实验2：Linux内核编译模块

目录

[基本概念 1](#_Toc28390963)

[1.内核态和用户态,特权级 1](#_Toc28390964)

[2.什么是单体式结构? 2](#_Toc28390965)

[3.什么是微内核结构 2](#_Toc28390966)

[部分源码理解 2](#_Toc28390967)

[1. task\_struct 2](#_Toc28390968)

[2. list\_head 2](#_Toc28390969)

[3. 一些函数的理解 3](#_Toc28390970)

## 基本概念

### 1.内核态和用户态,特权级

Linux中每个进程有两个栈，分别用于用户态和内核态的进程执行，其中的内核栈就是用于内核态的堆栈，它和进程的task*struct结构，更具体的是thread*info结构一起放在两个连续的页框大小的空间内。现在我们从特权级的调度来理解用户态和内核态就比较好理解了，当程序运行在**3级特权级**上时，就可以称之为运行在**用户态**，因为这是最低特权级，是普通的用户进程运行的特权级，大部分用户直接面对的程序都是运行在用户态；反之，当程序运行在**0级特权级**上时，就可以称之为运行在**内核态**。虽然用户态下和内核态下工作的程序有很多差别，但最重要的差别就在于特权级的不同，即权力的不同。运行在用户态的程序不能访问操作系统内核数据结构合程序。

当我们在系统中执行一个程序时，大部分时间是运行在用户态下的。在其需要操作系统帮助完成某些它没有权力和能力完成的工作时就会切换到内核态。 Linux进程的4GB地址空间，3G-4G部分大家是共享的，是内核态的地址空间，这里存放在整个内核的代码和所有的内核模块，以及内核所维护的数据。用户运行一个程序，该程序所创建的进程开始是运行在用户态的，如果要执行文件操作，网络数据发送等操作，**必须通过write，send等系统调用**，这些系统调用会调用内核中的代码来完成操作，这时，必须切换到**Ring0**，然后进入3GB-4GB中的内核地址空间去执行这些代码完成操作，完成后，切换回**Ring3**，回到用户态。这样，用户态的程序就不能随意操作内核地址空间，具有一定的安全保护作用。保护模式，通过内存页表操作等机制，保证进程间的地址空间不会互相冲突，一个进程的操作不会修改另一个进程的地址空间中的数据。

在内核态下，CPU可执行任何指令，在用户态下CPU只能执行非特权指令。当CPU处于内核态，可以随意进入用户态；而当CPU处于用户态，只能通过**中断的方式**进入内核态。一般程序一开始都是运行于用户态，当程序需要使用系统资源时，就必须通过调用软中断进入内核态.

Intel x86架构的cpu一共有0～4四个特权级，0级最高，3级最低，硬件上在执行每条指令时都会对指令所具有的特权级做相应的检查。硬件已经提供了一套特权级使用的相关机制，软件自然要好好利用，这属于操作系统要做的事情，**对于UNIX/LINUX来说，只使用了0级特权级别和3级特权级**。也就是说在UNIX/LINUX系统中，一条工作在0级特权级的指令具有了CPU能提供的最高权力，而一条工作在3级特权的指令具有CPU提供的最低或者说最基本权力

### 2.什么是单体式结构?

大内核，将OS的全部功能都做进内核中，包括调度、文件系统、网络、设备驱动器、存储管理。比如设备驱动管理、资源分配、进程间通信、进程间切换管理、文件系统、存储管理、网络等。单体内核是指在一大块代码中实际包含了所有操作系统功能，并作为一个单一进程运行，具有唯一地址空间。大部分UNIX（包括Linxu）系统都采用的单体内核。

### 3.什么是微内核结构

微内核只是将OS中最核心的功能加入内核，包括IPC通信、地址空间分配和基本的调度，这些东西处在内核态运行。如：WINCE系统。而其他功能如设备驱动、文件系统、存储管理、网络等作为一个个处于用户态的进程而向外提供某种服务来实现，而且这些处于用户态的进程可以针对某些特定的应用和环境需求进行定制

## 部分源码理解

### 1. task\_struct

[1.Linux进程管理之task\_struct结构体（上）](https://blog.csdn.net/npy_lp/article/details/7292563%20)

[2.Linux进程管理之task\_struct结构体（下）](https://blog.csdn.net/npy_lp/article/details/7335187%20)

[Linux pid与tgid概念](https://www.cnblogs.com/shihuvini/p/10043251.html)

1. 表示进程亲属关系的成员

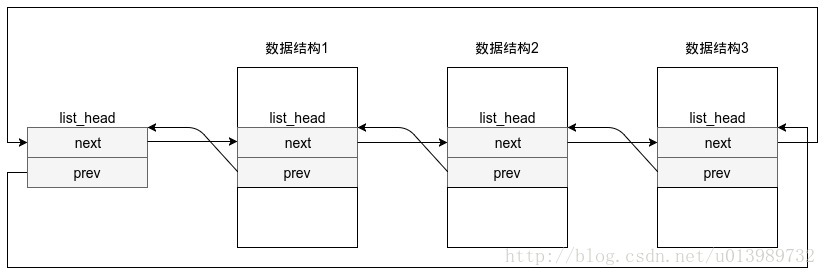
* struct task\_struct \*real\_parent; /\* real parent process \*/  
  struct task\_struct \*parent; /\* recipient of SIGCHLD, wait4() reports \*/  
  struct list\_head children; /\* list of my children \*/  
  struct list\_head sibling; /\* linkage in my parent's children list \*/  
  struct task\_struct \*group\_leader; /\* threadgroup leader \*/

1. 进程名

* char comm[TASK\_COMM\_LEN];

### 2. list\_head

#kernel/inclue/linux/types.h   
struct list\_head {  
 struct list\_head \*next, \*prev;  
};



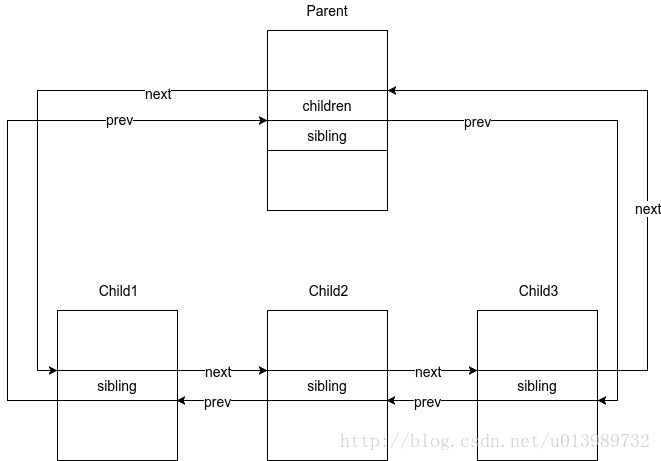
### 3. 一些函数的理解

#### 3.1 list *for*e ach

#head是一个list\_head  
#define list\_for\_each(pos, head) \  
 for (pos = (head)->next; pos != (head);pos = pos->next)

它实际上是一个 for 循环，利用传入的pos 作为循环变量，从表头 head开始，逐项向后（next方向）移动 pos ，直至又回到 head

注意：此宏必要把list\_head放在数据结构第一项成员，至此，它的地址也就是结构变量的地址。



#### 3.2 struct pid

很多时候在写内核模块的时候，需要通过进程的**pid**(int)找到对应进程的**task\_struct**,其中首先就需要通过进程的pid找到进程的**struct pid**，然后再通过**struct pid**找到进程的**task\_struct**,实现从pid到task\_struct的内核函数struct pid 是进程描述符，在include/linux/pid.h 中，其定义如下：

struct pid  
{  
 atomic\_t count;  
 unsigned int level;  
 /\* lists of tasks that use this pid \*/  
 struct hlist\_head tasks[PIDTYPE\_MAX];  
 struct rcu\_head rcu;  
 struct upid numbers[1];  
};

* struct pid \*find\_get\_pid(pid\_t nr)

//是通过进程号pid\_t nr得到进程描述符(结构体)，并将结构体中的count加1  
struct pid \*find\_get\_pid(pid\_t nr)   
{   
 struct pid \*pid;   
 rcu\_read\_lock();   
 pid = get\_pid(find\_vpid(nr));   
 rcu\_read\_unlock();   
 return pid;   
}

find\_get\_pid和find\_vpid有一点差异，就是使用find\_get\_pid将返回的struct pid中的字段count加1，而find\_vpid没有加1

static inline struct pid \*get\_pid(struct pid \*pid)  
{  
 if (pid)  
 atomic\_inc(&pid->count);  
 return pid;  
}

find\_vpid和find\_pid\_ns是一样的

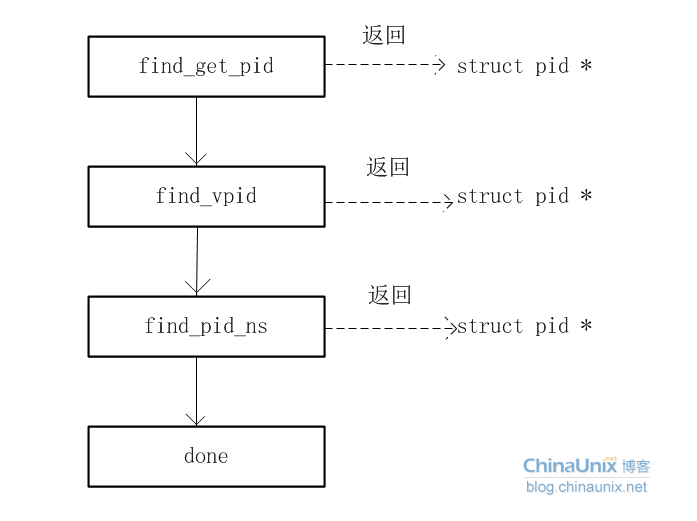
* struct pid \*find\_vpid(int nr)

struct pid \*find\_vpid(int nr)  
{  
 return find\_pid\_ns(nr, task\_active\_pid\_ns(current));  
}

* struct pid \*find\_pid\_ns(int nr, struct pid\_namespace \*ns)

[find*pid*ns详解](http://blog.chinaunix.net/uid-27033491-id-3291637.html)

struct pid \*find\_pid\_ns(int nr, struct pid\_namespace \*ns)  
{  
 return idr\_find(&ns->idr, nr);  
}



#### 3.3 pid\_task

根据PID(结构体)查找task*struct(PCB),一般`p = pid*task(find*vpid(pid), PIDTYPE*PID);`

struct task\_struct \*pid\_task(struct pid \*pid, enum pid\_type type)  
{  
 struct task\_struct \*result = NULL;  
 if (pid) {  
 struct hlist\_node \*first;  
 first = rcu\_dereference\_check(hlist\_first\_rcu(&pid->tasks[type]),  
 lockdep\_tasklist\_lock\_is\_held());  
 if (first)  
 result = hlist\_entry(first, struct task\_struct, pids[(type)].node);  
 }  
 return result;  
}

#### 3.4 list\_entry

#list\_entry 就是 container\_of   
#define list\_entry(ptr, type, member) \  
 container\_of(ptr, type, member)

那container\_of(ptr, type, member),又是什么呢

#### 3.5 container\_of

[资料](%20https://www.cnblogs.com/hicjiajia/archive/2012/07/01/2571791.html%20)

[通俗理解](%20https://blog.csdn.net/zsj100213/article/details/81843523%20)

container\_of(ptr, type, member)

#define container\_of(ptr, type, member) ({ \  
 const typeof( ((type \*)0)->member ) \*\_\_mptr = (ptr); \  
 (type \*)( (char \*)\_\_mptr - offsetof(type,member) );})

container\_of()的作用就是通过一个结构变量中一个成员的地址找到这个结构体变量的首地址

比如

Struct test  
{  
 int i;  
 int j;  
 char k;  
};  
Struct test temp;

现在呢如果我想通过temp.j的地址找到temp的首地址就可以使用container\_of(&(temp.j),struct test,j);

如果我能知道temp.j的地址A,也能知道temp.j的偏移量B,那么A-B就是temp的首地址

网上有一种很形象的比喻:x->member = ptr，求 x (结构体)的地址是多少

#### 3.6 for *each* process

#define for\_each\_process(p) \  
 for (p = &init\_task ; (p = next\_task(p)) != &init\_task ; )

就是个for循环,其中p = next\_task(p),这个next\_task是怎么实现的呢

#### 3.6 next\_task

Linux内核链表,其中next\_task也是宏循环控制语句，在/include/linux/sched/signal.h中：

#define next\_task(p) \  
 list\_entry\_rcu((p)->tasks.next, struct task\_struct, tasks)

而task\_struct中有：

struct task\_struct {   
...  
 struct list\_head tasks;  
...  
}

这里的list\_entry\_rcu作用和list\_entry是一样的,我通过task*struct中, tasks这个成员变量以及tasks这个成员变量的地址(p)->tasks.next,得到下一个task*struct的首地址,即下一个进程

使用 ps -u root 可以查看进程信息