北京郵電大學
Beijing University of Posts and Telecommunications

操作系统内核
-基于Linux

第4讲
进程管理

主讲：杨文川

内容

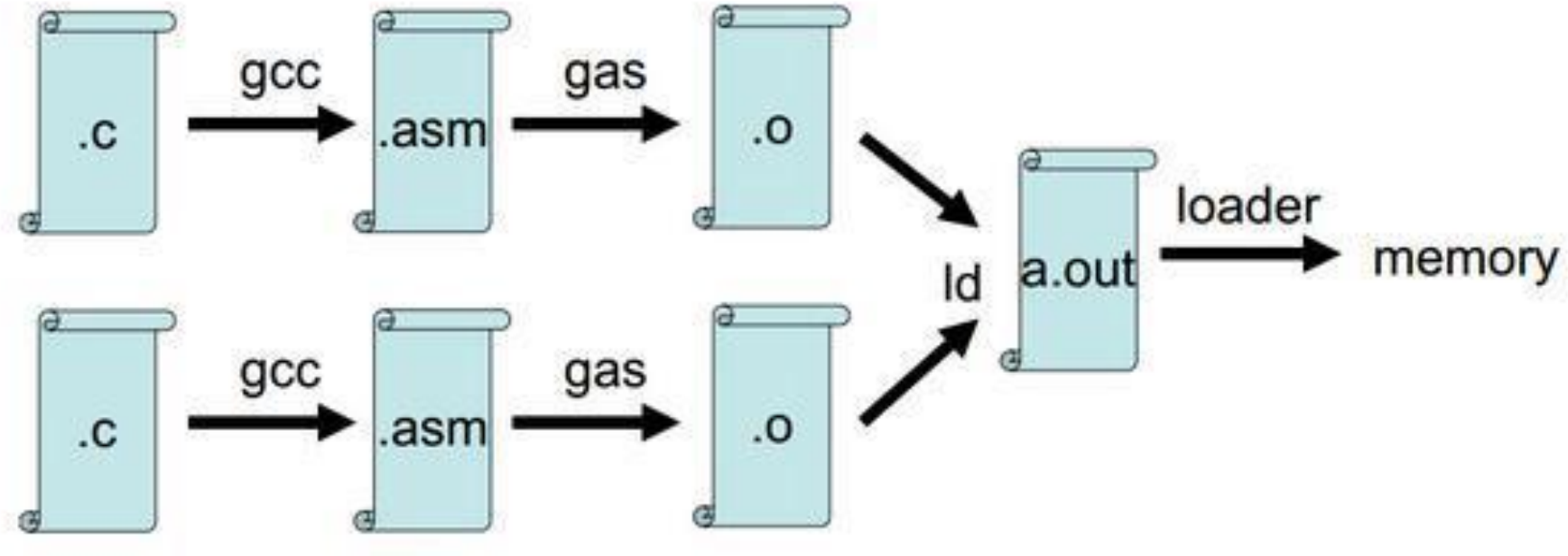
- 1 进程概述
- 2 Linux进程创建
- 3 Linux进程调度
- 4 实践-打印进程描述符task_struct中的字段
- 5 实践-基于内核模块的负载监控

内容导航：

1 进程概述

从程序到进程

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



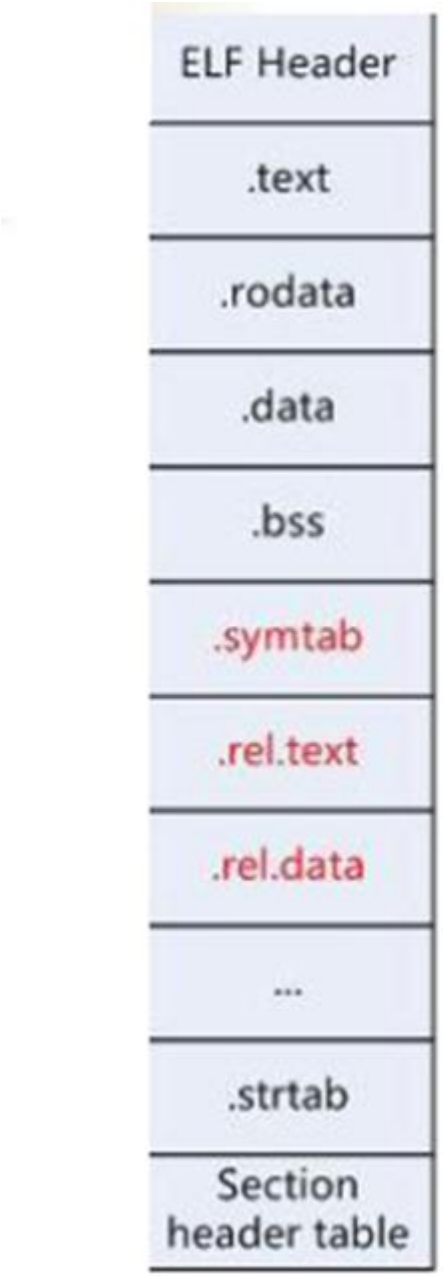
从程序到进程

- 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。

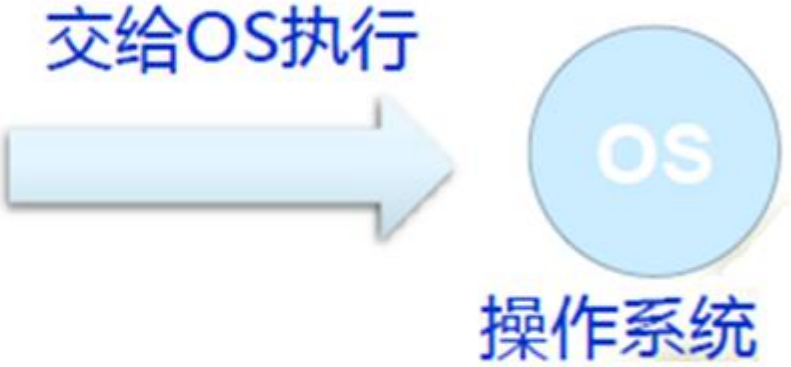
- 那么，操作系统如何应对千变万化的程序？

从程序到进程

- 一旦 程序 执行，程序就变成进程。
- 图中ELF(Executable and Linkable Format)是Linux的主要可执行文件格式。
- 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，
- 在内存管理那部分，我们将继续详细介绍进程的执行和加载。



ELF可执行文件格式



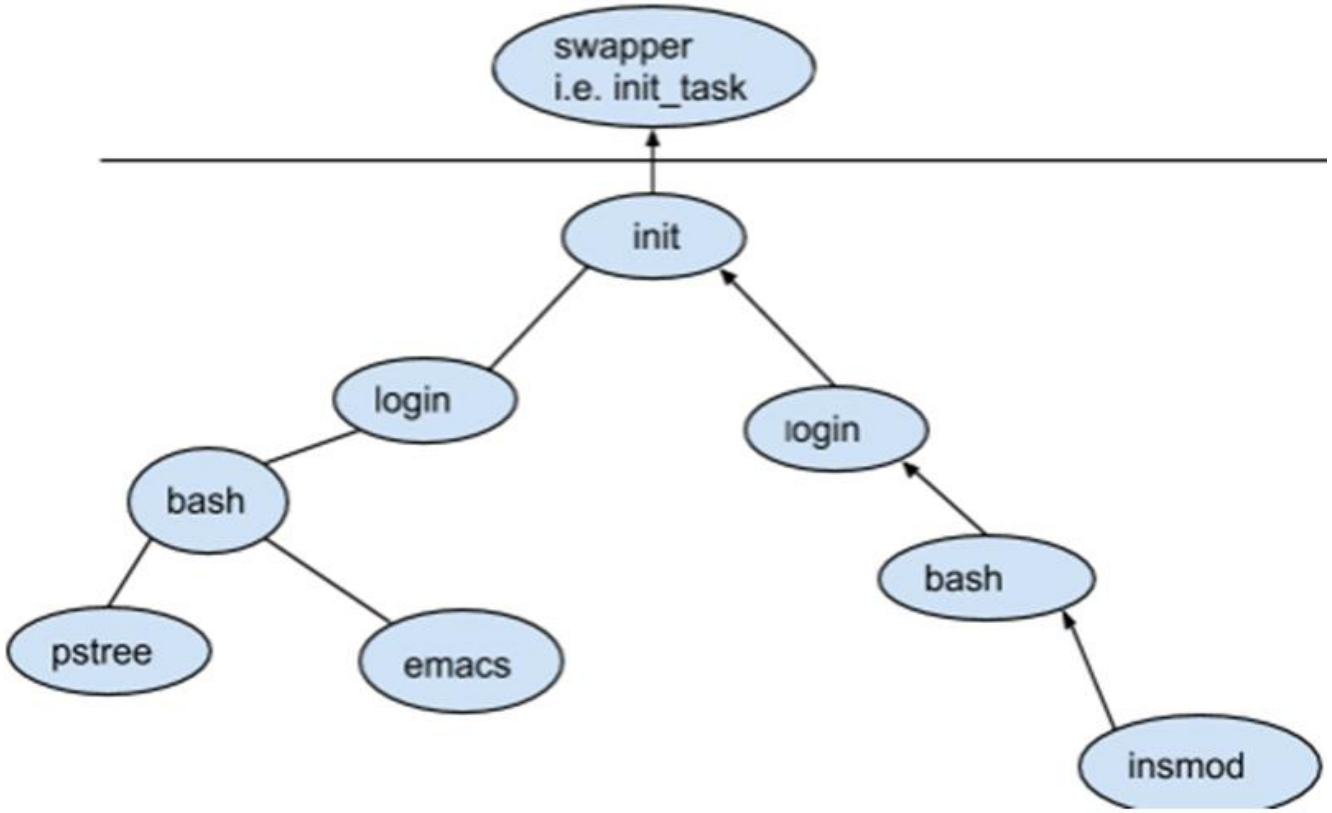
从程序到进程

- 可以通过top命令，感知系统中各个进程以及动态变化，如图：

```
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。
- 操作系统对进程的描述，模仿了人类的活动，一个进程总会有自己的父母。
- 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。
- 如图所示，这是Linux系统启动以后形成的一棵树，可以通过命令:
- ps -ejH
- 查看自己机子上的进程树。



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{
  - …
  - …
  - } ;

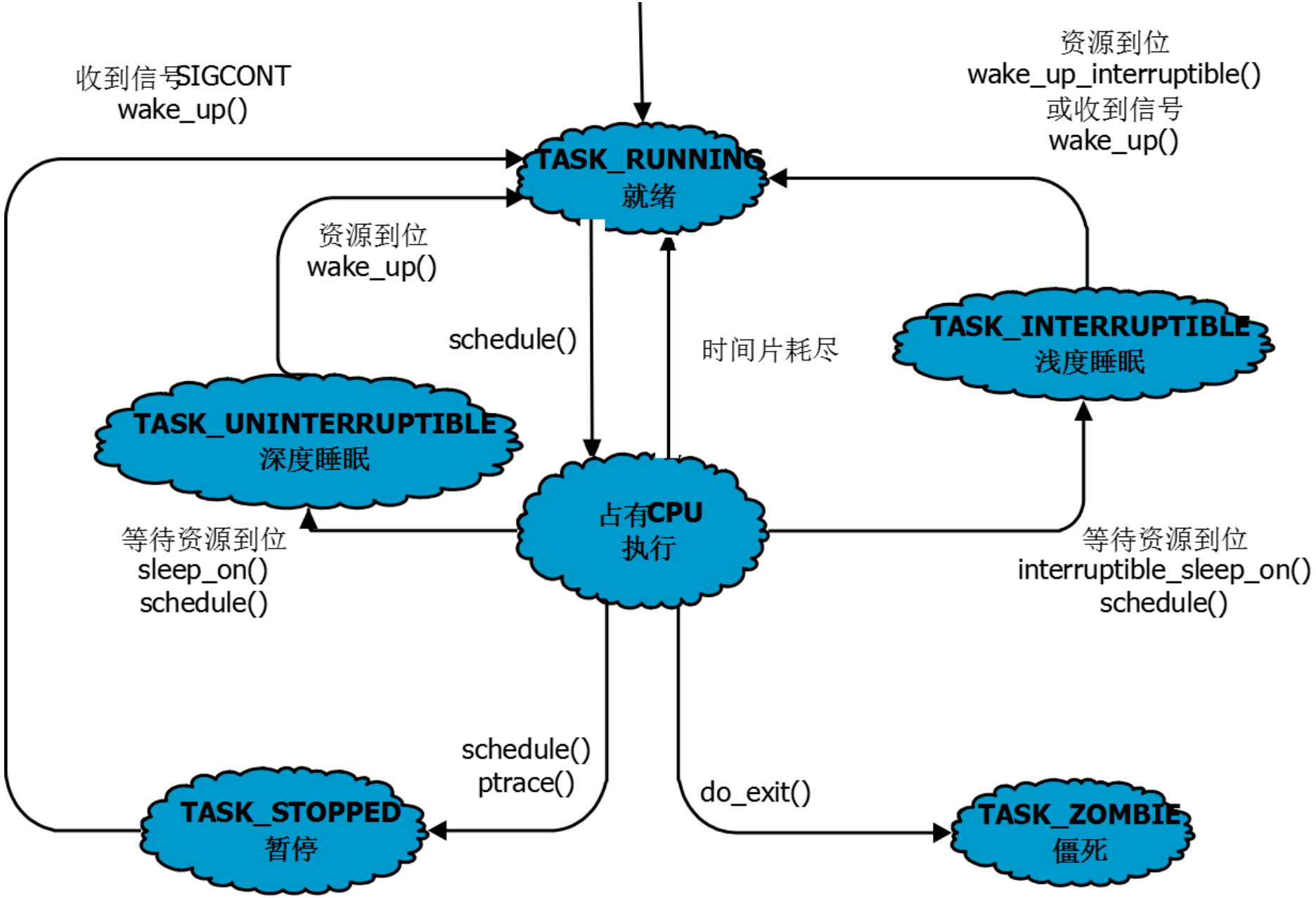
- 有些书上，这样的数据结构被叫做 **进程控制块** PCB (process control block)
- 在内核源代码中具体定义在include/linux/sched.h文件中

进程控制块 - 信息分类

- 因为进程控制块中的信息非常多，多达几百个字段，为了有助于对其认识，这里对其进行了分类，
- 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)
- 具体要通过查看源码对它们逐渐认识。

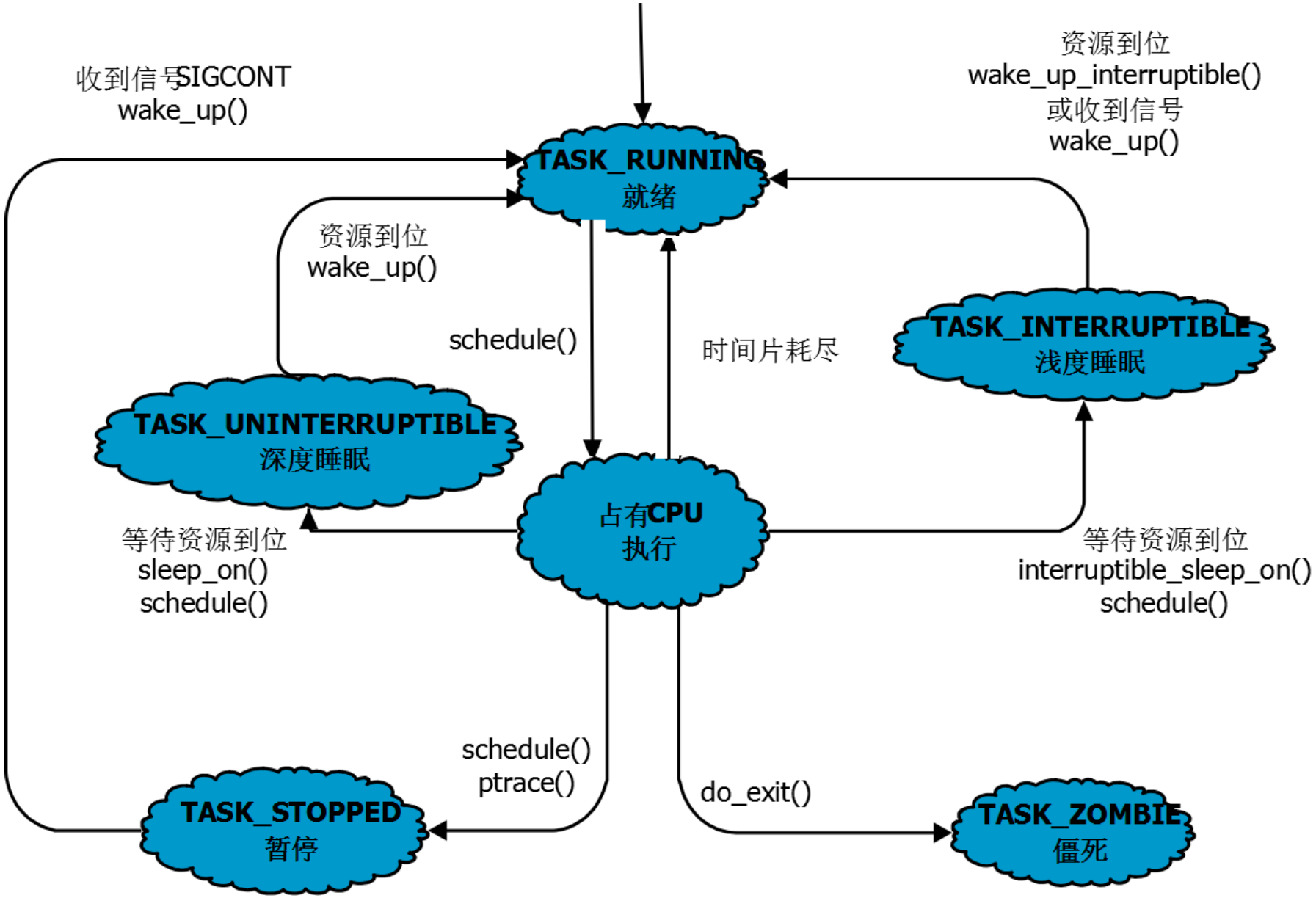
进程控制块 - Linux进程状态及转换

- 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，
- 右图给出Linux中进程5种状态



进程控制块 - Linux进程状态及转换

- 可以把 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 - 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，在include/linux/sched.h
- 注意每个状态的值定义为2的n次方

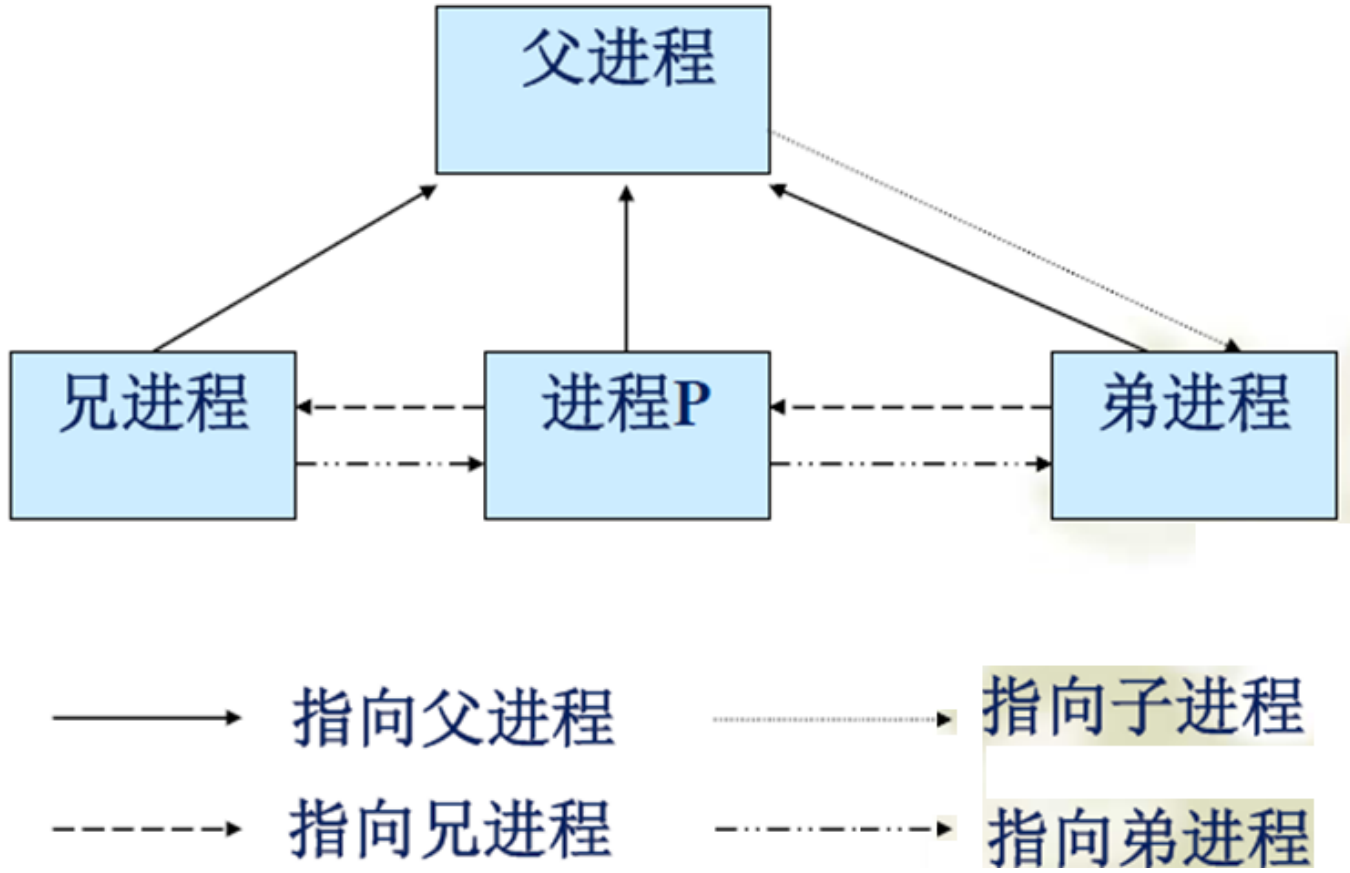
进程控制块 - 进程状态

```
/*
 * 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_ZOMBIE             16
#define EXIT_DEAD               32
/* in tsk- >state again */
#define TASK_DEAD               64

#define __set_task_state(tsk, state_value)              \
        do { (tsk)- >state = (state_value); } while (0)
#define set_task_state(tsk, state_value)                \
        set_mb((tsk)- >state, (state_value))
```

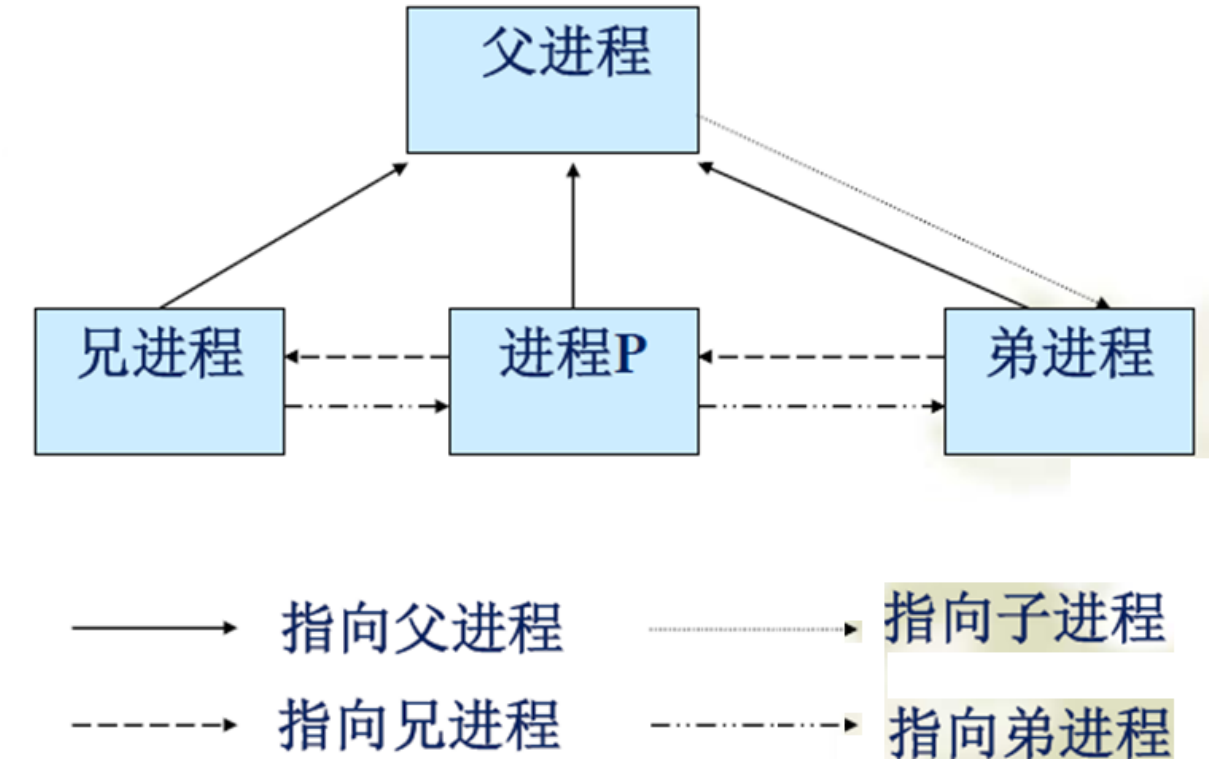
- 大家务必查看源代码，
- 可以看到不同版本其状态个数少有差异

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。
- 其中，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。
- 如图所示。



- 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。



进程控制块 - 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*/
  - **…**
- **}**；

在include/linux/sched.h

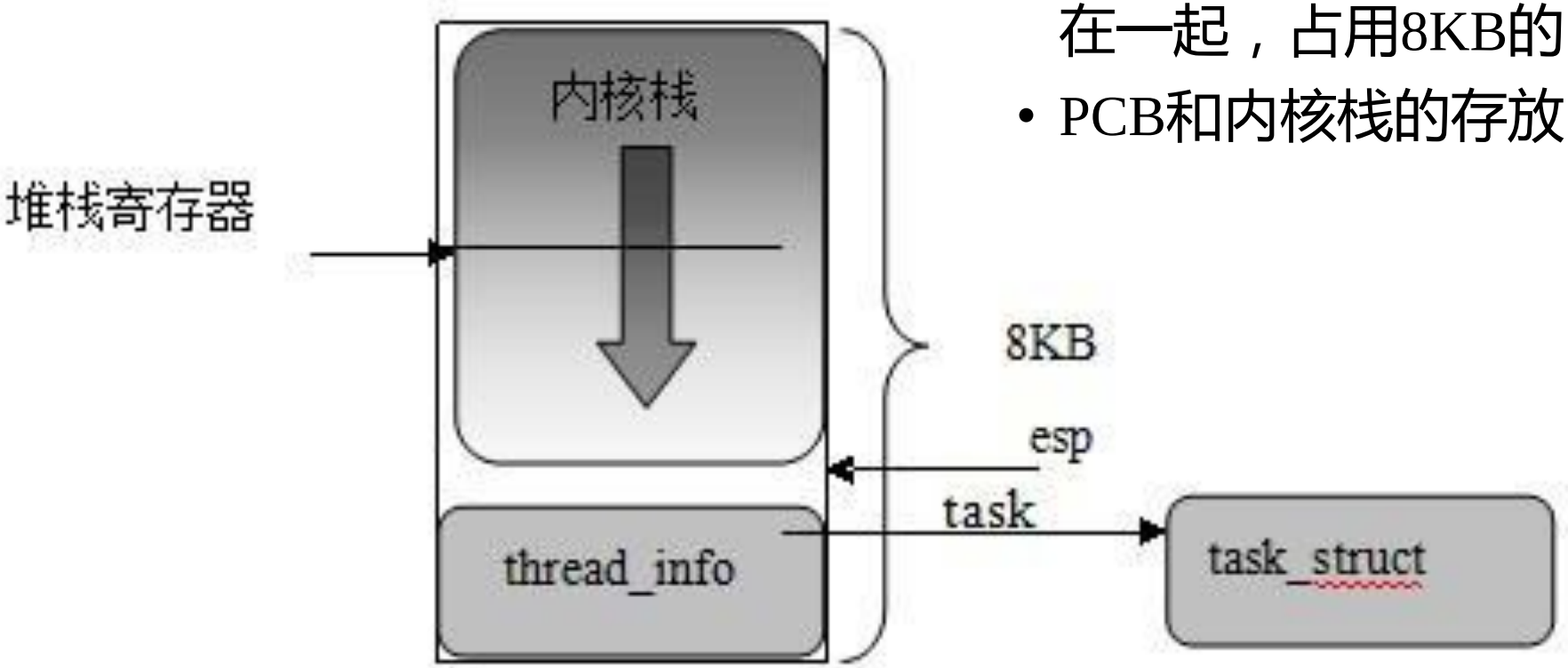
进程控制块 - 部分内容的描述

- 可以进入源代码查看，task_struct位于include/linux/sched.h中，这里是部 分代码片段：

```
struct task_struct {
        volatile long state;            / * - 1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned int flags;         / * per process flags, defined below */
        unsigned int ptrace;

#ifdef CONFIG_SMP
        struct llist_node wake_entry;
        int on_cpu;
        struct task_struct *last_wakee;
        unsigned long wakee_flips;
        unsigned long wakee_flip_decay_ts;

        int wake_cpu;
#endif
        int on_rq;

        int prio, static_prio, normal_prio;
        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

- 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的数据结构，thread_info放在一起，占用8KB的内存区，
- PCB和内核栈的存放，如下图所示：



进程控制块 - 如何存放

- 内核中用如下的联合结构，表示这个混合结构，在include/linux/sched.h:
  - 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*/
  - };

  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
- thread_info定义如下， x86上在arch/x86/include/asm/thread_info.h
  - struct thread_info {
  - struct task_struct *task;
  - struct exec_domain *exec_domain;
  - ……
  - };

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。

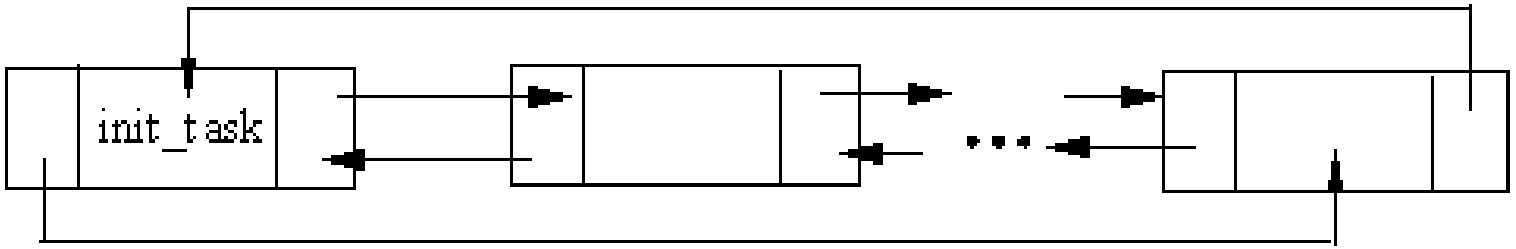
进程控制块 - 如何存放

- 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**
    - …
  - **}**;



- 如图所示，链表的头和尾都为init_task，这是0号进程(idle进程) 的PCB，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，模块中的代码如后页所示：
- 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
- 代码在samplec/34taskstruct/task_struct.c中

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;
- }

内容导航：

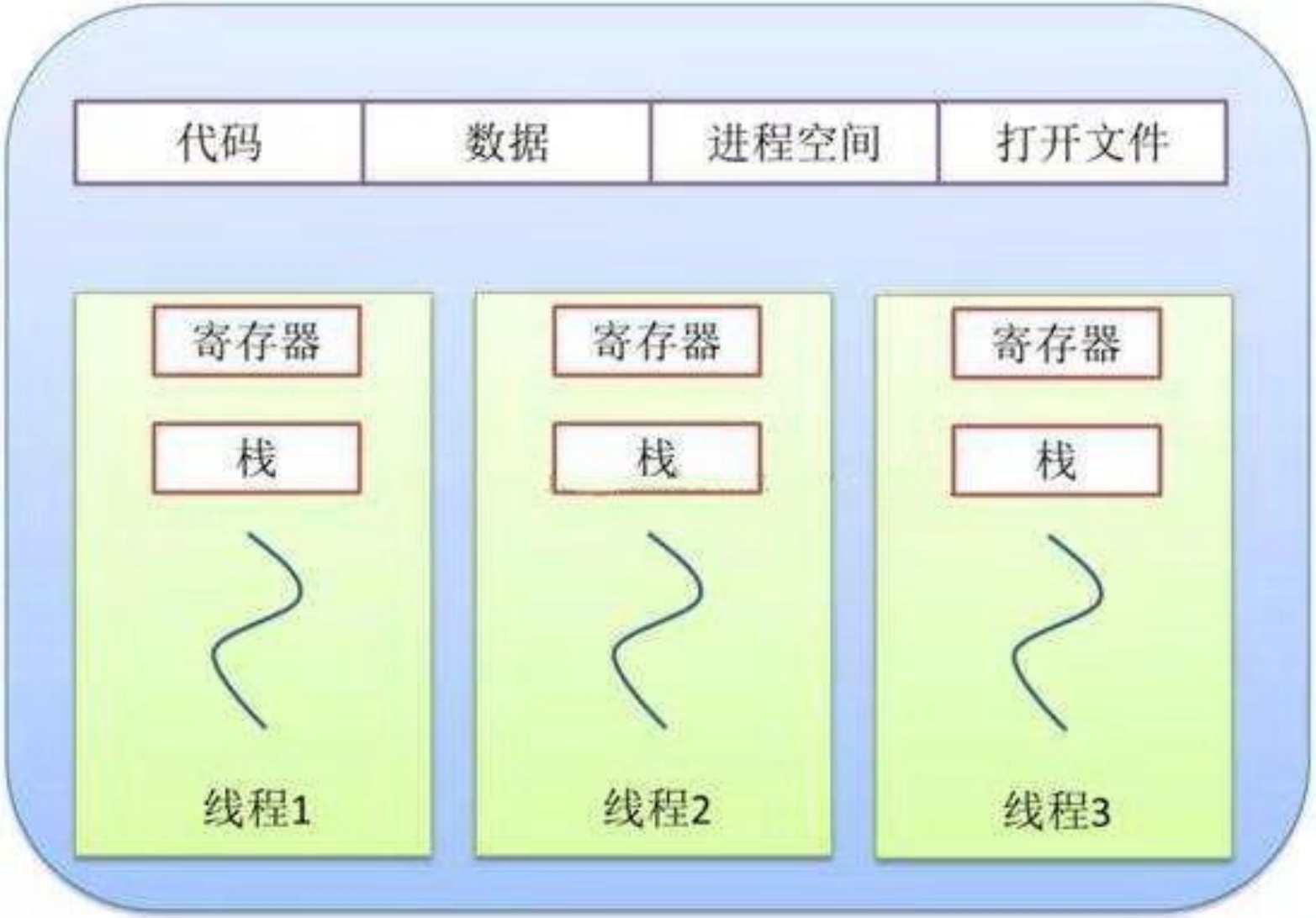
2 进程创建

进程和线程

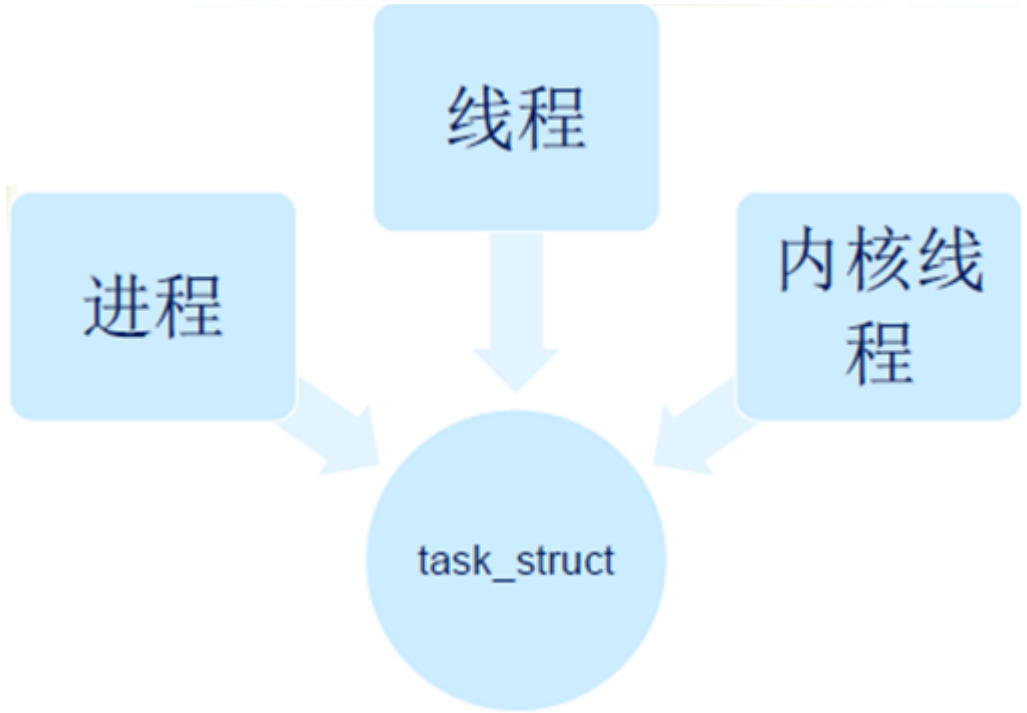
- 目前在用户态程序开发中，不仅仅涉及到进程，还涉及到线程和协程(Coroutine，又称微线程)，
- 他们到底是如何创建的，为什么创建了一个进程或者线程后，觉得自己对其没有控制权，
- 这是因为创建这件事完全由操作系统操控，你只是发出一个创建的请求，之后的事就交给操作系统了，
- 如果你对这个过程不了解，那么，一旦程序在运行的过程中出问题，你就可能就束手无策。
- 我们知道，进程是系统资源分配的基本单位，线程是独立运行的基本单位。
- 但进程的资源到底有哪些，线程为什么是轻量级的运行单位，如何体现呢？下面将对这个过程给予简要介绍。

进程和线程

- 如图所示，可以看出，进程和线程，几乎共享所有的资源，包括代码，数据，进程空间，打开的文件等，
- 线程只拥有自己的寄存器和栈。
- 这些概念上的代码段，数据段，进程空间，打开的文件，在内核中是如何表示的？



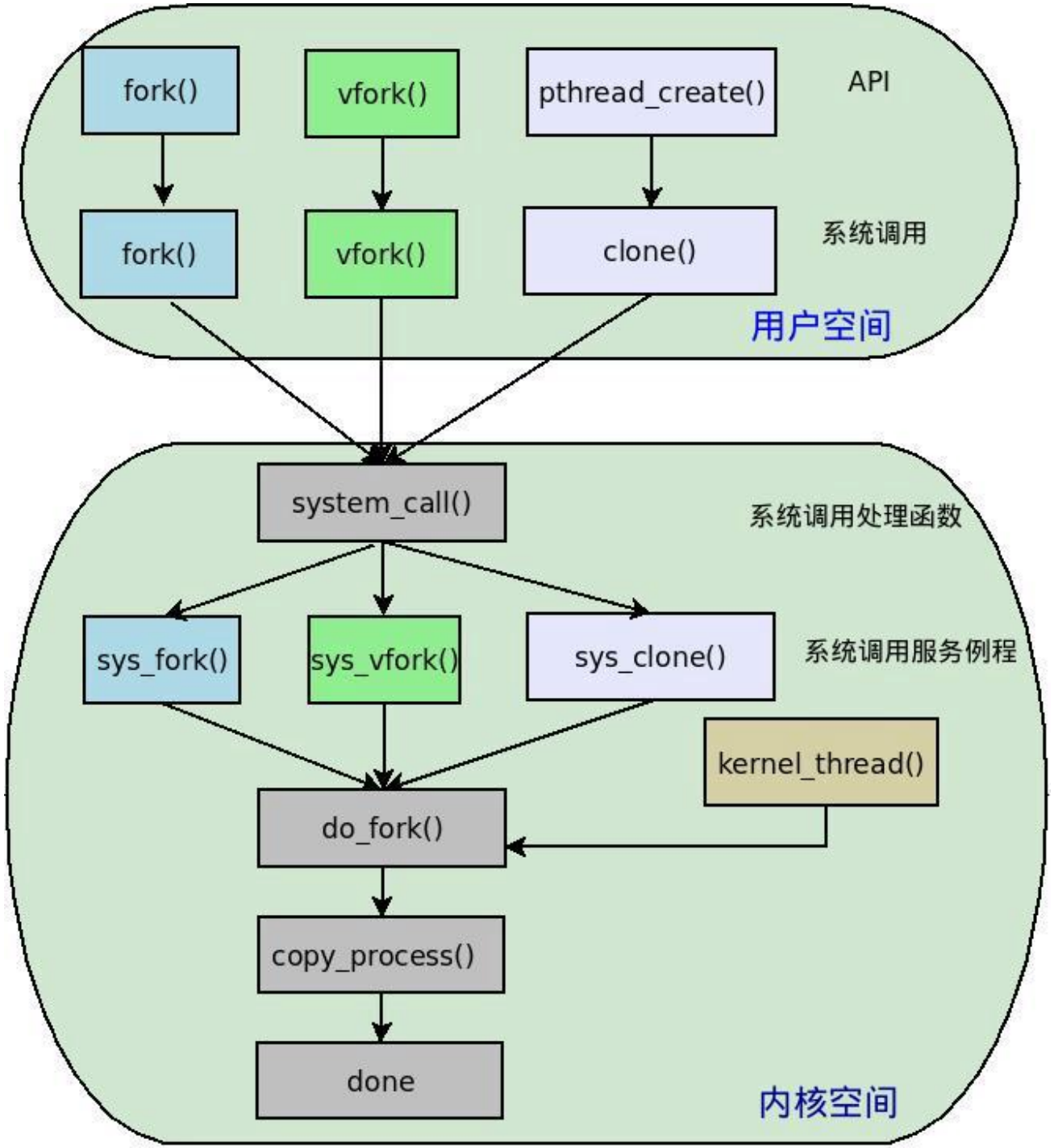
task_struct结构的统一性与多样性



- 内核如何对待进程，线程和内核线程呢？
- Linux内核坚持平等的原则，对它们一视同仁，
- 内核使用唯一的数据结task_struct来分别表示他们；
- 也使用相同的调度算法，对这三者进行调度；
- 尽管表面看起它们很不一样，但是在内核中最终都通过do_fork()分别创建
- 这样处理对内核来说简单方便，在统一的基础上又保持各自的特性
- 那么，这是如何做到的呢?

进程的API实现

- 我们站在用户态函数库角度看，创建 **进程** 和 **线程** 调用了不同的函数，分别为 fork() 和 pthread_create()，
- 而对应的系统调用为 fork() 和 clone()，vfork与fork类似，后面会讲到二者的差异。
- 所有的系统调用进入内核只有一个入口，进去以后，又各自进入自己的服务例程，
- 但最终会归到一处，do_fork就成为它们的聚合点。
- 如图所示。



do_fork()

- do_fork()在内核中是怎样的原型，在kernel/fork.c
  - long do_fork(unsigned long clone_flags,
  - unsigned long stack_start,
  - unsigned long stack_size,
  - int __user *parent_tidptr,
  - int __user *child_tidptr)
- 这一堆参数，非常丰富，难怪在用户态调用fork()时，轻松自由，一个参数也不需要给，但进入内核后这么麻烦

三个系统调用如何调用do_fork

参见include/linux/syscall.h

```
int sys_fork(struct pt_regs *regs)
{
        return do_fork(SIGCHLD, regs->sp, regs, 0, NULL, NULL);
}

int sys_vfork(struct pt_regs *regs)
{
        return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs->sp, regs, 0,
                       NULL, NULL);
}

long
sys_clone(unsigned long clone_flags, unsigned long newsp,
          void __user *parent_tid, void __user *child_tid, struct pt_regs *regs)
{
        if (!newsp)
                newsp = regs->sp;
        return do_fork(clone_flags, newsp, regs, 0, parent_tid, child_tid);
}
```

fork的实现

- 先看fork调用的do_fork，除了SIGCHLD参数外，有三个参数就是空手而来，有两个参数似乎也没有明确的目标
- 但作为子进程，它完全有自己的个性的，根本不想共享父进程的任何资源，而是让父亲把他所有资源，给自己复制一份，父亲没有真的给他复制一份，只是假装复制了一下，也就是用一个指针指过去而已，
- 等真正需要的时候，比如，要写一个页面，这时**写时复制**技术就登场了，只有父子进程中不管谁想写一个页面时，这个页面才被复制一份。

```
int sys_fork(struct pt_regs *regs)
{
        return do_fork(SIGCHLD, regs->sp, regs, 0, NULL, NULL);
}
```

vfork的实现

- vfork()，直接传递了两个标志过去，
- 第一个标志(CLONE_VFORK)，儿子优先，老爸等着。于是父进程就去睡觉，等子进程结束才能醒来。
- 第二个标志(CLONE_VM)儿子干脆与父亲待在一个进程的地址空间中，就是共享父进程的内存地址空间(父进程的页表项除外)。
- vfork()目前被取代了，因为写实复制技术的招数更高，也就是更高效，因此，**它没有了生存空间。**

```
int sys_vfork(struct pt_regs *regs)
{
        return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs->sp, regs, 0,
                       NULL, NULL);
}
```

clone的实现

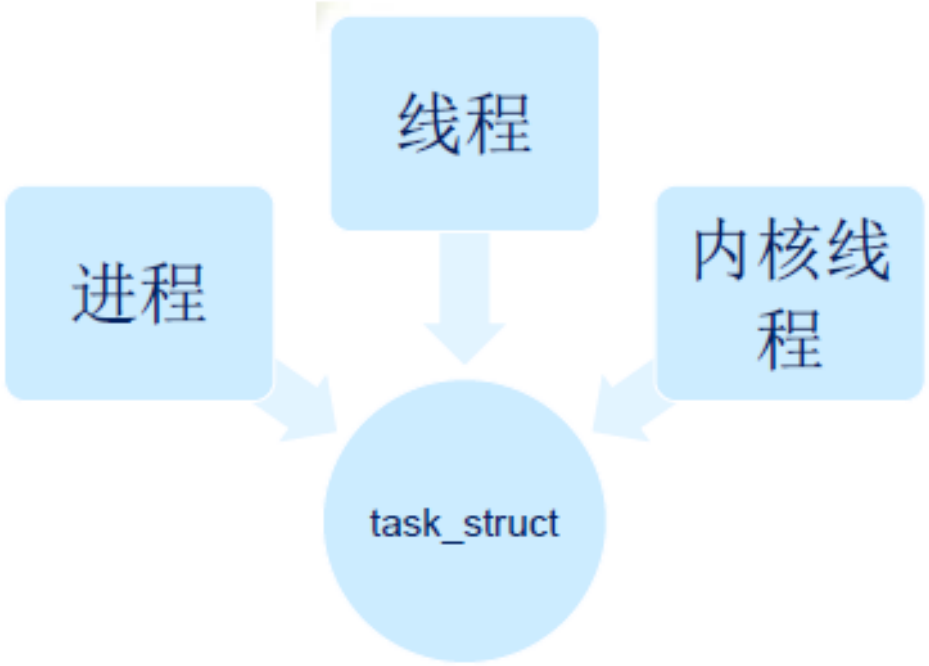
- clone()：也就是克隆技术，**线程就是这么诞生的。**
- 怎么克隆，方法上传了一堆参数，告诉老爸，你这我要共享，你那我也都要共享，
- 于是老爸的地址空间，文件系统，打开的文件，信号处理函数等，就都被儿子一句话说过来了，看过来就是这四个参数：
    - CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND

```
long
sys_clone(unsigned long clone_flags, unsigned long newsp,
          void __user *parent_tid, void __user *child_tid, struct pt_regs *regs)
{
        if (!newsp)
                newsp = regs->sp;
        return do_fork(clone_flags, newsp, regs, 0, parent_tid, child_tid);
}
```

内核线程的创建

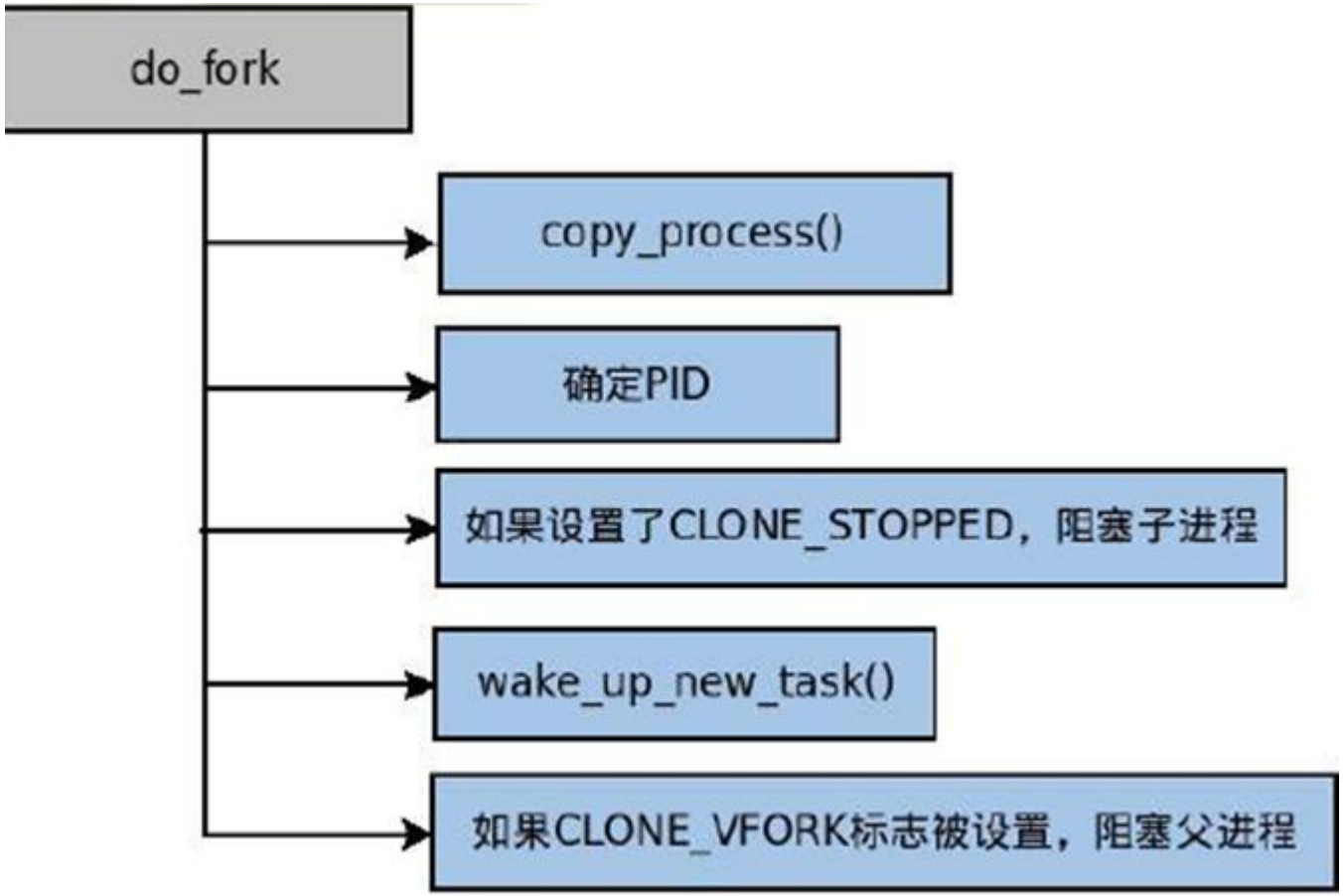
- 一个内核线程的出生更有优势，因为用户空间对它来说根本就没有意义，根本就不知道它的存在。
- 该调用哪个函数创建呢？
- 早期内核创建内核线程，是通过kernel_thread()而创建的，
- 目前内核中调用kthread_create()创建的，其本质也是向do_fork()提供特定的flags标志而创建的。

task_struct带来的统一性



- 到底是谁给进程，线程和内核线程的诞生带来了方便，这就是task_struct结构。
- 由此，它们的生命历程具有了诸多相似，不管是被调度到CPU上去跑，还是分配各种资源，到最终的诞生都是调用了相同的数do_fork()。
- 下面看看do_fork()的代码流程是什么样的

do_fork()代码流程



- 1.调用copy_process()复制父进程的进程控制块。
- 2.获得子进程的pid。
- 3.如果设置了暂停标志，则子进程的状态被设置为暂停。否则，通过唤醒函数，将子进程的状态设置为就绪，并且将子进程加入就绪队列。
- 4.如果使用vfork()创建进程，则阻塞父进程。

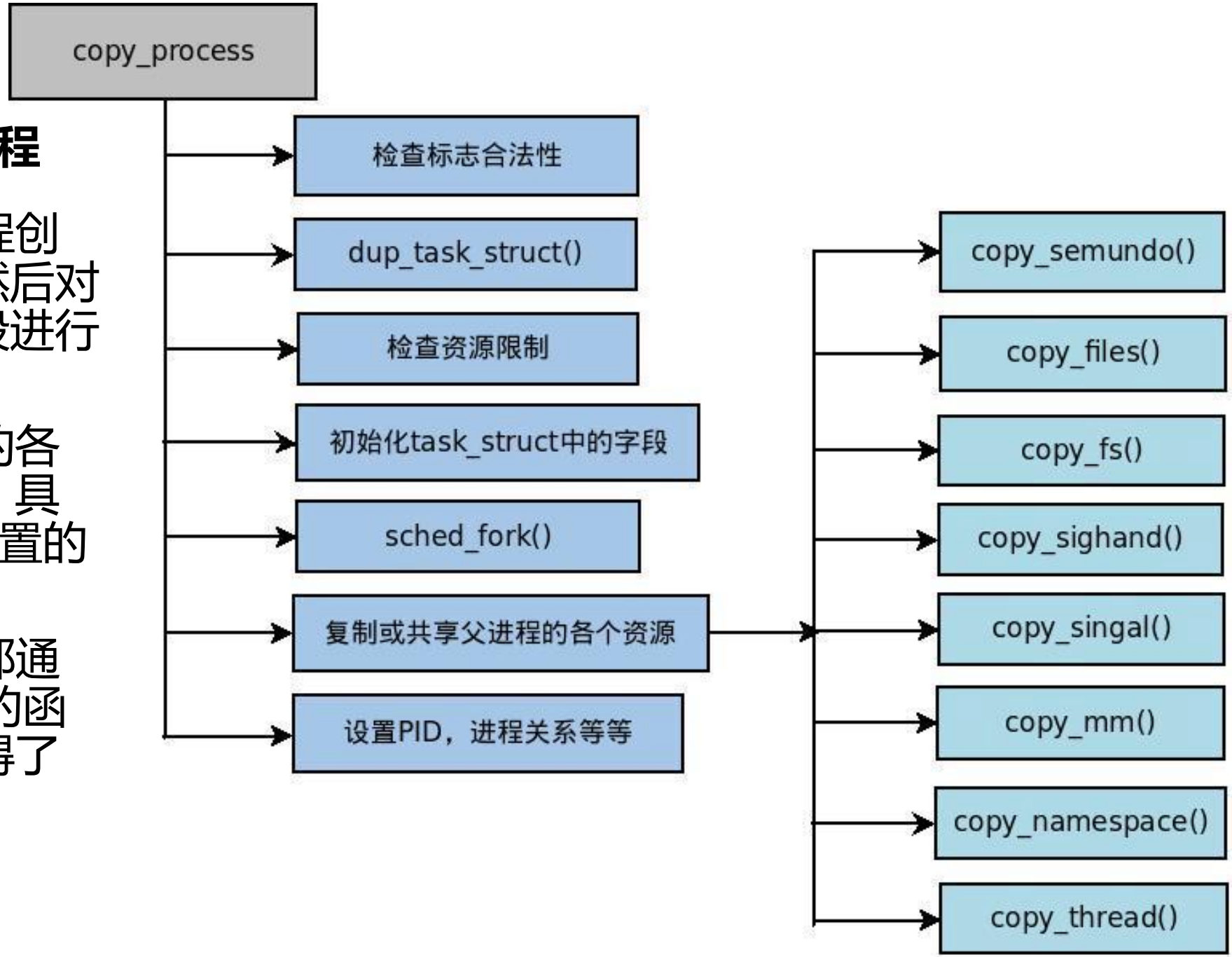
copy_process()

- copy_process()主要用于创建进程控制块，以及子进程执行时所需要的其他数据结构。
- 该函数的参数与do_fork()的参数大致相同，并添加了子进程的pid。
- copy_process()所做的处理，必须考虑到各种可能的情况，这些特殊情况即通过clone_flags来具体体现。
- 下面忽略特殊情况，给出一般的执行过程
  - 在kernel/fork.c

```
static struct task_struct *copy_process(unsigned long clone_flags,
                                        unsigned long stack_start,
                                        unsigned long stack_size,
                                        int __user *child_tidptr,
                                        struct pid *pid,
                                        int trace)
```

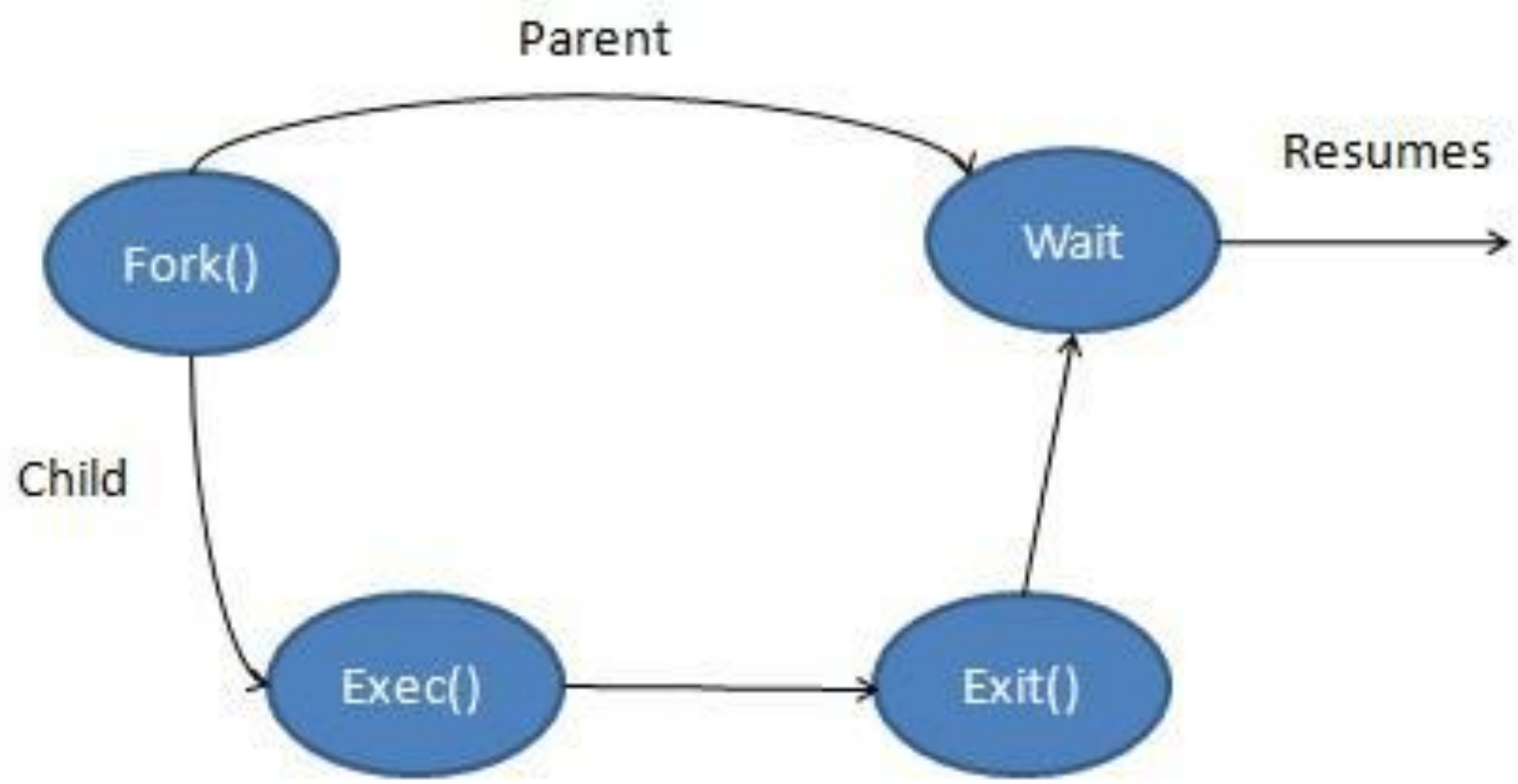
copy_process()主要实现过程

- 这个函数主要是为子进程创建父进程PCB的副本，然后对子进程PCB中的各个字段进行初始化。
- 同时，子进程对父进程的各种资源进行复制或共享，具体取决于clone_flags所设置的标志。
- 每种资源的复制或共享都通过形如copy_XYZ()这样的函数完成，当然子进程获得了新的pid。
- 如图所示。



进程的生命周期

- 这里比较详细的介绍了fork的创建过程，与之相应的还有exec,wait,exit三个系统调用，如图所示。这里就不一一详细介绍了，
-



进程的生命周期

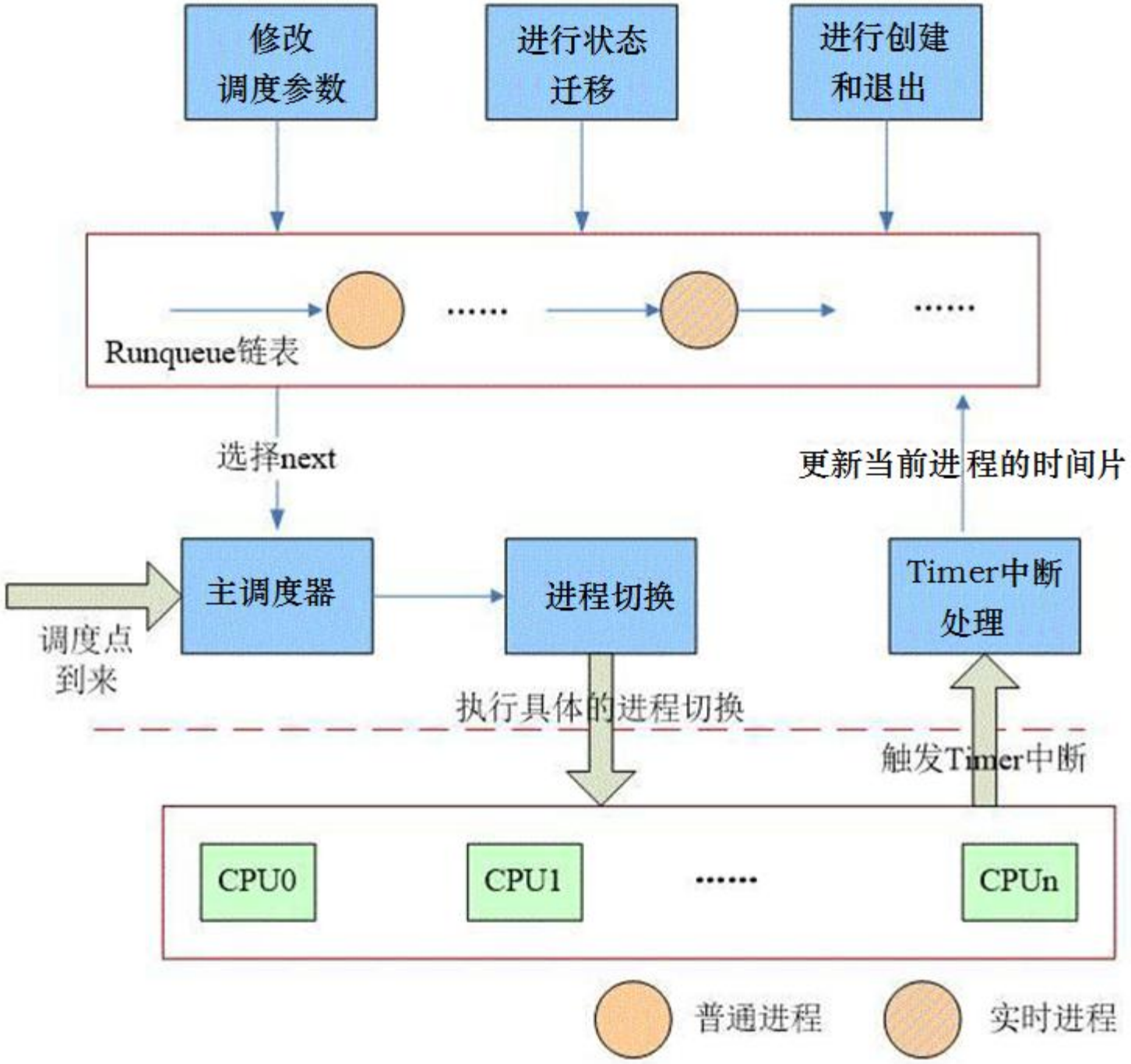
- 他们之间是如何配合的？下面让我们用一生作一个比喻，来对进程做一个总结：
- 随着fork的调用，一个新进程诞生，但这时它只是老进程的一个克隆。每当exec，新进程脱胎换骨，离家独立，开始了独立工作的职业生涯。
- 进程也有生老病死，它可以是自然死亡，即运行到主(main)函数的最后一个"}"，从容地离我们而去；也可以是中途退场，
- 退场有2种方式，一种是调用exit函数，一种是在主(main)函数内使用return，无论哪一种方式，它都可以留下留言，放在返回值里保留下来；甚至它还可能被谋杀，被其它进程通过另外一些方式结束它的生命。
- 进程死掉以后，会留下一个空壳，wait站好最后一班岗，打扫战场，使其最终归于无形。
- 这就是进程完整的一生。

内容导航：

3 Linux进程调度

调度基本模型

- 调度就是从就绪队列中，选择一个进程投入CPU运行。
- 调度的主战场是就绪队列，核心是调度算法，实质性的动作是进程切换，
- 对于以时间片调度为主的调度，时钟中断是驱动力，确保每个进程在CPU上运行一定的时间
- 在调度的过程中，用户还可以可以通过系统调用nice等调整优先级，比如降低自己的有优先级等，当然也涉及进程状态的转换。
- 新创建的进程加入就绪队列，退出的进程从队列中删除。如图所示。



调度基本模型

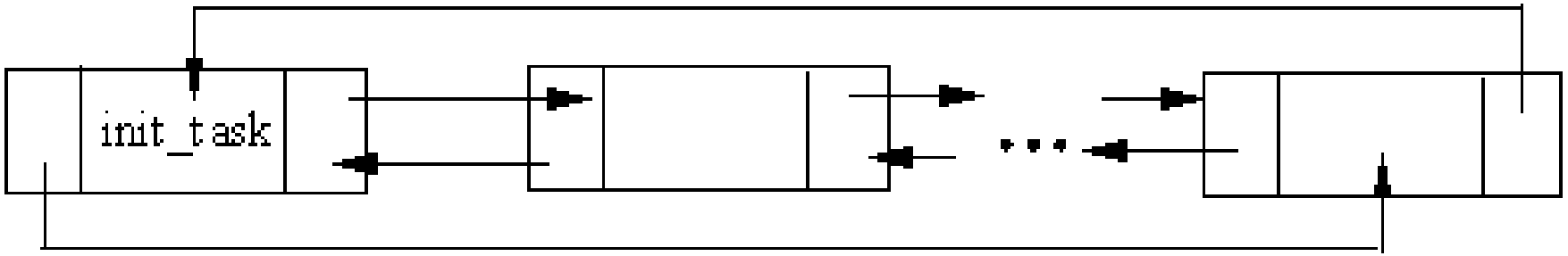
- 从前图可以看出，所有的CPU的所有的进程，都存放在一个就绪队列中，从中选中一个进程进行调度的过程，是从这队列上的一种线性查找，因此其算法复杂度为O(n)，
- 详细调度过程和代码分析请参见进程调度部分，
- 这是针对2.6版内核的代码分析，比较简单，看起来相对容易，
- 下面主要讲解进程调度的演变过程，以指导大家阅读高版本的内核源代码。

调度的主战场-就绪队列

- 在内核代码中，如何表示就绪队列？
- 把就绪状态的进程组成一个双向循环链表，也叫就绪队列(runqueue)，如2.6版本在task_struct结构中定义了队列结构。

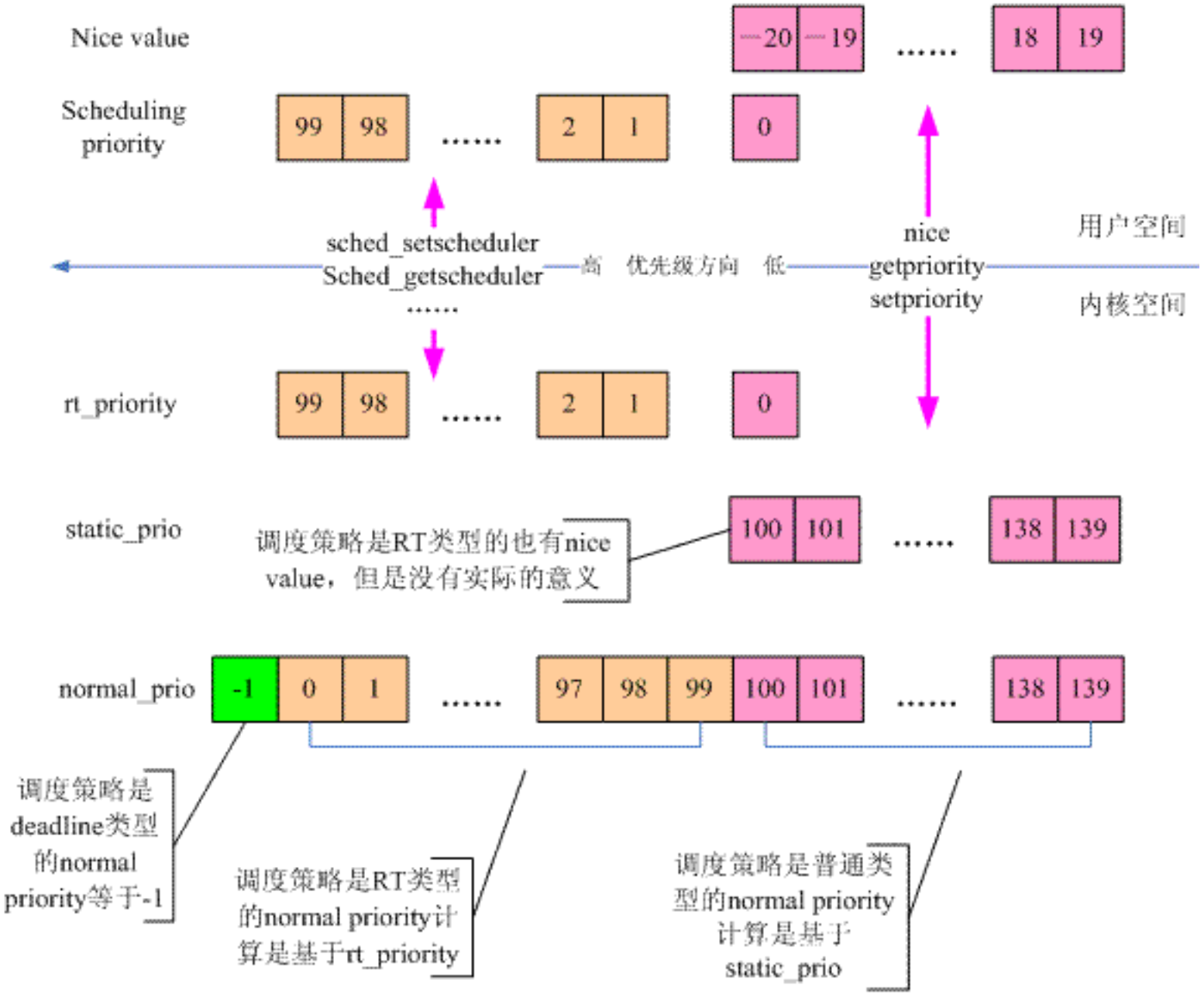
```
struct task_struct {
        volatile long state;
        void *stack;
        int prio, static_prio, normal_prio;
        struct list_head run_list;
```

- init_task(0号进程的PCB)为队头。

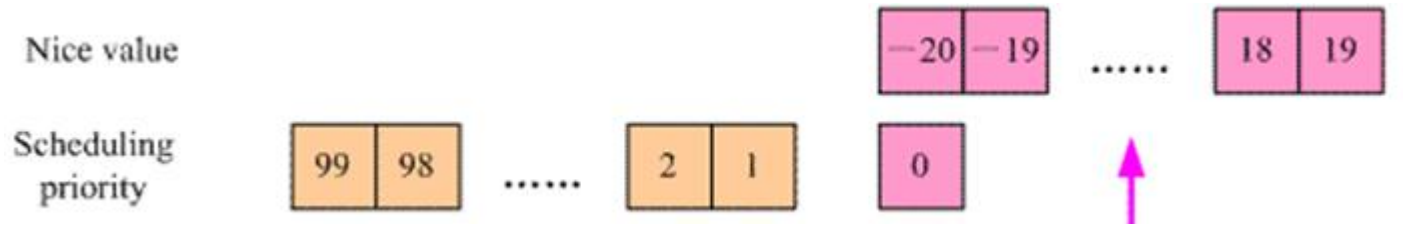


进程调度-进程优先级

- 在进程调度算法中，优先级是一个很重要的因素，可以从：
- 用户空间
- 内核空间
- 两个角度来看优先级，如图所示。
- 下面将详述

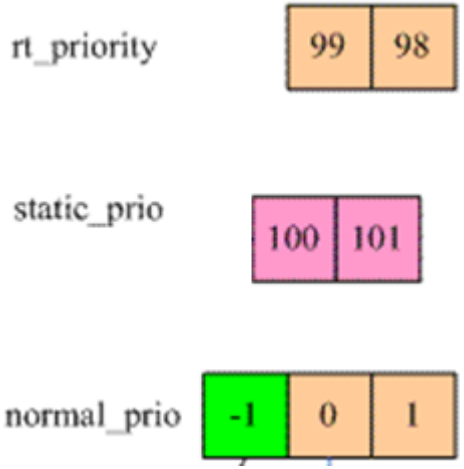


进程调度-进程的优先级



- 从用户空间，有两种优先级：
  - (1)普通优先级(nice)：从-20～19，数字越小，优先级越高，通过修改这个值，可以改变普通进程获取cpu资源的比例。
  - (2)调度优先级(scheduling priority)。从1(最低)~99(最高)，这是实时进程的优先级。当然，普通进程也有调度优先级(scheduling priority)，被设定为0。

- 从内核空间：
  - 动态优先级(prio)，静态优先级(static_prio)，归一化优先级(normal_prio)和实时优先级(rt_priority)，

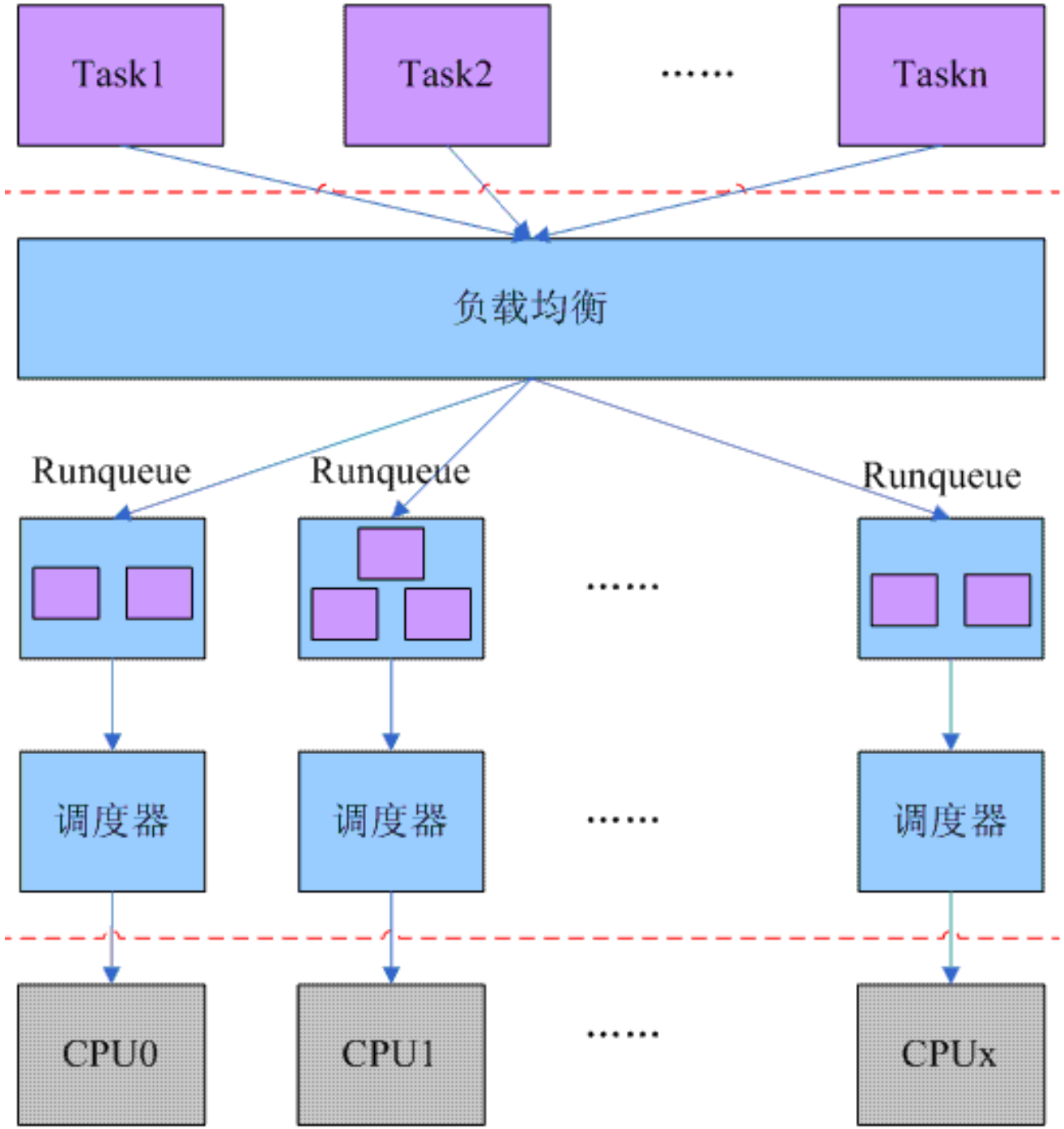


进程调度-内核空间优先级

- 这里特别介绍一下：
  - 归一化优先级(normal_prio)：它是根据静态优先级、调度优先级和调度策略来计算得到的
  - 动态优先级(prio)：运行时可动态调整
- 在task_struct结构中的表示如下，在include/linux/sched.h：
  - struct task_struct {
  - ......
  - int prio, static_prio, normal_prio;
  - unsigned int rt_priority; ......
  - unsigned int policy;
  - ......
  - }

进程调度-O(1)调度

- O(n)调度器中只有一个全局的就绪队列(Runqueue)，严重影响了扩展性，
- 为此，在O(1)调度器中引入了每CPU 一个就绪队列的概念。
- 系统中所有的就绪进程，首先经过负载均衡模块，挂入各个CPU的就绪队列上，
- 然后由主调度器和周期性调度器，驱动该CPU上的调度行为。
- 如图所示。

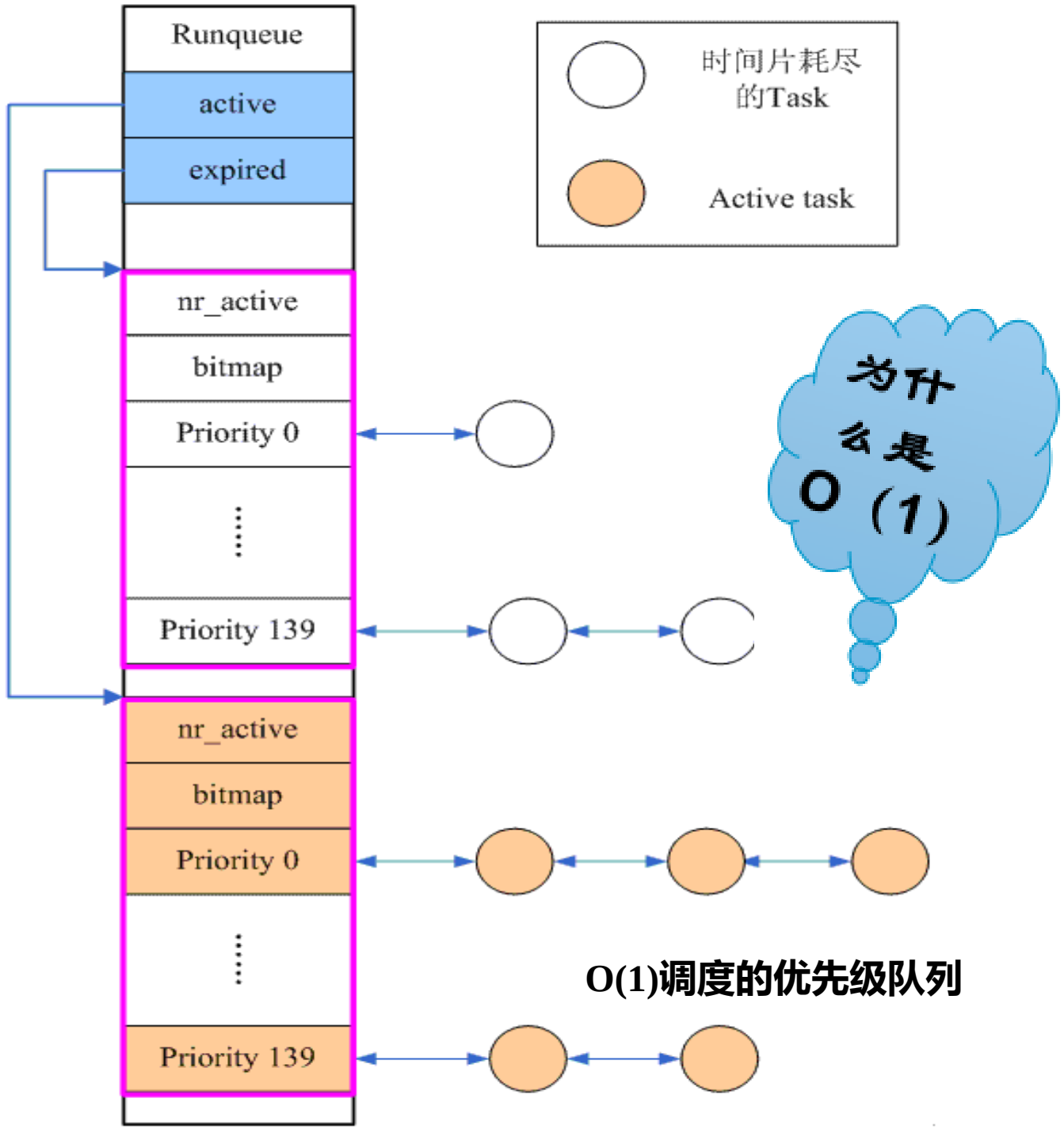


进程调度-O(1)调度

- O(1)调度器的基本优化思路，就是把原来就绪队列上的单链表，变成多个链表，即每一个优先级的进程，被挂入不同链表中。
- Linux2.4之前，早期的优先级数组结构表示如下：
  - struct prio_array {
  - unsigned int nr_active;
  - unsigned long bitmap[BITMAP_SIZE];
  - struct list_head queue[MAX_PRIO];
  - };

进程调度-O(1)调度

- O(1)由于支持140个优先级，因此队列成员中有140个分别表示各个优先级的链表头，不同优先级的进程挂入不同的链表中。
- bitmap 是表示各个优先级进程链表是空还是非空。nr_active表示这个队列中有多少个任务。
- 在这些队列中，100～139是普通进程的优先级，其他的是实时进程的优先级。
- 因此，在O(1)调度器中，实时进程和普通进程被区分开了，普通进程根本不会影响实时进程的调度。



进程调度-O(1)调度

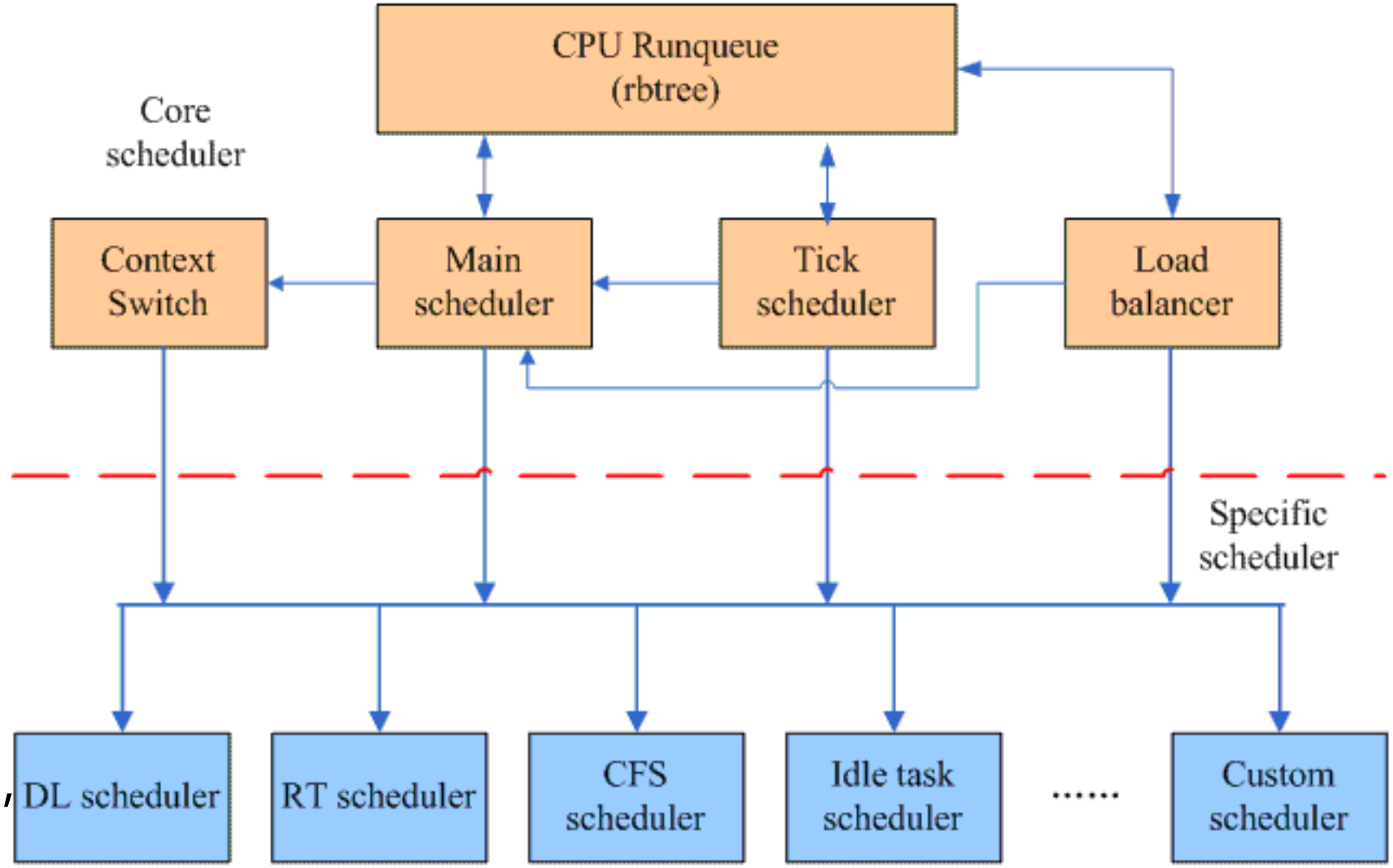
- 就绪队列中有两个优先级队列：(struct prio_array) 分别用来管理活跃队列active(也就是时间片还有剩余)和 expired(时间片耗尽)的进程。
- 随着系统的运行，活跃 (active)队列的任务一个个的耗尽其时间片，挂入到时间 片耗尽(expired)的队列。
- 当活跃(active)队列的任务为 空的时候，两个队列互换，开始一轮新的调度过程。
- 虽然在O(1)调度器中任务组织的形式发生了变化，但是 其核心思想仍然和O(n)调度器一致的，都是把CPU资源分成 一个个的时间片，分配给每一个就绪的进程。
- 进程用完其额 度后被抢占，等待下一个调度周期的到来

进程调度-O(1)调度

- 主调度器(就是schedule函数)的主要功能是从该CPU的 就绪队列中找到一个最合适的进程调度执行。
- 其基本的思路 就是从活跃(active)优先级队列中寻找。
- 首先在当前活跃 active队列的位图bitmap中寻找第一个非空的进程链表，然后从该链表中找到的第一个节点就是最适合下一个调度执行 的进程。
- 由于没有遍历整个链表的操作，因此这个调度器的 算法复杂度是一个常量，从而解决了O(n)算法复杂度的问 题。
- 但是，O(1)调度器使用非常复杂的算法，来判断进程是否是交互式进程，以及进程的用户交互指数，
- 即使这样，还会出现出现卡顿现象，如何解决？ 能否不要被用户的具体需求捆绑，而又能支持灵活多变的需求。

调度模型-机制与策略分离

- 可以采用机制与策略分离机制，来解决此问题。
- 这种机制功能层面上看，进程调度仍然分成两个部分，第一个部分是通过负载均衡模块，将各个就绪状态的任务，根据负载情况，平均分配到各个CPU就绪队列上去。
- 第二部分的功能是在各个CPU的主调度器(Main scheduler)和(周期性调度器)Tick scheduler的驱动下，进行单个CPU上的调度。



机制与策略分离-调度器类

- 调度器处理的任务各不相同，有实时任务(RT task)，普通任务(normal task)，最后期限任务(Deadline task)，但是无论哪一种任务，它们都有共同的)逻辑，这部分被抽象成核心调度器层(Core scheduler layer)
- 同时各种特定类型的调度器定义自己的调度类(sched_class)，并以链表的形式加入到系统中。
- 这样的机制与策略分离的设计，可以方便用户根据自己的场景定义特定的调度器，而不需要改动核心调度器层的逻辑。
- 先简单的介绍下struct sched_class部分成员作用。

机制与策略分离-调度器类

- 第一个：next：next指向下一个比自己低一个优先级调 度类。
- 第二个字段：enqueue_task：指向入队的函数。
- 第三个字段：dequeue_task：指向出队的函数。
- 第四个：check_preempt_curr：当前cpu上正在运行的进程是否可被强占。
- 第五个：pick_next_task：从就绪队列中选择一个最适合运行的。这也算是调度器比较核心的一个操作。
- 例如，我们依据什么挑选最适合运行的进程呢？这就是每一个调度器 需要关注的问题。

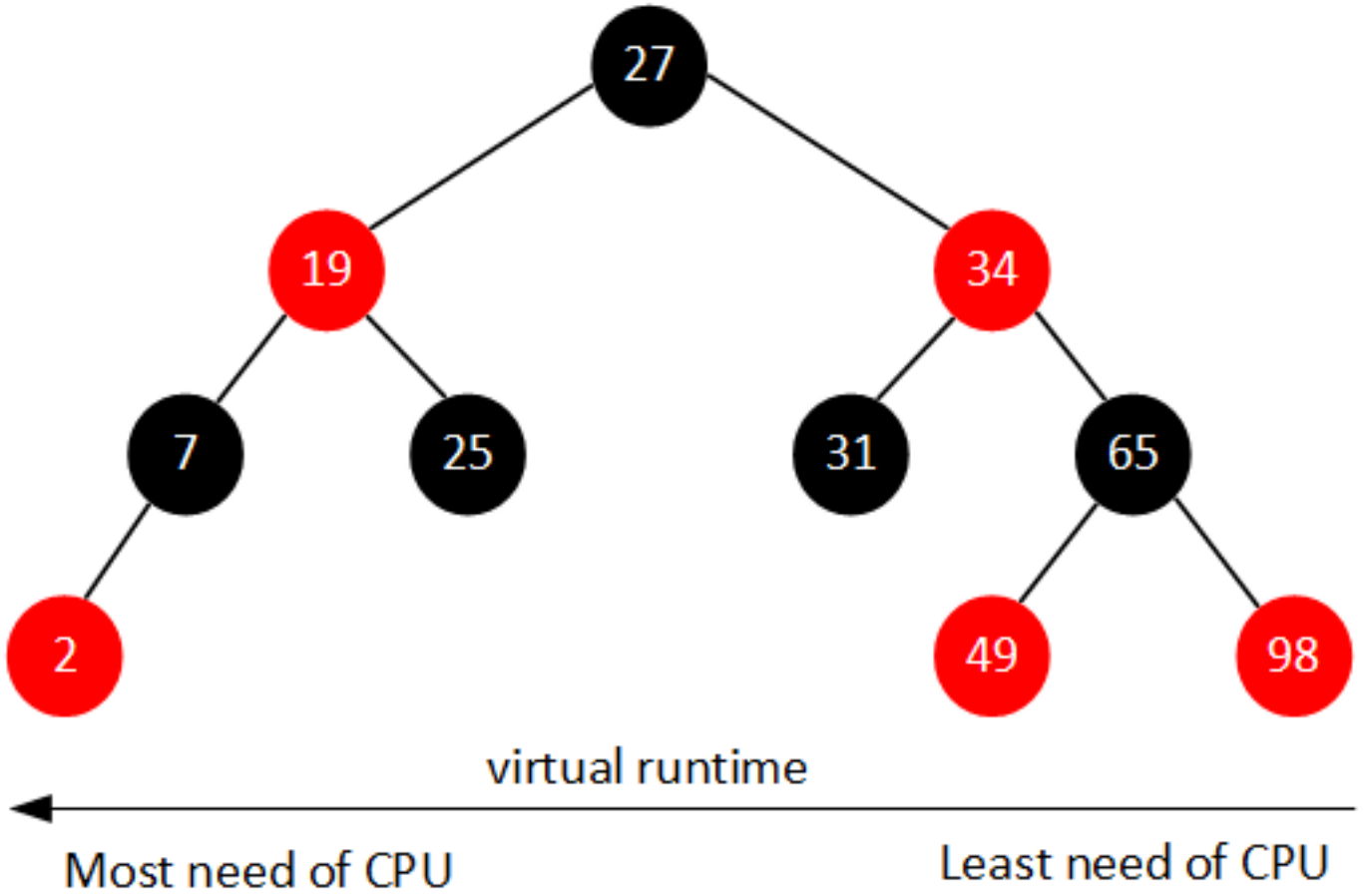
- struct sched_class {
  - const struct sched_class *next;
  - void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
  - void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
  - void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags);
  - struct task_struct * (*pick_next_task)(struct rq *rq, struct task_struct *prev, struct rq_flags *rf);
    - /* ... */
- };
- 在include/linux/sched.h

完全公平调度-CFS

- 接下来，简述一下完全公平调度CFS。
- CFS调度器的目标，是保证每一个进程的完全公平调度。
- CFS调度器和以往的调度器，不同之处在于，没有时间片的概念， 而是分配cpu使用时间的比例。
- 理想状态下，每个进程都能获 得相同的时间片，并且同时运行在CPU上，但实际上一个CPU 同一时刻运行的进程，只能有一个。
- 也就是说，当一个进程 占用CPU时，其他进程就必须等待。
- CFS为了实现公平，必须 惩罚当前正在运行的进程，以使那些正在等待的进程下次被 调度。

完全公平调度-红黑树

- 具体实现时，CFS通过每个进程的虚拟运行时间(vruntime)来衡量哪个进程最值得被调度。
- 如图所示， CFS中的就绪队列，是一棵以虚拟时间为键值的红黑树，虚拟时间越小的进程越靠近整个红黑树的最左端。
- 因此，调度器每次选择位于红黑树最左端的那个进程，该进程的虚拟时间最小。



- 虚拟运行时间，是通过进程的实际运行时间，和进程的权重(weight)计算出来的。
- 在CFS调度器中，将进程优先级这个概念弱化，而是强调进程的权重。
- 一个进程的权重越大，则说明这个进程更需要运行，因此它的虚拟运行时间就越小，这样被调度的机会就越大。

课程思政

课程思政

学会从唯物辩证法发展观的角度来看待问题

2018年以来，在“中兴事件”“华为事件”等的影响下，国人尤其觉得倘若核心技术受制于人，我国就无法在相应的科技领域中占据主导地位，而这必将会严重影响国家安全、社会稳定以及经济发展等。例如，采用Windows 10等知识产权非我国所有的操作系统，即有可能给我国信息安全带来潜在风险。

值得庆幸的是，近年来，在“核高基”（核心电子器件+高端通用芯片+基础软件产品）等国家科技重大专项的支持与引导下，国产操作系统领域研发人员不断增强自主创新能力，同时促进自主研发的操作系统充分参与市场竞争，使得国产操作系统市场占有率得以大幅提升。那么，国产操作系统究竟能否成功呢？

学会从唯物辩证法发展观的角度来看待问题

判定一个操作系统能否成功，主要得看它的生态建设情况。

在上述国际形势下，中国信息技术（information technology，IT）产业从基础硬件到系统软件，再到行业应用软件，在不同层面均迎来了国产替代潮。如今，国产操作系统的应用领域正在逐步拓展，金财、金农、金企等“十二金”工程以及通信、能源、交通等一系列关键基础设施，都已开始进行部分产品国产化替代。

坚信，随着国家战略向国产操作系统倾斜，行业市场规模必将越来越大，成熟的开源软件（如Linux等）所构成的生态圈，也必将会孕育出蓬勃发展的国产操作系统生态环境。但是与此同时，我们也要学会从唯物辩证法发展观的角度来看待问题，清晰地认识到研发国产操作系统依旧任重而道远，促使更多企业投入国产操作系统研发中来，同时不断培养IT产业高级人才，实现国产操作系统发展的良性循环。

课后练习题

1、进程和程序有什么区别与联系，请简述。
2、在Linux中，通过那种系统调用来创建一个新的进程？
3、进程包括5种状态，请对其做一个介绍。
4、请简述进程和程序有什么区别与联系。
5、fork 、clone与vfork函数，使用中有什么区别。
6、请简述do_fork()代码流程中的关键步骤。
7、调度就是从就绪队列中选择一个进程投入CPU运行，正确吗？
8、在进程调度算法中，从用户空间，有两种优先级，请对其进行简述。
9、请简述O(1)调度算法的核心思想是什么？
10、请对完全公平调度CFS的目标做一个简要介绍。

谢 谢！

THANKS