WRITING LINUX FS 4FUN

Part 2

Short story of Pages and Virtual Memory

Outline

- Why
- Main Concepts and bit of history
  - *VM as a start of the journey*
  - *Paging, Swapping*
- Caching vs Latencies
- VM caching problem statement
  - *Solaris Page Cache*
  - *Linux Page Cache*
- IO paths
- Code Fragments:
  - *Read/Write Page*
  - *mmap*

Why this talk?!

- **Cons**
  - Writing FS is quite time consuming (approx. 10 years…)
  - Just few production ready FS, many abandoned or not truly maintained
  - File Systems considered as passé: now is Object Stores era!

- **Pros**
  - ***Address specific Education gap***
  - ***Solving other complicated problems***
    - Memory Management is considered as difficult topic
    - Storage stack is complicated and usually became a bottleneck
    - Data is foundation of most todays application

VM in UNIX

- Virtual Memory: *"high level"* abstraction
- Multiple Processes with different address space
- Concept of Primary Storage: Main Memory (Fast - RAM) and Secondary Storage (Slow - Backing Device)

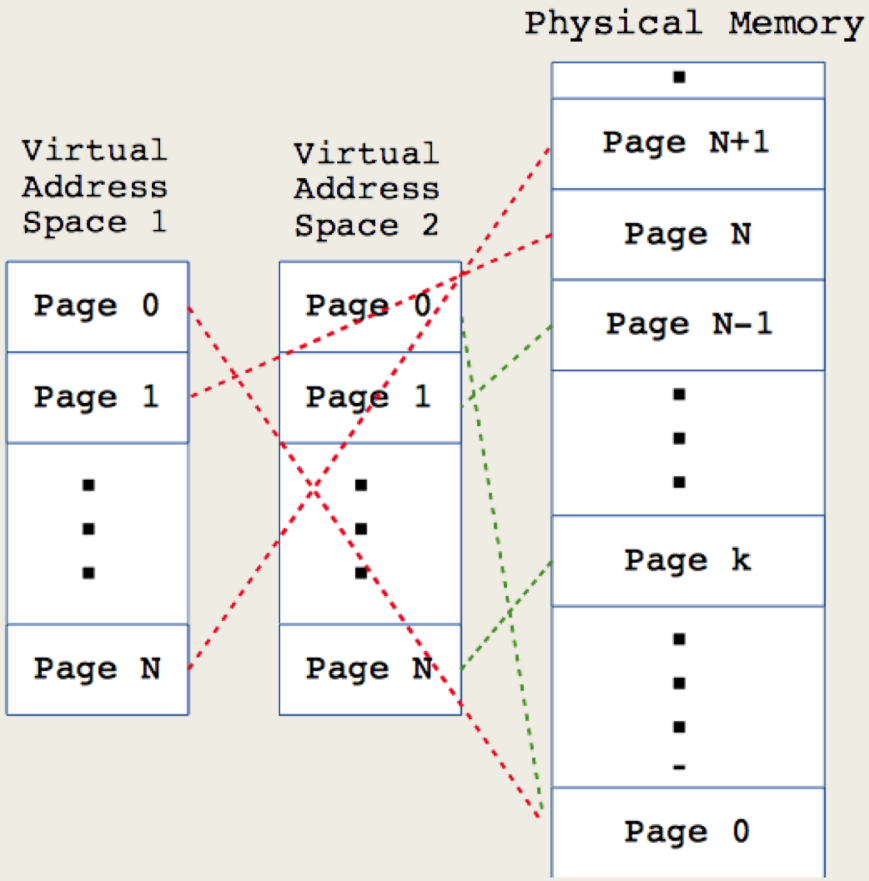
UNIX, like all time-sharing systems, and some multiprogramming systems uses "program swapping" (also called "rollin/roll-out") to share the limited resource of the main physical memory among several processes.

Processes which are suspended may be selectively "swapped out" by writing their data segments (including the "per process data") into a "swap area" on disk

From J.Lions[77] c14. Program Swapping

Paging

- Mechanism of Memory Management: Divide whole physical memory to small chunks called pages
- Pages referred by their number i.e. PFN
- Three important paging policies [1973]:
  - *Fetch policy*
  - *Placement policy*
  - *Replacement Policy*



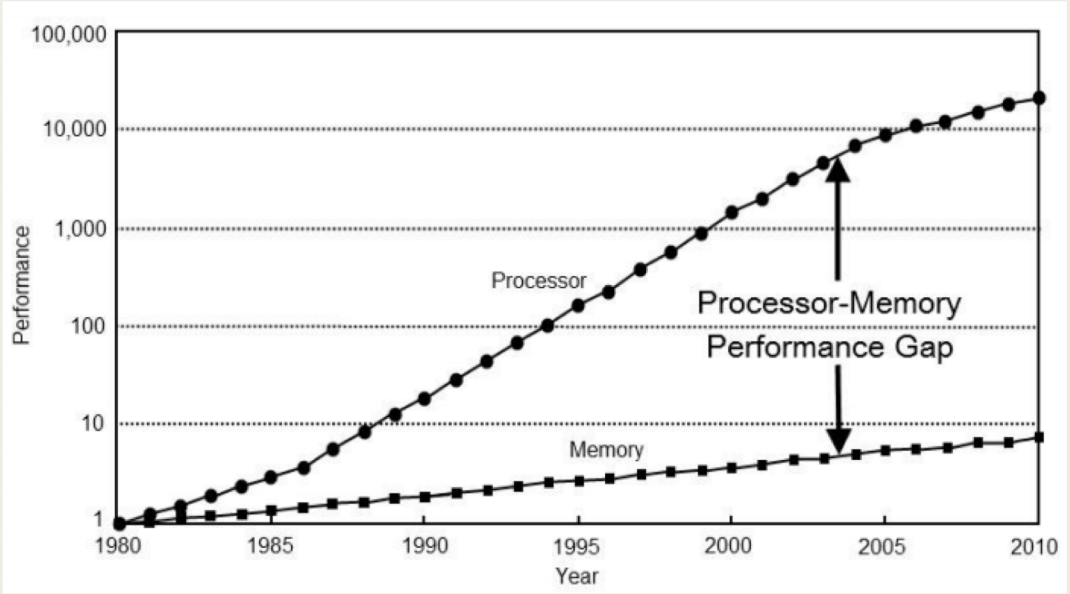
Demand Paging

- Introduced in : Demand paging made its appearance in UNIX with the introduction of the VAX-11/780 in 1978
- All versions of UNIX used demand paging as the primary memory management technique, with swapping relegated to a secondary role in 80s
- When referencing memory which is not present Page Fault: Minor, Major

Computing is all about caching

- Intel Laptop CPU can execute 50 Bilions instruction per second
  - *Unless it takes a cache miss*
- DRAM can deliver 500 Milions cache lines per second
- SSD can deliver 800 thousand 4KiB pages per second

- PDP 11/70 could execute
  400 Thousand IPS
- Its Memory could handle
  300 Thousand cache misses p/s
- The RK05 drive could deliver
  4 thousand 4KiB pages p/s



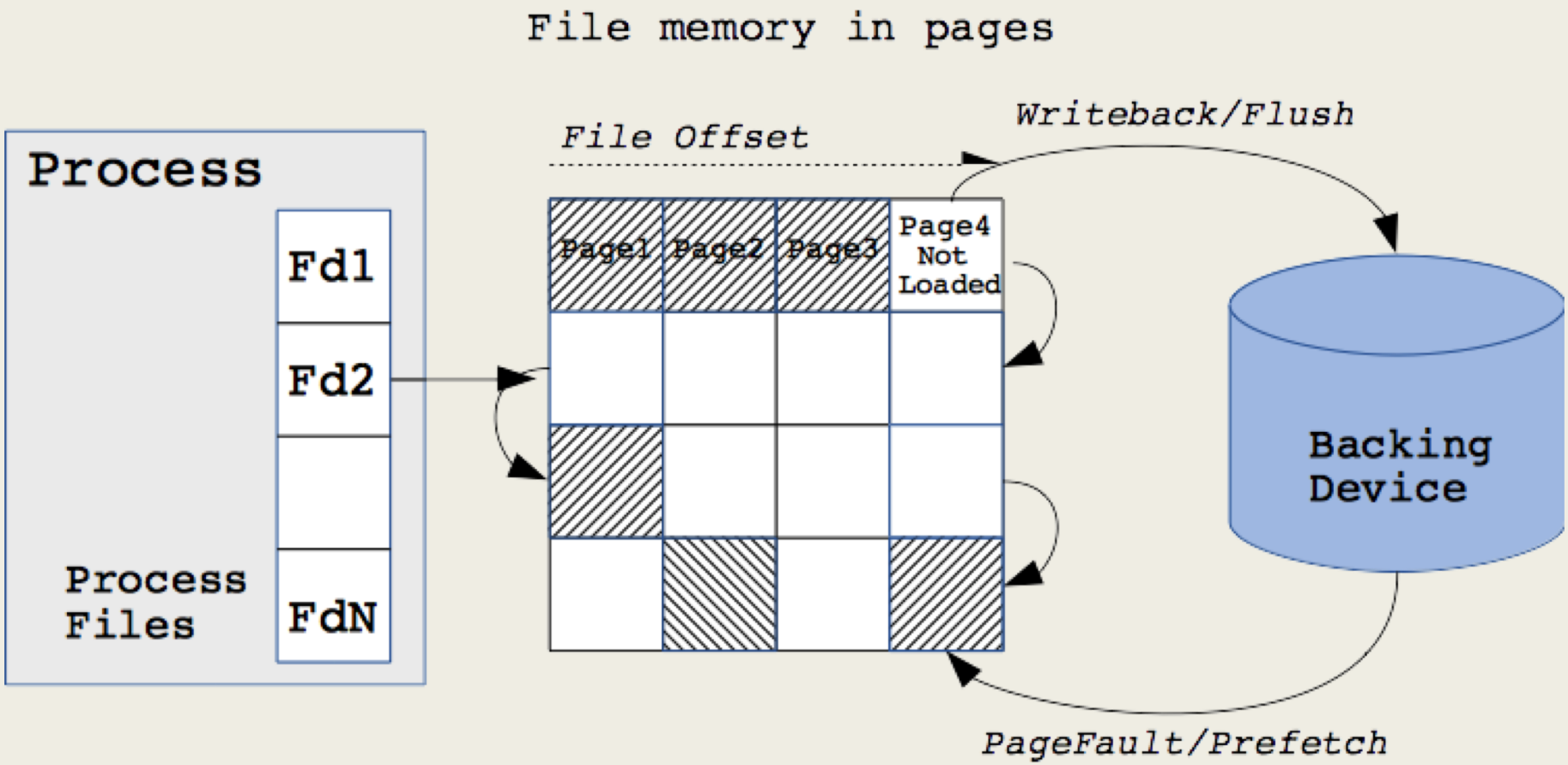
Latencies numbers that you should know

```
L1 cache reference ......................... 0.5 ns
Branch mispredict ............................ 5 ns
L2 cache reference ........................... 7 ns

Main memory reference ...................... 100 ns

SSD random read ........................ 150,000 ns  = 150 µs
Read 1 MB sequentially from memory ..... 250,000 ns  = 250 µs

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
```

Lets multiply all these durations by a billion:

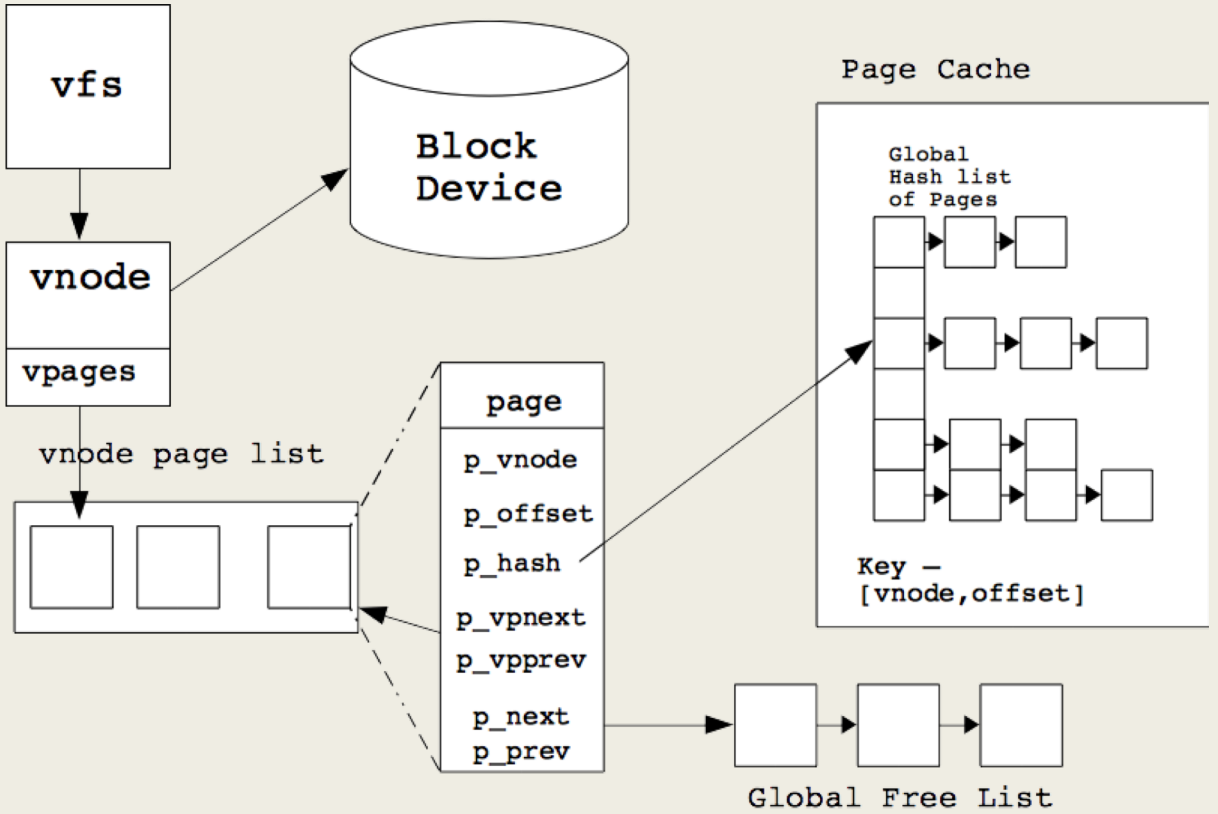
L1 cache reference	0.5 s	One heart beat (0.5 s)
Branch mispredict	5 s	Yawn
L2 cache reference	7 s	Dispatching expresso in coffee machine
Main memory reference	100 s	Doing a toasted sandwich
SSD random read	1.7 days	A normal weekend
Read 1 MB sequentially from memory	2.9 days	A long weekend
Read 1 MB sequentially from SSD	11.6 days	Waiting for 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	

General Caching problem



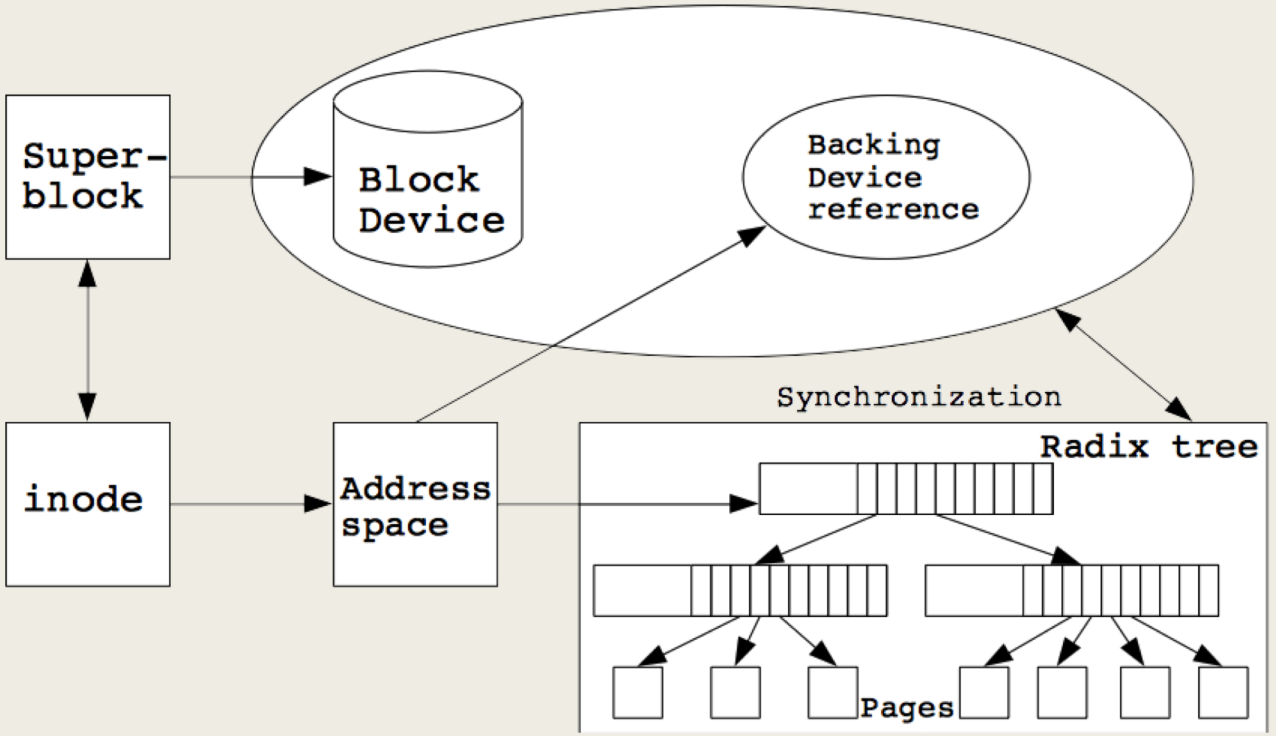
SOLARIS

- Page Cache implemented as a Hash Table
- Hashing of Pages done by key [vnode, offset]
- Each vnode maintain list of pages associated to this vnode



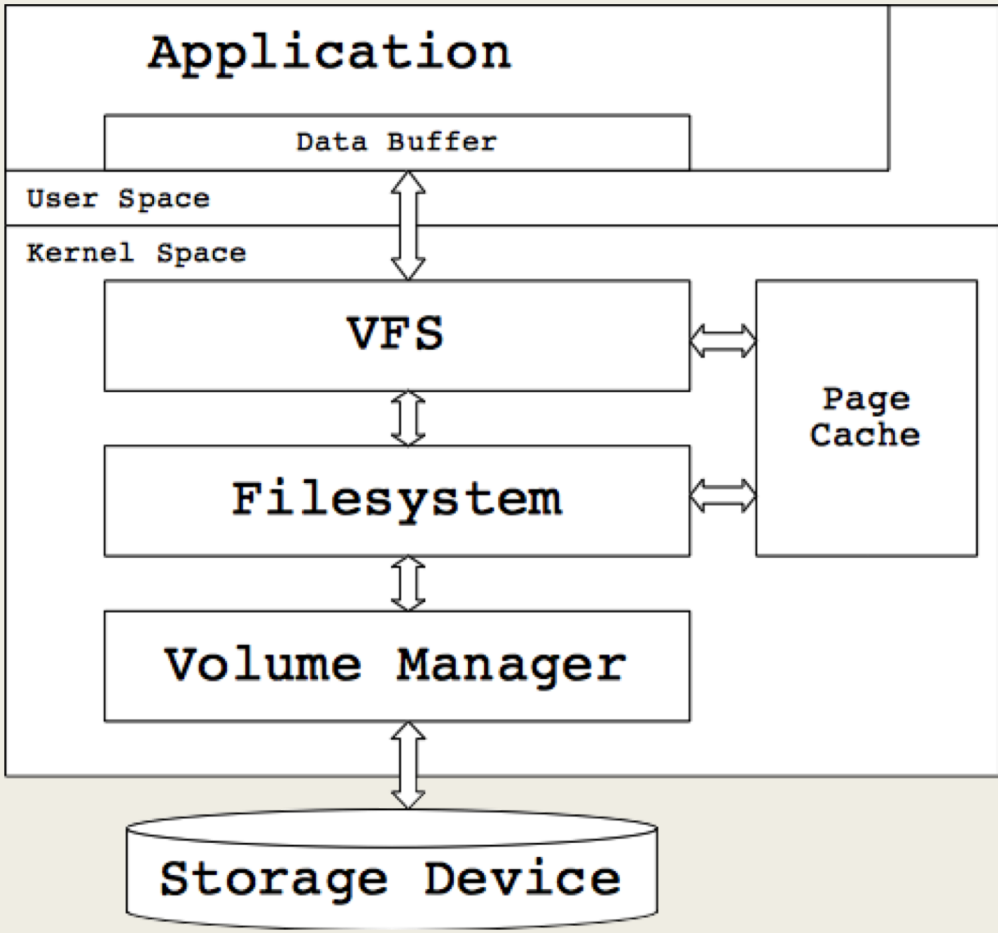
Linux

- Page Cache: inode connected to the address space
- Address space mapping of the pages
- Radix tree as Page Cache
- One Page Cache per inode (file)



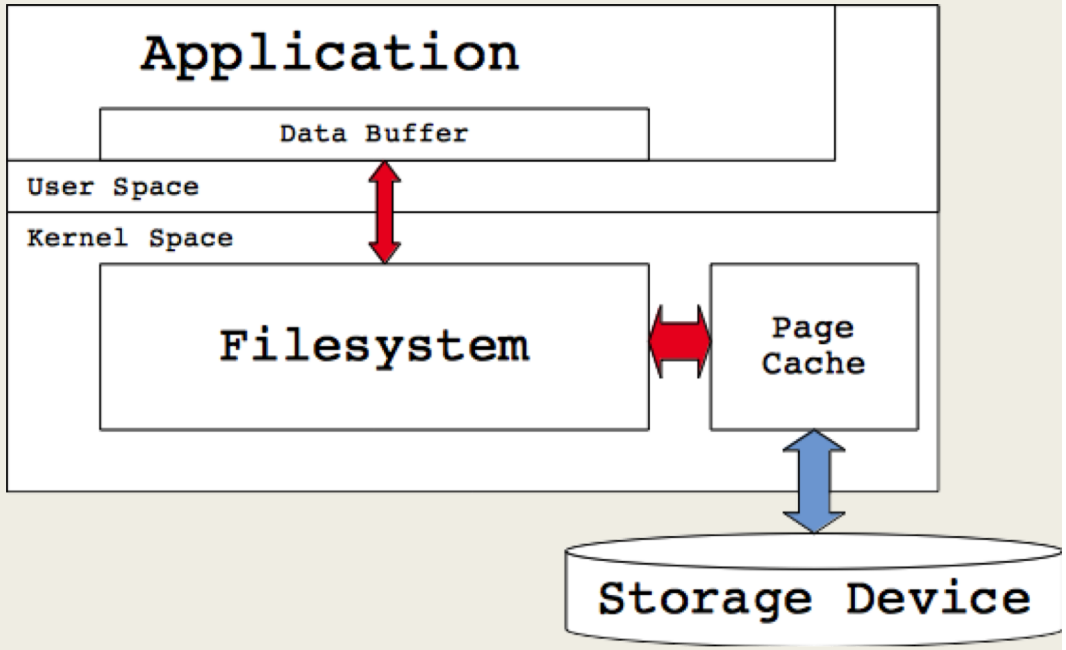
IO Paths:

- Data can be sent to the Storage device in many ways
- Multiple components inside kernel space
- Userspace can advise the kernel



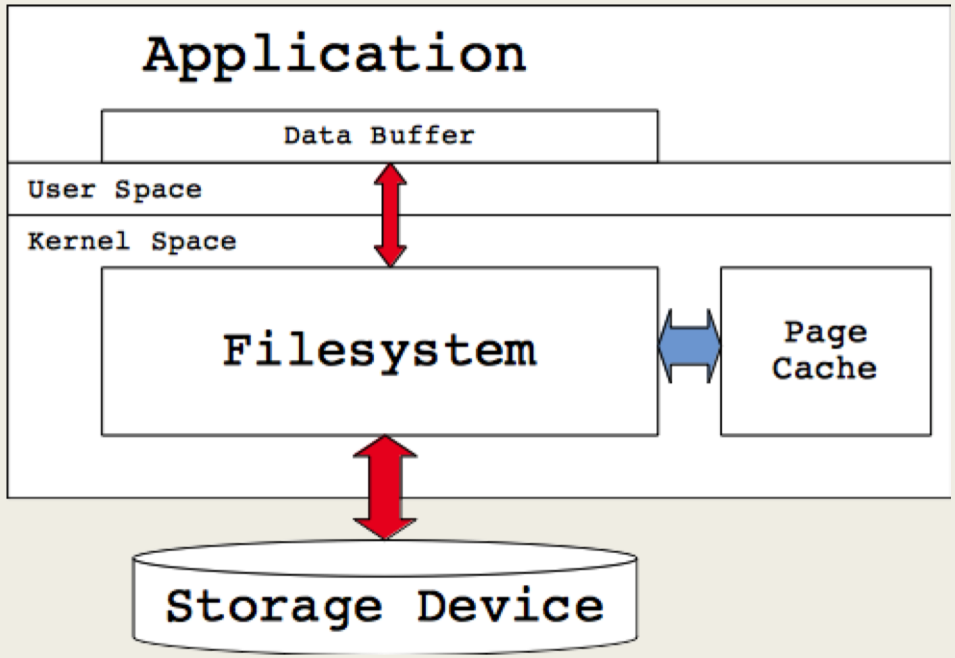
Buffered IO

- Application do frequent R/W
- Is fine for data to sync with disk after some time
- Read/Write directly from/to the Cache
- Cache do synchronization with storage by its own
- If cache miss during the Read need to wait



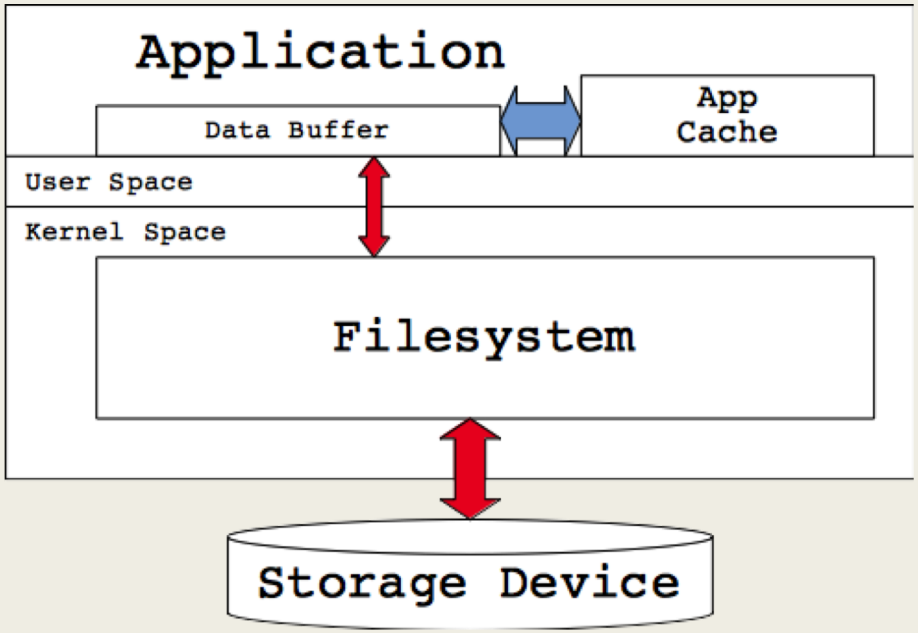
Synchronous Buffered IO

- Application want **Write** the data to reach the disk and get confirmation.
- Cache do store this data as well as it needs to sync data between it and Storage medium



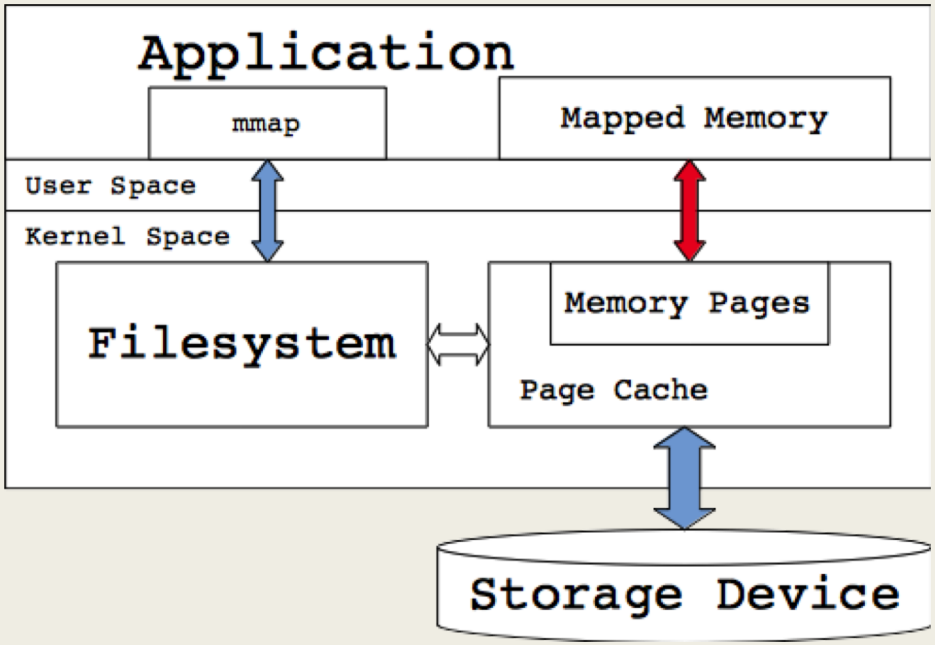
Direct IO

- Application wants to manage caching the data by its own
- Whenever IO performed it should go directly to the disk, no point for caching the same data inside kernel



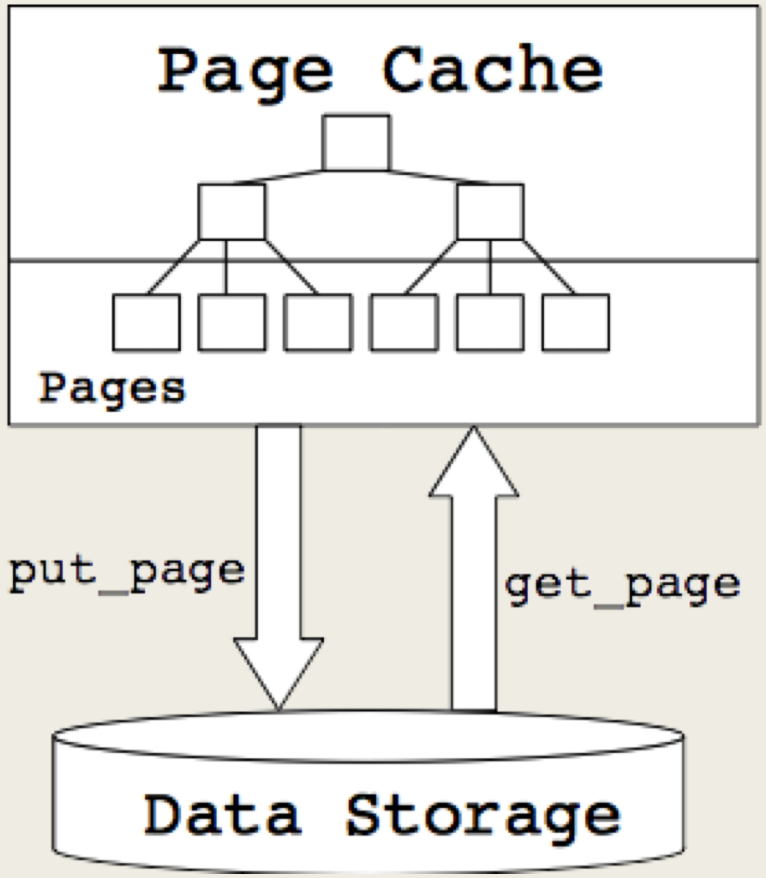
Memory mapped IO

- Application wants to use kernel Page Cache, but not keep a copy of the data on their site
- Same pages can be shared between Processes



Page Cache and Storage Device

- Page Cache is a set of pages per address space
- Possible to GET/PUT page to the page cache



What are we trying to achieve?

- Make use of the Page Cache for:
  - *Buffered IO*
  - *Sync IO*
- Make sure that Direct IO works and omit Page Cache
- Support mmap/munmap

Fragments: **mmap**

"support mmap(2)"

```
// file.c  fops for dm file
file_operations dummy_file_ops = {
…
  .mmap = dm_file_mmap,
…
// Handler of mmap
static int
dm_file_mmap(struct file *file,
             struct vm_area_struct *vma)

  return generic_file_mmap(file, vma);
```

```
//mm/filemap.c

/* This is used for a general mmap of a disk file
int generic_file_mmap(struct file * file, struct
vm_area_struct * vma)
{
…//
        file_accessed(file);
        vma->vm_ops = &generic_file_vm_ops;
        return 0;
}

/* These are filemap generic methods */
vm_operations_struct generic_file_vm_ops = {
  .fault          = filemap_fault,
  .map_pages      = filemap_map_pages,
  .page_mkwrite   = filemap_page_mkwrite,
};
```

Fragments: R/W through PC

```
/**
 * This is the "read_iter()"
 * routine for all filesystems
 * that can use the page cache
 * directly.                    */
ssize_t
generic_file_read_iter(struct
       kiocb *, struct iov_iter *)


file_operations dummy_file_ops = {
.read_iter  = dm_file_read_iter,
.write_iter = dm_file_write_iter,
 …
/* Plug it in */
static ssize_t
ext2_file_read_iter(struct kiocb *iocb,
                    struct iov_iter *to)
{
return generic_file_read_iter(iocb, to);
```

```
// mm/filemap.c
generic_file_read_iter(struct kiocb *iocb, struct iov_iter
*iter)
{
    struct file *file = iocb->ki_filp;
    struct address_space *mapping = file->f_mapping;
    struct inode *inode = mapping->host;
    loff_t size;

    size = i_size_read(inode);
    if (iocb->ki_flags & IOCB_NOWAIT) {
      if (filemap_range_has_page(mapping, iocb->ki_pos,
               iocb->ki_pos + count - 1))
        return -EAGAIN;
    } else {
      return filemap_write_and_wait_range(mapping,
            iocb->ki_pos, iocb->ki_pos + count - 1);
    }

    file_accessed(file);

    retval = mapping->a_ops->direct_IO(iocb, iter);

  }

  return generic_file_buffered_read(iocb, iter, retval);
```

Fragments: pages

"support r/w pages"

```
// inode as ops for dm_inode
struct address_space_operations
dm_as_ops = {
.readpage          = dm_readpage,
.readpages         = dm_readpages,
.writepage         = dm_writepage,
.writepages        = dm_writepages,
.write_begin       = dm_write_begin,
.write_end         = dm_write_end,
.bmap              = dm_bmap,
```

```
//inode.c
static int dm_readpage(struct file *file,
                       struct page *page)
{
  return mpage_readpage(page, dm_get_block);
}

static int
dm_readpages(struct file *file, struct address_space
*mapping, struct list_head *pages, unsigned nr_pages)
{
  return mpage_readpages(mapping, pages, nr_pages,
                         dm_get_block);
}

static int dm_writepage(struct page *page,
                        struct writeback_control *wbc)
{
  return block_write_full_page(page, dm_get_block, wbc);
}

static int
dm_writepages(struct address_space *mapping,
              struct writeback_control *wbc)
{
  return mpage_writepages(mapping, wbc, dm_get_block);
}
…
```

Fragments: get_block

```
/**
 * Get Block with block index
 * iblock
 * map it to the buffer_head
 * This is FS specific routine
 */
int dm_get_block(
    struct inode *inode,
    sector_t iblock,
    struct buffer_head *bh_result,
    int create)
```

```
int dm_get_block(struct inode *inode, sector_t iblock,
                        struct buffer_head *bh_result, int create)
{
            unsigned max_blocks =
                            bh_result->b_size >> inode->i_blkbits;
            bool new = false, boundary = false;
            u32 bno;
            int ret;

/**
 * return > 0, 'N' of blocks mapped or allocated.
 * return = 0, if plain lookup failed.
 * return < 0, error case.
 */
            ret = dm_get_blocks(inode, iblock, max_blocks, &bno,
                                        &new, &boundary, create);
            if (ret <= 0)
                        return ret;

            map_bh(bh_result, inode->i_sb, bno);
            bh_result->b_size = (ret << inode->i_blkbits);
            if (new)
                        set_buffer_new(bh_result);
            if (boundary)
                        set_buffer_boundary(bh_result);
            return 0;
}
```

Fragments: Get Blocks

```
/**
 * Return the blocks
 * routine for Filesystem
 * iblock is start block
 * maxblocks max number of blocks
 */
static int dm_get_blocks(
    struct inode *inode,
    sector_t iblock,
    unsigned long maxblocks,
    u32 *bno,
    bool *new,
    bool *boundary,
    int create)
```

```
/* Iterate all extends see if we can find a block range
inside existing range */
while (count < maxblocks && count <= blocks_to_boundary)
/* Simple case block found in existing blocks and do
not overflow return it */
if (err != -EAGAIN)
    goto got_it;

/* Next simple case - plain lookup or failed read of
indirect block  so we need to chain them. */
if (!create || err == -EIO)
    goto cleanup;

/* We do need to create new blocks */
err = dm_alloc_blocks(inode, indirect_blks, &count,
                      offsets + (partial - chain));

/* Truncate if size bigger than requested */
if (count > blocks_to_boundary)
    *boundary = true;

err = count;
```

Fragments: DirectIO

"support for DirectIO"

```
// inode as ops for dm_inode

struct address_space_operations
dm_as_ops = {
...
    .direct_IO = dm_direct_IO,
```

```
// We can again use magic get_blok function
// and plug to FS framework

static ssize_t
dm_direct_IO(struct kiocb *iocb, struct iov_iter *iter)
{
        struct file *file = iocb->ki_filp;
        struct address_space *mapping = file->f_mapping;
        struct inode *inode = mapping->host;
        size_t count = iov_iter_count(iter);
        loff_t offset = iocb->ki_pos;

        return blockdev_direct_IO(iocb, inode, iter,
                                  dm_get_block);
}
```

Other resources:

- J.Lions: "A commentary on the sixth edition UNIX Operating System" [1977]
- R McDougall, J. Mauro: "Solaris Internals Second edition" [2006]
- Uresh Vahalia: "UNIX Internals The new Frontiers" [1996]
- Steve D. Pate: "UNIX Filesystems: Evolution, Design and Implementation"
- Ext2, Ufs: Linux source code
- github.com/gotoco/dummyfs

Q&A

@mathfriday