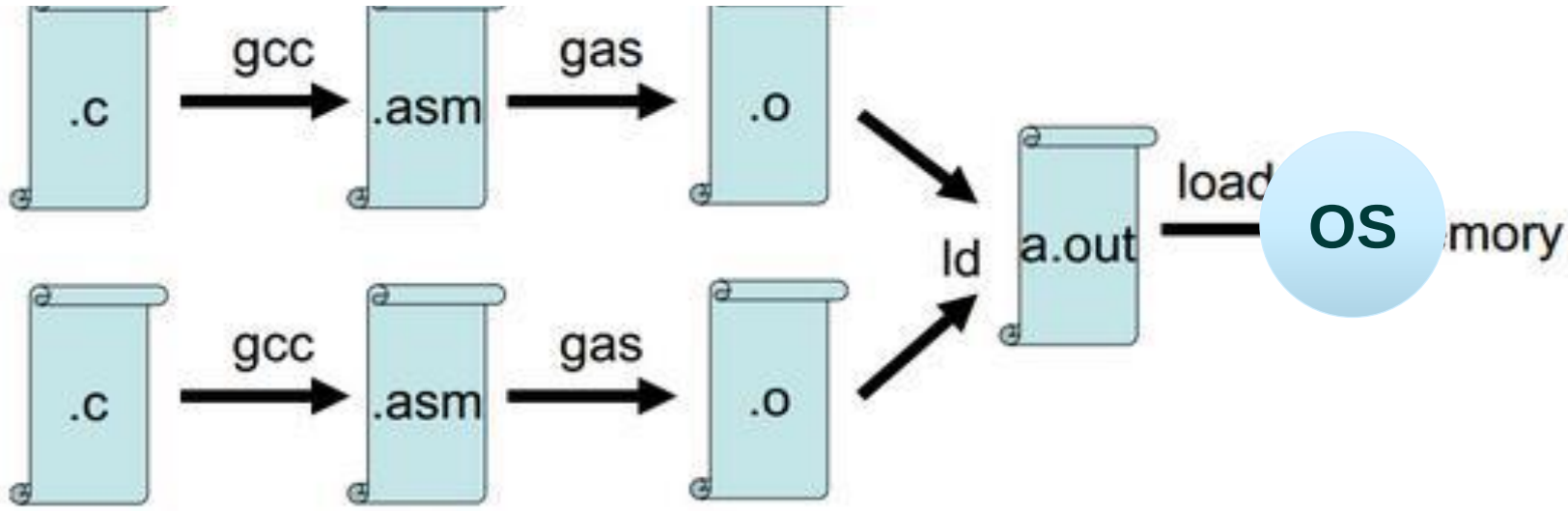
3.1 进程概述

西安邮电大学

从程序到进程



1. gcc -S hello.c -o hello.s // 编译
2. gcc -c hello.s -o hell.o // 汇编
3. gcc hello.c -o hello // 链接
4. ./hello // 装载并执行

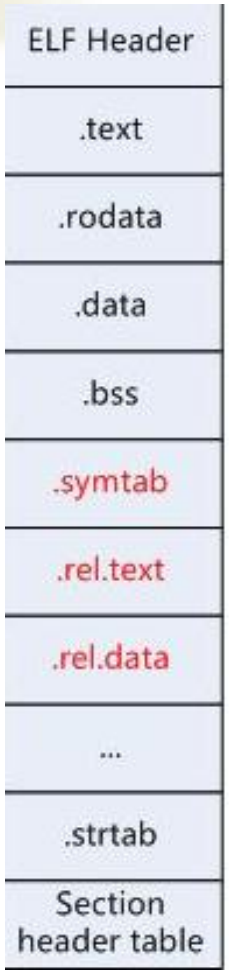
objdump -d hello //反汇编

从程序到进程

一个程序通过编译器GCC将其编译成汇编程序，经过汇编器gas将其汇编成目标代码，经过连接器ld形成可执行文件a.out或者ELF格式，最后交给操作系统来执行。

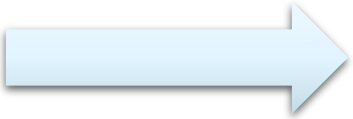
那么，问题来了，操作系统如何应对千变万化的程序？

从程序到进程



ELF可执行文件格式

交给OS执行



操作系统

从程序到进程

当一个程序一旦执行，程序也摇身一变成为进程。在OS看来，每个进程是没有多大差异性的，都被封装在这样的可执行文件格式中，在内存管理一章，我们将继续详细介绍进程的执行和加载。

我们也可以通过top命令感知系统中各个进程以及动态变化，如图：

从程序到进程

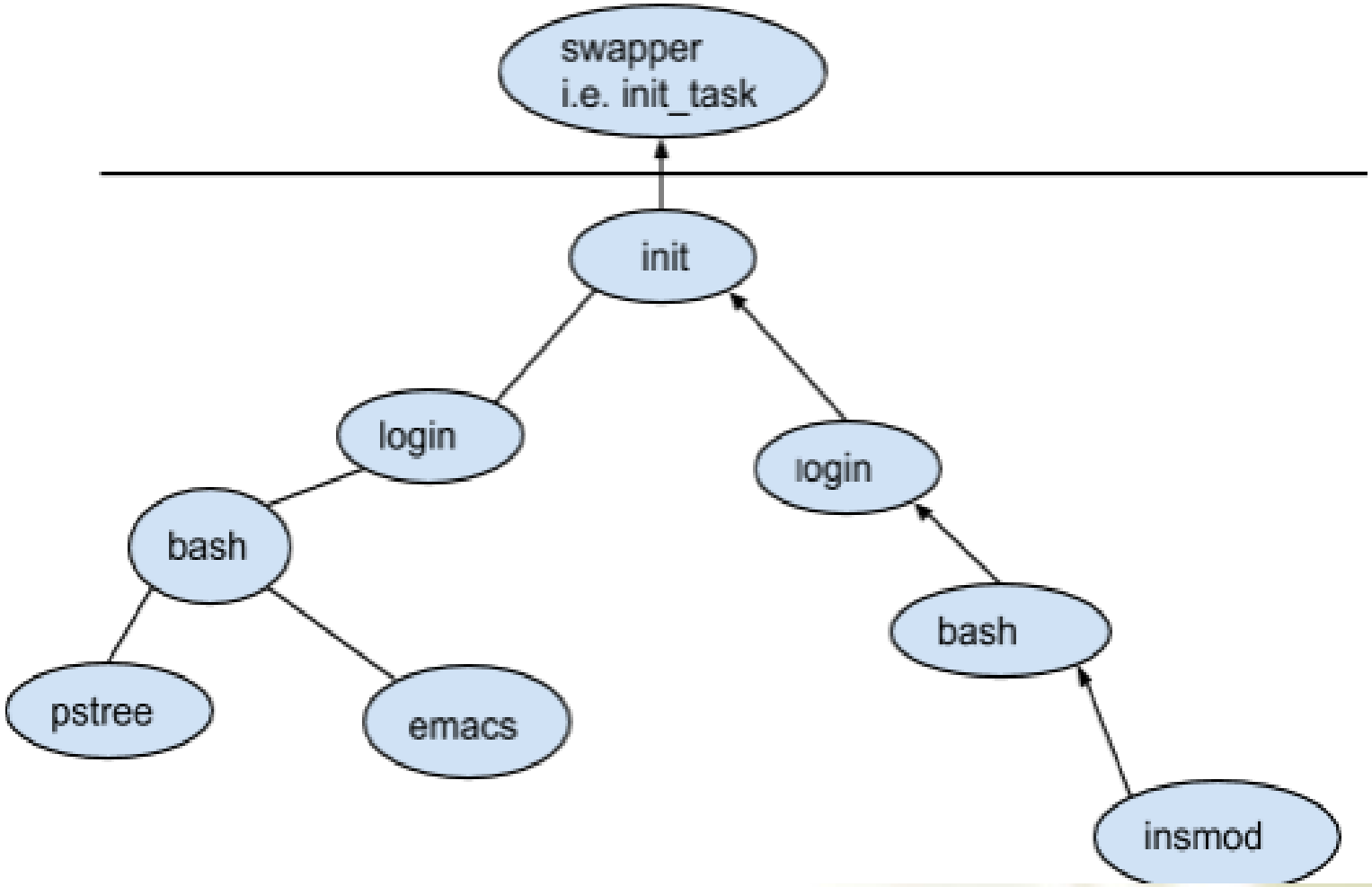
```
top - 18:18:41 up 44 days, 42 min,  1 user,  load average: 0.00, 0.01, 0.05
Tasks:  88 total,   1 running,  87 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.3 us,  0.7 sy,  0.0 ni, 99.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1882772 total,   124828 free,   598684 used,  1159260 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  1082964 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1069 root      20   0  133464   9288   5488 S   0.7  0.5 229:04.84 AliYunDun
 1155 root      20   0 2057492  68552   2992 S   0.7  3.6 332:01.70 java
 1426 root      20   0  267508   7740   1640 S   0.3  0.4  34:37.41 docker-cont+
13571 clj       20   0  161880   2180   1568 R   0.3  0.1   0:00.11 top
26300 root      20   0  365808  17180   4260 S   0.3  0.9  15:28.88 python
    1 root      20   0   43576   3796   2448 S   0.0  0.2   8:54.71 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.09 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:33.32 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    7 root      rt   0       0      0      0 S   0.0  0.0   0:00.00 migration/0
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      20   0       0      0      0 S   0.0  0.0  13:06.90 rcu_sched
   10 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 lru-add-dra+
   11 root      rt   0       0      0      0 S   0.0  0.0   0:16.40 watchdog/0
   13 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kdevtmpfs
   14 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 netns
   15 root      20   0       0      0      0 S   0.0  0.0   0:01.14 khungtaskd
```

进程的家族关系

进程是一个动态的实体，它具有生命周期，系统中进程的生生死死随时发生。因此，操作系统对进程的描述模仿人类活动。一个进程不会平白无故的诞生，它总会有自己的父母。在Linux中，通过调用fork系统调用来创建一个新的进程。新创建的子进程同样也能执行fork，所以，可以形成一颗完整的进程树。如图是Linux系统启动以后形成的一棵树，可以通过:ps -ejH命令，查看自己机子上的进程树。

进程的家族关系


swapper
i.e. init_task
init
login
login
bash
bash
pstree
emacs
insmod

进程控制块

如何描述进程的属性？

Linux内核中把对进程的描述结构叫task_struct:

```
struct task_struct {
…
…
    };
```

★传统的教科书中，这样的数据结构被叫做进程控制块PCB（process control blaock）

在内核源代码中具体定义在sched.h文件中

进程控制块—信息分类

- ★状态信息—描述进程动态的变化。
- ★链接信息—描述进程的父/子关系。
- ★各种标识符—用简单数字对进程进行标识。
- ★进程间通信信息—描述多个进程在同一任务上协作工作。
- ★时间和定时器信息—描述进程在生存周期内使用CPU时间的统计、计费等信息。
- ★调度信息—描述进程优先级、调度策略等信息。
- ★文件系统信息—对进程使用文件情况进行记录。
- ★虚拟内存信息—描述每个进程拥有的地址空间。
- ★处理器环境信息—描述进程的执行环境(处理器的寄存器及堆栈等)

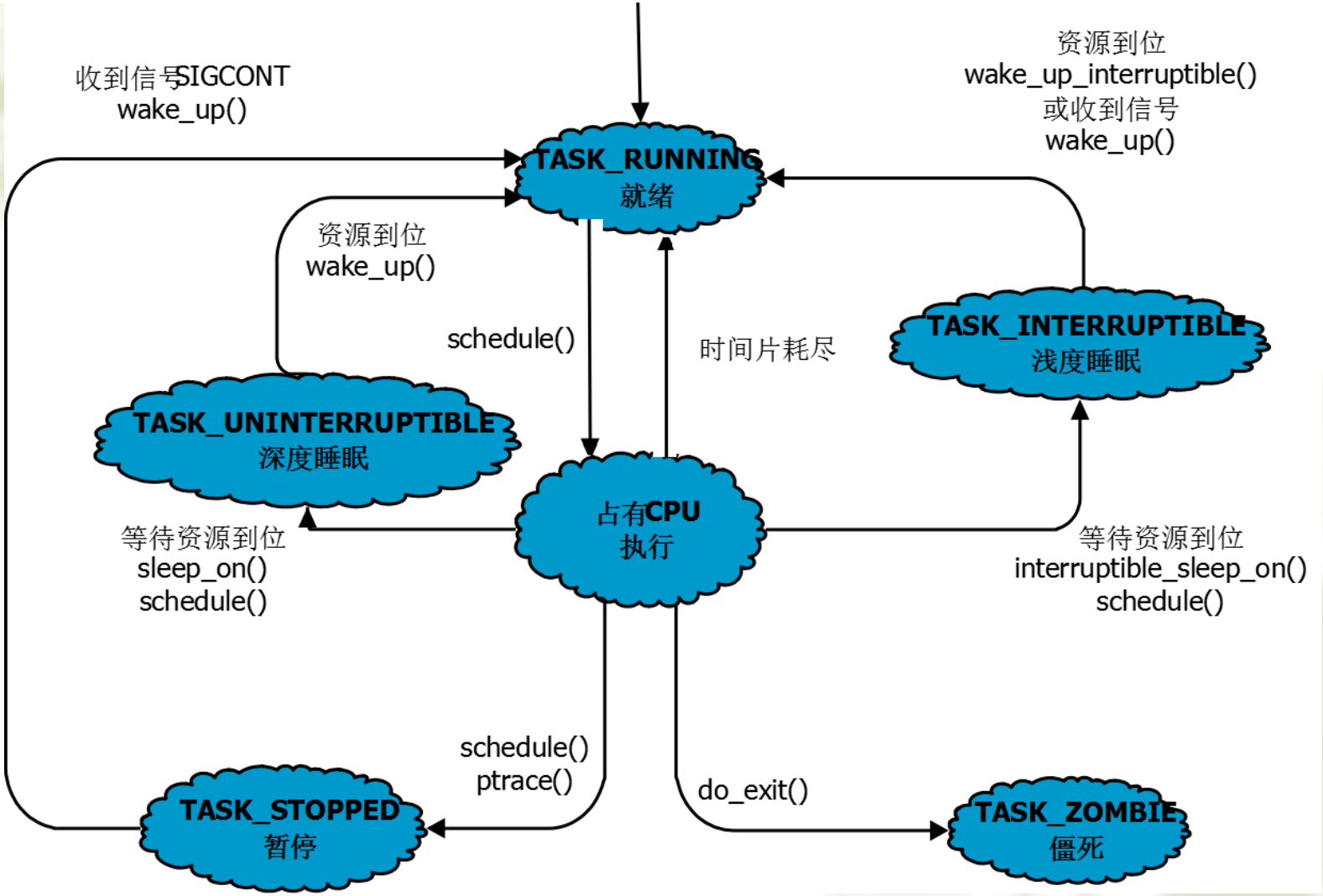
进程控制块—信息分类

因为进程控制块中的信息非常多，多大几百个字段，为了有助于对其认识，这里对其进行了分类，包括状态信息，链接信息，各种标识符，调度信息，进程间通信信息，虚拟内存信息，文件系统信息，处理器环境信息等等，涉及到进程方方面面，具体要通过查看源码对它们逐渐认识。

进程控制块—Linux进程状态及转换

状态是用来描述进程的动态变化的。最基本的状态有三种：就绪，睡眠和运行。在具体的操作系统中，可能实例化出多个状态，如图给出Linux中进程5种状态，把就绪态和运行态合并为一个状态叫就绪态，调度程序从就绪队列中选中一个进程投入运行。睡眠态又被划分为两种：浅度睡眠（很容易被唤醒）和深度睡眠。除此之外，还有暂停状态（比如调试程序时所处的状态）和僵死状态（进程死亡，但没有释放其PCB）

进程控制块—Linux进程状态及转换



进程控制块—进程状态

```
#define TASK_RUNNING                0
#define TASK_INTERRUPTIBLE          1
#define TASK_UNINTERRUPTIBLE        2
#define __TASK_STOPPED              4
#define __TASK_TRACED               8/* 由调试程序暂停进程的执行 */
/* in tsk->exit_state */
#define EXIT_ZOMBIE                 16
#define EXIT_DEAD                   32 /*最终状态，进程将被彻底
                                        删除，但需要父进程来回收*/
/* in tsk->state again */
#define TASK_DEAD                   64  /*与EXIT_DEAD类似，但不
                                        需要父进程回收*/
#define TASK_WAKEKILL               128/*接收到致命信号时唤醒
                                        进程，即使深度睡
```

以上是Linux源代码中对状态的定义，请观察有什么特点？

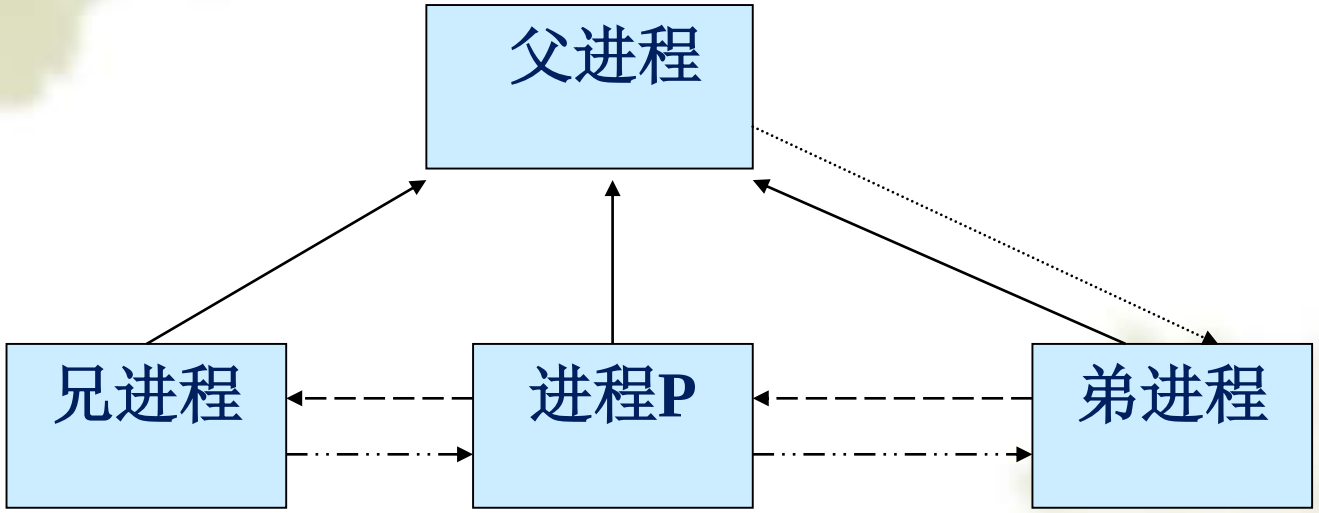
为什么每个状态的值定义为2的n次方?

进程控制块—进程状态

大家务必查看源代码，低版本可以查看2.6.18，可以看到不同版本其状态个数少有差异

```
/*
 * Task state bitmask. NOTE! These bits are also
 * encoded in fs/proc/array.c: get_task_state().
 *
 * We have two separate sets of flags: task->state
 * is about runnability, while task->exit_state are
 * about the task exiting. Confusing, but this way
 * modifying one set can't modify the other one by
 * mistake.
 */
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define __TASK_STOPPED          4
#define __TASK_TRACED           8
/* in tsk->exit_state */
#define EXIT_DEAD               16
#define EXIT_ZOMBIE             32
#define EXIT_TRACE              (EXIT_ZOMBIE | EXIT_DEAD)
/* in tsk->state again */
#define TASK_DEAD               64
#define TASK_WAKEKILL           128
#define TASK_WAKING             256
#define TASK_PARKED             512
#define TASK_STATE_MAX          1024
```

进程控制块—进程之间的亲属关系



指向父进程　　指向子进程

指向兄进程　　指向弟进程

进程控制块—进程之间的亲属关系

系统创建的进程具有父/子关系。因为一个进程能创建几个子进程，而子进程之间有兄弟关系。在PCB中引入几个域来表示这些关系。如前说述，进程1（init）是所有进程的祖先，系统中的进程形成一颗进程树。为了描述进程之间的父/子及兄弟关系，在进程的PCB中就要引入几个域。假设P表示一个进程，首先要有一个域描述它的父进程（parent）；其次，有一个域描述P的子进程，因为子进程不止一个，因此让这个域指向年龄最小的子进程（child）；最后，P可能有兄弟，于是用一个域描述P的长兄进程（old sibling），一个域描述P的弟进程（younger sibling）。

从这些关系看到，进程完全模拟人类的生存状态，你自己可以试图扮演进程的角色。

进程控制块一部分内容的描述

上面通过对进程状态、标识符及亲属关系的描述，我们可以把这些域描述如下：

```
struct task_struct {
        volatile long state;                    /*进程状态*/
        int pid, uid, gid;                      /*一些标识符*/
        struct task_struct *real_parent;        /*真正创建当前进程
                                                  的进程*/
        struct task_struct *parent;             /*相当于养父*/
        struct list_head children;              /*子进程链表*/
        struct list_head sibling;               /*兄弟进程链表*/
        struct task_struct *group_leader;/*线程组的头进程*/
        …
};
```

进程控制块一部分内容的描述

前面的描述是书本上的，同样。我们进入源代码查看，才有身临其境的感觉，task_struct位于sched.h中，这里列出部分代码片段：

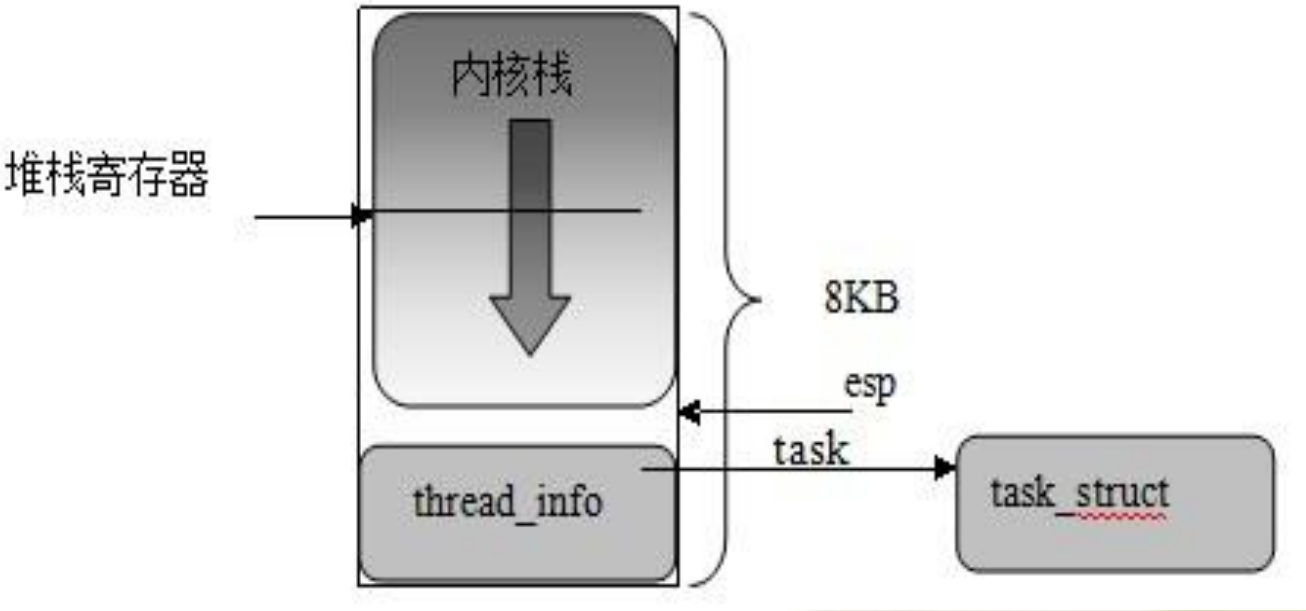
```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */

 pid_t pid;
 pid_t tgid;

  /*
   * pointers to (original) parent process, youngest child, younger sibling,
   * older sibling, respectively.  (p->father can be replaced with
   * p->real_parent->pid)
   */
  struct task_struct __rcu *real_parent; /* real parent process */
  struct task_struct __rcu *parent; /* recipient of SIGCHLD, wait4() reports */
  /*
   * children/sibling forms the list of my natural children
   */
  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 */
```

进程控制块—如何存放

那么，进程控制块存放在内存的什么地方？
进程为了节省空间，Linux把内核栈和一个紧挨近PCB的小数据结构thread_info放在一起，占用8KB的内存区，如下图所示：



PCB和内核栈的存放

进程控制块—如何存放

内核中使用下列的联合结构表示这样一个混合结构：

```
union thread_union {
        struct thread_info thread_info;
        unsigned long stack[THREAD_SIZE/sizeof(long)];
        /*大小一般是8KB，但也可以配置为4KB*/
};
```

x86上，thread_info的定义如下：

```
union thread_info {
        struct task_struct *task;
        struct exec_domain *exec_domain;
        ……
};
```

thread_info表示和硬件关系更密切的数据。第一个字段就是task_sturct结构

进程控制块—如何存放

从这个结构可以看出，内核栈占8KB的内存区。实际上，进程的PCB所占的内存是由内核动态分配的，更确切地说，内核根本不给PCB分配内存，而仅仅给内核栈分配8K的内存，并把其中的一部分让给PCB使用。

随着Linux版本的变化，进程控制块的内容越来越多，所需空间越来越大，这样就使得留给内核堆栈的空间变小，因此把部分进程控制块的内容移出这个空间，只保留访问频繁的thread_info。

进程控制块—如何存放

把PCB与内核栈放在有什么好处：
(1) 内核可以方便而快速地找到PCB，只要知道栈指针，就可以找到PCB的起始地址，用伪代码描述如下：

```
p = (struct task_struct *) STACK_POINTER &0xffffe000
```

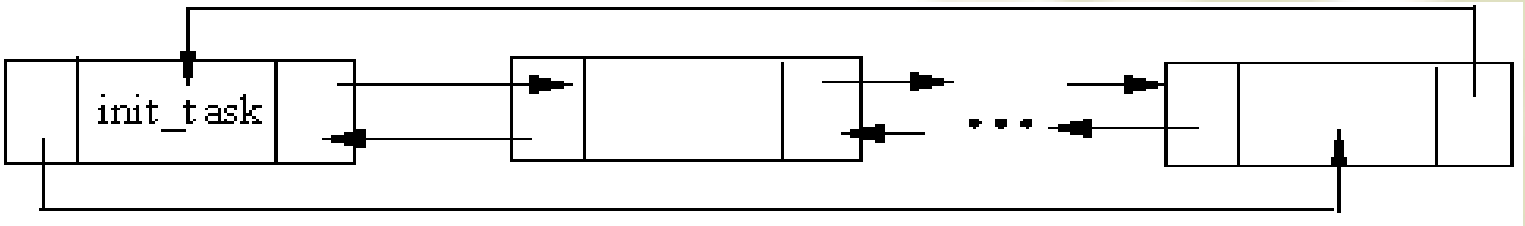
(2) 避免在创建进程时动态分配额外的内存

在Linux中，为了表示当前正在运行的进程，定义了一个current宏，可以把它看作全局变量来用，例如current->pid返回正在执行的进程的标识符

进程的组织方式-进程链表

在task_struct中定义如下：

```
struct task_struct {
        ...
  struct list_head tasks;
  char comm[TASK_COMM_LEN];/*可执行程序的名字
        ...
};
```



进程的组织方式-进程链表

链表的头和尾都为init_task，这是0号进程的PCB（进入源代码，查看其具体各个字段的值），0号这个进程永远不会被撤消，它的PCB被静态地分配到内核数据段中，也就是说init_task的PCB是预先由编译器分配的，在运行的过程中保持不变，而其它PCB是在运行的过程中，由系统根据当前的内存状况随机分配的，撤消时再归还给系统。

动手实践-打印进程控制块中的字段

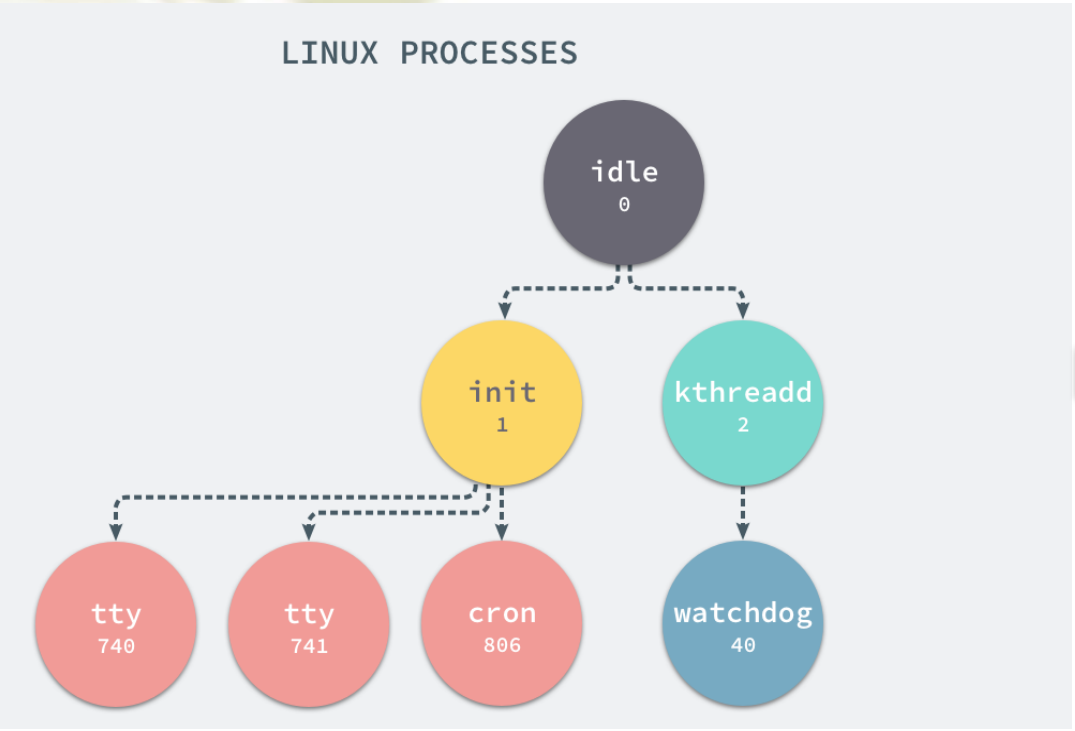
下面，我们自己编写一个内核模块，打印系统中所有进程的PID和进程名，模块中的代码如下：

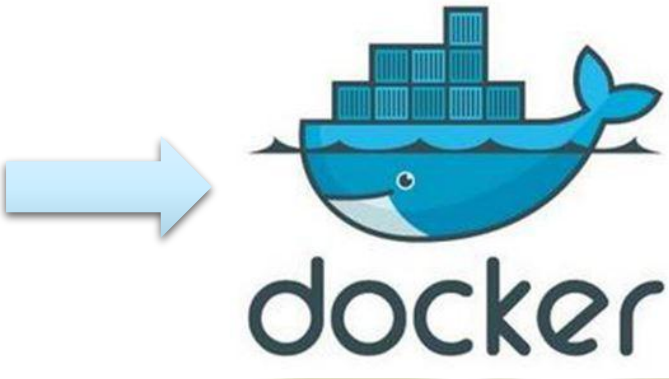
这个例子仅仅是打印出PCB中的2个字段，其实，你可以举一反三，打印出更多字段内容，来观察进程执行过程中其相关字段的具体值

动手实践-打印进程控制块中的字段

```
static int print_pid( void)
{
    struct task_struct *task,*p;
     struct list_head *pos;
     int count=0;
     printk("Hello World enter begin:\n");
     task=&init_task;
     list_for_each(pos,&task->tasks)   /*关键*/
            {
            p=list_entry(pos, struct task_struct, tasks);
            count++;
            printk("%d--->%s\n",p->pid,p->comm);
            }
     printk(the number of process is:%d\n",count);
     return 0;
}
```

从进程到容器


LINUX PROCESSES
idle
0
init
1
kthreadd
2
tty
740
tty
741
cron
806
watchdog
40


docker

从进程到容器

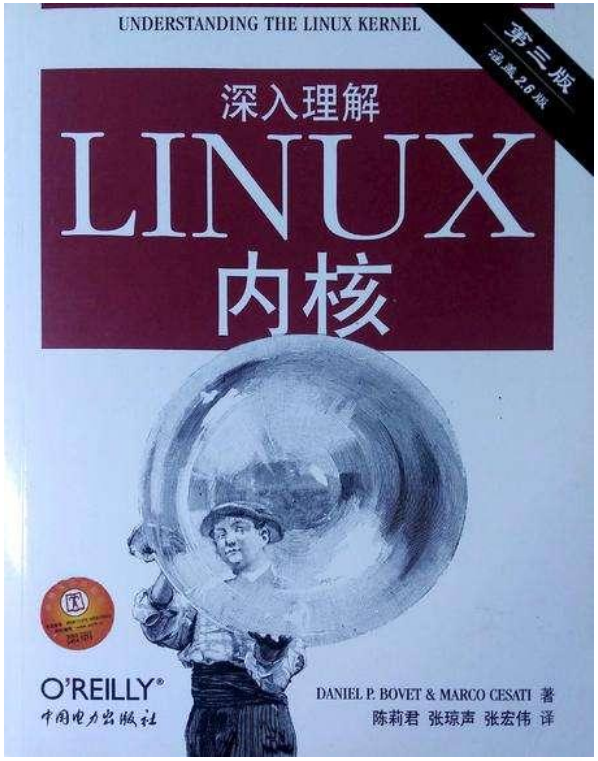
容器作为目前云技术的核心技术，与进程到底有多大关系？

对于进程来说，它的静态表现就是程序，平常都安安静静地待在磁盘上；而一旦运行起来，它就变成了计算机里的数据和状态的总和，这就是它的动态表现。

而容器技术的核心功能，就是通过约束和修改进程的动态表现，从而为其创造出一个“边界”。

对于Docker等大多数Linux容器来说，Cgroups技术是用来制造约束的主要手段，而Namespace技术则是用来修改进程视图的主要方法。在此我们抛砖引玉，感兴趣的同学可以进一步探讨下去。

参考资料



深入理解Linux内核 第三版第三章

Linux内核设计与实现第三版第三章

谢谢大家！

THANK YOU