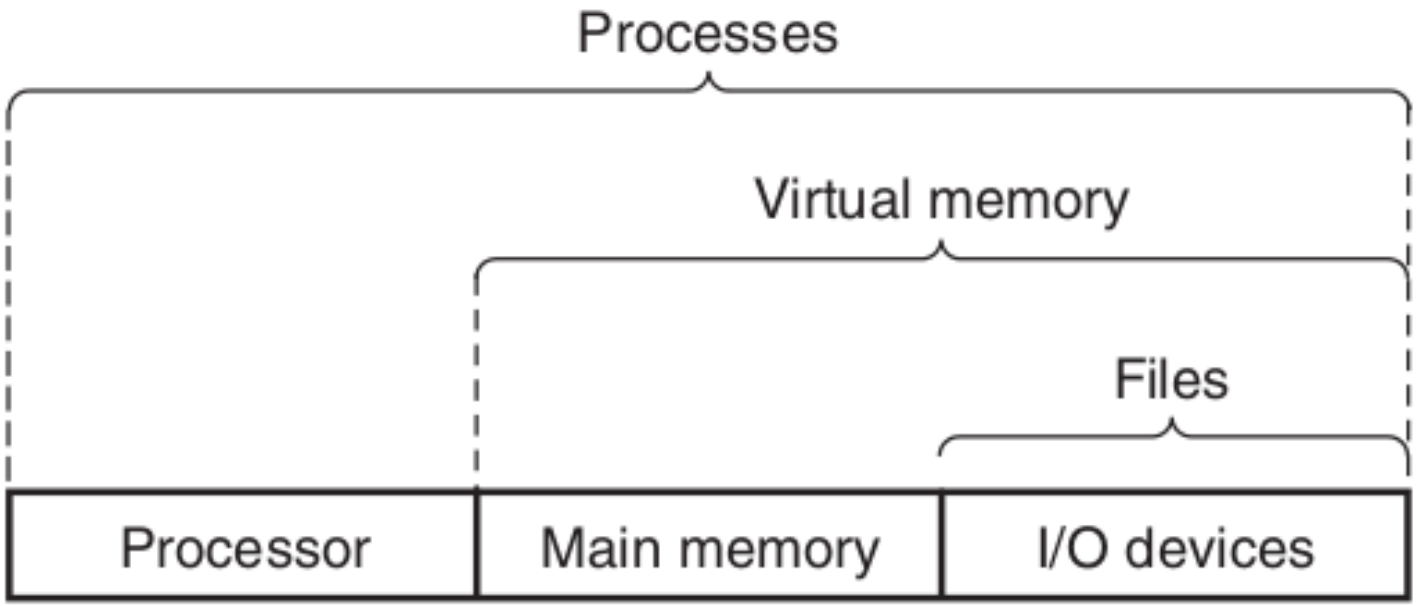
操作系统期末复习提纲

June 12, 2012

操作系统中三个最重要的概念

1. 把处理器CPU抽象为进程
2. 把物理内存抽象为虚拟内存和虚拟地址空间
3. 把永久存储介质（如硬盘）抽象为文件

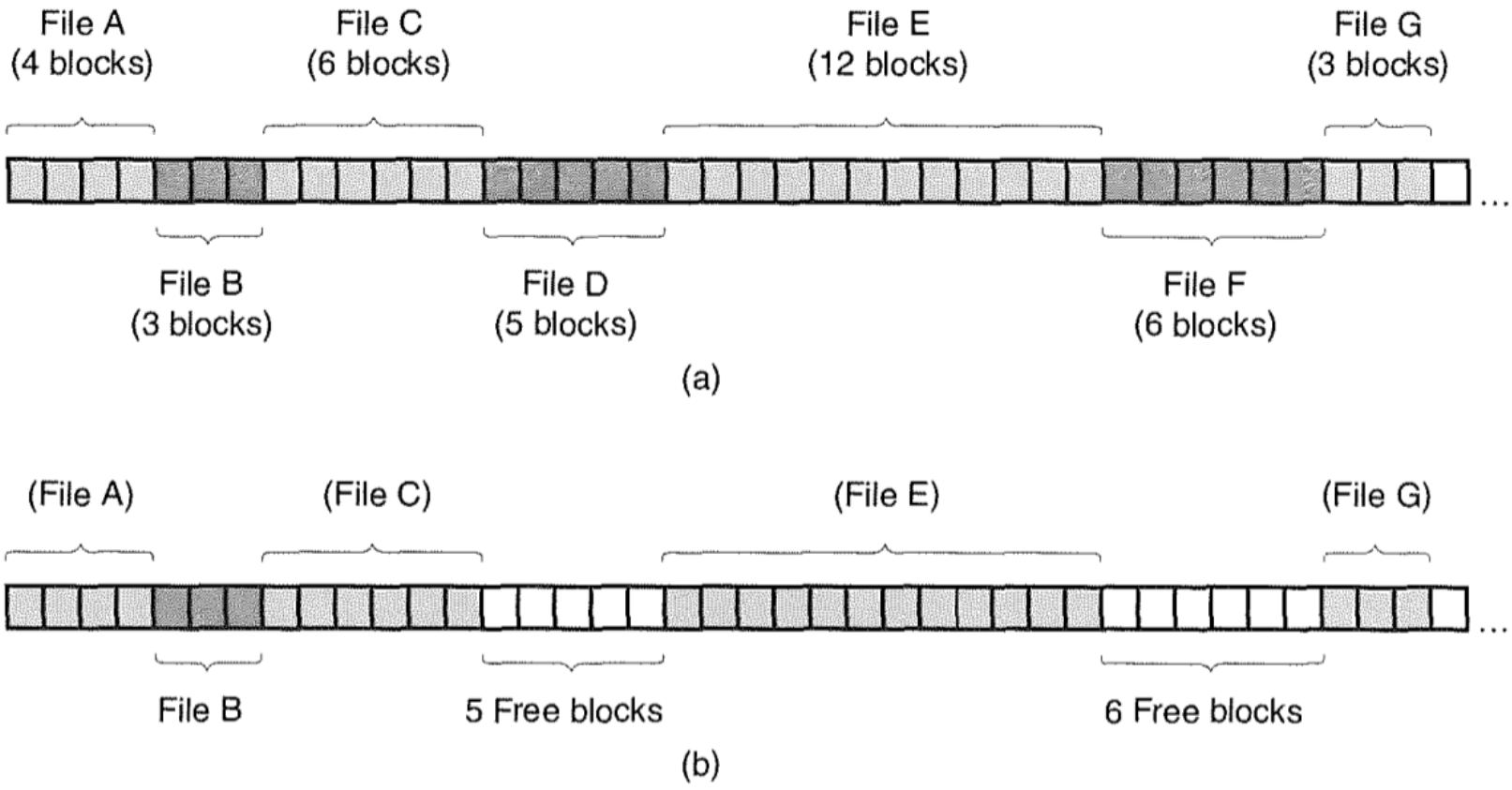
操作系统中三个最重要的概念



文件的属性

- protection
- password
- creator
- owner
- read-only flag
- lock flag
- creation time
- time of last access
- current size

文件的实现: 连续存储方式



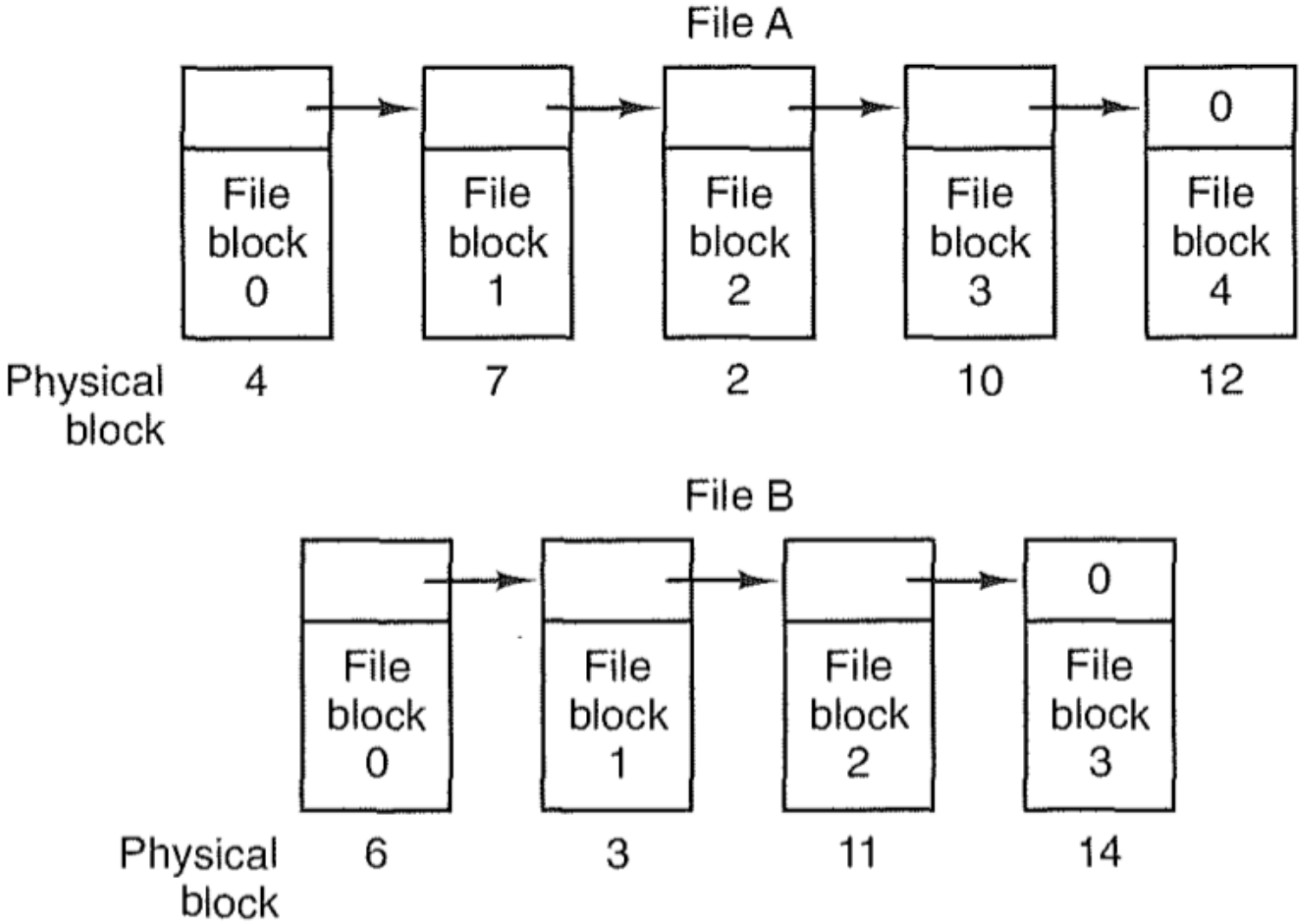
文件的实现: 连续存储方式

连续存储方式有两个重要优点:

- 实现容易：只需记录每个文件在磁盘上起始块地址，以及所占的总块数
- 读写速度非常块：每次读写仅需1次机械移动寻址过程

缺点? — 容易导致文件系统大量碎片(前图)
应用? — CD-ROM上的文件系统, why?

文件的实现: 线性链表存储方式



注: 每个磁盘块的前几个字节用于记录存储该文件的下一个磁盘块地址

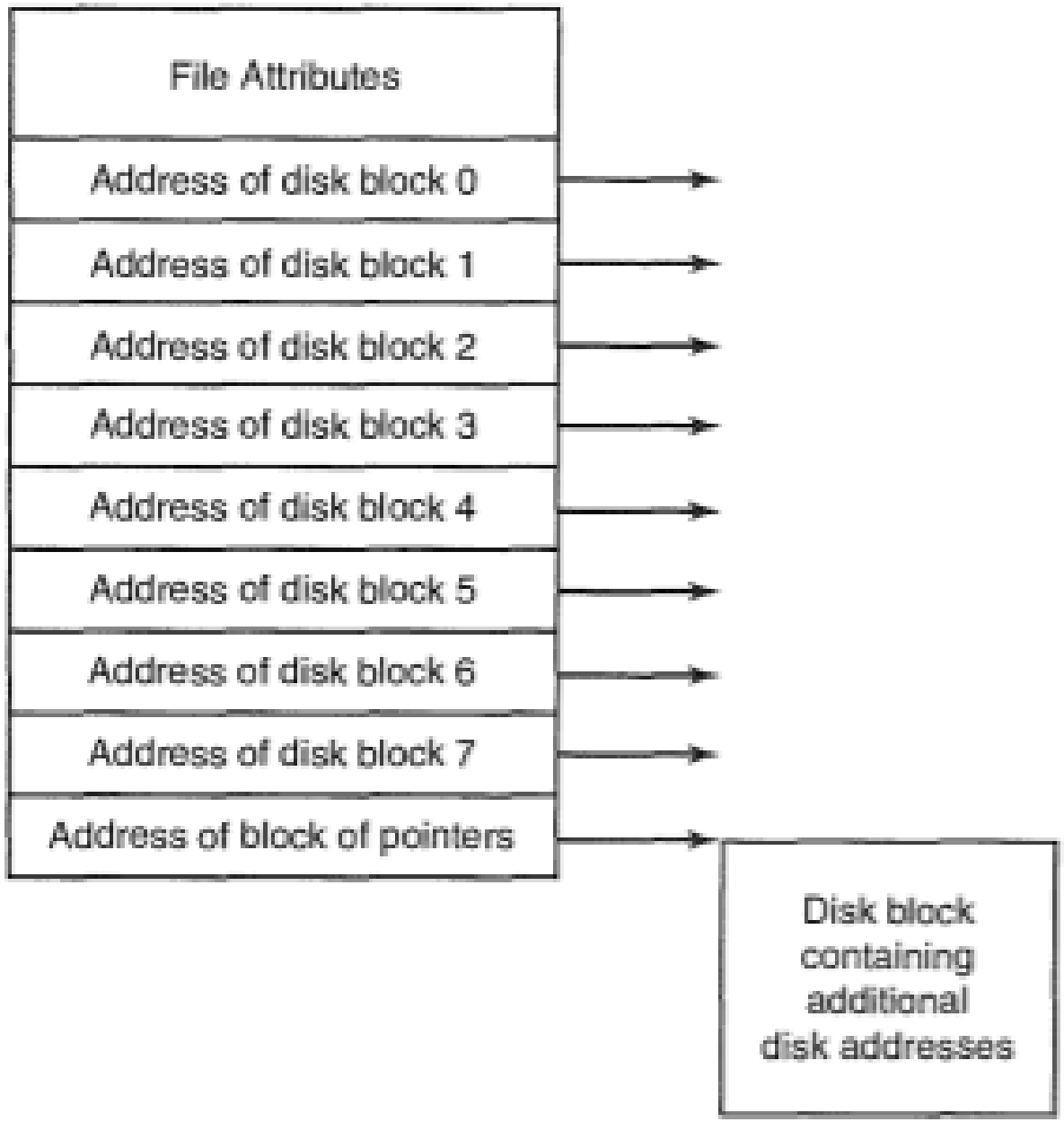
文件的实现: 线性链表存储方式

线性链表存储方式的优点:

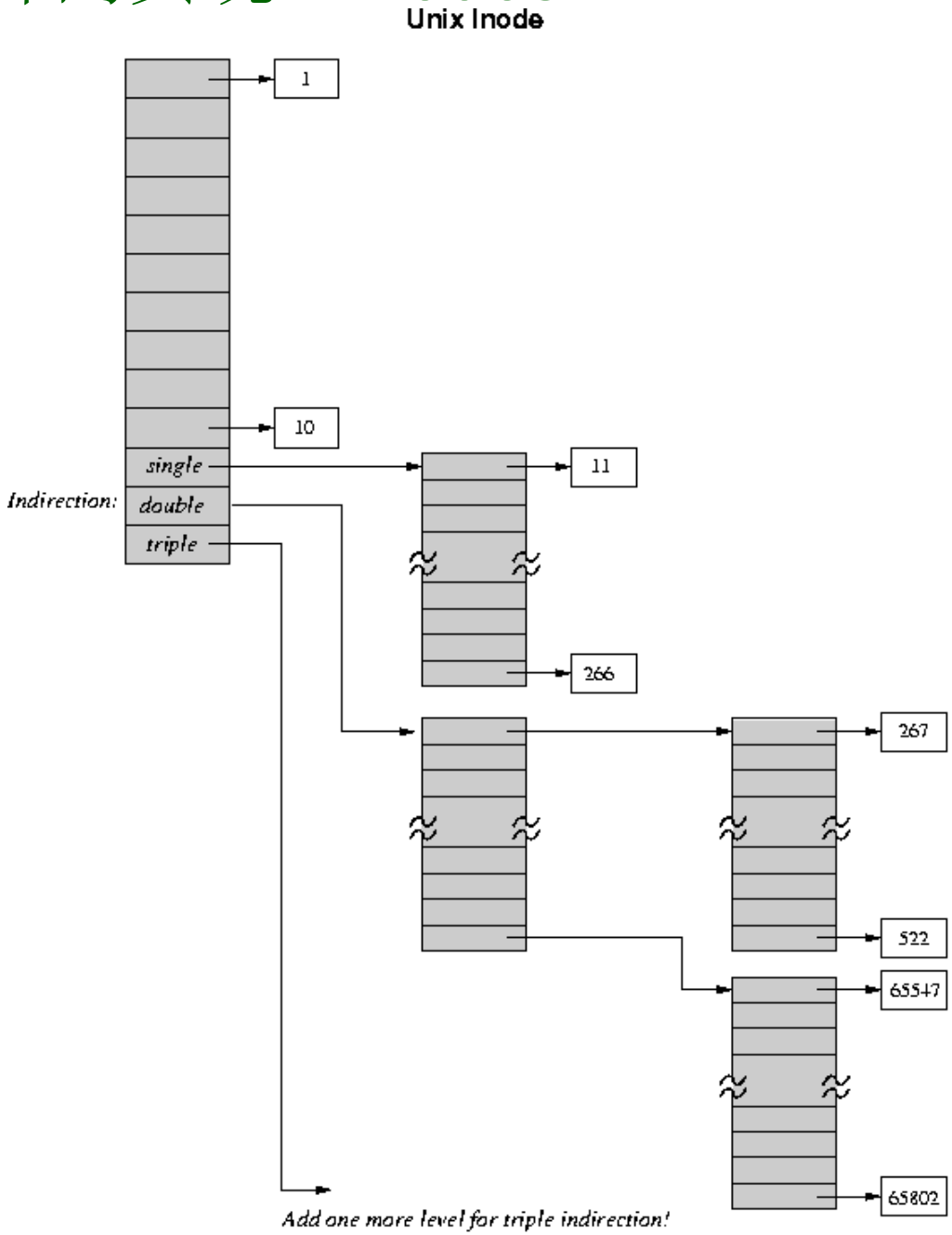
- 只需记录每个文件在磁盘上的起始地址
- 避免碎片问题

缺点：不利于随机访问, why?

文件的实现: I-nodes



文件的实现: I-nodes



文件系统的布局

考虑磁盘上的文件系统:

- 磁盘可以进行分区，用磁盘分区表记录每个分区的边界
- 第0扇区：MBR(Master Boot Record, 主引导记录), 存放引导程序及磁盘分区表
- MBR中的引导程序选择某个分区中的操作系统进行引导
- 每个分区中有一个磁盘块用于启动该分区上的操作系统(boot block)
- 超级块(superblock): 存放该分区上的文件系统参数(类型、块数等信息)

文件系统的布局

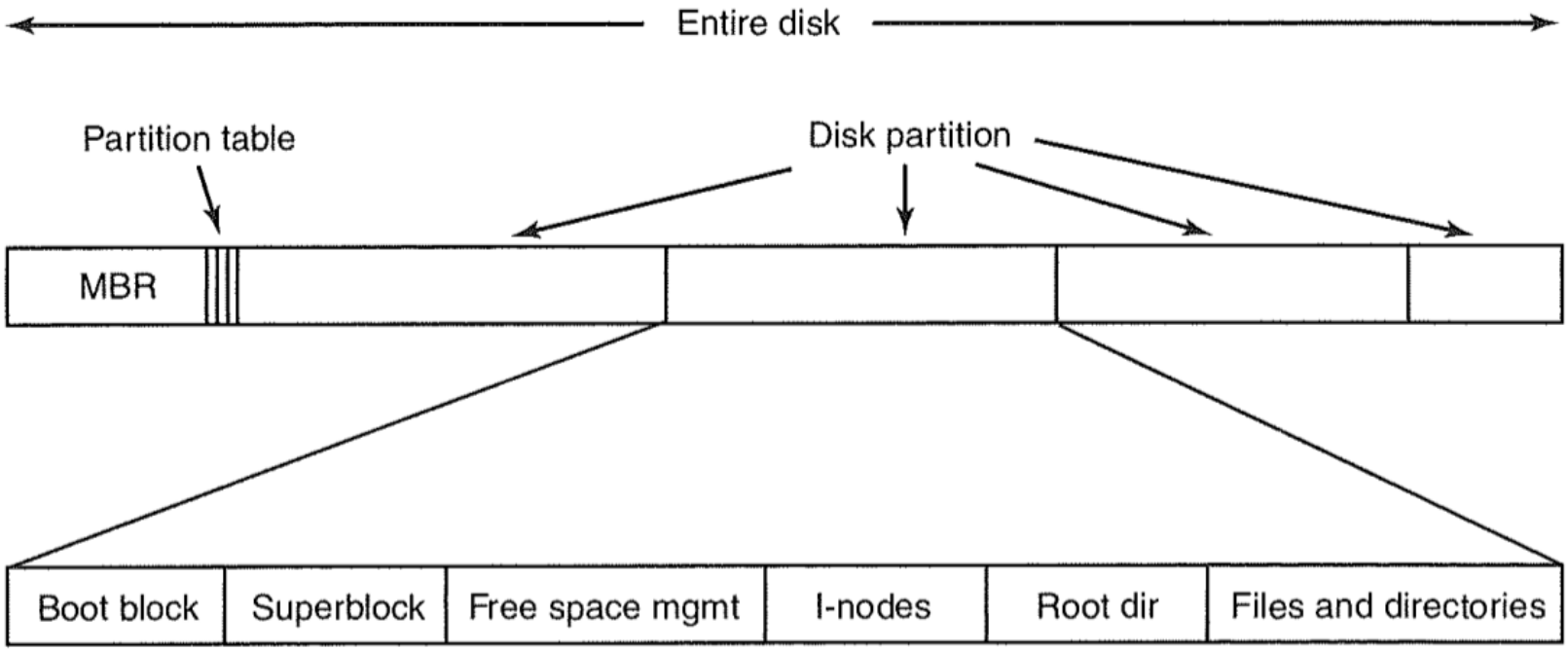


Figure 4-9. A possible file system layout.

Linux内核与普通应用程序的区别

- 内核不能使用C语言标注库函数（及C标准头文件）
- Linux内核编程使用的是GNU C而不是标准C语言
- 内核代码缺乏内存保护机制（应用程序则有）
- 与应用程序比，内核运行时的栈非常小
- 内核代码必须正确处理同步与并发问题
- 可移植性对内核而言非常重要

为什么内核不能使用C语言库函数及头文件?

原因

- chicken-and-egg问题
- C标准库对内核而言太庞大、低效

Linux内核的应对策略

- 自己实现部分库函数功能：`lib/string.c`，对应头文件`<linux/string.h>`
- 标准库中printf函数在内核中对应的是printk
  - 例如`printk(KERN_ERR "this is an error!\n")`

list_head的定义

include/linux/list.h

```
struct list_head {
    struct list_head *next, *prev;
};
```

思考

这种只有指针而没有数据的list_head能有什么用处?

list_head的使用方法举例

include/linux/sched.h

```
struct task_struct {
    ...
    struct list_head run_list;
    ...
};
```

思考

与fox结构相比，使用list_head构造链表有什么好处?

list_head结构的静态初始化

include/linux/list.h

```
#define LIST_HEAD_INIT(name) \
        { &(name), &(name) }

#define LIST_HEAD(name) \
    struct list_head name =
   LIST_HEAD_INIT(name)
```

LIST_HEAD的使用方法

例: `LIST_HEAD(packet_list)`声明一个名为`packet_list`的表头变量并将其初始化为空表。

针对list结构的操作

- `list_add(new, head)` 把new插入head后面
- `list_add_tail(new, head)` 把new插入head前面
- `list_del(entry)` 把entry从链表中删除
- `list_empty(head)` 判断链表head是否为空
- `list_splice(list, head)` 合并list和head
- `list_for_each(pos, head)` 遍历以head为头的链表

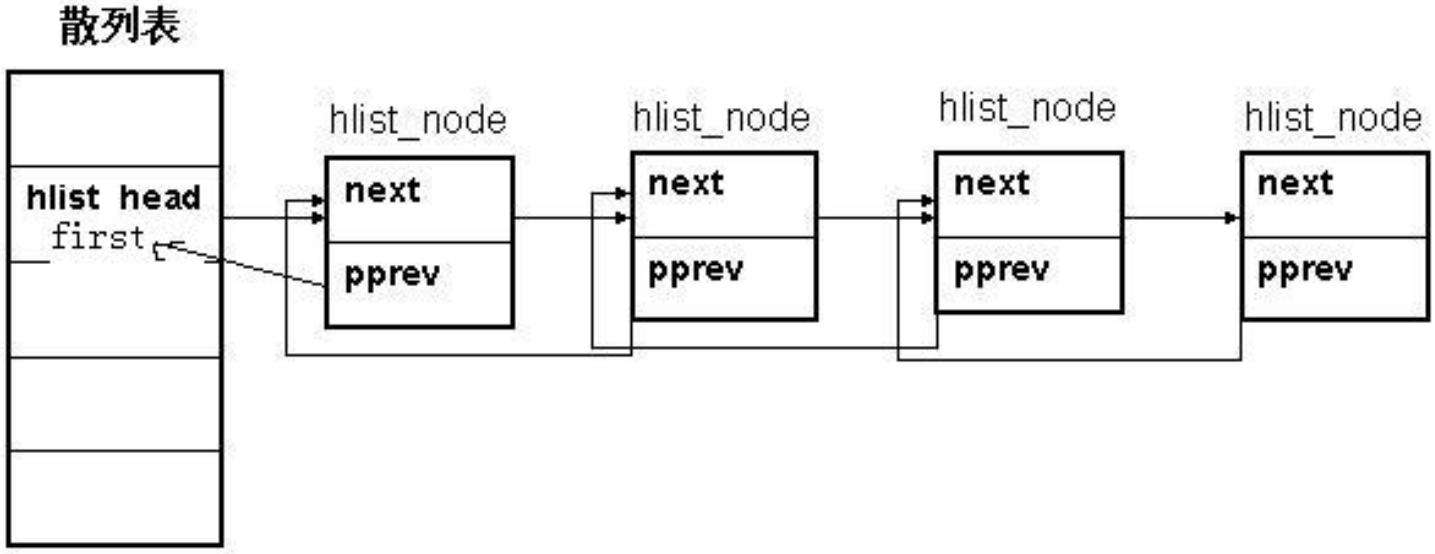
针对list结构的操作举例: list_for_each

```
struct list_head *p;

list_for_each(p, &list) {
    if (!condition) continue;
    return list_entry(p, struct task_struct,
                      run_list);
}

return NULL;
```

实现哈希表: hlist_head与hlist_node



include/linux/list.h

```
struct hlist_head {
    struct hlist_node *first;
};

struct hlist_node {
    struct hlist_node *next, **pprev;
};
```

针对hlist_head与hlist_node的操作

- `hlist_add_head`
- `hlist_add_before`
- `hlist_add_after`
- `hlist_del(entry)`
- `hlist_empty(head)`

Linux内核关键数据结构：hlist_head与hlist_node

思考及课外练习

1. hlist_node中的pprev字段指向什么内容?
2. 为什么hlist_head中只有一个成员first?
3. hlist_head可否直接用hlist_node代替?

Linux区别进程的方式: pid字段

- 在内核中，对进程的各种操作均通过task_struct实现
- 用户角度，每个进程有唯一编号(PID). 进程的PID保存于task_struct的pid字段中。
  - 例如：kill()系统调用的参数为PID
  - 内核需要快速从PID找到对应的task_struct进行操作(后面讲具体方法)

PID可取的最大值: include/linux/threads.h

```
/*
 * This controls the default maximum pid
 *   allocated to a process
 */
#define PID_MAX_DEFAULT 0x8000
```

Linux区别进程的方式: pid字段

思考

1. 为什么要限制PID可取的最大值?

2. 如果系统中进程个数超过这个最大值怎么办?

内核遍历所有进程的方式: tasks字段

task_struct中的tasks字段：将系统中所有进程连接起来

```
struct task_struct {
    ...
    struct list_head tasks;
    ...
};
```

系统中所有进程组成的链表

以init_task为表头，进程描述符的tasks字段将所有进程连接起来

遍历系统中所有进程的宏: for_each_process

阅读并理解include/linux/sched.h中for_each_process的实现。

与进程调度相关的字段

```
struct task_struct {
    ...
    int prio, static_prio;
    struct list_head run_list;
    prio_array_t *array;
    ...
};
```

各字段的含义

- prio 为进程当前优先级(0 – 139)
- run_list 用于连接所有相同优先级的进程
- array?

与进程调度相关的字段

kernel/sched.c

```
struct prio_array {
    unsigned int nr_active;
    unsigned long bitmap[BITMAP_SIZE];
    struct list_head queue[MAX_PRIO];
};
```

include/linux/sched.h

```
typedef struct prio_array prio_array_t;
```

各成员字段的含义

- nr_active: 当前列表中进程总数
- bitmap 用于记录哪个队列为非空(后面分析调度算法时用到)
- queue用于存储140个队列的表头

课外练习

1. 上页中，BITMAP_SIZE和MAX_PRIO两个宏分别在文件kernel/sched.c和include/linux/sched.h中定义，请确定这两个宏的具体数值。

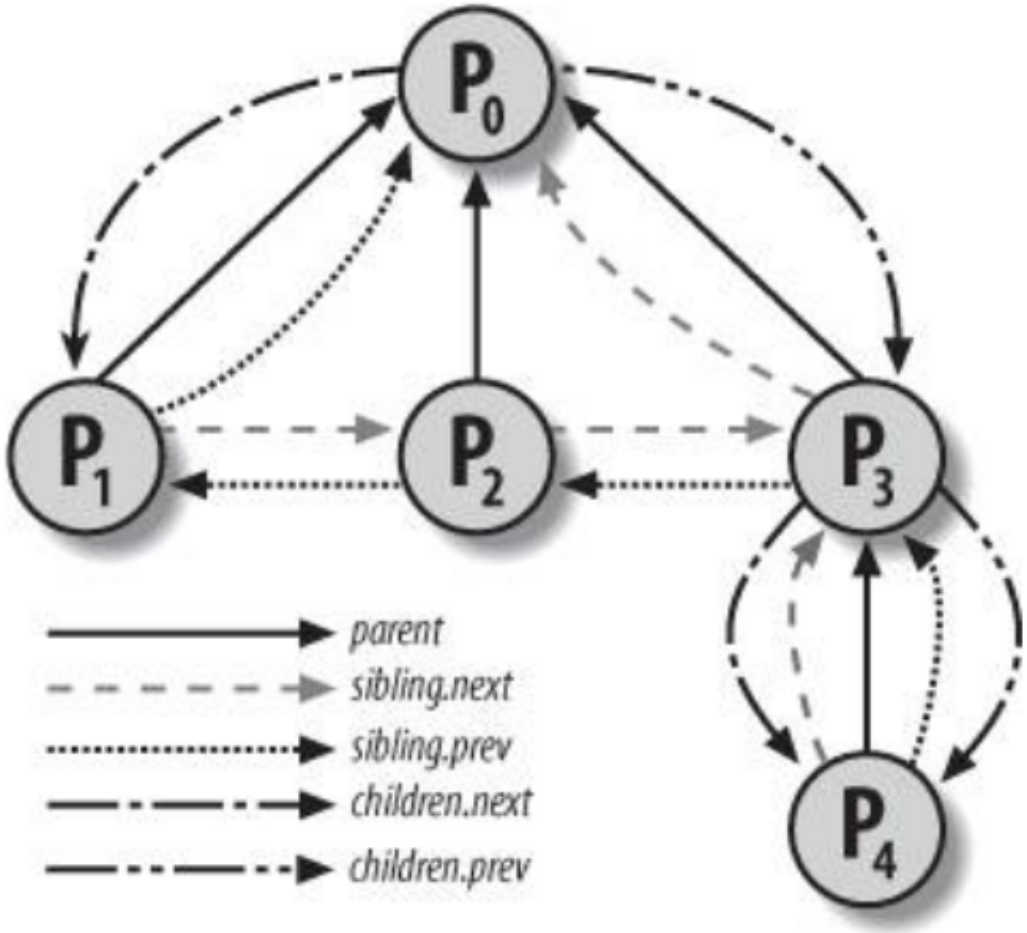
2. 在文件kernel/sched.c找出dequeue_task以及enqueue_task的定义并理解它。

记录进程之间层次关系的字段

进程管理中需要记录进程之间的层次关系(父母、兄弟姐妹).

```
struct task_struct {
    ...
    struct task_struct *real_parent;
    struct task_struct *parent;
    struct list_head children;
    struct list_head sibling;
    ...
};
```

记录进程之间层次关系的字段



思考

给定指向某进程描述符的指针p, 如何打印该进程所有子进程的ID? (实现这个函数，可使用`list_for_each`宏)

表示进程其他关系的字段

进程标识：

- 从内核角度，一律通过task_struct操作进程
- 从用户角度可能需要通过PID操作进程，如kill(pid)

所以需要做到从pid到task_struct的快速转换：

- task_struct到pid: `p->pid`
- pid到task_struct的快速转换：
  - 遍历所有进程，逐一检查其pid字段的值?
  - 通过散列表结构做到快速转换
- (重要)给定进程组、会话组或者线程组的leader ID, 如何快速找出该组中所有进程(线程)?

表示进程其他关系的字段

共有四种类型的PID：进程本身PID、线程组leader的PID、进程组leader的PID以及会话组leader的PID:

include/linux/pid.h

```
enum pid_type
{
    PIDTYPE_PID,
    PIDTYPE_TGID,
    PIDTYPE_PGID,
    PIDTYPE_SID,
    PIDTYPE_MAX
};
```

表示进程其他关系的字段

依据PID类型不同，共定义四个散列表：kernel/pid.c

```
static struct hlist_head
    *pid_hash[PIDTYPE_MAX];
```

思考

为什么定义这个数组时使用指向hlist_head的指针?

表示进程其他关系的字段

散列表中元素struct pid: include/linux/pid.h

```
struct pid
{
    int nr;
    struct hlist_node pid_chain;
    struct list_head pid_list;
};
```

task_struct中相应字段

```
struct task_struct
{
    ...
    /* PID/PID hash table linkage. */
    struct pid pids[PIDTYPE_MAX];
    ...
};
```

表示进程其他关系的字段

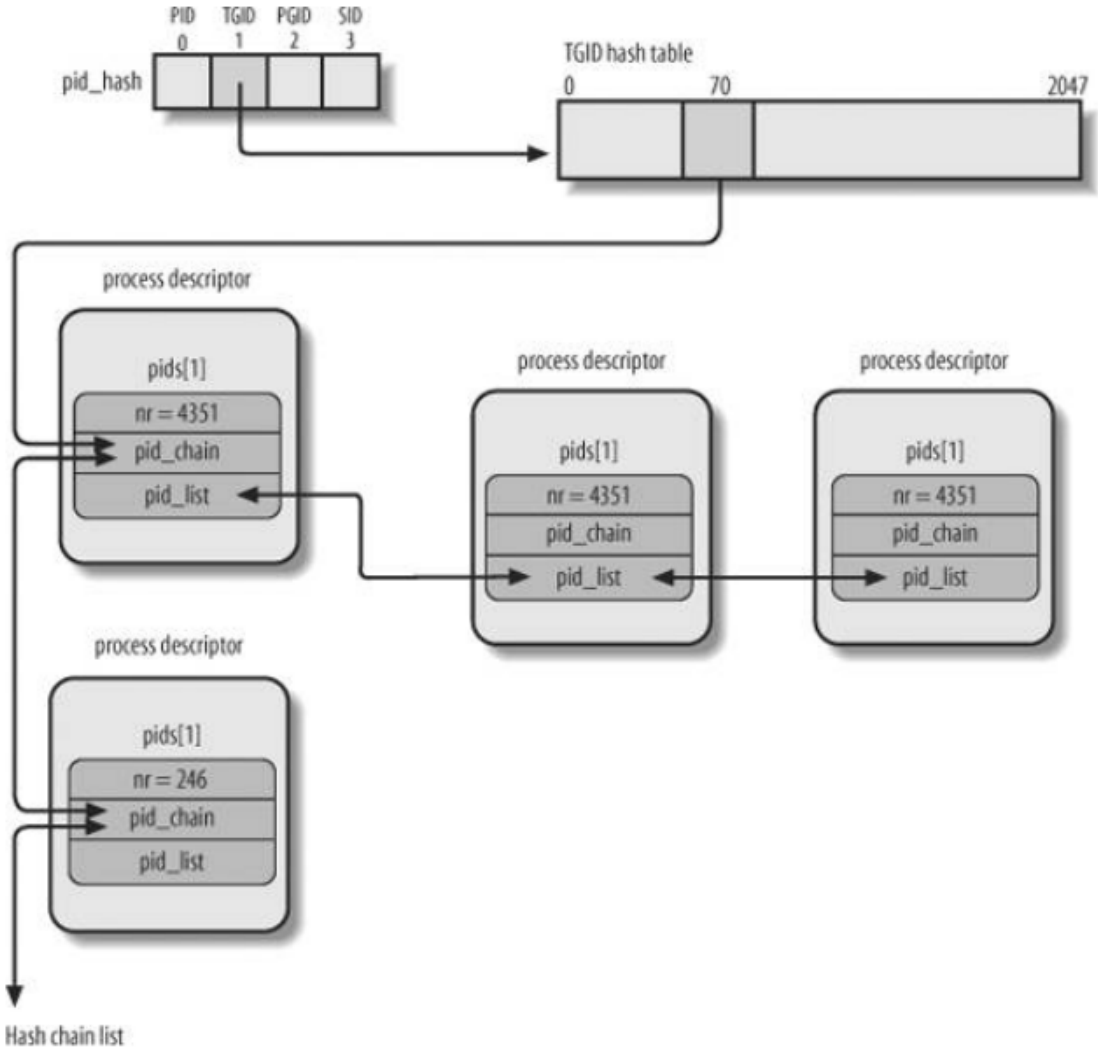
哈希函数的定义

```
unsigned long hash_long(unsigned long val,
    unsigned int bits)
{
     unsigned long hash = val * 0x9e370001UL;
     return hash >> (32 - bits);
}
```

思考

- 这里的参数bits起到什么作用?
- 对于含有2048个hlist_head元素的散列表而言，参数bits 应该是多少?

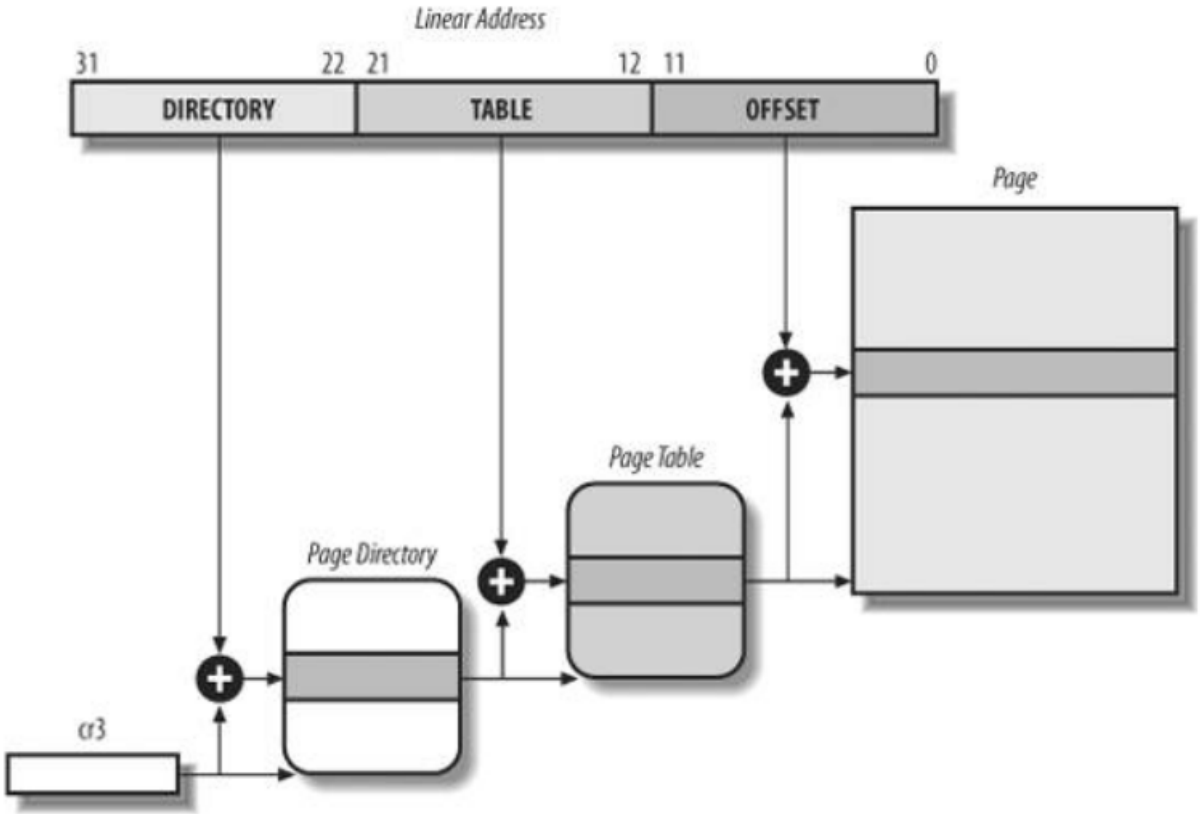
表示进程其他关系的字段: pid_hash结构图



考试要求

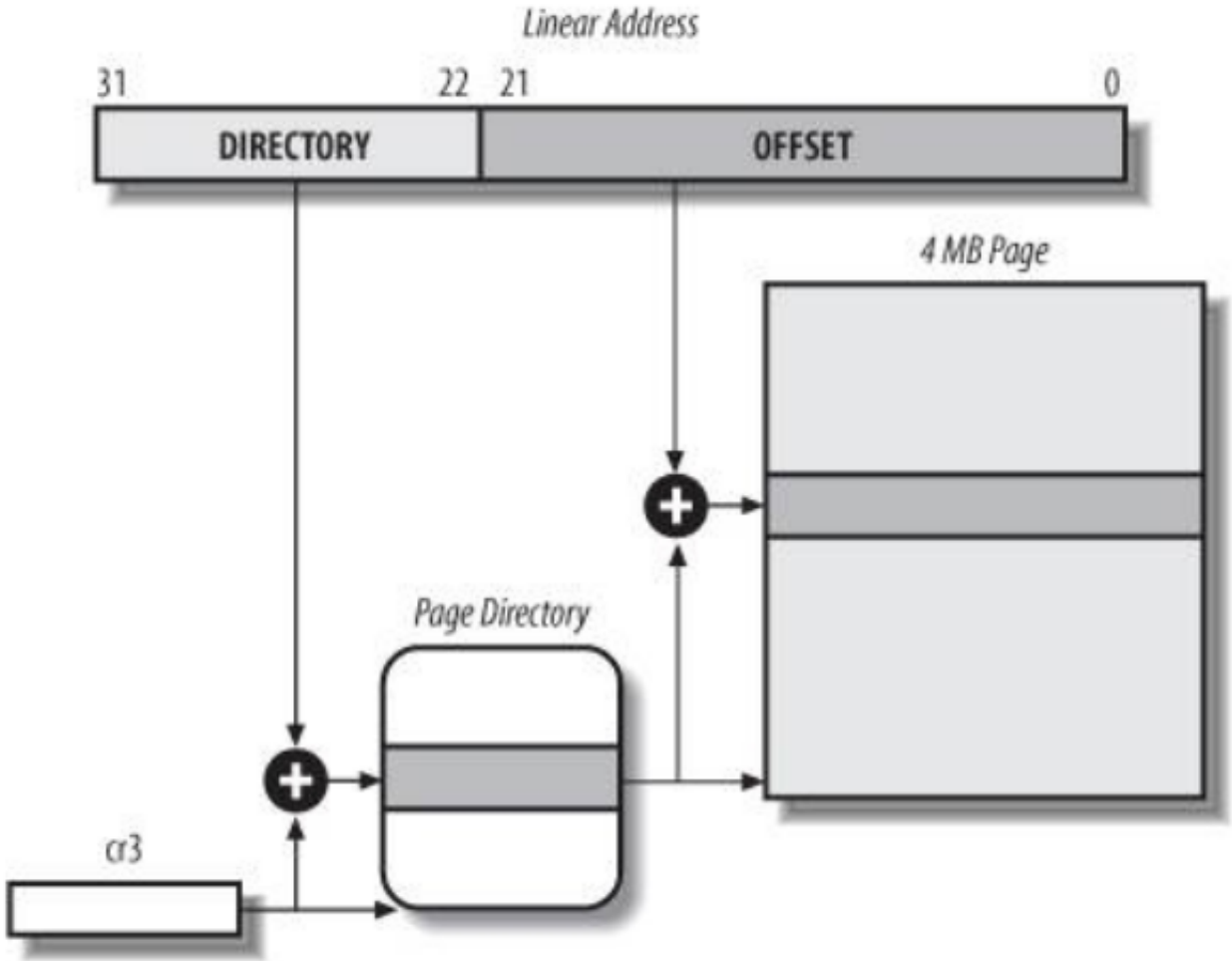
能够根据这个数据结构图，描述如何从pid确定进程描述符的地址，以及如何根据pid及其类型，确定小组内全体成员。

Intel平台上的分页技术



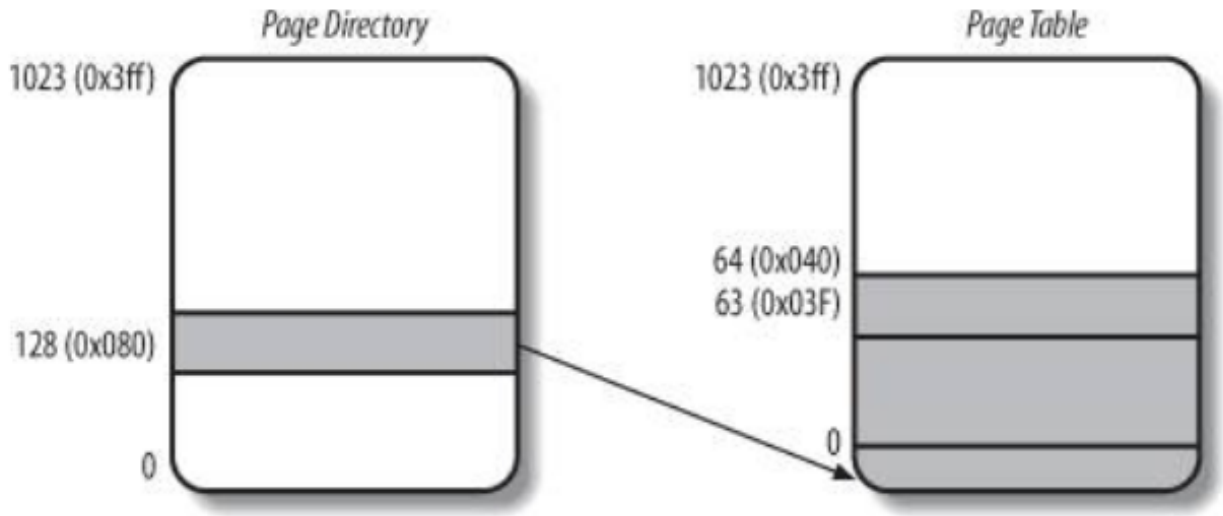
cr3寄存器用于存放当前进程的page directory(第一级页表).

Intel平台上的扩展分页模式



扩展分页模式下，页面大小为4M，用于处理大块连续地址空间，以节省页表所占空间及TLB空间。

分页的实际例子



假设内核把从0x20000000到0x2003ffff之间的线性地址(共64页)分配给某进程。

- 该段地址最高10位均为0010000000, 即0x080(十进制128)，因此第一级页表只有一个页表项有效。
- 该段地址中间10位范围为0 – 0x03f(即0 – 63). 所以第二级页表中开头64个页表项有效。(图示灰色区域)
- 给定线性地址0x20021406，如何确定其对应的物理地址?