

# 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 27, 2012

# 本课程Linux内核版本

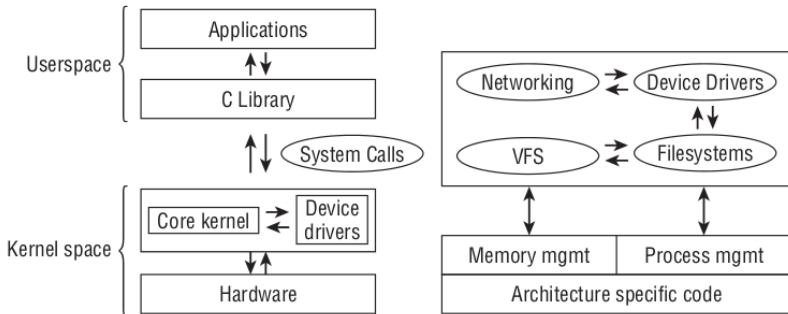
- ▶ linux kernel: 2.6.11.12
- ▶ 为什么使用这个版本的内核?
- ▶ 下载地址:

<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



## 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

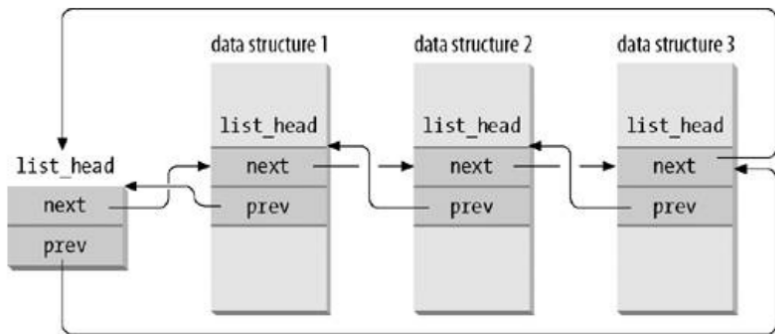
list\_head的使用方法举例: include/linux/sched.h

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

思考

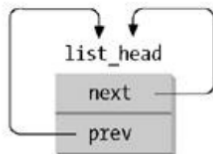
仍与fox结构相比，这样使用list\_head构造链表有什么好处？

# Linux内核关键数据结构: list



(a) a doubly linked list 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

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

## 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;
```

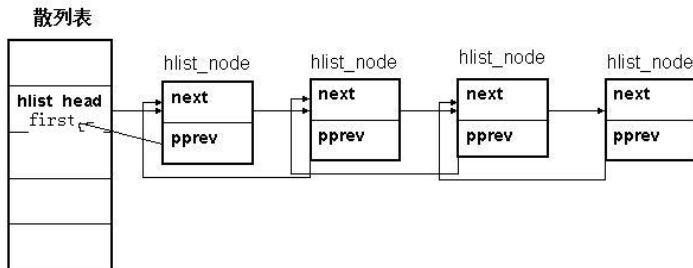
# 针对list结构的操作: 代码分析举例

## 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

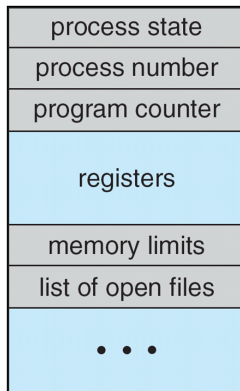
阅读include/linux/list.h中hlist\_add\_head(), hlist\_del(), hlist\_empty(), hlist\_add\_before()等函数的实现, 思考:

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

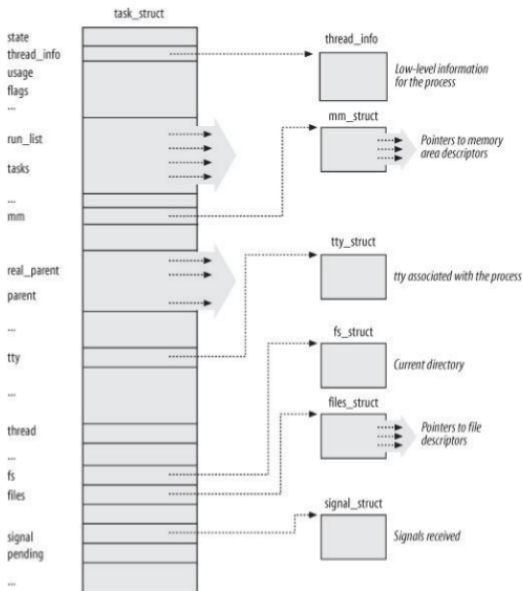
- ▶ 进程状态
- ▶ 程序计数器PC
- ▶ CPU寄存器
- ▶ CPU调度信息
- ▶ 内存管理信息
- ▶ I/O状态信息
- ▶ 记账信息



# 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

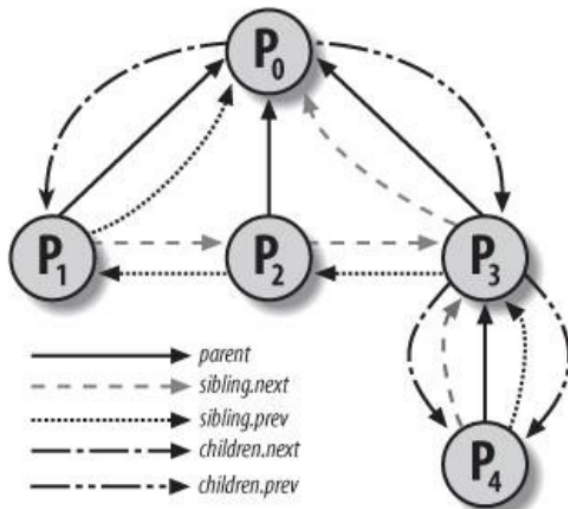
阅读以上两个函数的源代码(位于文件`include/linux/sched.h`), 并思考为什么要这样写。

## 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: `ps tree`

# 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

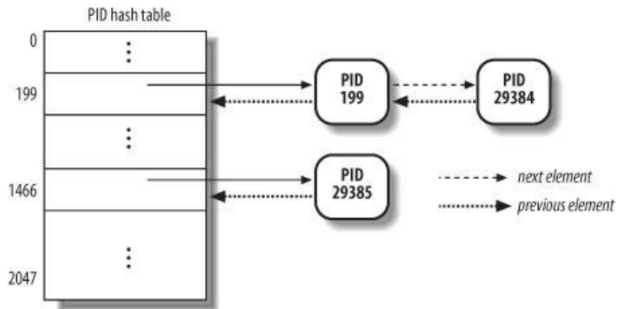
散列表中元素struct pid: include/linux/pid.h

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

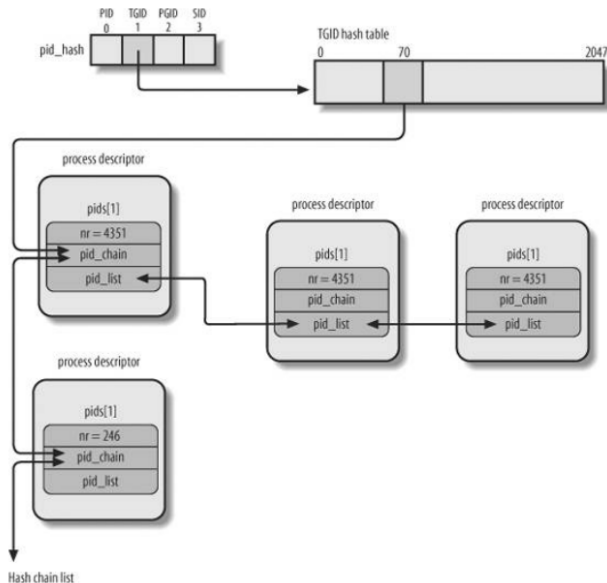
task\_struct中相应字段

```
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



## Linux进程表示: task\_struct结构

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

## Linux进程表示: task\_struct结构

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

# 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);
}
```