OS概念与Linux内核源代码分析之1

May 21, 2012

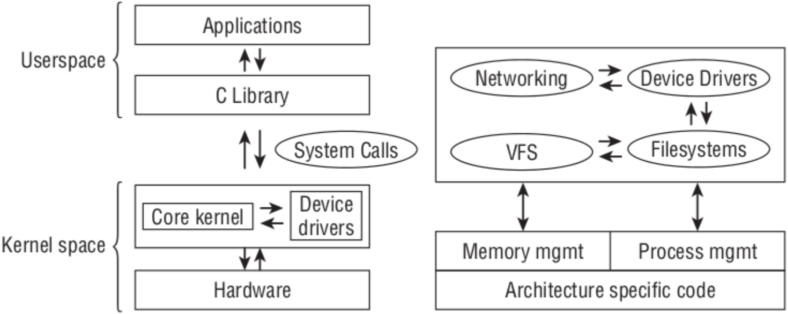
本课程Linux内核版本

- 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
- (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：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

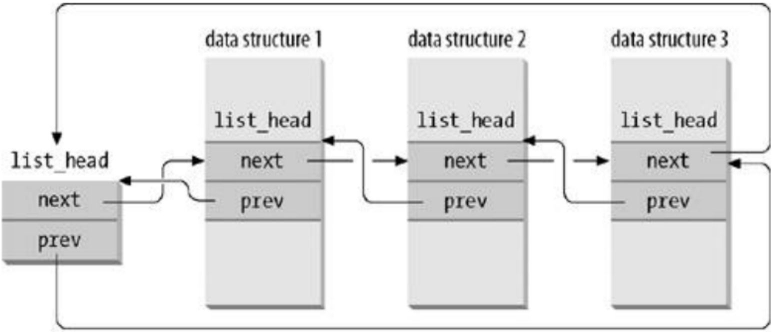
list_head的定义: include/linux/list.h

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

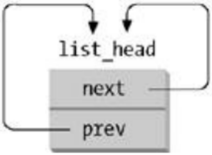
list_head的使用方法: include/linux/sched.h

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

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_entry(ptr, type, member) 计算包含ptr的结构体的地址
- 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

```c
typedef struct list_head list_head;

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

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

针对list结构的操作: 代码分析举例

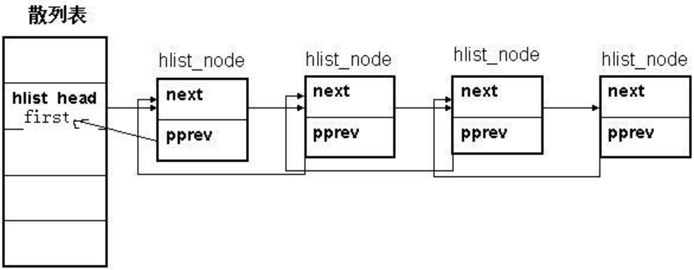
```
struct list_head *p;

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?

Linux进程描述符: Process Descriptor

task_struct结构: include/linux/sched.h

```
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

```
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];
};
```

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
- 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;
}
```

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

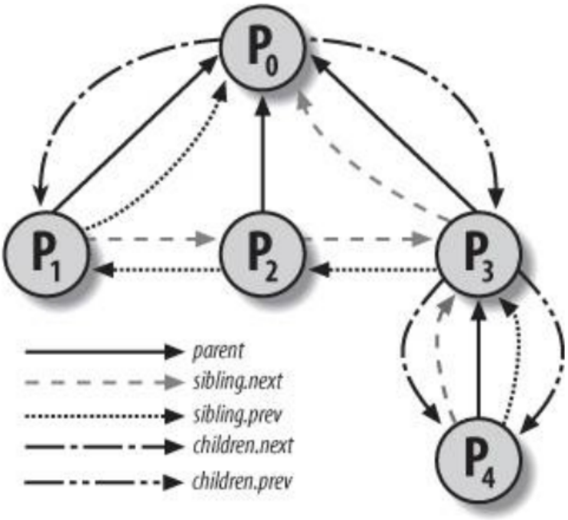
课外练习

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

```c
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

```
struct task_struct {
    ...
    struct task_struct *group_leader;
    pid_t tgid;
    struct list_head ptrace_children;
    struct list_head ptrace_list;
    struct signal_struct *signal;
    ...
};
```

Linux进程表示：task_struct结构

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

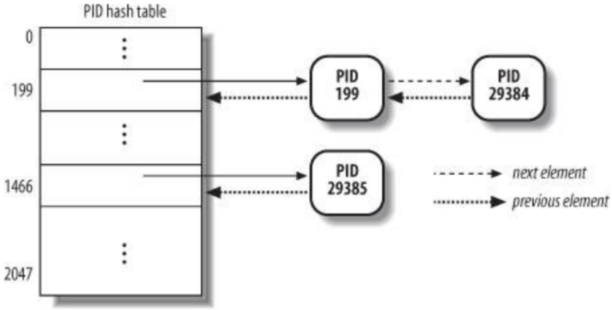
Linux进程表示：task_struct结构

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

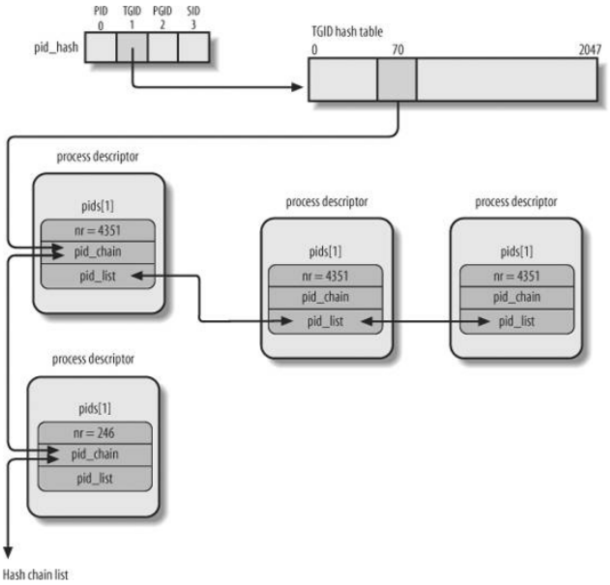
Linux进程表示：task_struct结构

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

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

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);
}
```