OS概念与Linux内核代码分析之一：进程管理

李中国

苏州大学计算机科学与技术学院

May 29, 2012

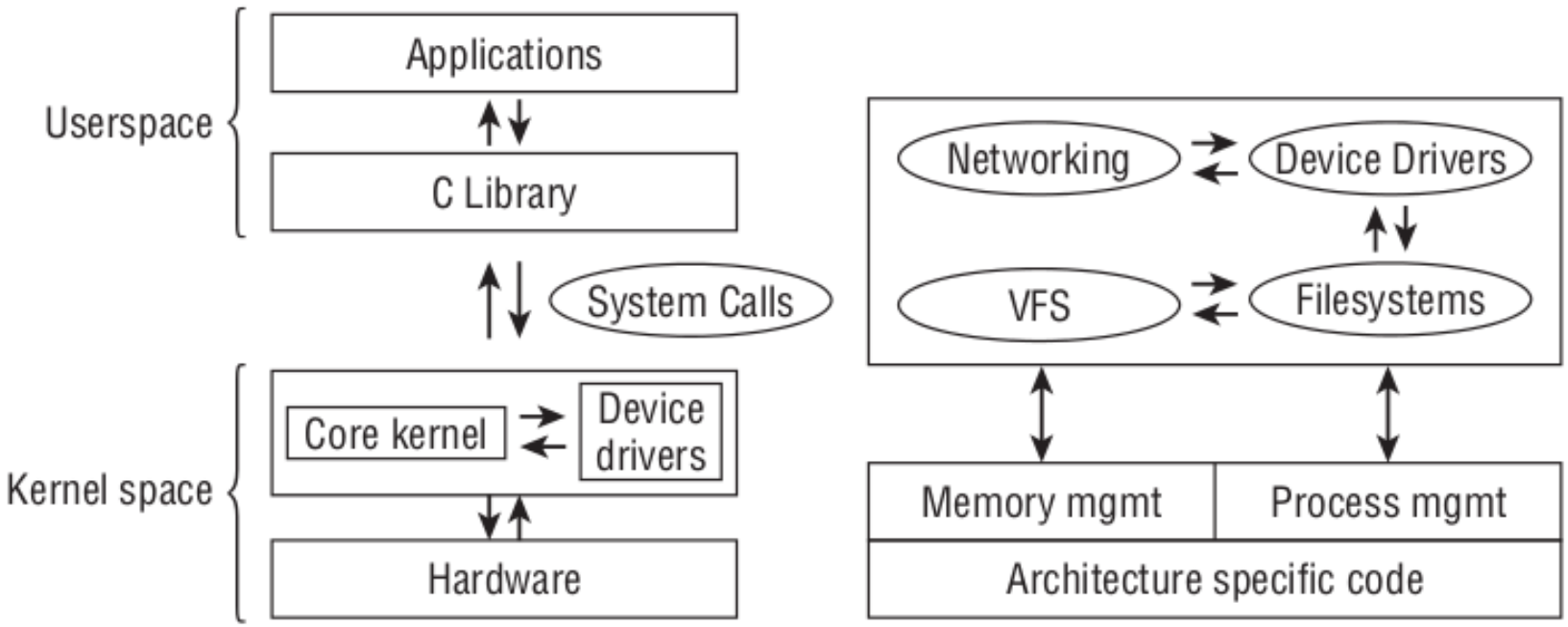
本课程Linux内核版本

- linux kernel: 2.6.11.12
- 为什么使用这个版本的内核?
- 下载地址:
  `http://www.kernel.org/pub/linux/kernel/v2.6`

为什么研究Linux内核代码

- 了解OS的概念如何应用于实际操作系统
- 学习顶尖高手写程序的思路与方法
- 学习数据结构与算法如何用于解决实际问题
- 学习大规模工程的设计与实施
  - 2.6.x内核开发成本: $1.14 billion USD (欧盟)
- (possibly)成为一名内核开发人员

Linux内核整体架构



Outline

Outline

Outline

Linux内核关键数据结构：双向循环链表(list)

```
struct fox {
    unsigned long tail_length;
    unsigned long weight;
    bool      is_fantastic;
    struct fox *next;
    struct fox *prev;
};
```

思考:
这样定义链表有什么主要缺点?

Linux内核关键数据结构

list_head的定义: include/linux/list.h

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

思考

与上页中的fox结构相比，这样定义的list_head能有什么用处?

Linux内核关键数据结构

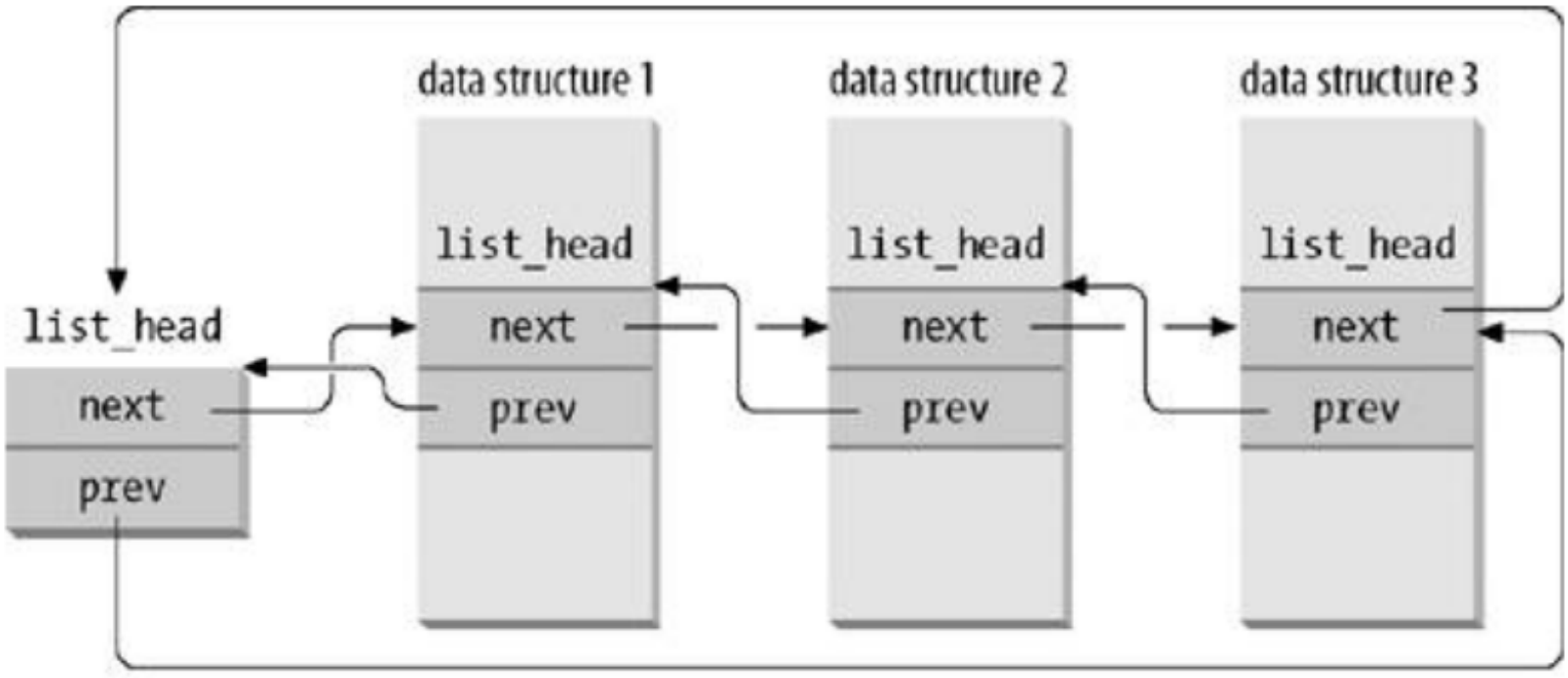
list_head的使用方法举例: include/linux/sched.h

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

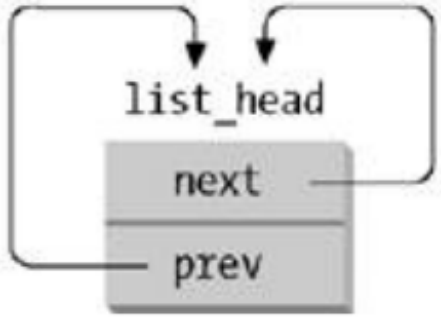
思考

仍与fox结构相比，使用list_head构造链表有什么好处?

Linux内核关键数据结构：list



(a) a doubly linked listed with three elements



(b) an empty doubly linked list

Linux内核关键数据结构：list

list_head的定义: include/linux/list.h

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

关键问题

如何根据list_head的地址确定包含它的数据结构的地址?

Linux内核关键数据结构：list

宏list_entry(ptr, type, member)

计算包含ptr(指向list_head结构的指针)的结构体的地址

例子: 结合上述task_struct结构

list_entry(ptr, struct task_struct, run_list)

思考

宏list_entry的实现原理是什么?

Linux内核关键数据结构：list

list_entry的定义: include/linux/list.h

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

container_of的定义: include/linux/kernel.h

练习：请在该头文件中找到container_of的定义并试着理解它。

Linux内核关键数据结构：list

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)
```

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
- list_for_each_entry(pos, head)

针对list结构的操作: 代码分析举例

```
typedef struct list_head list_head;

static inline void list_add(list_head *new,
                            list_head *head)
{
    __list_add(new, head, head->next);
}

static inline void __list_add(list_head *new,
                      list_head *prev,
                      list_head *next)
{
    next->prev = new;
    new->next  = next;
    new->prev  = prev;
    prev->next = new;
}
```

针对list结构的操作: 代码分析举例

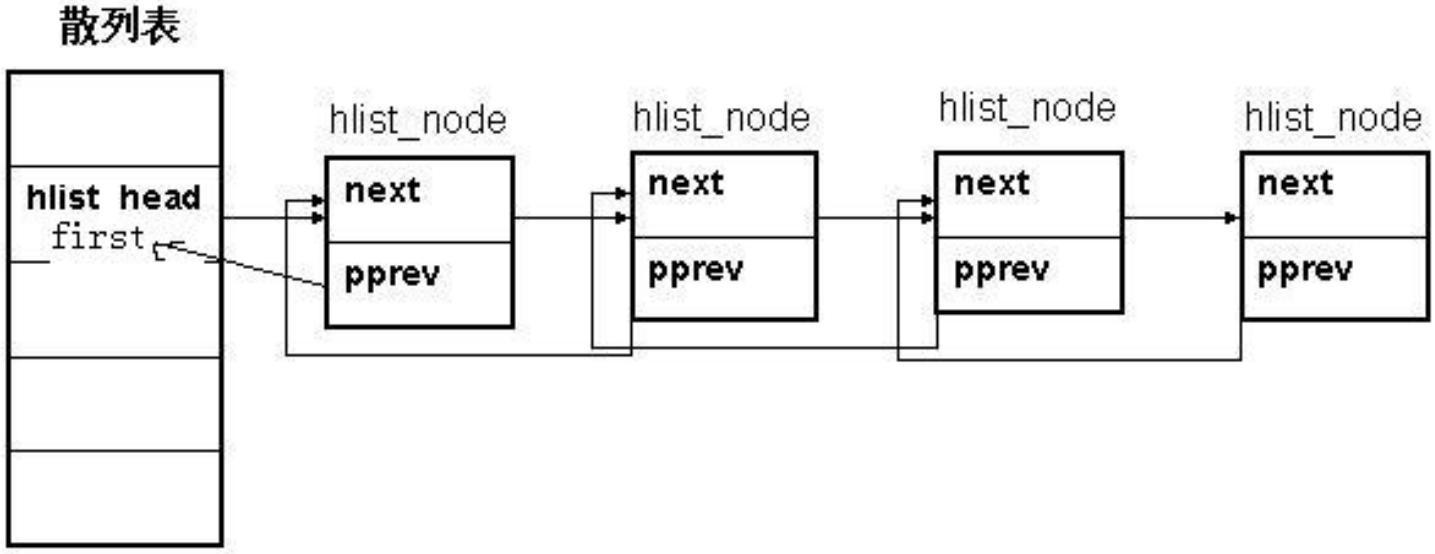
```
struct list_head *p;

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

return NULL;
```

针对list结构的操作: 代码分析举例

课外练习

从源文件include/linux/list.h中找出上述list操作的函数代码，阅读并理解其实现。

Linux内核关键数据结构：hlist_head与hlist_node



用于实现散列表：include/linux/list.h

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

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

Linux内核关键数据结构：hlist_head与hlist_node

思考及课外练习

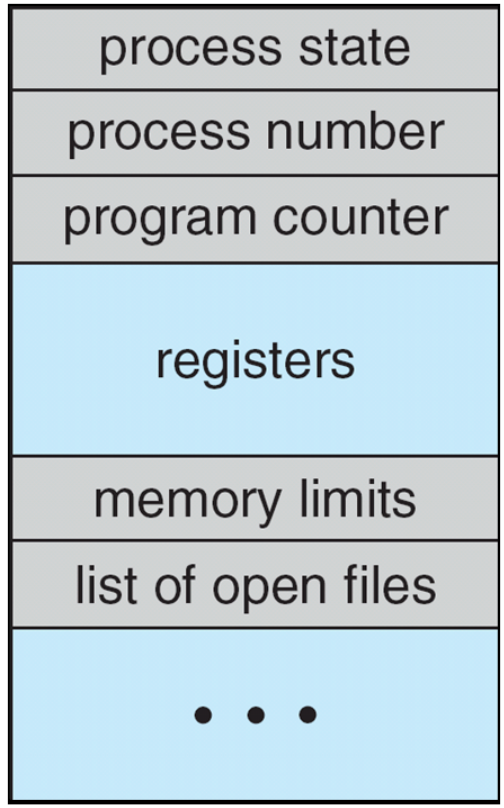
阅读include/linux/list.h中hlist_add_head(), hlist_del(), hlist_empty(), hlist_add_before()等函数的实现，思考：

1. hlist_node中的pprev字段指向什么内容?
2. 为什么pprev采用二重指针?
3. 为什么hlist_head中只有一个成员first?

进程控制块(PCB)

用于存放与每个进程相关的信息

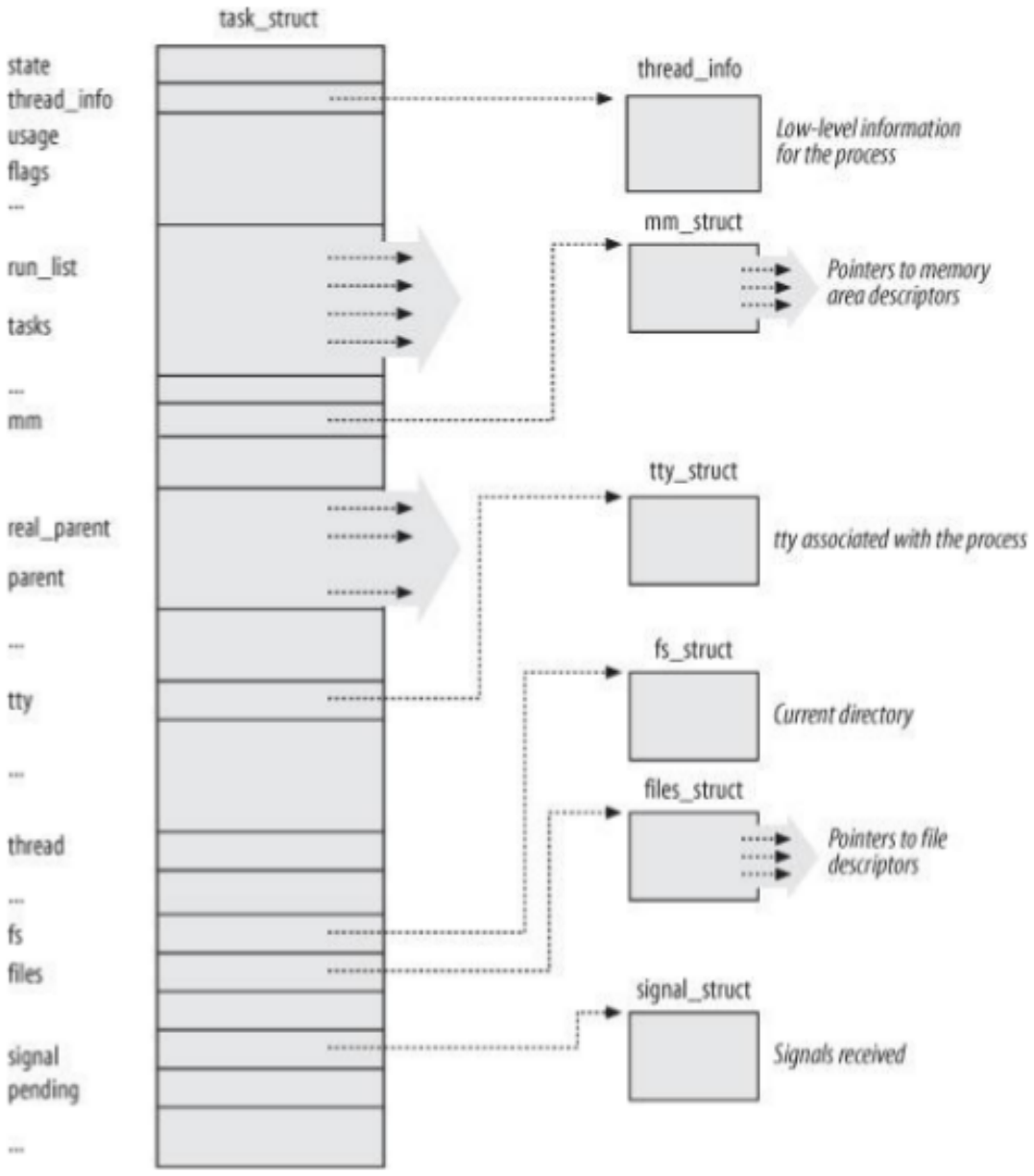
- 进程状态
- 程序计数器PC
- CPU寄存器
- CPU调度信息
- 内存管理信息
- I/O状态信息
- 记账信息



Linux进程描述符: Process Descriptor

- 进程控制块在Linux内核中被成为进程描述符(Process Descriptor)
- 结构名字: struct task_struct;
- 这是一个非常大的数据结构，其中包含:
  - 94行成员字段
  - 其中36行成员本身又是struct或者指向struct的指针
- 后面我们将对task_struct结构进行逐行分析

Linux进程描述符: Process Descriptor



Linux区别不同进程的方式

- 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区别不同进程的方式

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?

Linux区别不同进程的方式

pidmap_array用于管理pid值的回收利用: kernel/pid.c

```
typedef struct pidmap {
    atomic_t nr_free;
    void *page;
} pidmap_t;

static pidmap_t pidmap_array[PIDMAP_ENTRIES] =
     { [ 0 ... PIDMAP_ENTRIES-1 ] =
       { ATOMIC_INIT(BITS_PER_PAGE), NULL } };
```

Linux进程描述符: Process Descriptor

OS有时需要遍历系统中全部进程，所以进程描述符中需要相应字段，用于连接系统中的各个进程。

- 这就用到前面所说的list_head这个关键结构
- 回想一下list_head中包含的两个字段

Linux进程描述符: Process Descriptor

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。

Linux进程描述符: Process Descriptor

OS在进程调度时，需要从全部处于TASK_RUNNING（可运行）状态的进程中选择一个进程，让它占有CPU并运行。

2.6.x之前的Linux内核

将所有处于TASK_RUNNING状态的进程放到一个链表中，每次需要遍历整个链表，并从中选择优先级最高的进程。进程调度时间复杂度为$O(n)$.

Linux 2.6.x内核的$O(1)$调度算法

- 在进程描述符中记录每个进程的优先级
- 以有效的数据结构按照优先级组织所有进程
- 调度算法时间复杂度为$O(1)$, 与进程数n没有直接关系

Linux进程描述符: Process Descriptor

```
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?

Linux进程描述符: Process Descriptor

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

Linux进程描述符: Process Descriptor

```
void enqueue_task(struct task_struct *p,
                  prio_array_t *array)
{
    sched_info_queued(p);
    list_add_tail(&p->run_list,
                  array->queue[p->prio]);
    __set_bit(p->prio, array->bitmap);
    array->nr_active++;
    p->array = array;
}
```

特别注意

list_add_tail的第一个参数.

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的定义并理解它。

Linux进程描述符: Process Descriptor

task_struct结构: include/linux/sched.h

```
struct task_struct {
        volatile long state;
        ...
};
```

进程可能的状态: include/linux/sched.h

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_STOPPED            4
#define TASK_TRACED     8
#define TASK_DEAD       64
```

Linux进程描述符: Process Descriptor

改变进程状态的函数/宏

- set_task_state 改变指定进程的状态
- set_current_state 改变当前执行的进程的状态

课外练习

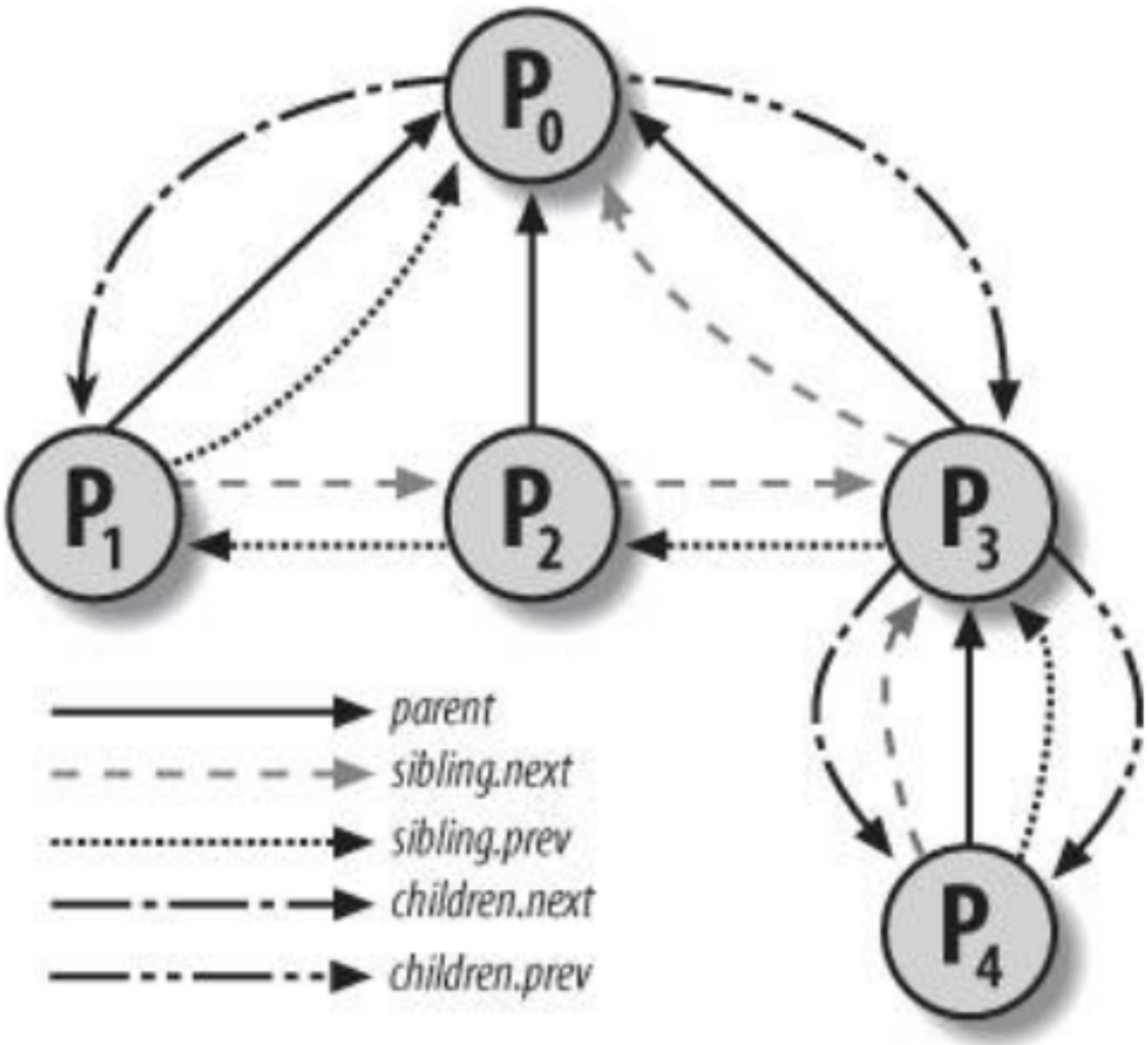
阅读以上两个函数的源代码(位于文件include/linux/sched.h)，并思考为什么要这样写。

Linux进程表示：task_struct结构

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;
    ...
};
```

Linux进程表示：task_struct结构



Linux进程表示：task_struct结构

查看进程结构树的命令：pstree

Linux进程表示：task_struct结构

除了上述关系外，进程之间还存在如下关系:

- 多个进程可以形成一个组(group)，每个组都有唯一的leader.
  - 例如: `cat file | grep 'abc' | wc -l`，其中三个进程形成一个组
- 类似，进程间可以形成一个会话(session)，每个会话中指定唯一的leader
- 同一进程创建的多个线程组成线程组(thread group)，每组也有唯一leader
  - 线程组对应多线程程序里的所有线程

Linux进程表示：task_struct结构

task_struct中的group_leader和tgid分别表示进程组的leader、线程组的leader:

```
struct task_struct {
    ...
    struct task_struct *group_leader;
    pid_t tgid;
    pid_t pid;
    struct signal_struct *signal;
    ...
};
```

Linux进程表示：task_struct结构

pgrp和session字段分别表示进程组leader的PID、会话组leader的PID：

include/linux/sched.h

```
struct signal_struct {
    ...
    /* job control IDs */
    pid_t pgrp;
    pid_t session;
    ...
};
```

Linux进程表示：task_struct结构

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
- 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)?

Linux进程表示：task_struct结构

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
};
```

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];
```

Linux进程表示：task_struct结构

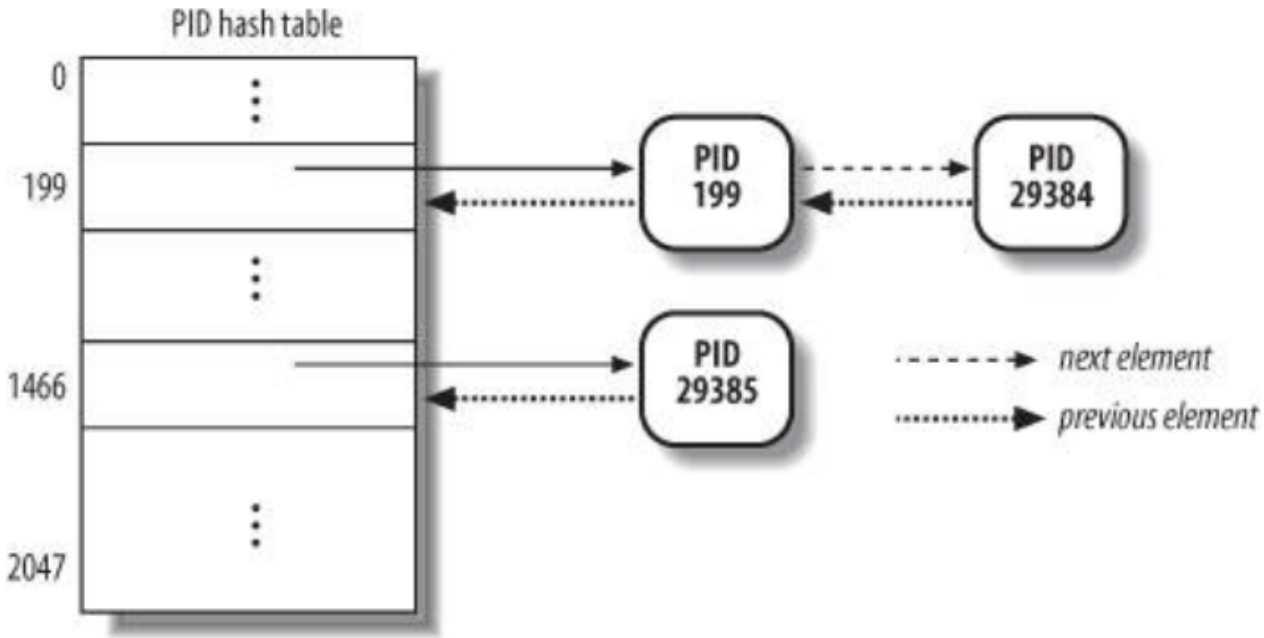
散列表中元素struct pid: include/linux/pid.h

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

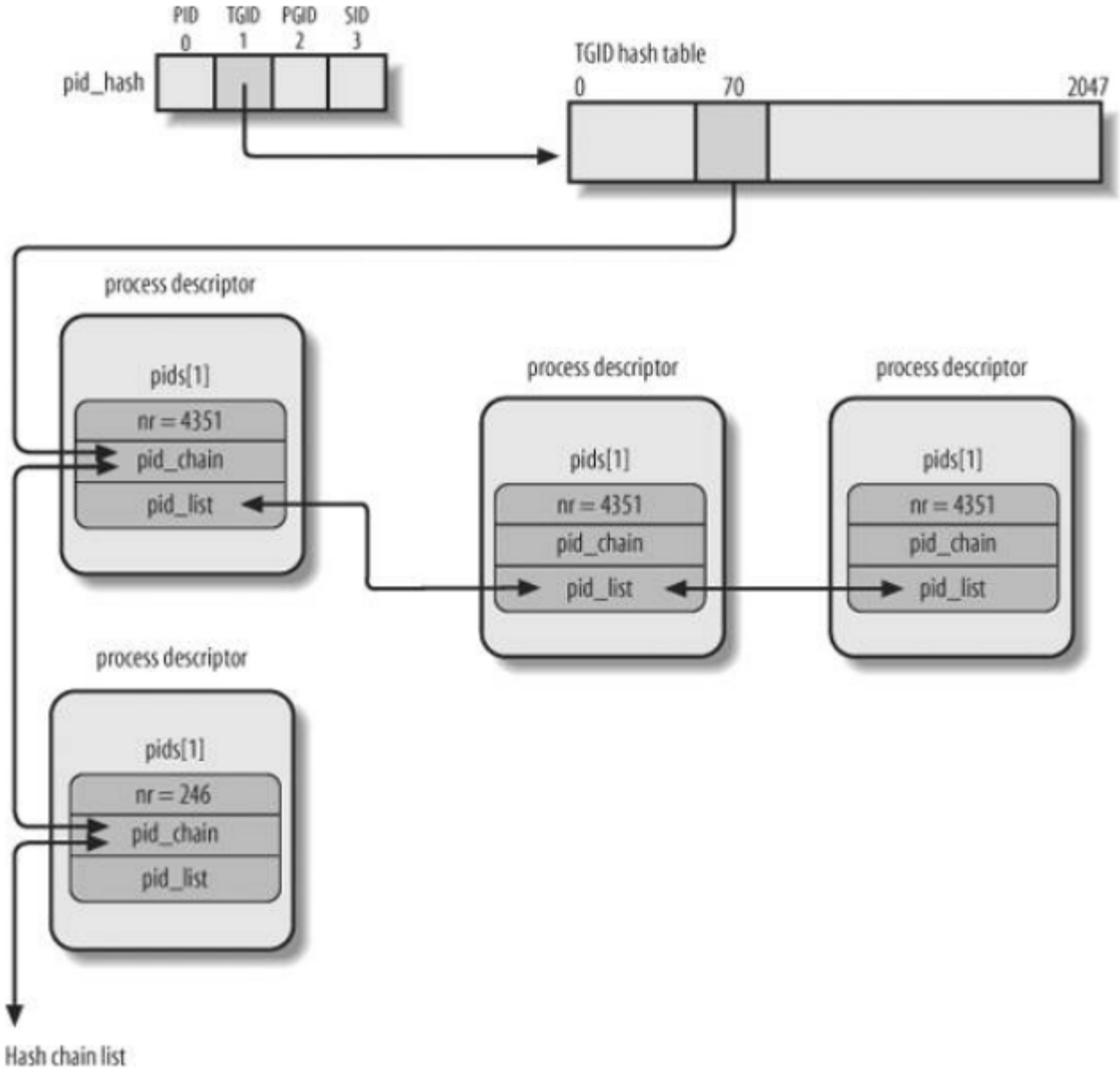
task_struct中相应字段

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

Linux进程表示：task_struct结构



Linux进程表示：task_struct结构



进程管理: 处于等待态的进程如何组织

Linux中等待态进程的表示

- `TASK_INTERRUPTIBLE`
- `TASK_UNINTERRUPTIBLE`

进入等待态的原因(所要等待的事件)

- 等待磁盘操作结束
- 等待固定时间（如1秒）
- 等待互斥锁的释放
- 等待其他临界资源

进程管理: 处于等待态的进程如何组织

等待队列(wait queue)

把等待同一事件的进程放到一个等待队列中去。一旦所等待的事件发生，则可以唤醒该队列中的进程。

wait queue表头的定义: include/linux/wait.h

```
struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
```

其中，字段lock用来实现对等待队列的互斥访问（因为内核中有多个地方可能需要对该队列进程操作）。

进程管理: 处于等待态的进程如何组织

等待队列中的元素：include/linux/wait.h

```
struct __wait_queue {
    unsigned int flags;
    struct task_struct * task;
    wait_queue_func_t func;
    struct list_head task_list;
};
```

- flags表示该进程等待的事件是否为exclusive
- task为进入等待队列的进程
- func为唤醒该进程的方式（函数）
- task_list用于将所有进程连接成队列

进程管理: 处于等待态的进程如何组织

exlusive进程和non-exlusive进程

- exlusive进程: 该进程等待的资源不能共享, flags设为1
  - 例如该进程等待的是进入临界区的机会
- non-exclusive进程: 该进程等待的事件/资源可以共享, flags设为0
  - 例如该进程等待的是磁盘数据传输

思考

为什么将flags放到`__wait_queue`而不是放到`__wait_queue_head`中去?

对等待队列(wait queue)的操作: 初始化

include/linux/wait.h

- `DECLARE_WAIT_QUEUE_HEAD(name)` 声明名为name的队头并初始化
- `init_waitqueue_head(q)` 对动态分配的队头q初始化
- `init_waitqueue_entry(p,q)` 对队列元素q初始化:

```
q->flags = 0;
q->task = p;
q->func = default_wake_function;
```

- `DEFINE_WAIT(name)` 将当前占有CPU的进程封装为名为name的队列元素

课外练习

阅读并理解上述对队列头和队列元素初始化的函数或者宏。

对等待队列的操作: 队列元素的插入和删除

kernel/wait.c

- `add_wait_queue(q, wait)`
- `add_wait_queue_exclusive(q, wait)`
- `remove_wait_queue(q, wait)`
- `waitqueue_active(q)` 功能是什么？(include/linux/wait.h)

对等待队列的操作: 进程如何将自己放入等待队列?

- `sleep_on`
- `interruptible_sleep_on`
- `sleep_on_timeout`
- `interruptible_sleep_on_timeout`

对等待队列的操作: 进程如何将自己放入等待队列?

```
void sleep_on(wait_queue_head_t *wq)
{
    wait_queue_t wait;
    init_waitqueue_entry(&wait, current);
    current->state = TASK_INTERRUPTIBLE;
    add_wait_queue(wq, &wait);
    schedule();
    remove_wait_queue(wq, &wait);
}
```

对等待队列的操作: 进程如何将自己放入等待队列?

- `prepare_to_wait`
- `prepare_to_wait_exclusive`
- `finish_wait`

对等待队列的操作: 进程如何将自己放入等待队列?

```
DEFINE_WAIT(wait);
prepare_to_wait(&wq,&wait,TASK_INTERRUPTIBLE);
...
if (!condition)
    schedule();
finish_wait(&wq, &wait);
```

对等待队列的操作: 进程如何将自己放入等待队列?

```
DEFINE_WAIT(wait);
for (;;) {
    prepare_to_wait(&wq, &wait,
                    TASK_UNINTERRUPTIBLE);
    if (condition)
        break;
    schedule();
}
finish_wait(&wq, &wait);
```

对等待队列的操作: 如何把队列中的进程唤醒?

- `wake_up`
- `wake_up_nr`
- `wake_up_all`
- `wake_up_interruptible`
- `wake_up_interruptible_nr`
- `wake_up_interruptible_all`
- `wake_up_interruptible_sync`
- `wake_up_locked`

对等待队列的操作: 如何把队列中的进程唤醒?

- 所有non-exclusive的进程都被唤醒
- 如果名字中没有interruptible, 则唤醒所有睡眠进程(`TASK_INTERRUPTIBLE`及`TASK_UNINTERRUPTIBLE`). 否则，只唤醒队列中`TASK_INTERRUPTIBLE`状态的进程。
- 名字中含有nr的宏唤醒指定个数的exclusive进程
- 名字中含有all的宏唤醒全部指定状态的exclusive进程
- 名字中没有nr或者all的宏只唤醒一个exclusive进程

对等待队列的操作: 如何把队列中的进程唤醒?

Linux进程表示：task_struct结构

task_struct结构的rlim字段: include/linux/resource.h

```
struct rlimit {
    unsigned long   rlim_cur;
    unsigned long   rlim_max;
};

struct task_struct {
    ...
    struct rlimit rlim[RLIM_NLIMITS];
    ...
};
```

相关系统调用:

```
int getrlimit(int res, struct rlimit *rlim);
int setrlimit(int res, const struct rlimit *rlim);
```

Linux进程表示：task_struct结构

task_struct结构的rlim字段: include/asm-generic/resource.h

```
#define RLIMIT_CPU      0
#define RLIMIT_FSIZE        1
#define RLIMIT_DATA     2
#define RLIMIT_STACK        3
#define RLIMIT_CORE     4
#define RLIMIT_NOFILE       7
...
#define RLIM_NLIMITS        15
```

相关命令

cat /proc/self/limits

Linux进程表示：task_struct结构

```
struct task_struct {
    ...
    struct user_struct thread;
    ...
};
```

Linux进程表示：task_struct结构

include/asm-x86_64/processor.h

```
struct thread_struct {
    unsigned long   rsp0;
    unsigned long   rsp;
    unsigned long   fs;
    unsigned long   gs;
    unsigned short  es, ds, fsindex, gsindex;
    ...
};
```

通用寄存器的值保存在哪里?

eax, ebx等通用寄存器在上下文切换时保存在kernel mode stack (核心态下的栈中)

创建进程的系统调用

1. fork用于创建新进程
2. vfork创建新进程，并且只有当子进程运行完毕，父进程才能运行；二者共享存储空间(deprecated)
3. clone用于创建进程或者线程

创建进程的系统调用: 写拷贝技术(copy-on-write)

历史上，unix中调用fork创建新进程时，OS需要将父进程的存储空间完全拷贝一份给子进程，这有以下缺点:

- 拷贝内存的过程非常耗时
- 需要占用大量内存空间
- 子进程一旦执行exec，则上述拷贝完全浪费

写拷贝(copy-on-write): 创建新进程时，仅拷贝父进程的页表，并将所有页表项对应的页面设置成只读，只有当父亲或子进程需要写入某页面时，才拷贝相应页面的内容。

创建进程: do_fork函数

kernel/fork.c

```
long do_fork(unsigned long clone_flags,
             unsigned long stack_start,
             struct pt_regs *regs,
             unsigned long stack_size,
             int __user *parent_tidptr,
             int __user *child_tidptr)
```

- clone_flags用于描述进程的哪些属性将被复制(进程/线程!)
- start_stack为用户态下进程的栈起始位置, stack_size为栈的总长度
- regs, parent_tidptr等参数后面讲

创建进程: do_fork函数

arch/x86/kernel/process_32.c

```
asmlinkage int sys_fork(struct pt_regs regs)
{
    return do_fork(SIGCHLD, regs.esp, &regs,
                   0, NULL, NULL);
}
```

创建进程: do_fork函数

arch/x86/kernel/process_32.c

```
asmlinkage int sys_clone(struct pt_regs regs)
{
    unsigned long clone_flags;
    unsigned long newsp;
    int __user *parent_tidptr, *child_tidptr;
    clone_flags = regs.ebx;
    newsp = regs.ecx;
    parent_tidptr = (int __user *)regs.edx;
    child_tidptr = (int __user *)regs.edi;
    if (!newsp)
        newsp = regs.esp;
    return do_fork(clone_flags, newsp, &regs,
                   0, parent_tidptr, child_tidptr);
}
```