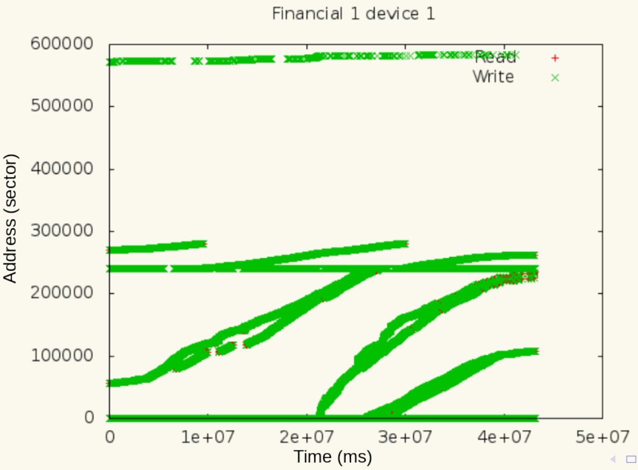
| | |
|---|---|
| execute typical instruction | 1/1,000,000,000 sec = 1 nanosec |
| fetch from L1 cache memory | 0.5 nanosec |
| branch misprediction | 5 nanosec |
| fetch from L2 cache memory | 7 nanosec |
| Mutex lock/unlock | 25 nanosec |
| fetch from main memory | 100 nanosec |
| send 2K bytes over 1Gbps network | 20,000 nanosec |
| read 1MB sequentially from memory | 250,000 nanosec |
| fetch from new disk location (seek) | 8,000,000 nanosec |
| read 1MB sequentially from disk | 20,000,000 nanosec |
| send packet US to Europe and back | 150 milliseconds = 150,000,000 nanosec |

- Source: `http://norvig.com/21-days.html#answers`

# General notions about caching

## Page cache: general presentation

- Why caching?




- Traces from `http://traces.cs.umass.edu/index.php/Storage/Storage`

# General notions about caching

## Page cache: general presentation

- Page cache: **physical pages in RAM holding disk content** (blocks)
  - Disk is called the *backing store*
  - Works for regular files, memory mapped files, and block devices files
- **Dynamic size:**
  - Grows to consume free memory unused by kernel and processes
  - Shrinks to relieve memory pressure
- In case of a `read()` operation, data presence in the page cache is first checked
  - If data is present in the cache, ***cache hit***
  - Otherwise, ***cache miss***: VFS asks the (concrete) filesystem to read the data from disk
    - Read (and write) operations populates the page cache
    - Files are cached on a per-page basis

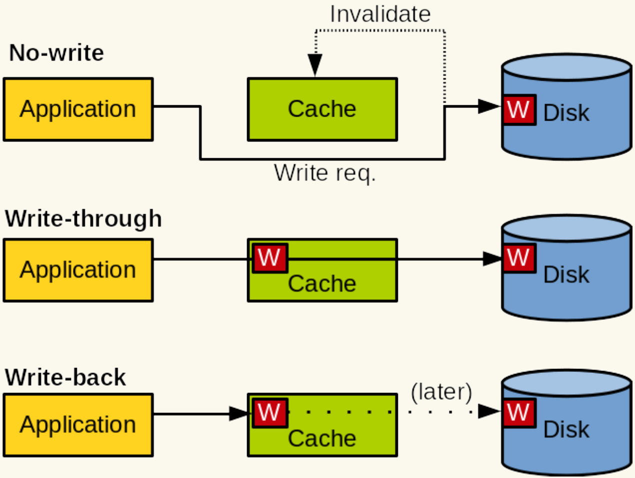
# General notions about caching

## Write caching

- 3 main policies for cache write implementation:
  - **No-Write**: all writes are directed to disk and cached (read) data is invalidated
    - Costly because no write caching + invalidation
  - **Write-through**: writes are directed to disk and also update the cache
    - Cache is kept coherent with disk, no need to invalidate
  - **Write-back**: writes update the cache, and disk is not directly updated
    - This is the Linux page cache policy
    - Pages written are marked *dirty*
    - Regularly synchronized with the disk and unmarked as dirty through a process called ***writeback***
    - Benefit is performance as the cache absorbs temporal locality to reduce disk access
    - Downside is complexity in implementation and management

# General notions about caching

## Write caching (2)



# General notions about caching

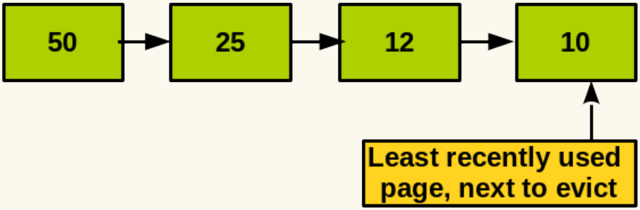
## Cache eviction: generalities

- Evicting data from the cache is needed when:
  - The cache needs to shrink (memory pressure)
  - The cache cannot grow and we need to make space for upcoming data
- In Linux:
  - Select *clean* (not dirty) pages and replace them/release the memory
  - Not enough clean pages $\rightarrow$ force writeback
- **Eviction policy**: how to select which data to remove from the cache
  - Ideal cache: evict pages that will not be accessed in the future (*clairvoyant algorithm*)

# General notions about caching

## Cache eviction: Least Recently Used

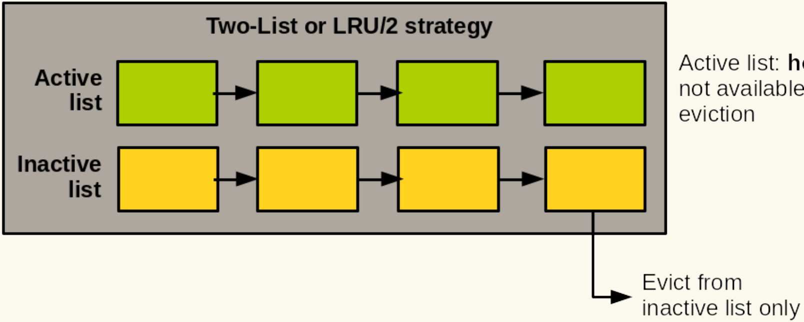
- The *clairvoyant* algorithm is not implementable in reality
  - ***Least Recently Used*** (LRU) tries to approach it with information from the past
- LRU keeps track of when each page in the cache is accessed
  - Pages are sorted in timestamp usage order




- LRU issue: multiple files are accessed only once

# General notions about caching

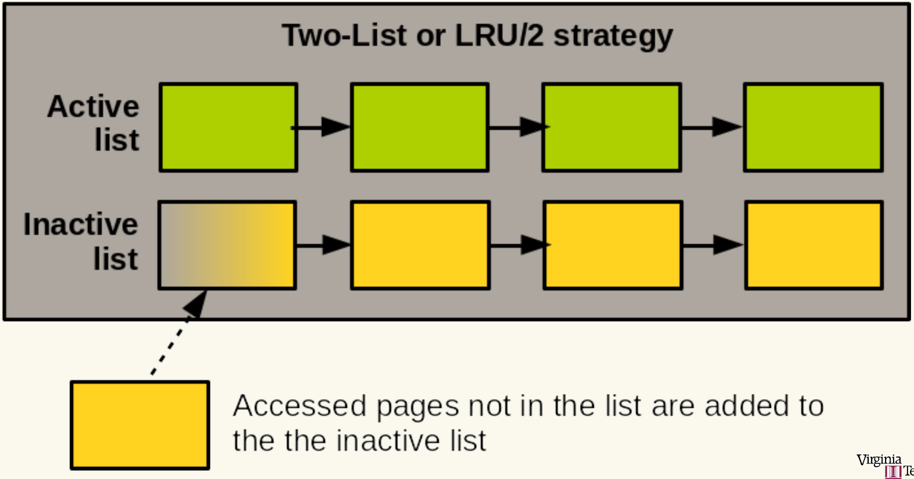
## Cache eviction: the two-list strategy




Active list: **hot pages,** not available for eviction

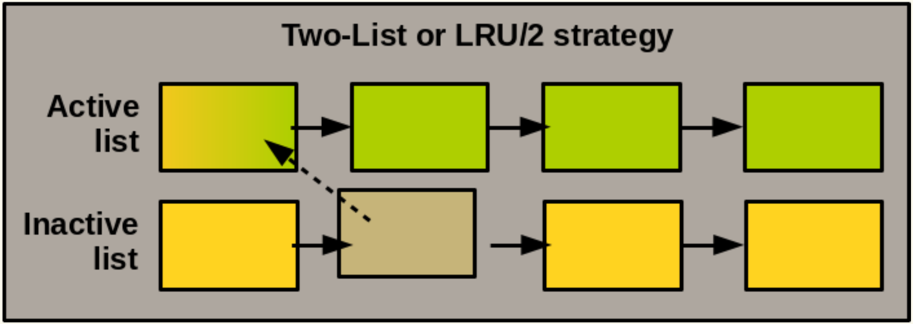
# General notions about caching

## Cache eviction: the two-list strategy



# General notions about caching

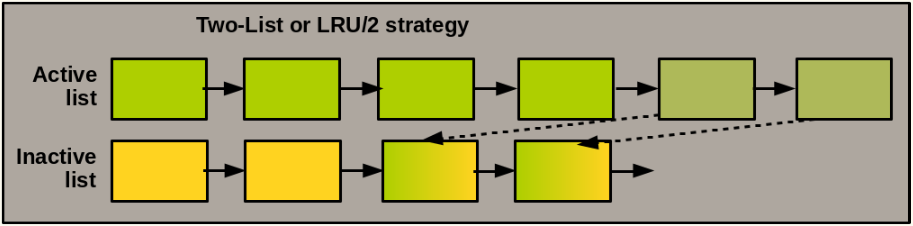
## Cache eviction: the two-list strategy




Inactive page accessed
are added to the active list

# General notions about caching

## Cache eviction: the two-list strategy



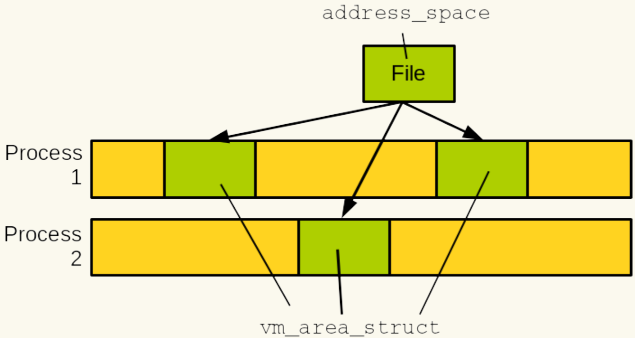

Lists are balanced and active pages are evicted in the inactive list

# Outline

# The Linux page cache

## address_space object

- The address_space object represents an entity present in the page cache
  - **A file**
  - 1 address space per entity (represent the physical pages containing the entity)



# The Linux page cache

## address_space object (2)

- Defined in `include/linux/fs.h`:

```
struct address_space {
  struct inode              *host;          /* owning inode */
  struct radix_tree_root    page_tree;      /* radix tree of all pages */
  spinlock_t                tree_lock;      /* page tree lock */
  unsigned int              i_mmap_writable; /* VM_SHARED (writable) mapping count */
  struct rb_root            i_mmap;         /* list of all mappings */
  unsigned long             nrpages;        /* total number of pages */
  pgoff_t                   writeback_index; /* writeback start offset */
  struct address_space_operations a_ops;    /* operations table */
  unsigned long             flags;          /* error flags */
  gfp_t                     gfp_mask;       /* gfp mask for allocation */
  struct backing_dev_info   backing_dev_info; /* read-ahead info */
  spinlock_t                private_lock;   /* private lock */
  struct list_head          private_list;   /* private list */
  struct address_space      assoc_mapping;  /* associated buffers */
  /* ... */
}
```

# The Linux page cache

## address_space object (3)

- Interesting fields of the `address_space` structure:
  - `i_mmap`: *priority search tree* of all shared and private mappings concerning this address space
  - `nrpages`: total number of pages in the address space
  - `host`: points to the inode of the corresponding file
  - `a_ops`: address space operations table
    - Similar to VFS operations on inodes, dentries, etc.Similar to VFS operations on inodes, dentries, etc.

# The Linux page cache

## address_space operations

- address_space_operations defined in include/linux/fs.h

```
struct address_space_operations {
  int (*writepage)(struct page *page, struct writeback_control *wbc);
  int (*readpage)(struct file *, struct page *);
  int (*writepages)(struct address_space *, struct writeback_control *);
  int (*set_page_dirty)(struct page *page);
  int (*readpages)(struct file *filp, struct address_space *mapping,
      struct list_head *pages, unsigned nr_pages);
  int (*write_begin)(struct file *, struct address_space *mapping,
        loff_t pos, unsigned len, unsigned flags,
        struct page **pagep, void **fsdata);
  int (*write_end)(struct file *, struct address_space *mapping,
        loff_t pos, unsigned len, unsigned copied,
        struct page *page, void *fsdata);
  /* ... */
}
```

- Functions implement page I/O for this cached object
  - Each backing store implements its own address_space_operations instance (ex: filesystems)

# The Linux page cache

## address_space operations

- **Page read operation**: `read()` function from the `file_operations`
  - Search the data in the page cache:

```
page = find_get_page(mapping, index);
```

    - `mapping` is the corresponding `address_space`
    - `index` is the searched page index
    - Returns `NULL` is the page is not present
- Adding the page to the page cache:

```
struct page *page;
int error;
/* allocate the page */
page = page_cache_alloc_cold(mapping);
if(!page)
  /* allocation error */
/* add the page to the page cache */
error = add_to_page_cache_lru(page, mapping,
      index, GFP_KERNEL);
if(error)
  /* error during page insertion in the page
      cache */
```

- Then, read data from disk:

```
error = mapping->a_ops->readpage(file,
      page);
```

# The Linux page cache

## address_space operations (2)

- **Page write operation**:
  - When a page is modified in the page cache, it is set as dirty:

```
SetPageDirty(page);
```

    - It will be written later (writeback)
  - Default write path: in `mm/filemap.c`

```
/* search the page cache for the desired page. If the page is not present,
an entry is allocated and added: */
page = __grab_cache_page(mapping, index, &cached_page, &lru_pvec);
/* Set up the write request: */
status = a_ops->write_begin(file, mapping, pos, bytes, flags, &page, &fsdata);
/* Copy data from user-space into a kernel buffer: */
copied = iov_iter_copy_from_user_atomic(page, i, offset, bytes);
/* write data to disk: */
status = a_ops->write_end(file, mapping, pos, bytes, copied, page, fsdata);
```

# The Linux page cache
## The radix tree

- For any page I/O (read/write) the concerned page is searched in the page cache
  - **Page cache lookup must be fast**
- Searching in the page cache is done with an `address_space` plus an offset value, a page index
- Each `address_space` has a **radix tree** indexing its content (`page_tree` member)
  - Specific type of binary tree
  - Allows quick searching for a page given the file offset
    - `radix_tree_lookup()`
- More info on the radix tree: `https://0xax.gitbooks.io/linux-insides/content/DataStructures/radix-tree.html`

# The Linux page cache

## The old page hash table

- The radix tree was introduced in 2.6 to replace a hash table mechanism:
  - Searching for a hash returned a doubly linked list of pages hashing to the same value
  - If the page was in the page cache, then it was contained in the list
- Hash table had 3 main problems:
  1. Protected by a single lock, high contention
  2. Large hash as it covered *all* the pages in the page cache $\rightarrow$ large memory consumption
  3. High performance cost for searching a page that is not in the page cache

# Outline

# Flusher threads

## Generalities

- Write operation are deferred, data is marked *dirty*
  - RAM data is out-of-sync with the storage media
- Dirty page writeback occurs:
  - Free memory is low and the page cache needs to shrink
  - Dirty data grows older than a specific threshold
  - User process calls `sync()` or `fsync()`
- Multiple ***flusher threads*** are in charge of syncing dirty pages from the page cache to disk
  1. To shrink the page cache when free memory amount becomes low
  2. To sync data that has been dirty for a given time

# Flusher threads

## Generalities (2)

- **Flusher threads writeback on low memory:**
  - When the free memory goes below a given threshold, the kernel calls `wakeup_flusher_threads()`
    - Wakes up one or several flusher threads performing writeback though `bdi_writeback_all(num_pages_to_write)`
  - Thread write data to disk until
    1. `num_pages_to_write` have been written *and*
    2. The amount of memory drops below the threshold
  - Can consult and modify the threshold by reading and writing to: `/proc/sys/vm/dirty_background_ratio`
    - Percentage of total memory

# Flusher threads

## Generalities (3)

- **Flusher threads writeback of *old* data:**
  - Page cache content is lost on power cut
    - Pages should not stay dirty for too long
- At boot time a timer is initialized to wake up a flusher thread calling `wb_writeback()`
  - Writes back all data older than a given value:
    - `/proc/sys/vm/dirty_expire_interval`
  - Timer reinitialized to expire at a given time in the future: now + period
    - `/proc/sys/vm/dirty_writeback_interval`
- Multiple other parameters related to the writeback and the control of the page cache in general are present in `/proc/sys/vm`
  - More info: `Documentation/sysctl/vm.txt`

# Flusher threads

## Laptop mode

- **Laptop mode** is a writeback strategy designed to save power
- When not used, hard disk enters sleep state and stop spinning
  - Saves a significant amount of power compared to active state
- *Laptop mode* tries to minimize spinning as much as possible:
  - When a flusher thread wakes up to write back old data, *all* dirty data is synced with the disk
  - `dirty_expire_interval` and `dirty_writeback_interval` are set to very large values (several minutes)