| 文档版本 | 说明 | 作者 | 创建日期 |
| --- | --- | --- | --- |
| V0.1 | Linux系统编程：入门篇视频配套PPT | 王利涛 | 2018年10月14日 |
| V0.2 | 第01期：揭开文件系统的神秘面纱 | 王利涛 | 2018年11月07日 |
| V0.3 | 第02期：文件IO编程实战 | 王利涛 | 2018年11月25日 |
| V0.4 | 第03期：IO缓存与内存映射 | 王利涛 | 2018年12月11日 |

# Linux系统编程

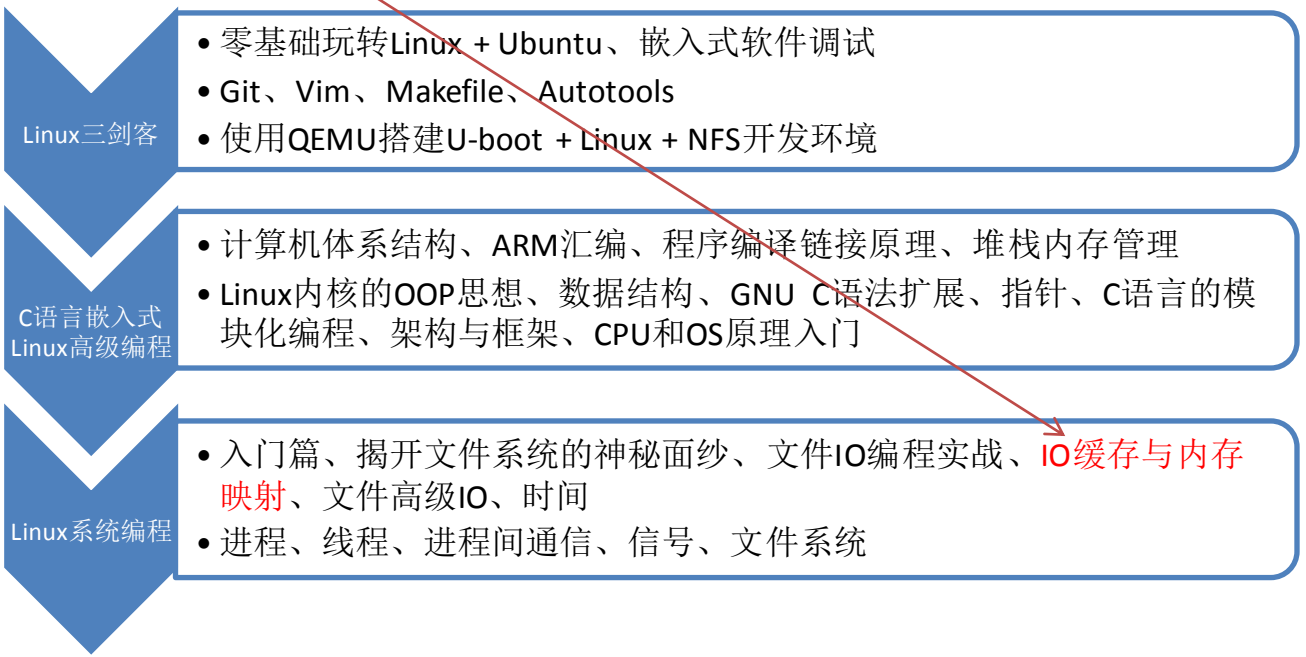
## 第03期：IO缓存与内存映射

# 《嵌入式工程师自修养》视频教程

- 第00步：Linux三剑客
- 第01步：C语言嵌入式Linux高级编程
- 第03步：Linux系统编程
- 第04步：Linux内核编程
- 第05步：嵌入式驱动开发
- 第06步：项目实战
- ----------
- 详情咨询QQ：3284757626
- 视频淘宝店：wanglitao.taobao.com
- 博客：www.zhaixue.cc
- 微信公众号：

# 学习路线图

- We are here…



# Linux系统编程第03期
# I/O缓存与内存映射

# 计算机中的缓存

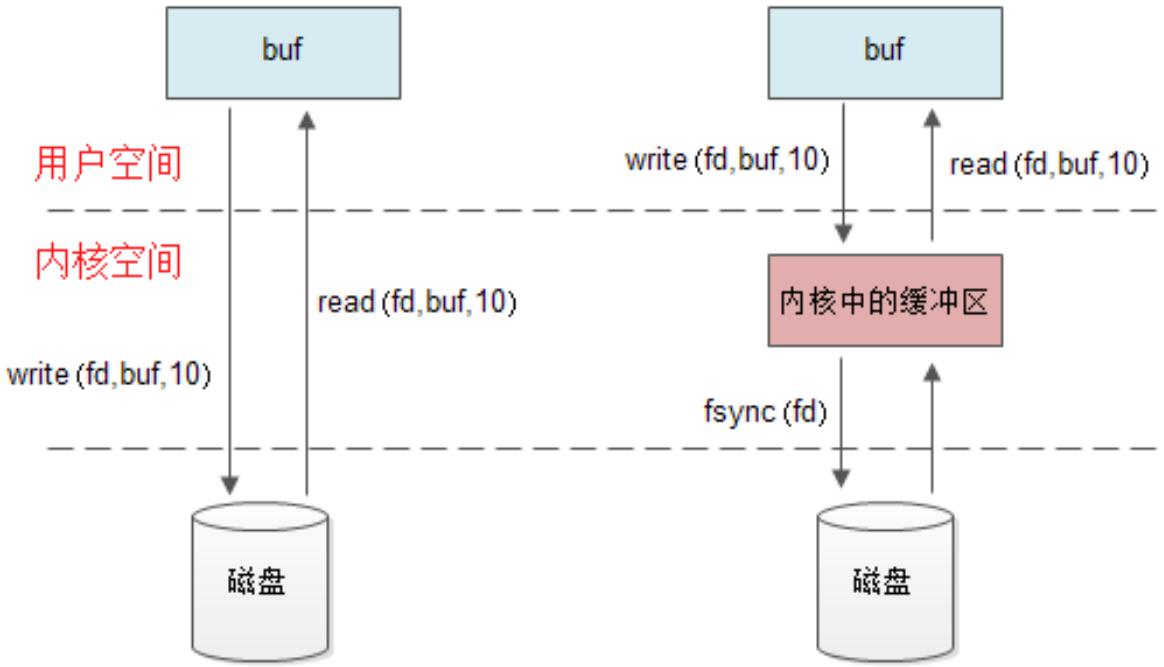
- 无处不在的缓存
  - CPU级别的缓存：cache、TLB，让CPU更高效运行
  - 操作系统的缓存：页缓存、slab，让系统更高效运行
  - 应用层的缓存：内存管理、IO应用缓存，让应用更高效运行

# I/O缓存

- 提高文件I/O性能
  - 操作系统级的缓存
  - 应用层的缓存
  - 内存映射
  - API系统调用、参数

# 页高速缓存(上)
# 页缓存读写流程

# 内核中的缓冲区

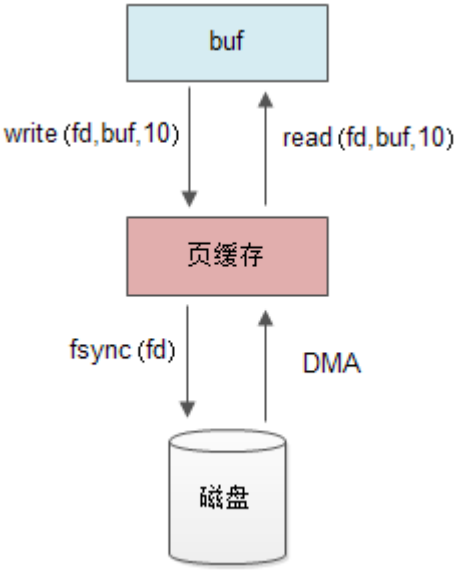

buf
buf
用户空间
write (fd,buf,10)
read (fd,buf,10)
内核空间
read (fd,buf,10)
内核中的缓冲区
write (fd,buf,10)
fsync (fd)
磁盘
磁盘

# 页高速缓存

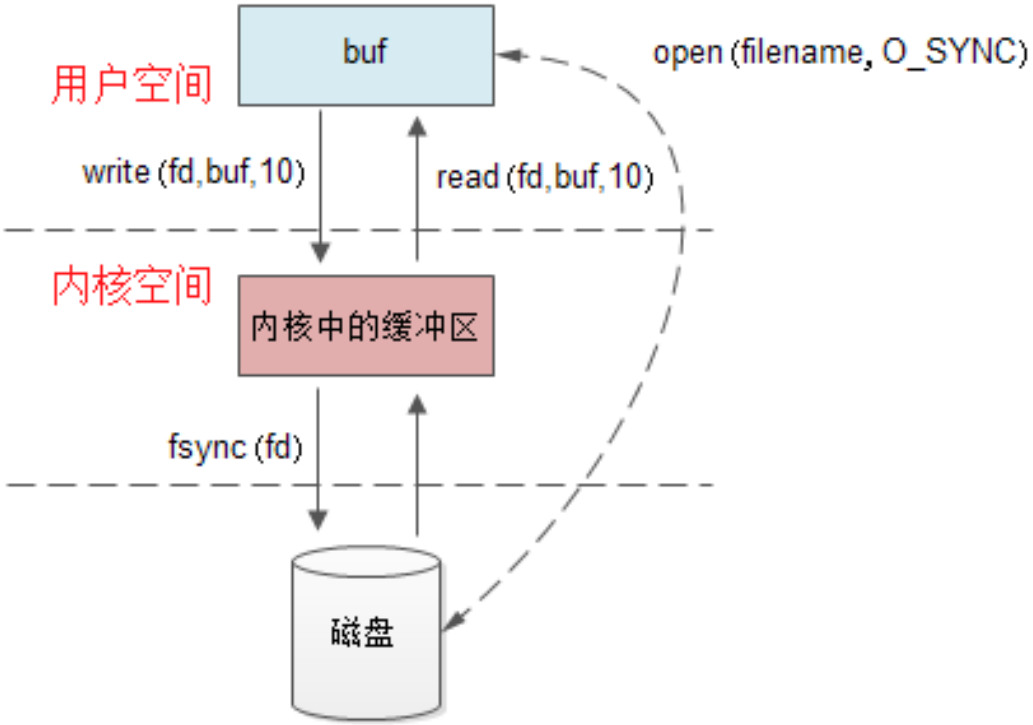
- 页缓存
  - 通过Linux内核缓冲区实现了一种磁盘缓存机制
  - 基本原理：局部原理、时间局部、空间局部
  - 缓存基于页的对象：普通文件、块设备、内存映射
  - 优点：一定程度上分离了应用程序空间和物理设备、减少IO读盘次数

# 页缓存和回写

- 读流程
  - 先到缓存中看数据是否存在，存在直接读取返回到用户空间
  - 若不存在，将磁盘数据缓存到page cache中
  - 然后从page cache中读取数据到用户空间
- 写流程
  - 将用户空间数据写到page cache中
  - 当page cache数据达到阈值或时间超时，将将数据回写到磁盘



# 同步方式


buf
open (filename, O_SYNC)
用户空间
write (fd,buf,10)
read (fd,buf,10)
内核空间
内核中的缓冲区
fsync (fd)
磁盘

# 实验

- 页缓存读写实验
  - $ dd if=/dev/zero of=test.dat bs=1M count=10 写一个文件
  - $ cat /proc/meminfo | grep Dirty 查看当前脏页
  - $ sync 刷新
  - $ cat /proc/meminfo | grep Dirty 再次查看

# 作业

- 写一个文件写的程序：
  - 1）普通方式写
  - 2）同步写

todo 3.3

# 页高速缓存(中)
# 内存管理

# 物理内存管理

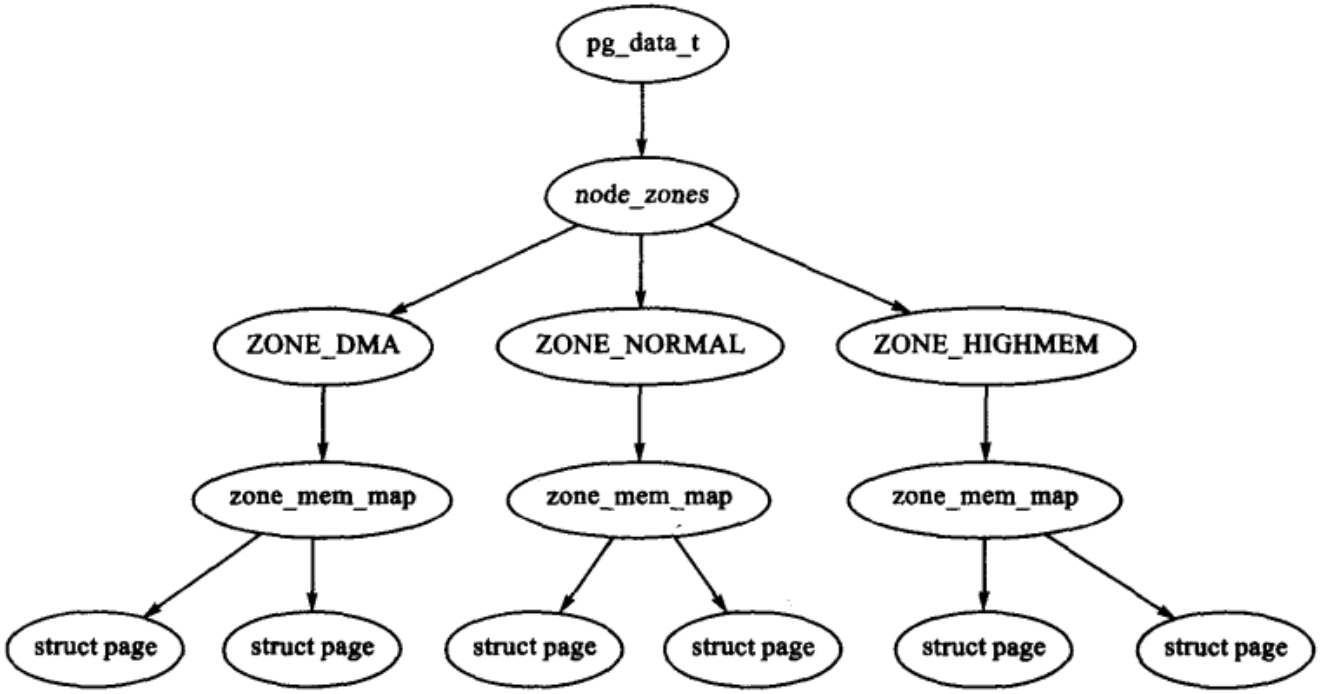
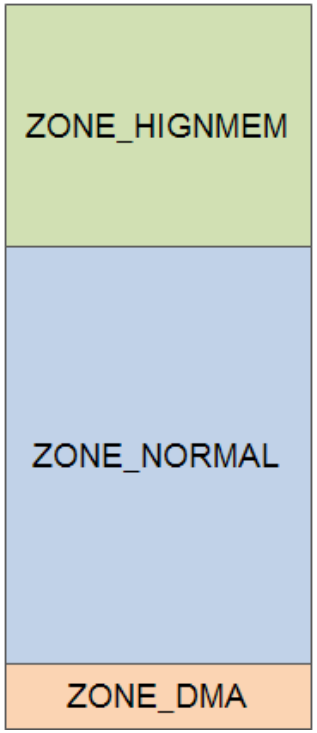
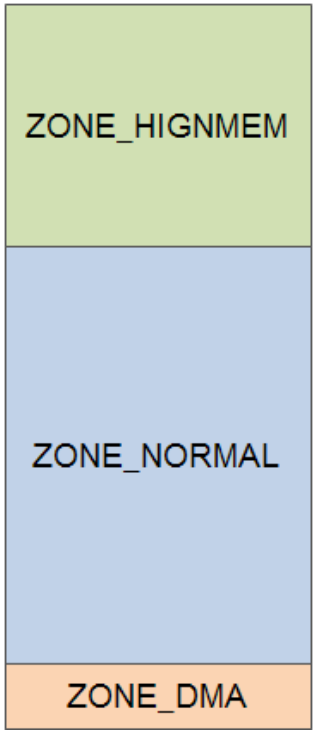


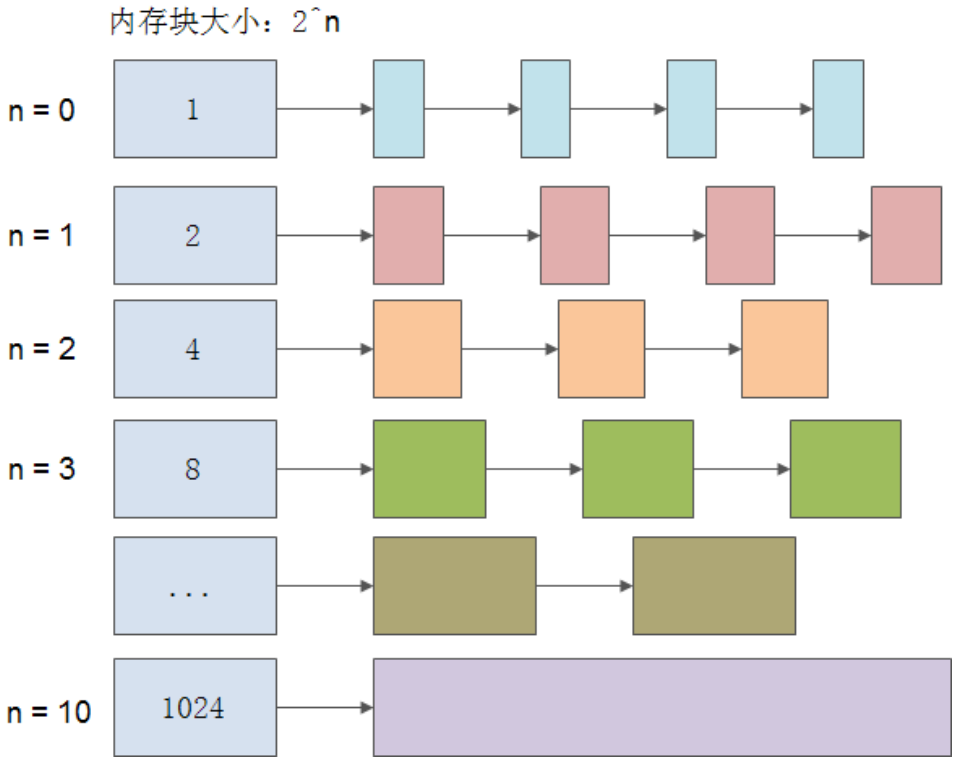

图 2.1 节点、管理区和页面的关系



# 物理内存管理

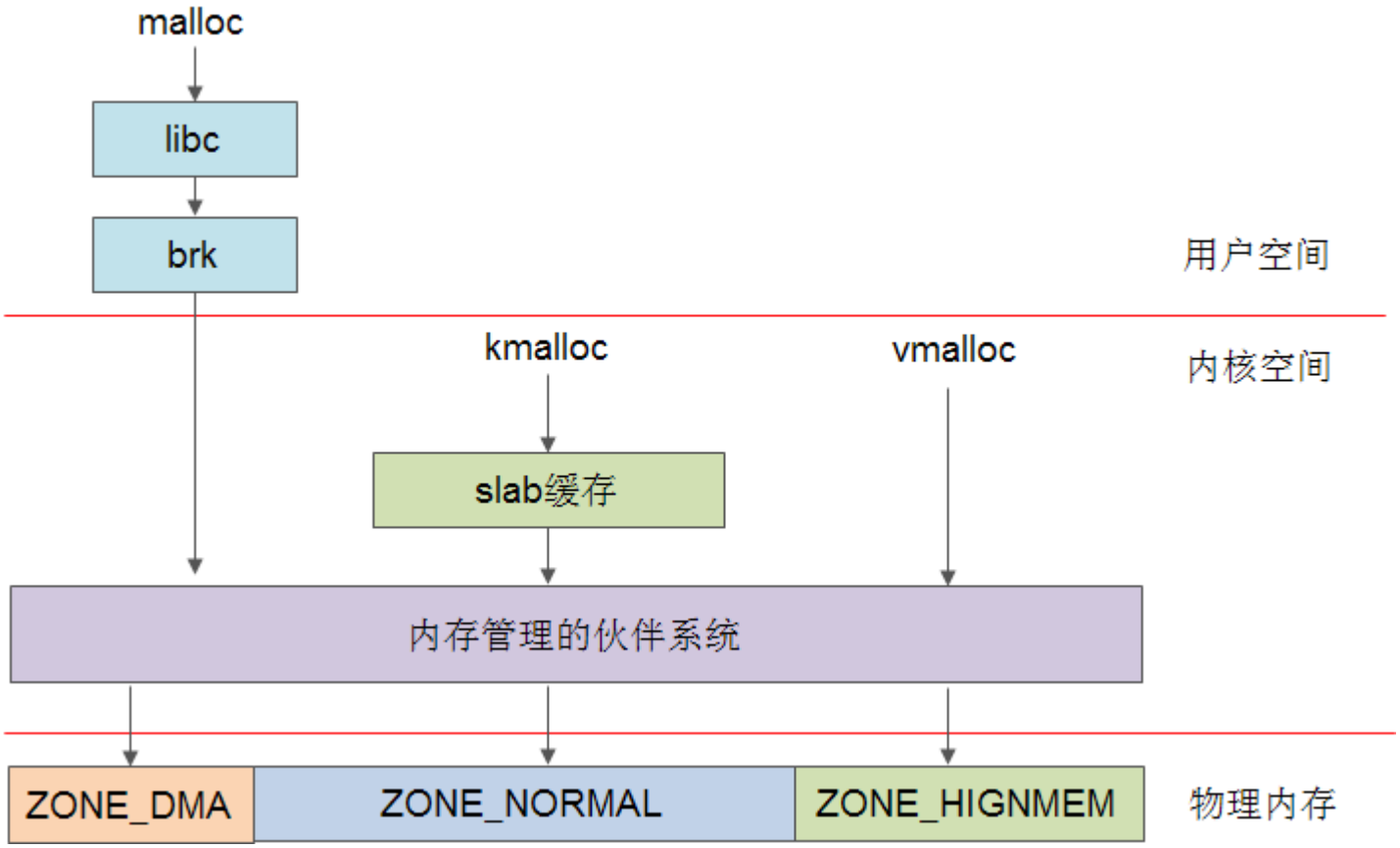
- 伙伴算法


ZONE_HIGNMEM
ZONE_NORMAL
ZONE_DMA

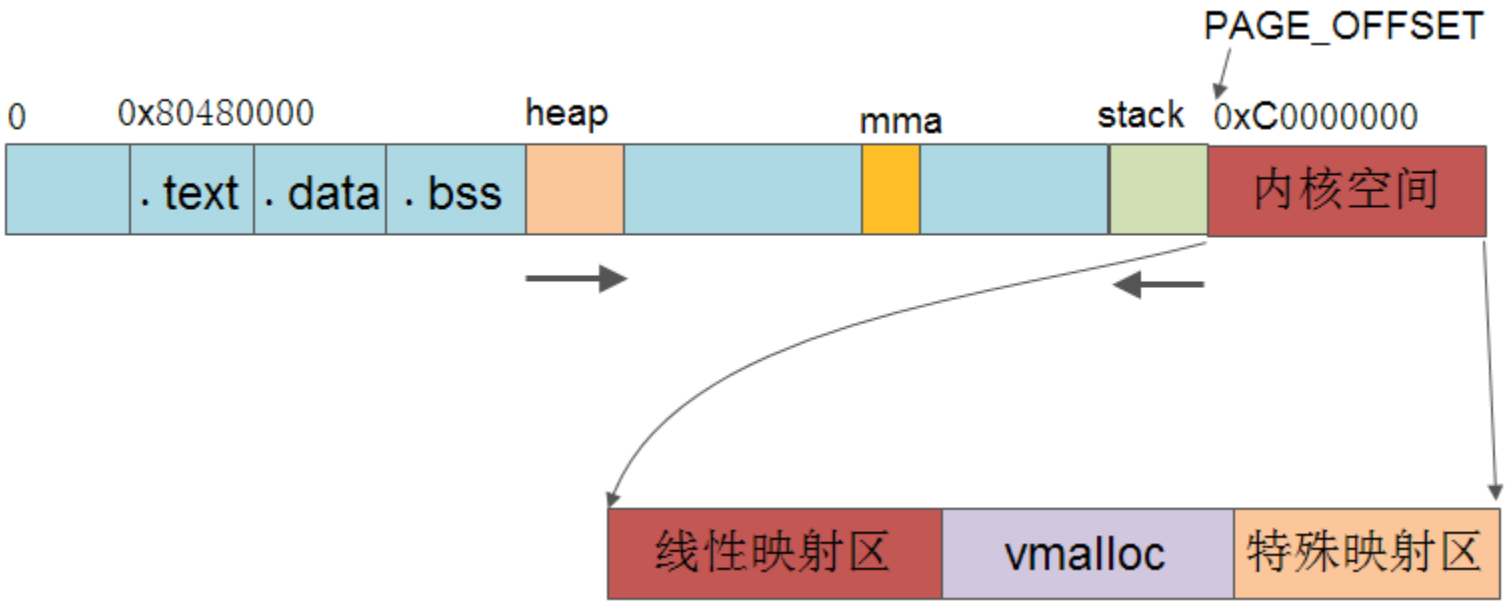

内存块大小：2^n
n = 0
1
n = 1
2
n = 2
4
n = 3
8
...
n = 10
1024

# 物理内存管理

- 内存申请

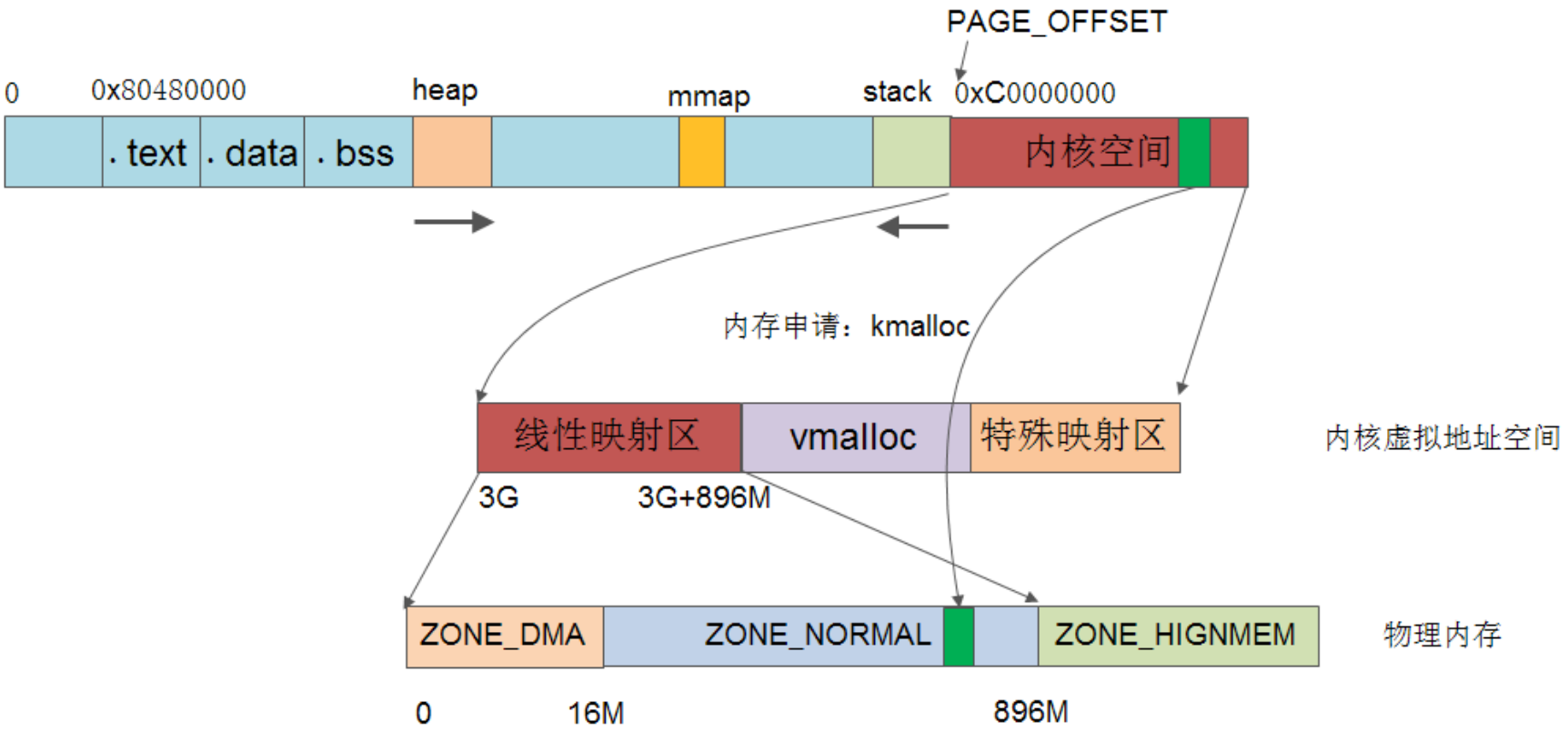

malloc
libc
brk
用户空间
kmalloc
vmalloc
内核空间
slab缓存
内存管理的伙伴系统
ZONE_DMA
ZONE_NORMAL
ZONE_HIGNMEM
物理内存

# Linux虚拟地址空间



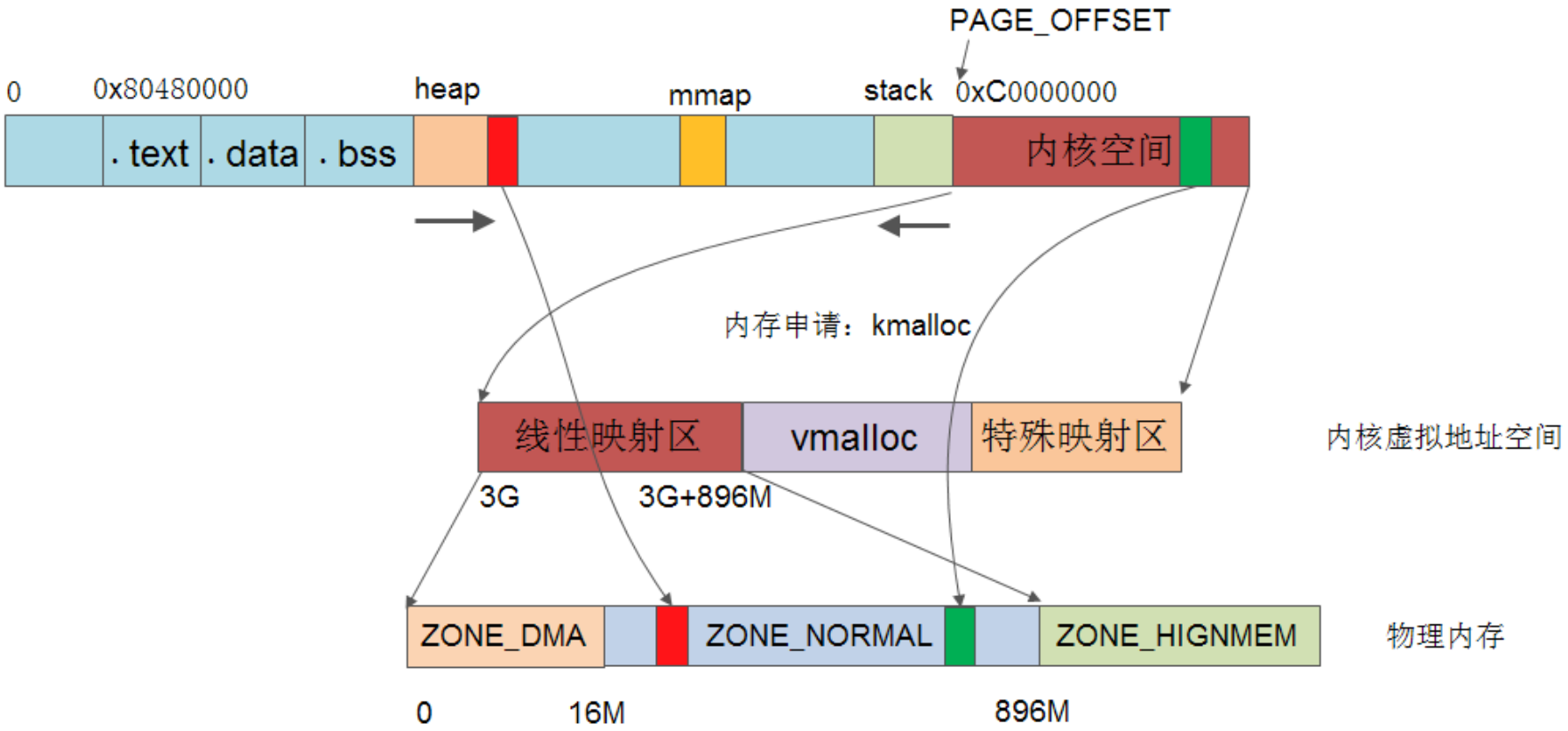
# Linux虚拟地址空间

- 内核空间到物理内存的映射



# Linux虚拟地址空间

- 用户空间到物理内存的映射



todo 3.4

# 页高速缓存(下)
# 页缓存的实现

# 页缓存对象：属性

```
struct address_space {
        struct inode            *host;          /* owner: inode, block_device */
        struct radix_tree_root page_tree; /* radix tree of all pages */
        spinlock_t              tree_lock;  /* and lock protecting it */
        atomic_t                i_mmap_writable;/* count VM_SHARED mappings */
        struct rb_root          i_mmap;         /* tree of private and shared mappings */
        struct rw_semaphore  i_mmap_rwsem;      /* protect tree, count, list */
        /* Protected by tree_lock together with the radix tree */
        unsigned long           nrpages;    /* number of total pages */
        unsigned long           nrshadows;      /* number of shadow entries */
        pgoff_t                 writeback_index;/* writeback starts here */
        const struct address_space_operations  *a_ops;          /* methods */
        unsigned long           flags;          /* error bits/gfp mask */
        spinlock_t  private_lock;           /* for use by the address_space  */
        struct list_head        private_list;       /* ditto */
        void                    *private_data;      /* ditto */
}; //通过该对象，可以将文件系统、内存管理系统进行关联
```

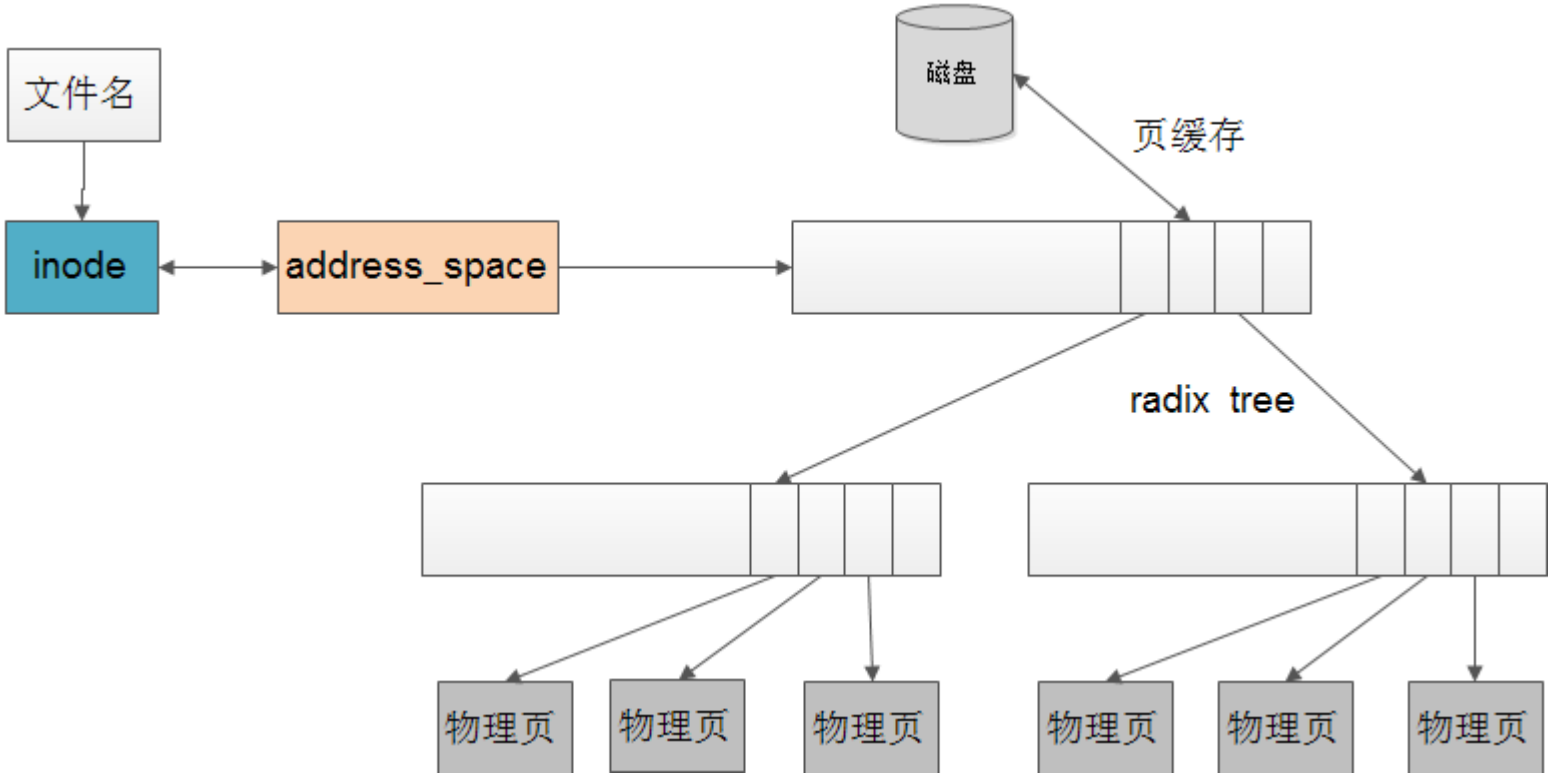
# 页缓存对象：方法

```
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);
        …
        sector_t (*bmap)(struct address_space *, sector_t);
        void (*invalidatepage) (struct page *, unsigned int, unsigned int);
        int (*releasepage) (struct page *, gfp_t);
        void (*freepage)(struct page *);
        ssize_t (*direct_IO)(struct kiocb *, struct iov_iter *iter, loff_t offset);
};  // 文件位置偏移 → 页偏移量 → 文件系统块号：block → 磁盘扇区号
```

# 页缓存对象：物理页

```
struct  page {
        unsigned long flags;
        union {
                struct  address_space  *mapping;
                void *s_mem;                        /* slab first object */
        };
        union {
                struct list_head lru;
                struct {                            /* slub per cpu partial pages */
                        struct page *next;          /* Next partial slab */
                        short int pages;
                        short int pobjects;
                };
        union {
                unsigned long private;
                spinlock_t ptl;
                struct kmem_cache *slab_cache;
        };
} //一个页可以是文件页缓存、可以是交换缓存、也可以是普通内存
```

# 页缓存数据结构图

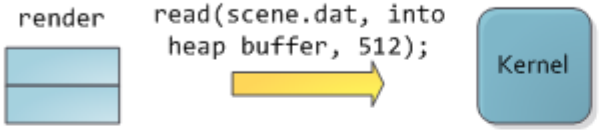

文件名
磁盘
页缓存
inode
address_space
radix tree
物理页
物理页
物理页
物理页
物理页
物理页

# 读文件

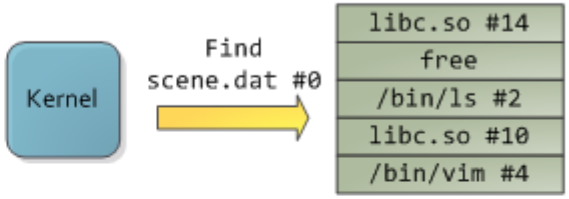
- 基本流程
  - 数据结构关联：inode->i_mapping指向address_space对象
  - 数据结构关联：address_space->host指向inode
  - 数据结构关联：page->mapping指向页缓存owner的address_space
  - 系统调用传参：文件描述符 + 文件位置偏移
  - 系统找到文件的address_space，根据偏移量到页缓存中查找page
  - 若查找到，返回数据到用户空间
  - 若没查到，内核新建一个page并加入页缓存，数据从磁盘载入该页
  - 调用readpage方法将数据返回给用户空间

# 读文件示例

1. Render asks for 512 bytes of scene.dat starting at offset 0.

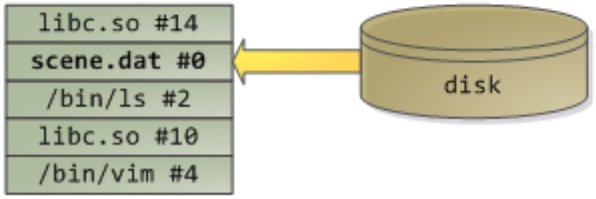



2. Kernel searches the page cache for the 4KB chunk of scene.dat satisfying the request. Suppose the data is not cached.

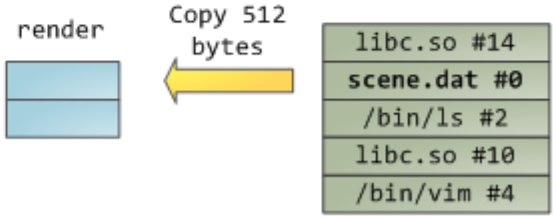





3. Kernel allocates page frame, initiates I/O requests for 4KB of scene.dat starting at offset 0 to be copied to allocated page frame




4. Kernel copies the requested 512 bytes from page cache to user buffer, read() system call ends.



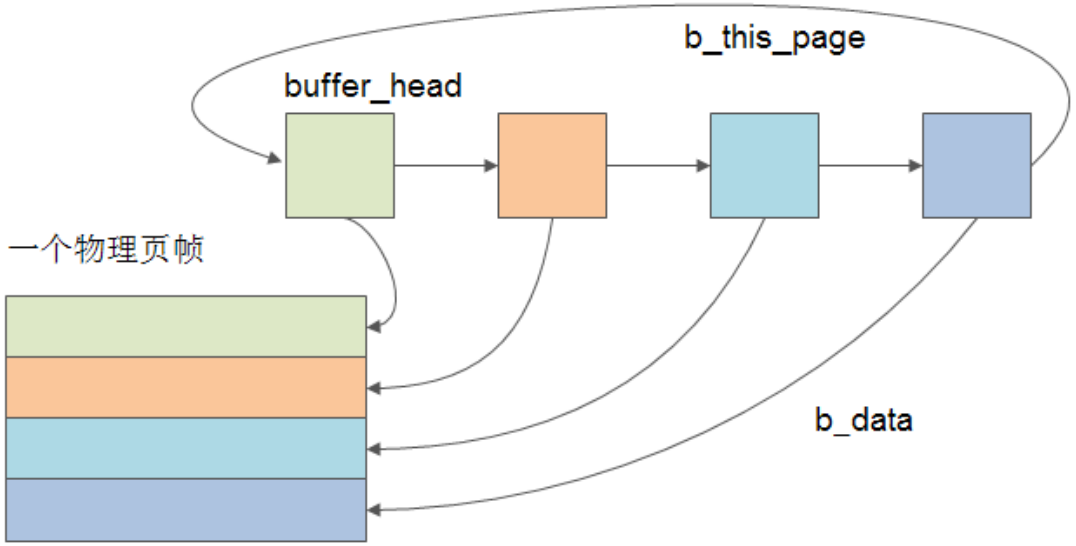
# 写文件

- 基本流程
  - 数据结构关联：inode->i_mapping指向address_space对象
  - 数据结构关联：address_space->host指向inode
  - 数据结构关联：page->mapping指向页缓存owner的address_space
  - 系统调用传参：文件描述符+ 文件位置偏移
  - 系统找到文件的address_space，根据偏移量到页缓存中查找page
  - 若查找到，将数据写入到该页缓存中，该页成为“脏页”
  - 若没查到，从缓存中分配空闲页，数据从用户空间写入该页
  - 当数据满足一定空间或时间阈值，将脏页中的数据回写到磁盘
  - 守护进程：pdflush

# 块设备驱动架构

# 块缓存

- Linux内核中的缓存
  - 早期使用块缓存，新的内核逐渐使用页缓存
  - 块缓存：比页缓存小、长度可变，依赖于设备(或文件系统)
  - 块缓存的实现：基于页缓存
    - 缓冲头：buffer_head，缓冲区的元数据信息：块号、块长度
    - 每个buffer_head指向一个缓冲区，一个页可以细分为若干个缓冲区
    - 内核物理页与磁盘物理块之间的桥梁



# 块缓存

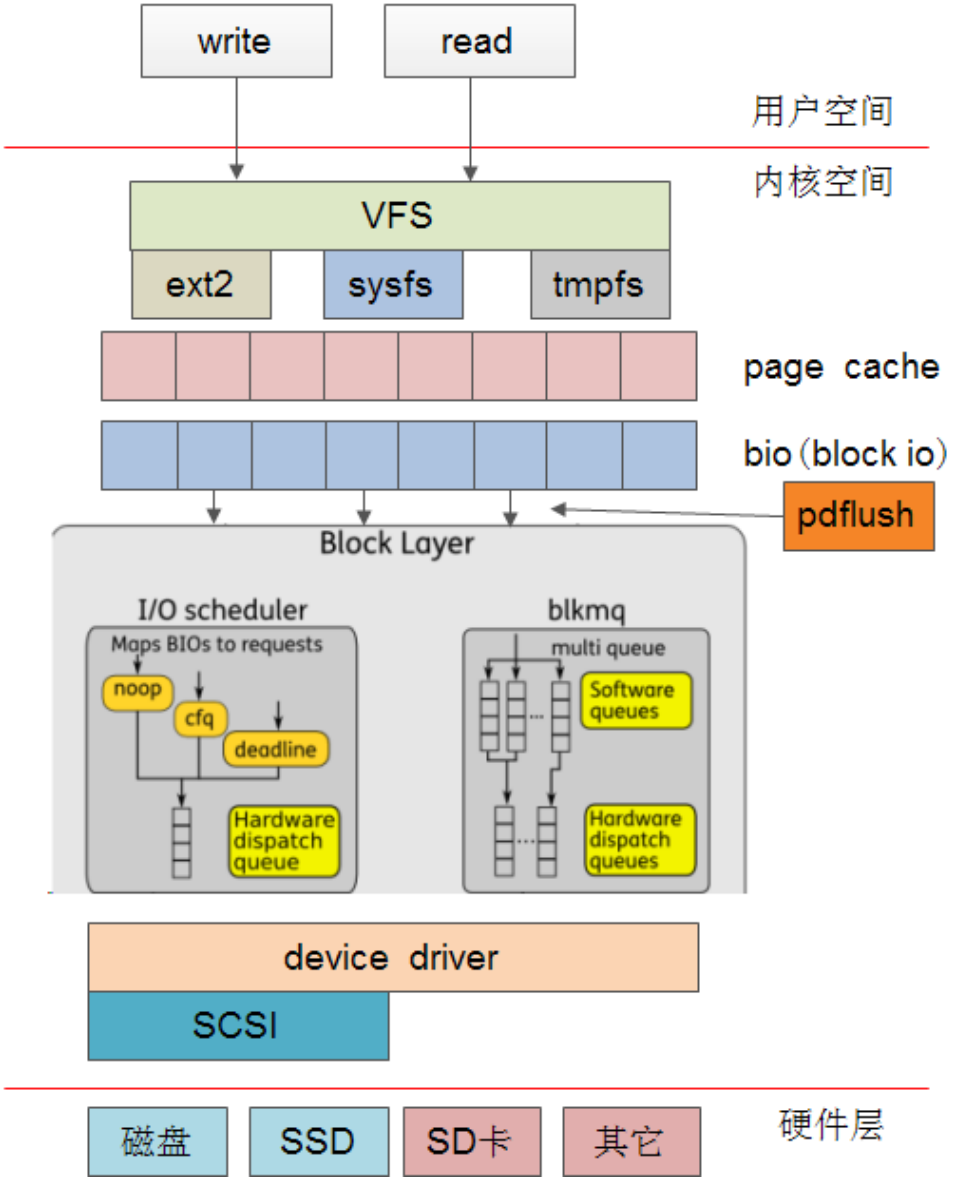
```
struct  buffer_head {
        unsigned long           b_state;         /* buffer state bitmap (see above) */
        struct buffer_head      *b_this_page; /* circular list of page's buffers */
        struct page             *b_page;        /* the page this bh is mapped to */
        sector_t                b_blocknr;      /* start block number */
        size_t                  b_size;         /* size of mapping */
        char                    *b_data;        /* pointer to data within the page */

        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 */
        struct address_space *b_assoc_map;  /* mapping this buffer is
                                                                associated with */
        atomic_t                b_count;        /* users using this buffer_head */
};
```

# bio结构体

- main unit of I/O for the block layer and lower layers
- struct gendisk -> block_device -> /dev/sdax -> inode

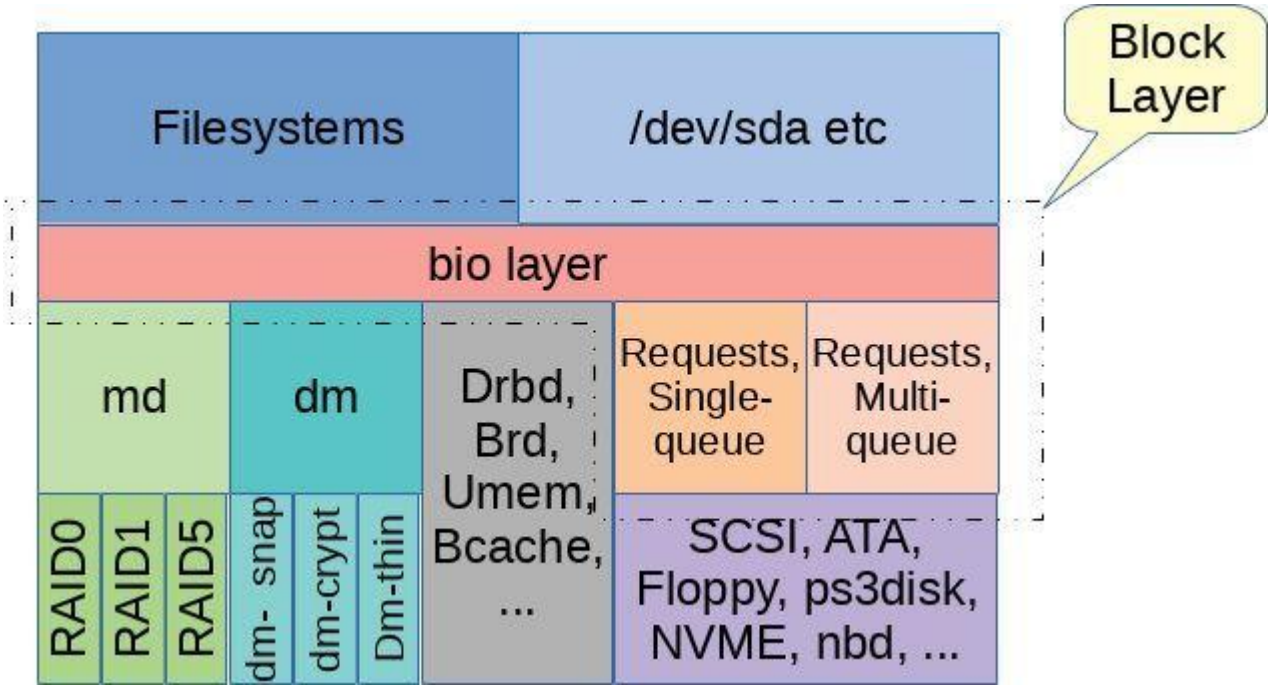
```
struct bio {
        struct bio              *bi_next;       /* request queue link */
        struct block_device     *bi_bdev;
        unsigned int            bi_flags;       /* status, command, etc */
        unsigned long           bi_rw;
        struct bvec_iter        bi_iter;
        unsigned int            bi_phys_segments;
        unsigned int            bi_seg_front_size;
        unsigned int            bi_seg_back_size;
        atomic_t                __bi_remaining;
        bio_end_io_t            *bi_end_io;
        void                    *bi_private;
        unsigned short          bi_vcnt;        /* how many bio_vec's */
        unsigned short          bi_max_vecs;    /* max bvl_vecs we can hold */
        atomic_t                __bi_cnt;       /* pin count */
        struct bio_vec          *bi_io_vec;     /* the actual vec list */
        struct bio_set          *bi_pool;
        struct bio_vec          bi_inline_vecs[0];
};
```

# 块设备驱动架构



# 块设备驱动

- 各种各样的块设备



# 用户空间的IO缓存

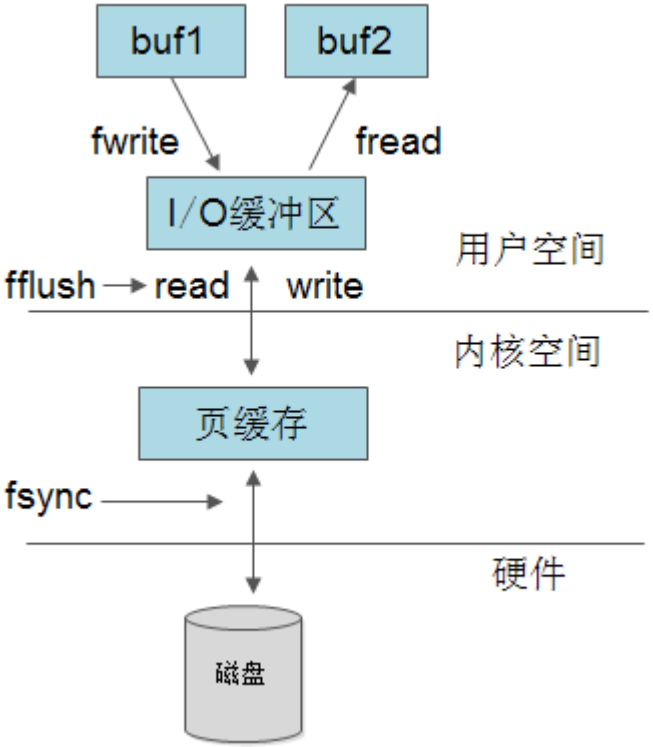
# 用户空间的IO缓冲区

- 系统调用的开销
  - 切换CPU到内核模式
  - 数据拷贝
  - 切换CPU到用户模式
- C标准库IO缓冲区
  - 在用户空间，为每个打开的文件
    - 分配一个I/O缓冲区
    - 分配一个文件描述符
    - I/O缓冲区信息和文件描述符一起封装在FILE结构体中
  - size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
  - size_t fwrite(const void *ptr, size_t size, size_t nmemb,FILE *stream);

# FILE结构体

```
struct _IO_FILE {
        int          _flags;
        char* _IO_read_ptr;   /* Current read pointer */
        char* _IO_read_end;   /* End of get area. */
        char* _IO_read_base;  /* Start of putback+get area. */
        char* _IO_write_base;         /* Start of put area. */
        char* _IO_write_ptr;  /* Current put pointer. */
        char* _IO_write_end;  /* End of put area. */
        char* _IO_buf_base;   /* Start of reserve area. */
        char* _IO_buf_end;    /* End of reserve area. */
        struct _IO_FILE *_chain;
        int          _fileno;
        _IO_off_t _old_offset; /* This used to be _offset but it's too small. */
        unsigned short _cur_column;
        signed char _vtable_offset;
        char _shortbuf[1];
        _IO_lock_t *_lock;
};
typedef struct _IO_FILE  FILE;
```

# 文件读写流程


buf1
buf2
fwrite
fread
I/O缓冲区
用户空间
fflush
read
write
内核空间
页缓存
fsync
硬件
磁盘

# 用户空间的IO缓存

- 三种模式
  - 块缓冲(block_buffered)：
    - 固定字节的缓冲区大小，比如跟文件相关的流都是块缓冲
    - 标准IO称块缓冲为完全缓冲(full buffering)
  - 行缓冲(line_buffered)：
    - 遇到换行符，缓冲区的数据会拷贝到内核缓冲区
  - 无缓冲(unbuffered)：
    - 数据直接拷贝到内核缓冲区
    - 如：标准错误stderr采用无缓冲模式

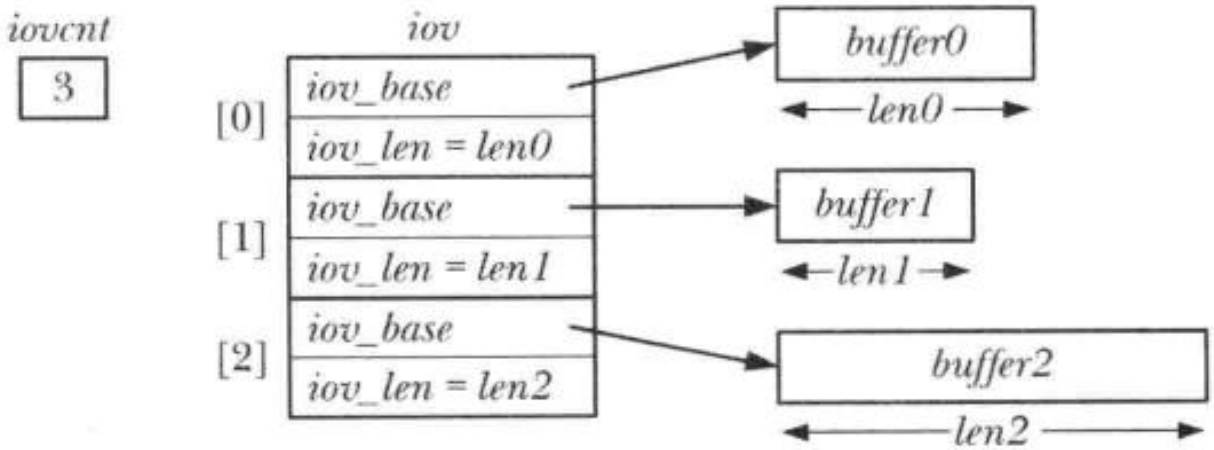
# 自定义缓冲区

- C标准库函数
  - int setvbuf (FILE *stream, char *buf, int mode, size_t size);
  - stream：指向流的指针
  - buf：缓冲区地址
  - mode：缓冲区类型
    - __IOFBF：当缓冲区为空，从流中读入数据；缓冲区满时向流写入数据
    - __IOLBF：每次从流中读入一行数据或向流中写入一行数据
    - __IONBF：直接从流中读入数据或直接向流中写入数据，无缓冲区
  - size：缓冲区内字节的数量

# Scatter-gather I/O

# Scatter-gather I/O

- 分散/聚集IO
  - 优点：更进一步减少了系统调用的次数
  - 单个向量I/O操作取代多个线性I/O操作，效率更高



# 分散/聚集 I/O

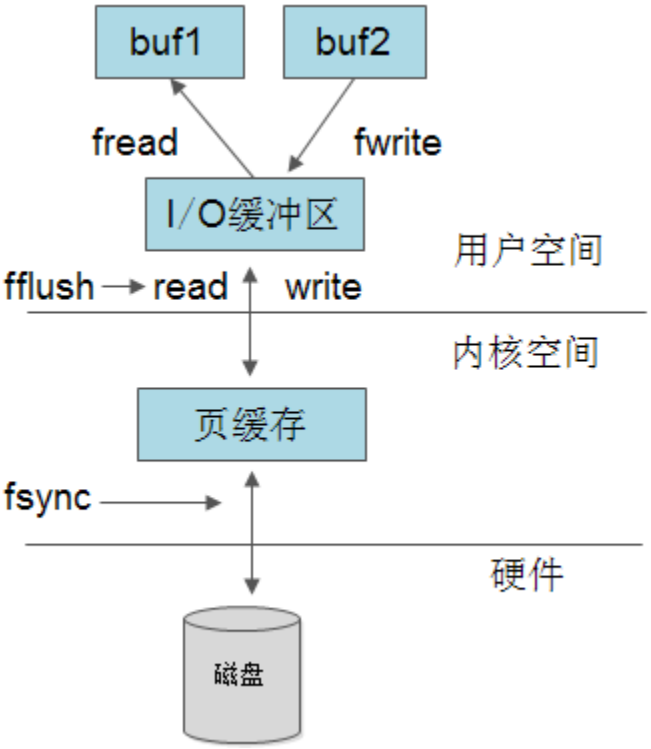
- 系统调用函数原型
  - ssize_t readv (int fd, const struct iovec *iov, int iovcnt);
  - ssize_t writev (int fd, const struct iovec *iov, int iovcnt);

```
struct iovec
 {
        void    *iov_base;   /* Pointer to data. */
        size_t  iov_len;     /* Length of data. */
 };
```

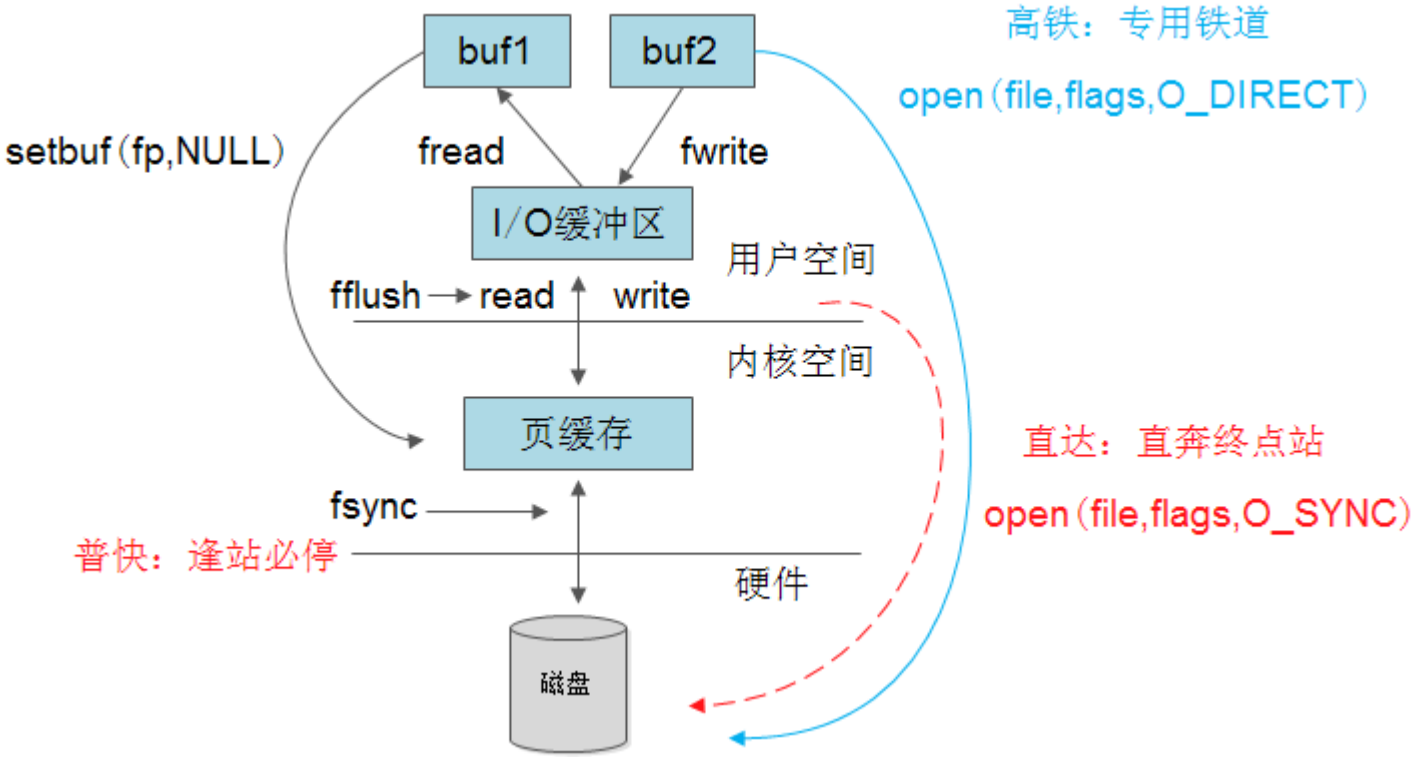
# 直接I/O

# 用户空间的IO缓存

- 优缺点：
  - 优点：减少了系统调用的次数，减少了系统开销
  - 缺点：增加了数据拷贝次数，增大了CPU和内存开销
    - 读：数据从内核页缓存拷贝到标准IO缓存，再拷贝到用户的buffer
    - 写：数据从用户的buffer拷贝到标准IO缓存，再拷贝到内核缓冲区



# 飞越“缓冲区”


buf1
buf2
高铁：专用铁道
open(file,flags,O_DIRECT)
setbuf(fp,NULL)
fread
fwrite
I/O缓冲区
用户空间
fflush
read
write
内核空间
页缓存
直达：直奔终点站
open(file,flags,O_SYNC)
fsync
普快：逢站必停
硬件
磁盘

# 自缓存应用程序

- 数据库
  - 采用直接IO，绕过缓冲区，直接读写磁盘
  - 在用户空间对IO进行缓存和读写优化
  - 频繁读写、小批量数据

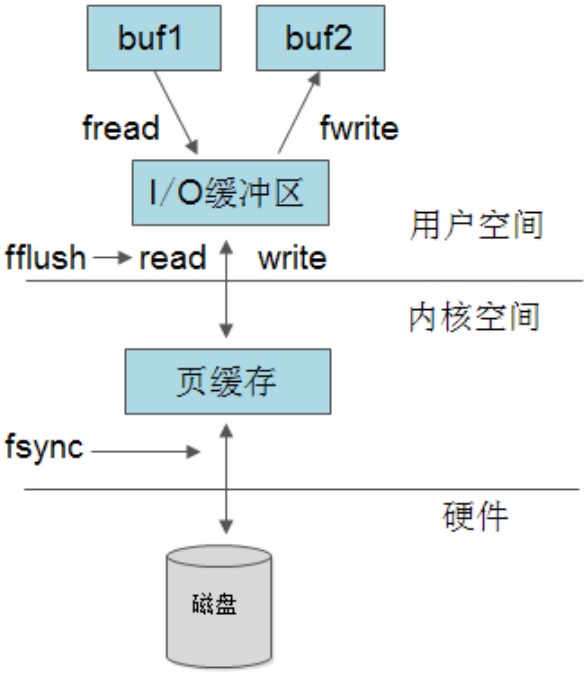
# 直接IO

- 遵守的原则
  - 用于传递数据的缓冲区，其内存边界必须对齐：块的整数倍
  - 数据传输的开始点，即文件和设备的偏移量，必须是块大小的整数倍
  - 待传递数据的长度必须是块大小的整数倍
  - 如果用户空间不优化的话，可能会降低系统性能

# 将文件映射到内存

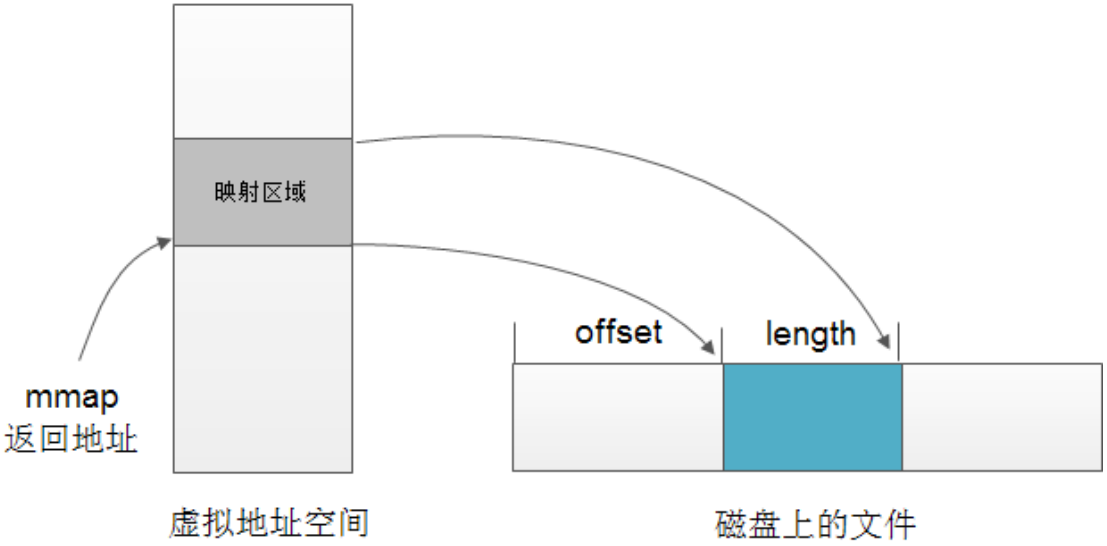
# 用户空间的IO缓存

- 优缺点：
  - 优点：减少了系统调用的次数，减少了系统开销
  - 缺点：增加了数据拷贝次数，增大了CPU和内存开销
    - 读：数据从内核页缓存拷贝到标准IO缓存，再拷贝到用户的buffer
    - 写：数据从用户的buffer拷贝到标准IO缓存，再拷贝到内核缓冲区



# 内存映射

- 将文件映射到内存
  - 内存地址与文件数据一一对应
  - 通过内存代替read/write 等I/O系统调用接口来访问文件
  - 减少内存拷贝、减少系统调用次数，提高系统性能

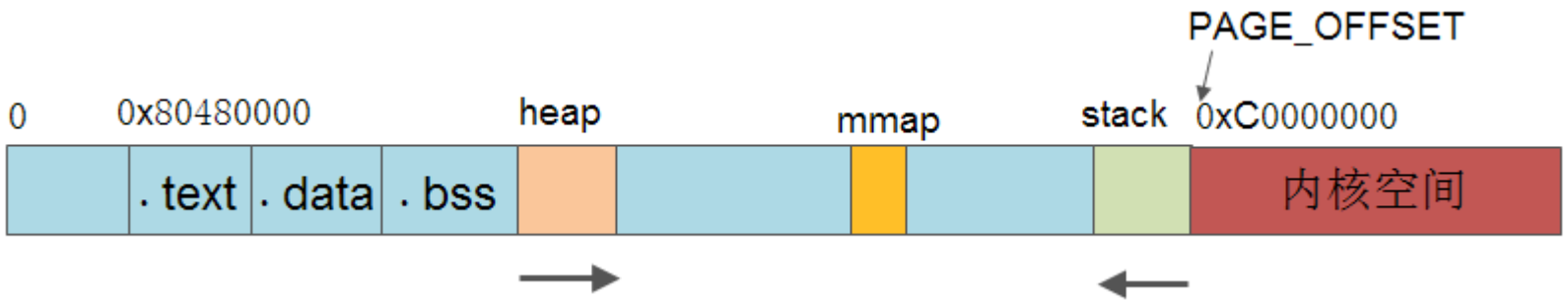


# 内存映射

- 系统调用接口
  - void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
    - addr：进程中要映射的虚拟内存起始地址，一般为NULL
    - length：要映射的内存区域大小
    - prot：内存保护标志： PROT_EXEC、 PROT_READ、 PROT_WRITE
    - flags：映射对象类型： MAP_FIXED、 MAP_SHARED、MAP_PRIVATE
    - fd：要映射文件的文件描述符
    - offset：文件位置偏移
    - mmap以页为单位操作：参数addr和offset必须按页对齐

# 文件映射内存的实现

# Linux进程虚拟地址空间



# Linux进程虚拟地址描述

- 结构体：task_struct

```
struct task_struct {
        volatile long state;
        void *stack;
        int prio, static_prio, normal_prio;
        unsigned int rt_priority;
        const struct sched_class *sched_class;
        struct sched_entity se;
        cpumask_t cpus_allowed;
        struct list_head tasks;

        struct mm_struct *mm, *active_mm;
        u32 vmacache_seqnum;
        int exit_state;
        int exit_code, exit_signal;
        …
}
```

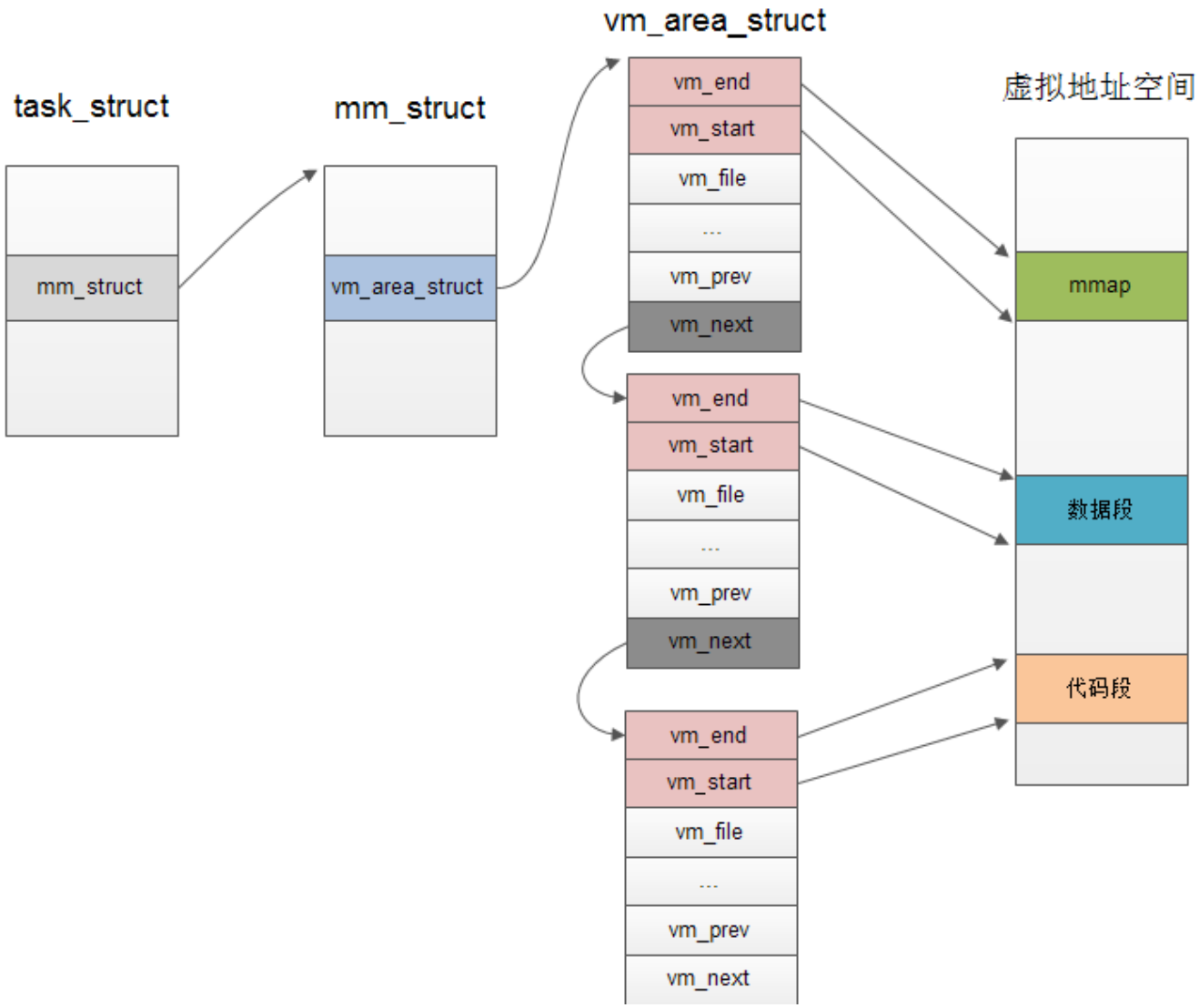
# Linux进程虚拟地址描述

```
struct mm_struct {
        struct vm_area_struct *mmap;    /* list of VMAs */
        struct rb_root mm_rb;
        u32 vmacache_seqnum;                   /* per-thread vmacache */
        unsigned long mmap_base;                       /* base of mmap area */
        unsigned long mmap_legacy_base; /* base of mmap area in bottom-up allocations */
        unsigned long task_size;                       /* size of task vm space */
        unsigned long highest_vm_end;                  /* highest vma end address */
        pgd_t * pgd;
        atomic_t mm_users;                     /* How many users with user space? */
        atomic_t mm_count;  /* How many references to "struct mm_struct" (users count as 1) */
        atomic_long_t nr_ptes;                                      /* PTE page table pages */
        int map_count;                                              /* number of VMAs */
        spinlock_t page_table_lock;                    /* Protects page tables and some counters */
        struct rw_semaphore mmap_sem;
}
```

# Linux进程虚拟地址描述

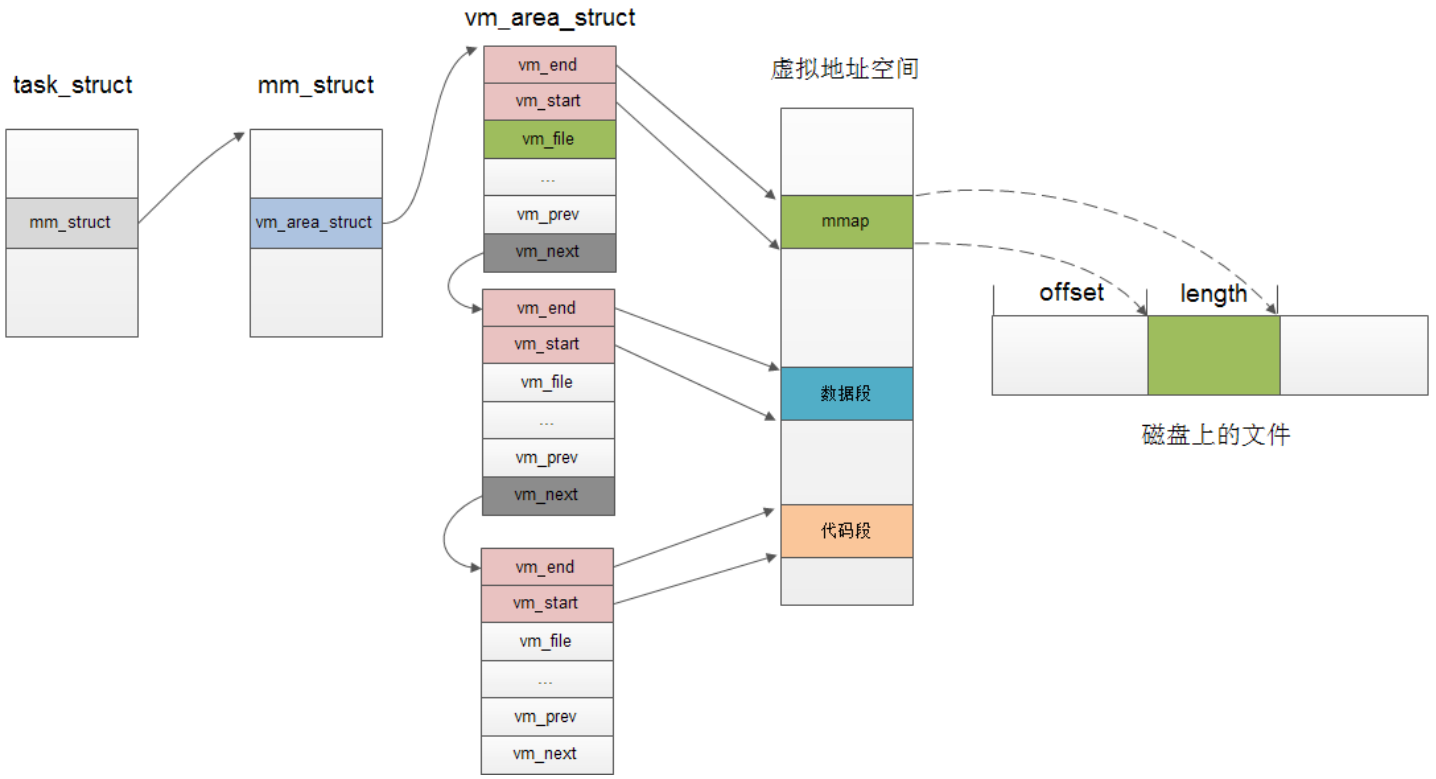
```
struct vm_area_struct {
        /* The first cache line has the info for VMA tree walking. */
        unsigned long    vm_start;                  /* Our start address within vm_mm. */
        unsigned long    vm_end;                    /* The first byte after our end address
                                                       within vm_mm. */
        struct vm_area_struct   *vm_next, *vm_prev; /* linked list of VM areas per task*/
        struct rb_node    vm_rb;
        unsigned long rb_subtree_gap;
        struct mm_struct    *vm_mm;                /* The address space we belong to. */
        pgprot_t     vm_page_prot;                 /* Access permissions of this VMA. */
        unsigned long    vm_flags;                 /* Flags, see mm.h. */
        const struct vm_operations_struct     *vm_ops;
        unsigned long vm_pgoff;                    /* Offset (within vm_file) in PAGE_SIZE
                                                       units, *not* PAGE_CACHE_SIZE  */
        struct file * vm_file;                     /* File we map to (can be NULL). */
        void * vm_private_data;                    /* was vm_pte (shared mem) */
        …
};
```

# Linux进程虚拟地址描述

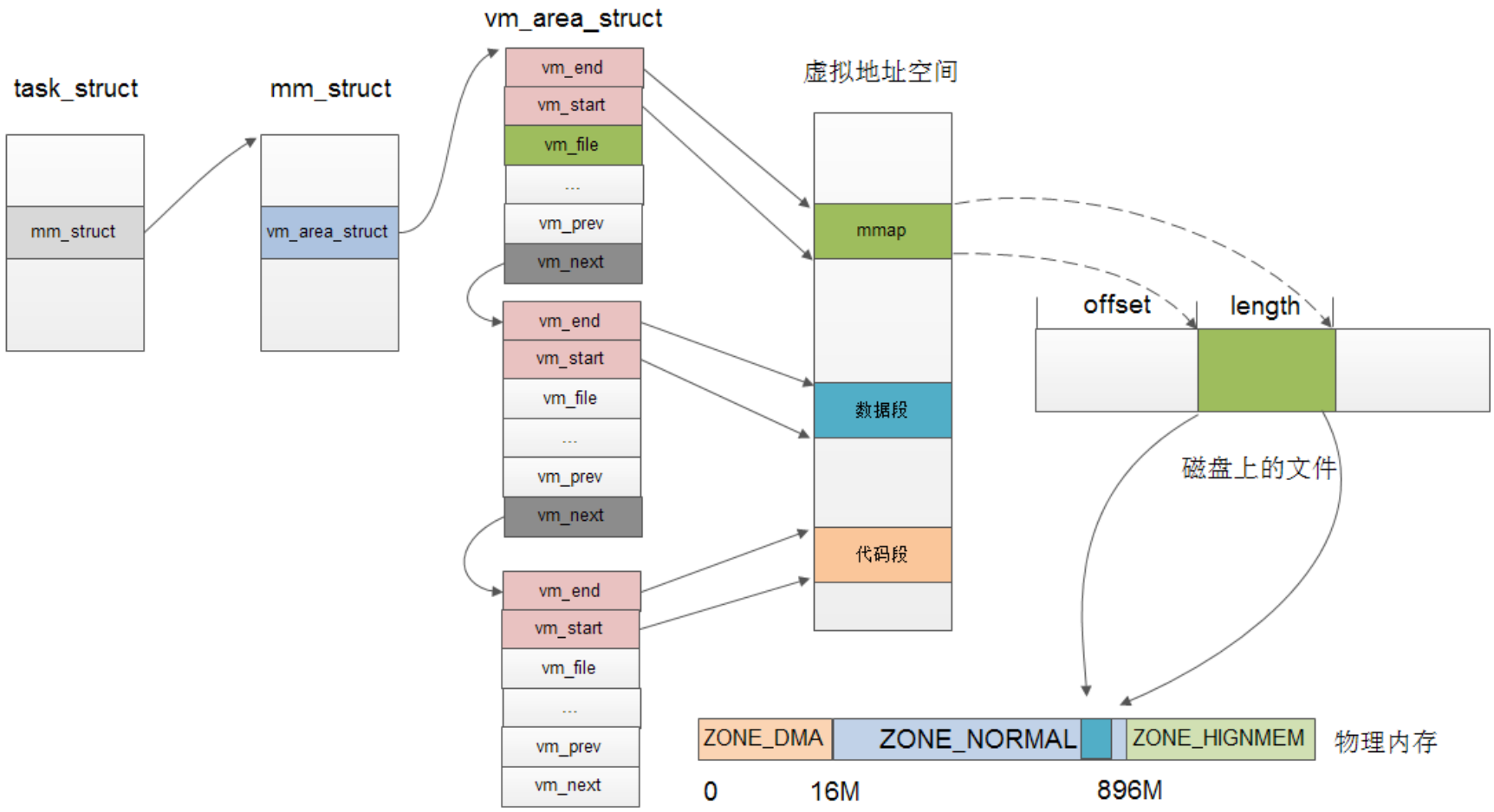


# Linux进程虚拟地址到文件的映射

- 函数：mmap
  - 磁盘文件的逻辑地址与Linux进程虚拟地址建立关联



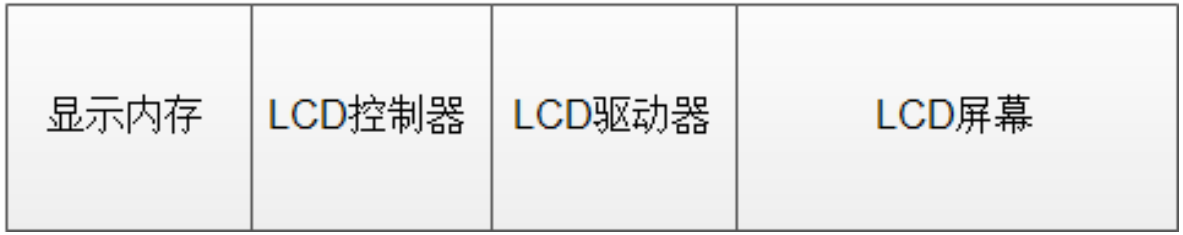
# 内存物理地址与Linux进程虚拟地址建立关联



# 将设备映射到内存

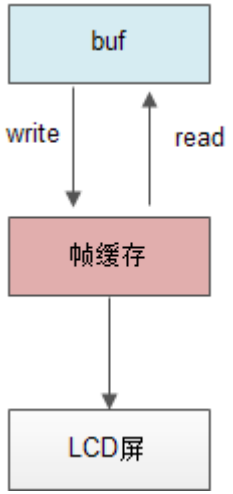
# LCD显示原理

- 显示内存
  - X86下的显存：独立显卡、集成显卡
  - 嵌入式下显存：1602、SOC中的显存

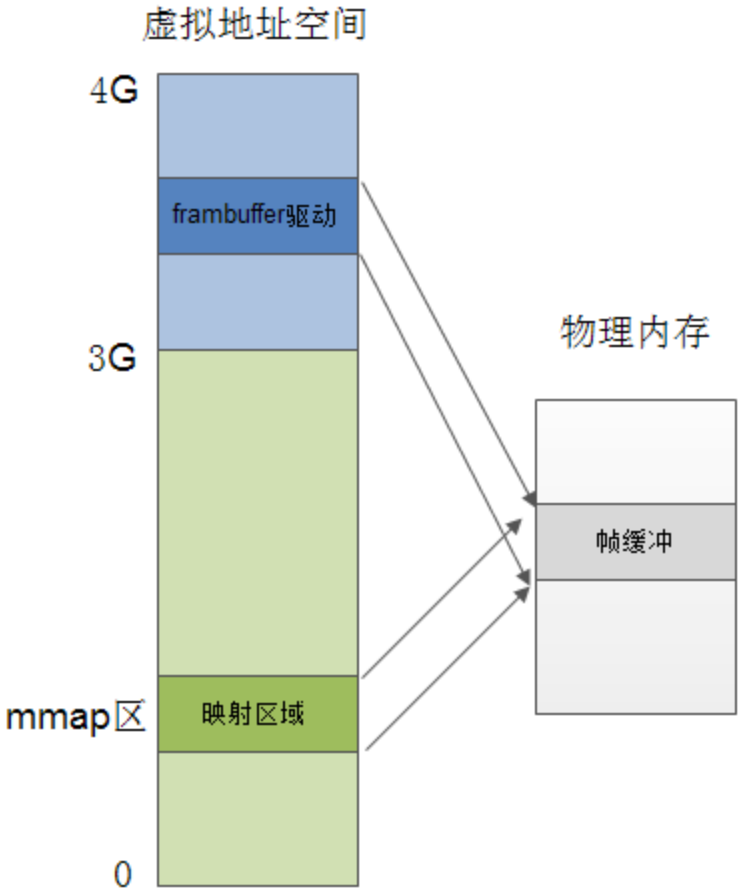


# Frambuffer设备

- 对不同LCD硬件设备的抽象
  - 对不同的显存进行抽象
  - 屏蔽底层各种硬件差异、操作细节
  - 采用统一接口进行操作：显示位置、换页机制



# 将设备映射到内存



# 共享文件映射