



東北大學 秦皇島分校
Northeastern University at Qinhuangdao



Linux系统与内核分析

-- 进程

于七龙



目录

Part 1

进程简介

Part 2

Linux的进程控制块

Part 3

Linux进程组织方式

Part 4

Linux进程调度

Part 5

Linux进程创建

Part 6

Linux进程相关系统调用



Part 1 进程简介



进程特征

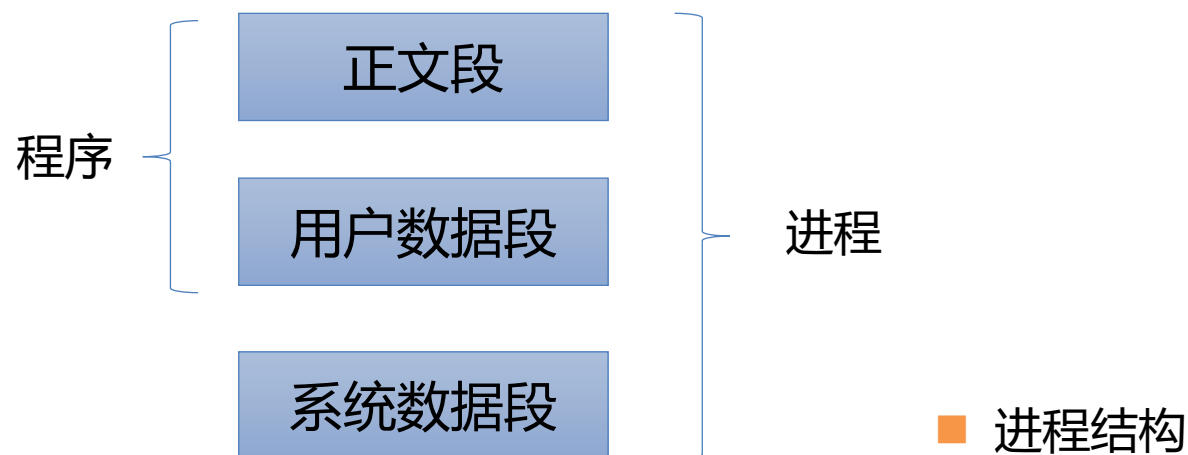
- ◆ 并发性
 - 与其它进程在宏观上同时向前推进
- ◆ 动态性
 - 进程是执行中的程序，动态产生、动态消亡、动态变化
- ◆ 独立性
 - 进程是系统资源分配的基本单位
- ◆ 交互性
 - 进程可与其他进程交互
- ◆ 异步性
 - 各个进程不统一推进
- ◆ 结构性
 - 每个进程都有一个控制块（PCB）
- ◆ 并行性
 - 多处理机上特有



进程结构

◆ 进程主要包含三部分内容

- 正文段：只读，存放被执行的机器指令，可多进程共享
- 用户数据段：进程执行过程中直接操作的数据，如变量，每个进程有私有数据段
- 系统数据段：程序运行的环境(上下文)，如进程控制信息





进程控制块

- ◆ 进程控制块PCB (Process Control Block) 是用于描述进程信息的数据结构
 - PCB是进程存在和运行的唯一标志
- ◆ 进程控制块包含信息
 - 进程标识符、处理机状态、进程调度信息、进程控制信息.....

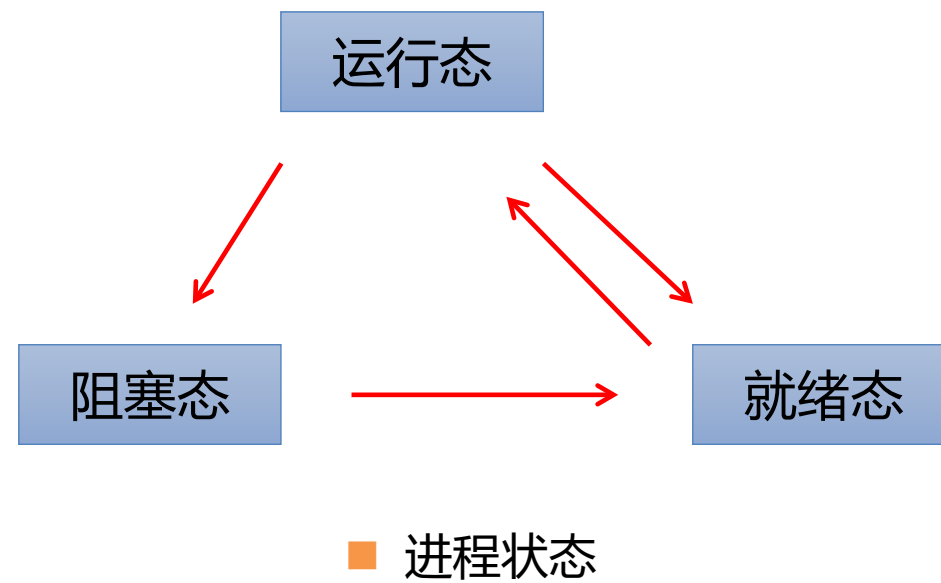


进程状态

◆ 进程最基本状态

- 运行态：正在运行
- 就绪态：万事俱备只欠CPU
- 阻塞态：暂时缺少运行条件，即使CPU空闲

◆ 三种状态，四种转换





Linux进程的层次结构

◆ Linux启动时创建名为init的起始进程

- init进程是系统所有进程的祖先

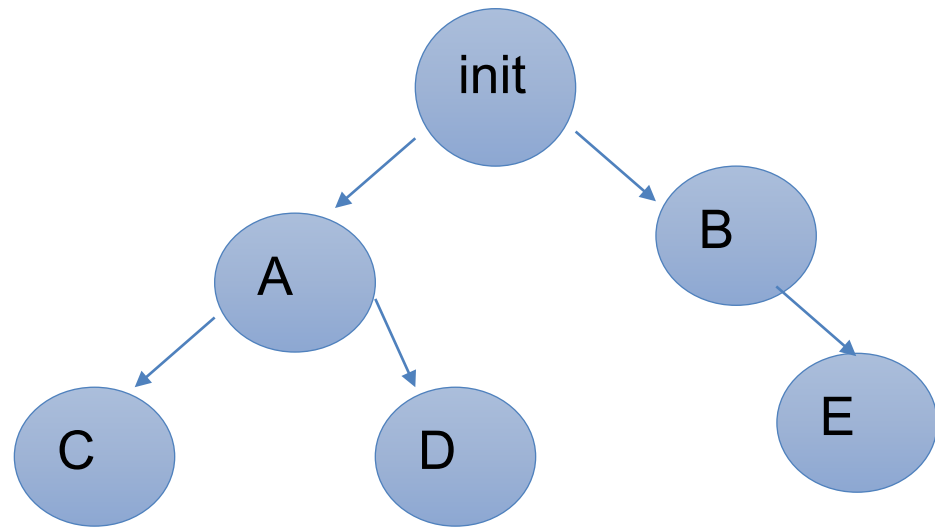
◆ init还负责托管系统其它“孤儿”进程

- 本功能是为保持进程数完整性

◆ 每个进程只有一个父进程，但可有多个子进程

◆ 查看进程树

- pstree、ps -elH



■ 进程树



Linux进程举例

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
```

```
int main(){
    pid_t pid;
    printf("PID before fork():%d\n",getpid());
    pid=fork();  //创建进程
    pid_t npid = getpid();
    if(pid<0)
        printf("error in fork!");
    else if(pid==0){
        printf("I am the child process, my process ID is %d\n", npid);
        sleep(1000);
    } else {
        printf("I am the parent process, my process ID is %d\n",npid);
        sleep(1000);
    }
    return 0;
}
```



Linux进程举例

```
root@bogon:/code# ./forktest
PID before fork():342655
I am the parent process, my process ID is 342655
I am the child process, my process ID is 342656
```

■ 创建进程

```
root@bogon:~/Desktop# ps lf
```

UID	PID	PPID	PRI	NI	VSZ	RSS	WCHAN	STAT	TTY	TIME	COMMAND
0	336598	336595	20	0	7764	4764	-	Ss	pts/2	0:00	/bin/bash
0	342931	336598	20	0	8500	3056	-	R+	pts/2	0:00	_ ps lf
0	328694	315081	20	0	7976	3560	-	S	pts/1	0:00	su root
0	328707	328694	20	0	8480	5184	-	S	pts/1	0:00	_ bash
0	342655	328707	20	0	2280	764	-	S+	pts/1	0:00	_ ./forktest
0	342656	342655	20	0	2280	88	-	S+	pts/1	0:00	_ ./forkte
0	312352	312349	20	0	8384	5104	-	Ss	pts/0	0:00	/bin/bash
0	313107	312352	20	0	1070828	116108	-	Sl+	pts/0	7:24	_ qemu-system-arm
0	775	677	20	0	293304	38616	-	Ssl+	tty7	0:12	/usr/lib/xorg/Xorg :
0	776	1	20	0	2636	1468	-	Ss+	tty1	0:00	/sbin/agetty -o -p -

■ 进程树



Part 2 Linux的进程控制块



Linux的进程控制块

- ◆ 操作系统通过PCB对每个进程在其生命周期内涉及的所有事情进行全面的描述和控制
- ◆ Linux通过结构体对进程进行描述

```
struct task_struct  
{  
    ...  
    ...  
};
```

```
1340 struct task_struct {  
1341     volatile long state;      /* -1 unrunnable, 0 runnable, >0 stopped */  
1342     void *stack;  
1343     atomic_t usage;  
1344     unsigned int flags;      /* per process flags, defined below */  
1345     unsigned int ptrace;
```

■ 进程描述



Linux的进程控制块

◆ PCB数据结构主要区域

- 状态信息：描述进程状态
- 链接信息：描述进程亲属关系
- 各种标识符：进程标识符、用户标识符等
- 进程间通信信息：描述多个进程在同一任务上的写作工作，如管道、共享内存等
- 时间和定时器信息：描述进程生存周期内使用CPU时间的统计等
- 调度信息：描述进程优先级、调度策略等
- 文件系统信息：描述进程的文件使用情况，如打开文件动作
- 虚拟内存信息：描述进程拥有的地址空间，即编译连接后形成的空间
- 处理器环境信息：描述进程执行环境，如堆栈等



Linux的进程状态

◆ task_struct状态定义域

- state = -1: unrunnable
- state = 0 :runnable
- state >0:stopped

volatile是类型修饰符，标识编译程序不必优化，从内存区读取数据，以确保状态变化同步

```
1339
1340 struct task_struct {
1341     volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
1342     void *stack;
1343     atomic_t usage;
1344     unsigned int flags;    /* per process flags, defined below */
1345     unsigned int ptrace;
1346 }
```



Linux的进程状态

```
201 #define TASK_RUNNING          0
202 #define TASK_INTERRUPTIBLE     1
203 #define TASK_UNINTERRUPTIBLE   2
204 #define __TASK_STOPPED         4
205 #define __TASK_TRACED           8
206 /* in tsk->exit_state */
207 #define EXIT_ZOMBIE             16
208 #define EXIT_DEAD               32
209 /* in tsk->state again */
210 #define TASK_DEAD               64
211 #define TASK_WAKEKILL           128
212 #define TASK_WAKING             256
213 #define TASK_PARKED             512
214 #define TASK_STATE_MAX         1024
215
```

■ Linux进程状态定义



Linux的进程状态

◆ Linux进程状态分类

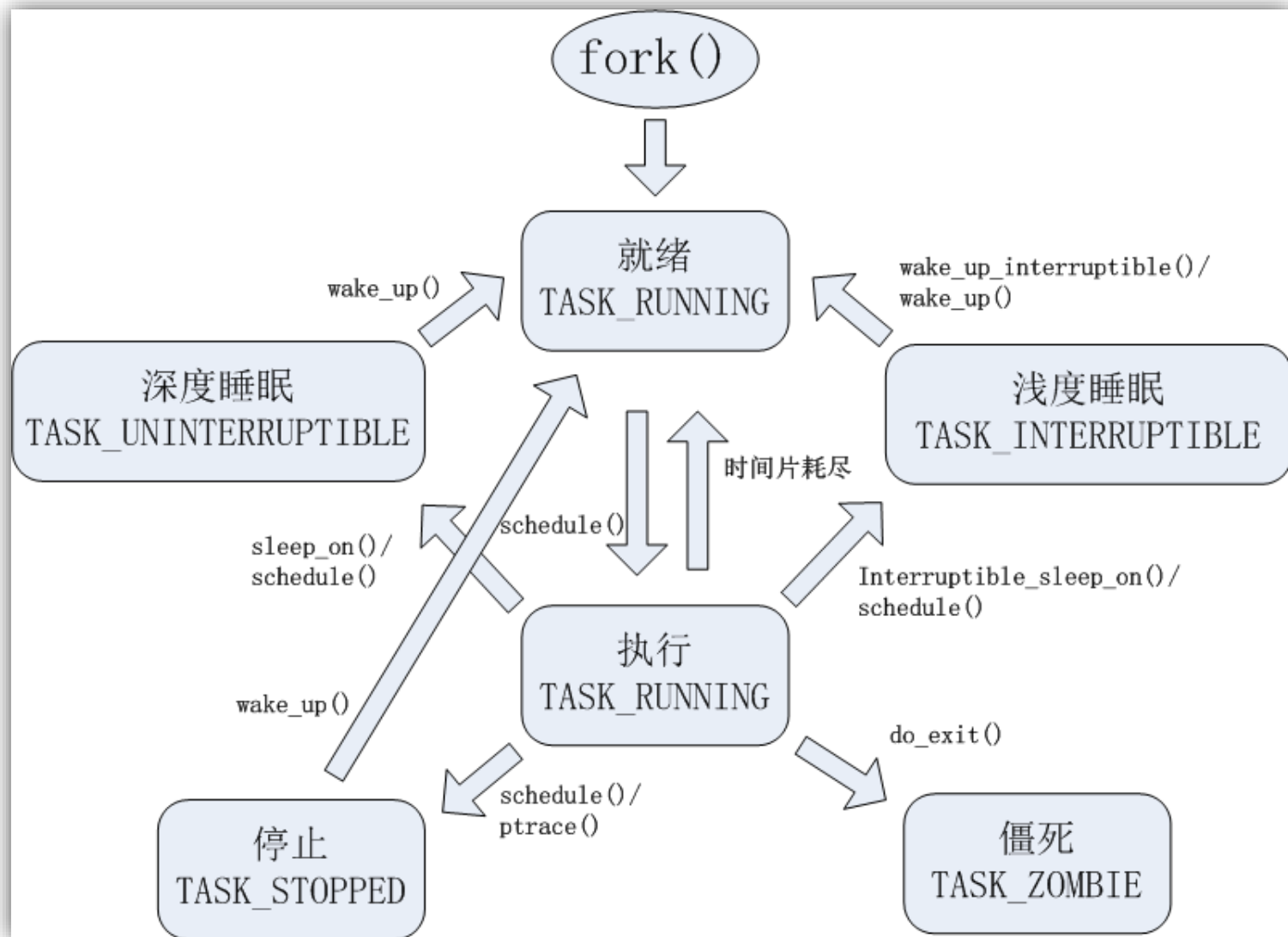
- 就绪态(TASK_RUNNING):正在运行或准备运行
- 浅度睡眠(TASK_INTERRUPTIBLE):进程被阻塞, 等待资源有效时唤醒
- 深度睡眠(TASK_UNINTERRUPTIBLE): 进程被阻塞, 资源有效时仍保持阻断
- 暂停态(TASK_STOPPED): 进程暂停执行
- 僵死态(TASK_ZOMBIE):进程执行结束, 等待释放资源

tips

Linux对操作系统中常规的就绪态和运行态进行了合并, 都称为就绪态



Linux的进程状态



Linux通过内核函数
实现进程状态转换



Linux的进程标识符

- ◆ 每个进程都有进程标识符(PID)、用户标识符(UID)、组标识符(GID)
 - PID是进程的唯一标识符，同时也是内核向用户程序提供的接口
 - UID和GID用于系统安全控制，可通过UID与GID控制进程对系统中文件的访问权限
- ◆ Linux中PID是32位的无符号整数
 - PID最大值代表系统中允许同时存在的进程的最大数目

```
[root@localhost 3.10.0-957.10.1.el7.x86_64]# cat /proc/sys/kernel/pid_max  
32768
```

■ Linux进程最大数目



Linux进程间亲属关系

◆ task_struct数据块中对Linux进程所有亲属关系进行了描述

- “生父进程”：创建该进程的父进程
- “养父进程”：非创建该进程的父进程
- 子进程
- 兄弟进程

子进程与兄弟进程均用
双向循环链表管理

```
struct task_struct {
    int pid, uid, gid;
    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*/
    ...
};
```



Linux进程控制块的存放

- ◆ 系统创建进程时首先需内核为其分配PCB结构
- ◆ 进程被执行时进程从**用户态**进入**内核态**
 - 用户态与内核态具有不同的特权级别
 - 进程在运行自己的代码时属于用户态，特权级为3(最低)
 - 当进程因为系统调用陷入内核代码中执行时处于内核运行态（内核态），特权级为0(最高)



Linux进程控制块的存放

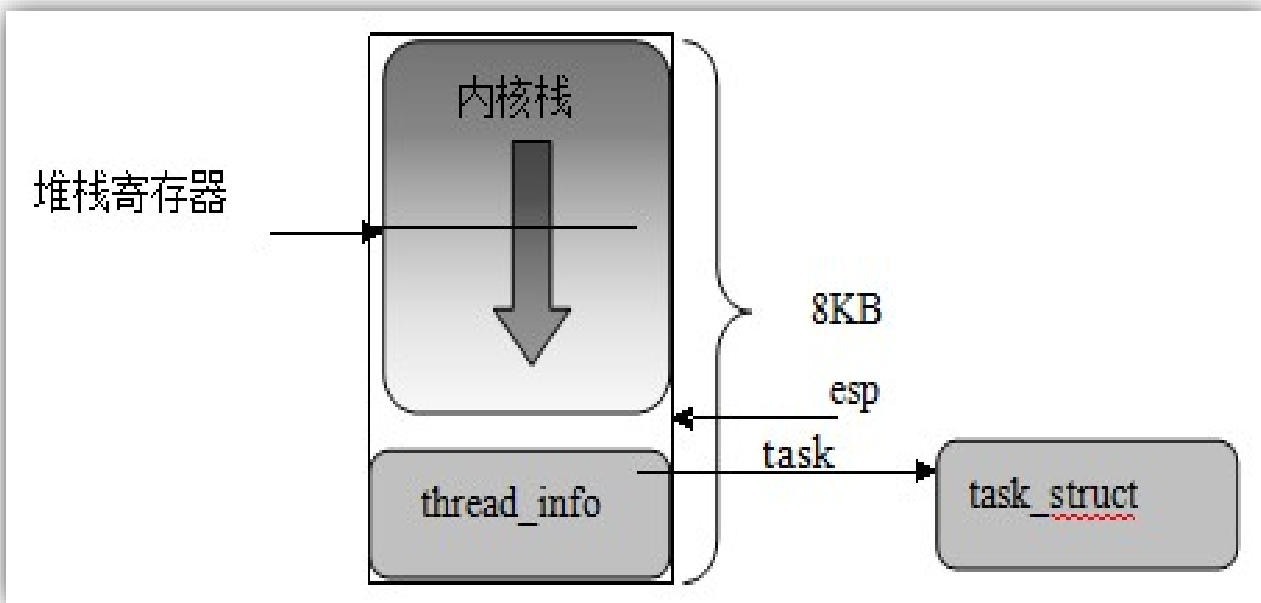
- ◆ 进程进入内核态都需使用栈，称为进程的**内核栈**
 - 进程是动态存在不断变化的实体，用于描述进程运行信息的进程描述符必须常驻内存，当进程从用户态切换到内核态后，也需要栈空间用于进行函数调用，所以内核为每个进程都分配固定大小的内核栈
 - 进程的内核栈位于内核的数据段上



Linux进程控制块的存放

◆ 为节省内核空间，Linux定义小数据结构thread_info用于指向PCB

- 内核栈空间一般占用8K空间
- thread_info紧贴所属进程的内核栈



■ PCB和内核栈的存放

```
union thread_union {  
    struct thread_info thread_info;  
    unsigned long  stack[THREAD_SIZE/sizeof(long)];  
    //定义内核栈空间大小 (8K)  
};
```


■ PCB和内核栈混合结构定义



Linux进程控制块的存放

◆ 内核栈使用联合体结构定义

```
union thread_union
{
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
    //定义内核栈空间大小
};
```



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



Linux进程控制块的存放

```
struct task_struct
{
    .....
    void *stack;
    .....
};
```

```
union thread_union
{
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

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




Linux进程控制块的存放

◆ 内核栈与thread_info能有效节省内核空间

- 理论上，内核态的进程所有数据都应常驻内存，但实际上，进程PCB所占内存是由内核控制分配，准确说，内核根本不对PCB分配内存，而仅仅给内核栈分配8K空间，并把其中一部分让给PCB使用，即thread_info部分



当前进程

◆ thread_info置于内核栈另一好处是内核可以快速获得CPU上正在运行的

thread_info结构对应进程的地址

- thread_union结构长度为8K(即 2^{13} 字节), 则内核屏蔽ESP的低13位有效位就是thread_info结构的基地址
- 内核函数current_thread_info()产生寻址汇编指令
- 执行指令后, 指针p就指向进程的thread_info结构

```
movl $0xffffe000, %ecx  
andl %esp, %ecx  
movl %ecx, p
```



当前进程

- ◆ 在Linux中，为了表示当前正在运行的进程，定义了一个current宏，这个宏本质上等价`current_thread_info()->task`,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

类似于C++中的this指针



Part 3 Linux进程组织方式

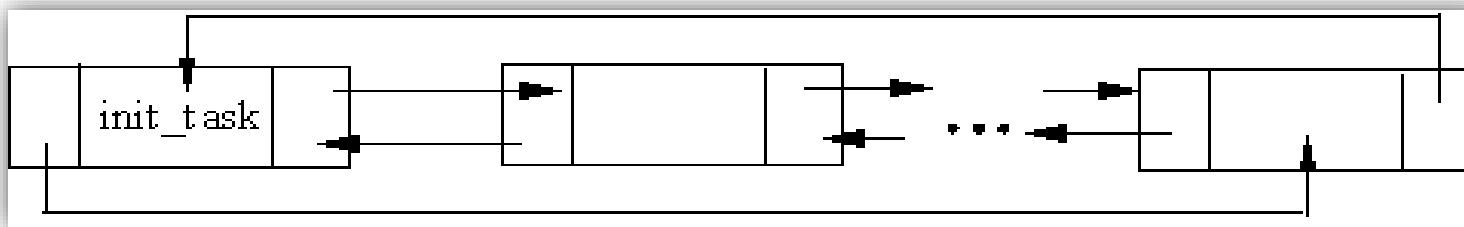


进程链表

◆ 在Linux中，进程通过双向循环链表对所有进程进行有序组织

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

进程链表头和尾都是
init_task进程。该进程
系统运行期间永远不
被撤销



■ 进程链表



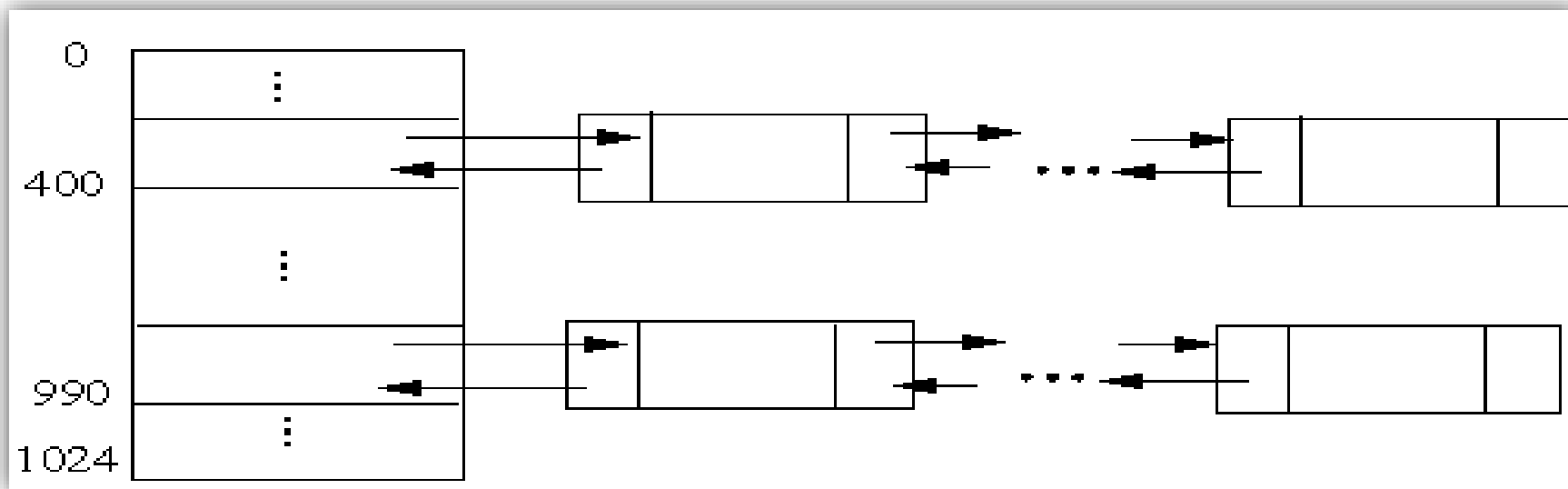
进程哈希表

- ◆ 在拥有大量进程的系统中遍历进程非常耗时，哈希表可加快查找速度
 - 系统通过pid_hashfn()函数，将进程PID均匀散列在哈希表中
 - 不同的PID可能散列到相同的索引上
- ◆ 对于给定的PID，可通过find_task_by_pid()函数定位相应进程



进程哈希表

- ◆ task_struct结构中的pidhash_next和pidhash_prev域用于实现本链表
- ◆ 同一链表中PID由小到大排列



■ 链地址法处理冲突时的哈希表



就绪队列

- ◆ 内核需要寻找新进程到CPU上运行时，只需考虑就绪状态的进程
 - 就绪队列容纳了系统中所有准备运行的进程
- ◆ 就绪队列是由可运行状态的进程组成的双向循环链表

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




等待队列

- ◆ 等待队列是一组睡眠的进程，当进程所需条件为真时由内核唤醒
 - 深度睡眠和浅度睡眠状态进程在同一队列中

```
struct __wait_queue {  
    unsigned int flags;  
    #define WQ_FLAG_EXCLUSIVE    0x01  
    void *private;  
    wait_queue_func_t func;  
    struct list_head task_list;    /*双向链表*/  
};  
typedef struct __wait_queue wait_queue_t;
```

func域指向唤醒函数



等待队列

- ◆ 每个等待队列都有一个等待队列头 (wait_queue head)

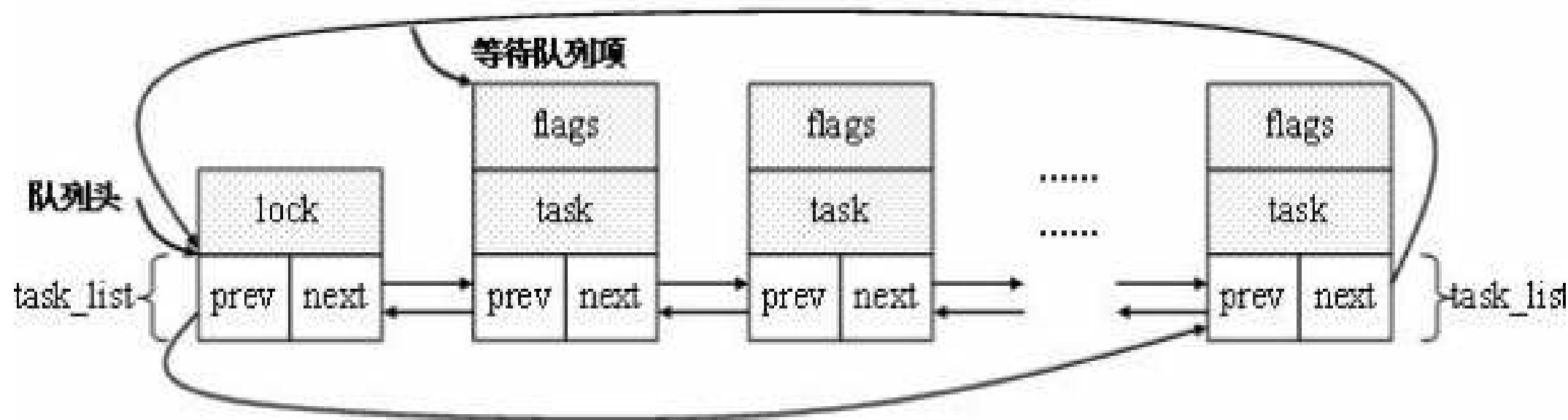
```
struct __wait_queue_head {  
    spinlock_t lock;  
    struct list_head task_list;  
};  
  
typedef struct __wait_queue_head  
wait_queue_head_t;
```

等待队列由终端处理程序和主要内核函数修改，必须进行保护



等待队列

- ◆ 每个等待队列都有一个等待队列头 (wait_queue head)



- 等待队列



等待队列

◆ 等待队列相关函数

- sleep_on(): 进程睡眠函数
- wake_up(): 进程唤醒函数



Part 4 Linux进程调度



进程调度

- ◆ 进程调度的实质是资源的分配
- ◆ 调度算法取决于资源的分配策略
- ◆ 调度算法考虑因素
 - 公平：保证每个进程得到合理的CPU时间
 - 高效：使CPU保持忙碌状态，即总是有进程在CPU上运行
 - 响应时间：使交互用户的响应时间尽可能短
 - 周转时间：使批处理用户等待输出的时间尽可能短
 - 吞吐量：使单位时间内处理的进程数量尽可能多



常见调度算法

◆ 时间片轮转调度算法

- 每个进程依次地按时间片轮流地执行

◆ 优先权调度算法

- 非抢占式优先权算法：常应用于实时性要求不强的系统中
- 抢占式优先权调度算法：用于实时性要求强的系统，本算法容易导致饿死

Linux主要采用优先权调度算法

◆ 多级反馈队列调度

- 前两者折中，优先权高的进程先运行给定的时间片，相同优先权的进程轮流运行给定的时间片

◆ 实时调度

- “有求必应，尽快响应”，一般采用抢占式调度算法



时间片

- ◆ 时间片标识进程在被抢占前所能持续运行的时间
- ◆ 调度策略必须规定一个默认的时间片
 - 普遍使用20ms
- ◆ Linux采用动态的时间片
 - Linux调度程序提高了交互式程序的优先级，所以调度程序提供较长的默认时间片给交互式程序
 - Linux调度程序根据进程的优先级动态调整分配的时间片，从而保证优先级高的进程得到更高的执行频率



进程调度时机

◆ Linux进程调度时机

- 进程状态转换
- 进程时间片用完
- 设备驱动程序运行时(任务执行时间长且重复, 不调度将引起其它进程 “饿死”)
- 从内核态返回到用户态
- 高优先级进程到来



进程优先级

◆ Linux对进程进行了分类

◆ 限期进程优先级 > 实时进程 > 普通进程

- 限期进程优先级为-1
- 实时进程优先级为1-99，数值越大，优先级越高
- 普通进程优先级为100-139，数值越小，优先级越高，可通过nice值改变



进程调度策略

◆ Linux内核已经定义的调度策略，即policy

● 针对限期进程

- SCHED_DEADLINE: 限期调度策略

限期调度有运行时间runtime、截止时间deadline、周期period三个参数，每个周期运行一次，在截止时间前执行完，一次运行时间长度是runtime

● 针对实时进程

- SCHED_FIFO: 先入先出调度
- SCHED_RR: 时间片轮转调度

先入先出很“霸道”，如果没有更高优先级进程将一直占据CPU

轮流调度有时间片，进程用完时间片后让出CPU，并加入优先级对应运行队列的尾部

● 针对普通进程

- SCHED_NORMAL: 标准轮流分时调度
- SCHED_IDLE: 最低优先级调度

标准轮流分时采用完全公平调度算法，把CPU时间公平分配给每个进程

最低优先级调度适合低优先级进程，以避免饿死



进程调度优先级

◆ task_struct中与调度有关域

```
struct task_struct
{
    .....
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    .....
};
```



进程调度优先级

◆ task_struct中与调度有关域

优先级	含义	限期进程	实时进程	普通进程
prio	调度优先级， 数值越小，优先级越高	大多数情况下prio=normal_prio 特殊情况：如果进程a占有互斥锁，进程b正在等待锁，而b优先级高于a，则将a的prio等于b的prio，是a与b优先级相等		
static_prio	静态优先级	无意义，0	无意义，0	120+nice值，值 越小优先级越高
normal_prio	正常优先级： 数值越小，优先级越高	-1	99-rt_priority	static_prio
rt_priority	实时优先级	无意义，0	实时进程优先级（1-99）， 值越大，优先级越高	无意义，0



进程调度优先级

◆ nice值

- 进程可被执行的优先级的**修正数值**
- $PRI(new) = PRI(old) + nice$, PRI值越小, 进程优先级越高
- nice值的范围从-20到+19
- 子进程会继承父进程的nice值



nice值

◆ 权重由nice值确定

kernel/sched/sched.h

```
static const int prio_to_weight[40] = {  
    /* -20 */ 88761, 71755, 56483, 46273, 36291,  
    /* -15 */ 29154, 23254, 18705, 14949, 11916,  
    /* -10 */ 9548, 7620, 6100, 4904, 3906,  
    /* -5 */ 3121, 2501, 1991, 1586, 1277,  
    /* 0 */ 1024, 820, 655, 526, 423,  
    /* 5 */ 335, 272, 215, 172, 137,  
    /* 10 */ 110, 87, 70, 56, 45,  
    /* 15 */ 36, 29, 23, 18, 15,  
};
```

nice值为-20时权重为
88761, -19时为71755



调度类

- ◆ Linux内核引入调度类 (struct sched_class)将调度策略模块化
- ◆ schedule()只需调用调度类接口，实现具体的调度任务

调度类	调度策略	调度算法	调度对象
停机调度类 stop_sched_class	无	无	停机进程
限期调度类 dl_sched_class	SCHED_DEADLINE	最早期限优先	限期调度
实时调度类 rt_sche_class	SCHED_FIFO	先入先出	实时调度
	SCHED_RR	轮流调度	
公平调度类 cfs_sched_class	SCHED_NORMAL	完全公平调度算 法	普通进程
	SCHED_IDLE		
空闲调度类 idel_sched_class	无	无	处理器上的空闲进程



停机调度类

- ◆ 停机调度类是优先级最高的调度类，**停机进程 (stop-task)** 是优先级最高的进程，可以抢占其他所有进程，其它进程不可抢占停机进程
- ◆ 目前只有迁移线程属于停机调度类，每个处理器一个迁移线程，该线程对外显示是优先级为99的实时进程
- ◆ 停机进程没有时间片，如果不主动退出将一直占据CPU
- ◆ 引入停机调度类目的是支持限期调度类
 - 迁移线程的优先级必须比限期进程的优先级高，能够抢占所有其他进程，才能快速处理调度器发出的迁移请求，把进程从当前处理器迁移到其他处理器



限期调度类

- ◆ 限期调度类使用最早期限优先算法，使用红黑树把进程按照绝对截止时间从小到大排序，每次调度选择绝对截止时间最小的进程
- ◆ 如果限期进程用完其运行时间，则让出CPU，并从运行队列中删除，在下一个周期的开始，重新加入运行队列



实时调度类

- ◆ 实时调度类为每个调度优先级分别维护一个队列
- ◆ 每次调度，先找到优先级最高的第一个非空队列，然后从队列中选择第一个进程
- ◆ 使用先进先出调度策略没有时间片，如果没有更高优先级进程，将一直霸占CPU
- ◆ 轮流调度策略的时间片可修改

```
root@bogon:/# cat /proc/sys/kernel/sched_rr_timeslice_ms  
100
```

- 轮流调度策略时间片



公平调度类

◆ 公平调度类引入虚拟运行时间的概念

- 虚拟运行时间 += 实际运行时间 * nice 0 对应的权重 / 进程的权重

```
if (se.load.weight != NICE_0_LOAD)
    vruntime += delta * NICE_0_LOAD / se.load.weight;
else
    vruntime += delta;
```

权重越大，虚拟运行时间越小

◆ 完全公平调度算法使用红黑树把进程按照虚拟运行时间从小到大排序，每次调度时选择虚拟运行时间最小的进程



公平调度类

- ◆ 使用空闲调度类策略的普通进程权重为3， nice值对权重无影响

```
kernel/sched/sched.h
```

```
#define WEIGHT_IDLEPRIO      3
```

- ◆ 显然，进程的静态优先级越高（nice值越小），权重越大，在实际运行时间相同的情况下，虚拟运行时间就越短，进程累计的虚拟运行时间增加越慢，在红黑树中移动的速度越慢，被调度器选中的机会越大，被分配的运行时间相对越多



公平调度类效率保障

◆ 调度的最小粒度

- 为防止进程切换太频繁而设置进程调度后运行的最短时间

```
root@bogon:/# cat /proc/sys/kernel/sched_min_granularity_ns  
750000
```

■ 调度最小粒度

◆ 调度周期：某个时间长度可以保证运行队列中每个进程至少执行一次

- 如：若运行队列中进程数量大于8，则 调度周期 = 调度最小粒度 * 进程数量，否则6ms

◆ 进程时间片 = （调度周期 * 进程权重 / 运行队列中所有进程权重总和）

(每个进程应该运行的时间)

$$ideal_time = sum_runtime * se.weight / cfs_rq.weight$$



空闲调度类

- ◆ 每个处理器上有一个空闲线程
- ◆ 空闲线程优先级最低，仅当没有其它进程可以调度时调用空闲进程



运行队列

- ◆ 每个处理器有一个运行队列
 - 运行队列结构体是rq，系统定义了其全局变量

```
kernel/sched/core.h
```

```
#DEFINE_PER_CPU_SHARED_ALIGNED(struct rq, runqueues);
```




运行队列

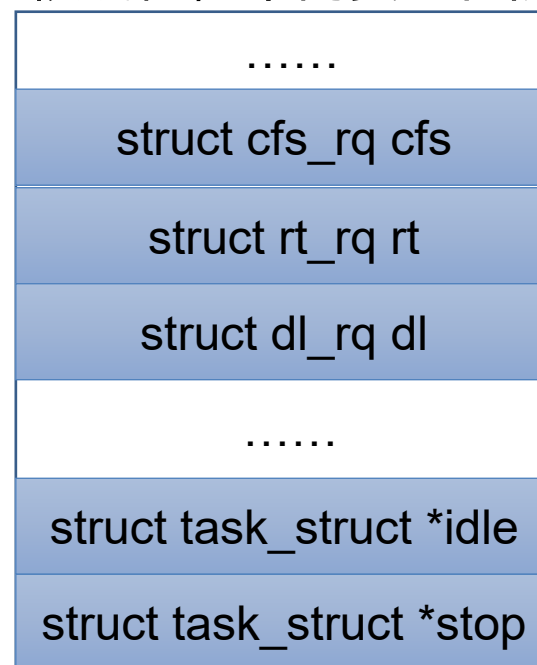
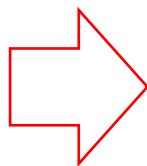
◆ 每个处理器有一个运行队列

- 结构体rq中嵌套了cfs、rt、dl队列
- 停机调度类和空闲调度类在每个处理器上只有一个内核线程，不需要运行队列，直接定义成员

stop和idle分别指向迁移线程和空闲线程

kernel/sched/sched.h

```
struct rq {  
    struct cfs_rq cfs;  
    struct rt_rq rt;  
    struct dl_rq dl;  
    .....  
};
```



■ 运行队列rq



进程调度过程

◆ Linux内核通过schedule()进行进程调度

- schedule()调用内核函数__schedule()

```
...  
next = pick_next_task(rq, prev); //选择调度算法  
...  
context_switch(rq, prev, next);  
...
```

◆ context_switch()用于实现上下文切换

- switch_mm()切换页表
- switch_to()切换处理器寄存器内容和内核栈，本部分通常由汇编语言实现



Part 5 Linux进程创建



进程创建

◆ Linux通过fork()函数实现进程的创建

- fork()就像细胞的裂变，调用fork()的进程就是父进程，而新裂变出的进程就是子进程
- 调用fork()后，fork()对父进程返回子进程的进程标识符（PID），同时向子进程，返回0

```
...
pid=fork();
if(pid<0)
    printf("error in fork!");
else if(pid==0){
    printf("I am the child process, my process ID is %d\n", npid);
} else {
    printf("I am the parent process, my process ID is %d\n", npid);
}
...
```



进程创建

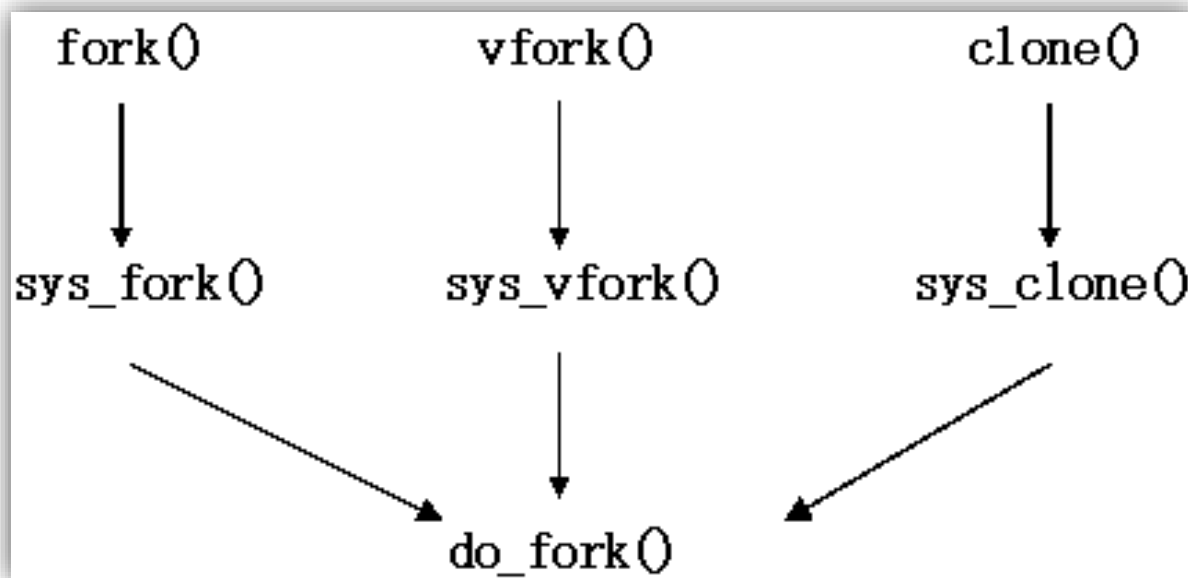
- ◆ 除了fork()外，Linux的C库中还有vfork()和 clone(),三个函数均是用于进程创建
 - fork(): 完全复制父进程的资源，并独立于父进程
 - vfork(): 完全共享父进程的地址空间，并阻塞父进程
 - clone():通过参数对父子进程之间的共享、复制进行精确控制



进程创建

◆ `fork()`、`vfork()`和 `clone()`均通过调用`do_fork()`函数实现

- `do_fork()`通过参数控制实现不同功能



■ `fork()`、`vfork()`与`clone()`

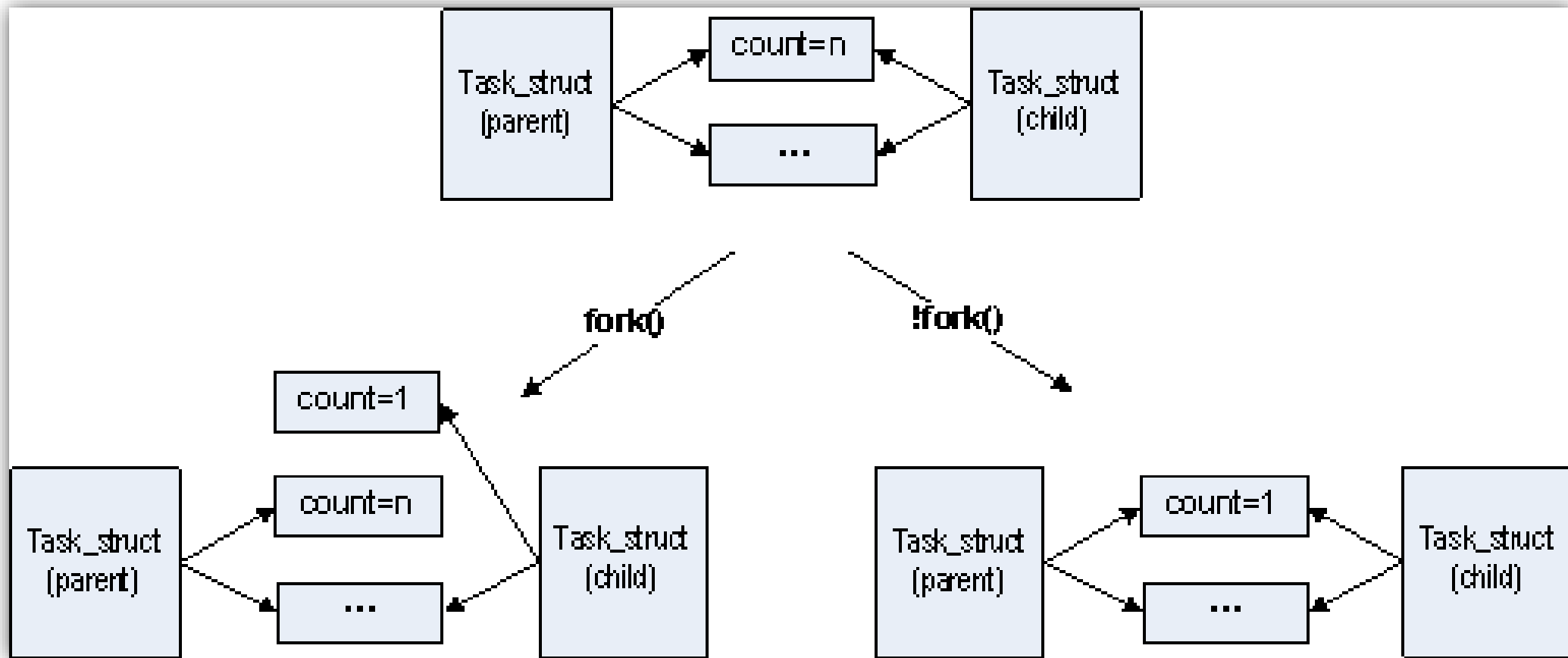


写时复制

- ◆ 传统的fork()系统调用直接把所有的资源复制给新创建的进程
- ◆ 为了节省空间，调用fork()时内核并没有把父进程的全部资源复制，只是以只读的方式共享父进程资源，当父进程或子进程需要修改地址空间上的某一页时，才将该页进行复制
- ◆ **fork()的实际开销仅仅是复制父进程的页表与修改PCB**



写时复制



■ 写时复制



创建进程

◆ 创建进程 (do_fork()) 主要操作

1. 调用`alloc_task_struct()`函数以获得8KB的`union task_union`内存区,用以存放新进程的内核栈
2. 让当前指针指向父进程的PCB, 并把父进程PCB的内容拷贝到刚刚分配的新进程的PCB中
3. 检查新创建子进程, 确定当前用户所拥有的进程数目没有超出给他分配的资源限制
4. 将子进程的状态被设置为`TASK_UNINTERRUPTIBLE`以保证不会马上投入运行
5. 调用`get_pid()`为新进程获取一个有效的PID
6. 更新不能从父进程继承的PCB的其他所有域, 例如, 进程间亲属关系的域



创建进程

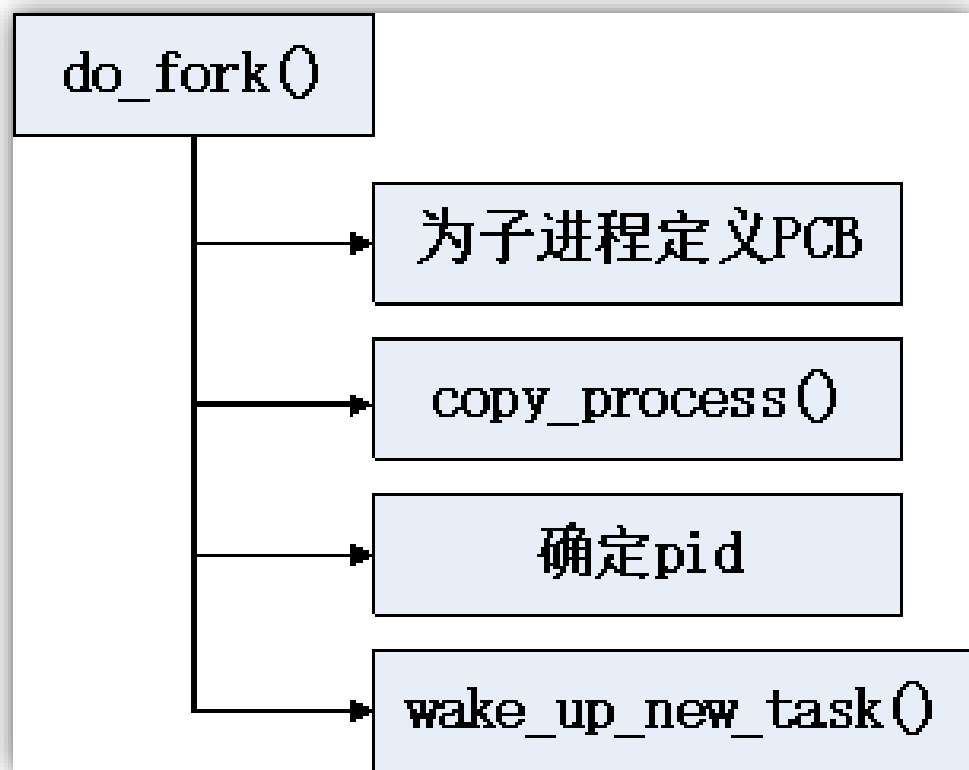
◆ 创建进程 (do_fork()) 主要操作

7. 把新的PCB插入进程链表, 以确保进程之间的亲属关系
8. 把新的PCB插入pidhash哈希表
9. 把子进程PCB的状态域设置成TASK_RUNNING, 并调用wake_up_process()把子进程插入到运行队列链表
10. 让父进程和子进程平分剩余的时间片
11. 返回子进程的PID, 这个PID最终由用户态下的父进程读取

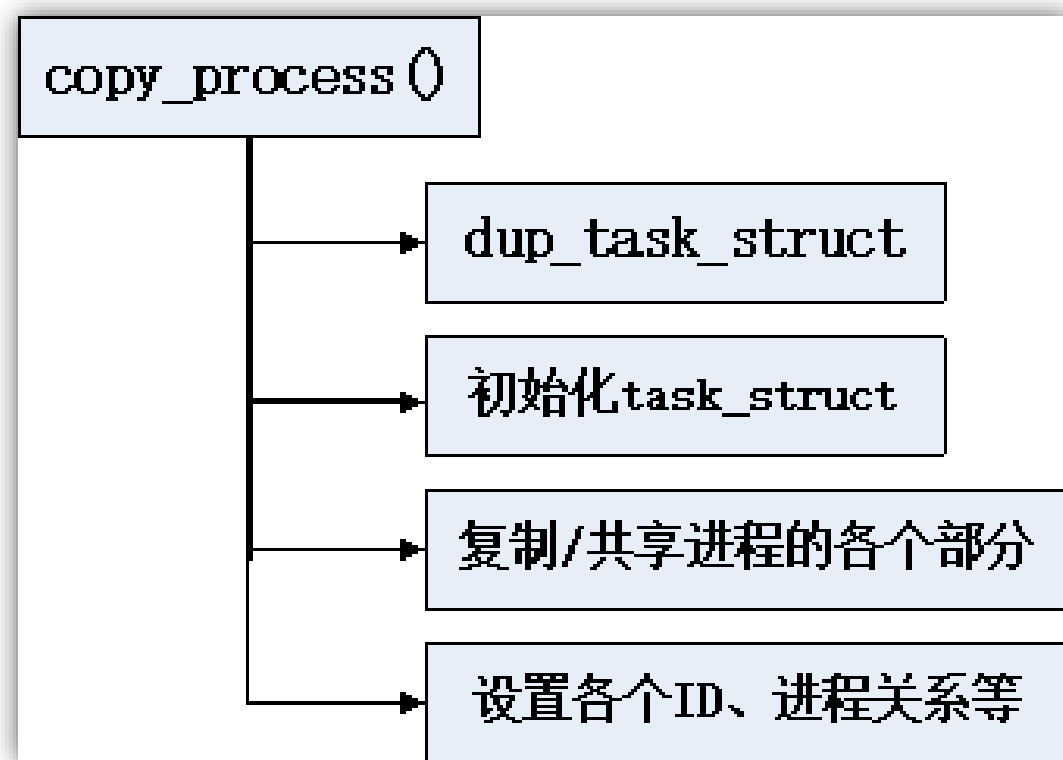


创建进程

◆ 创建进程 (do_fork()) 主要操作



■ do_fork()



■ copy_process()



内核线程

- ◆ Linux内核线程指运行在内核中的**轻量级进程**，也叫内核任务
- ◆ 内核线程与进程又有一定区别
 - 内核线程执行的是内核中的函数，而普通进程只有通过系统调用才能执行内核中的函数
 - 内核线程只运行于内核态，普通进程即可运行于内核态，也可运行于用户态
 - 内核线程使用的地址空间小于普通进程



Linux特殊的内核线程

◆ Linux特殊的内核线程

- 进程0：内核初始化创建的第一个线程
- 进程1：系统第一个用户进程
- kflushd (即bdf flush)线程：守护进程
- kupdate线程：守护进程
- kpiod线程：守护进程
- kswapd线程：守护进程



0号进程

- ◆ 内核初始化创建的第一个线程
- ◆ 内核初始化入口是函数start_kernel,函数首先初始化基础设施, 即初始化内核的各个子系统, 然后调用函数rest_init
 - 创建1号线程, 即init线程, 线程函数是kernel_init
 - 创建2号线程, 即kthreadd线程, 负责创建内核线程
 - 0号线程最终变成空闲进程, 称为“闲逛线程” (Idle Process)
- ◆ 闲逛进程可减少系统耗能
 - 线程执行cpu_idle函数, 函数只有汇编指令hlt (处理器暂停指定)
 - 当就绪队列没有其它进程时, 闲逛进程0就被调度程序选中, 以此达到省电目的



1号进程

- ◆ 系统第一个用户进程，本进程从不终止，用于创建和监控内核外层所有进程
- ◆ init线程初始化后执行以下操作
 - smp_prepare_cpus():在启动从处理器以前执行准备工作
 - do_pre_smp_initcalls():执行必须在初始化SMP系统以前执行的早期初始化
 - smp_init:初始化SMP系统，启动所有从处理器
 - do_initcalls:执行级别0-7的初始化
 - 打开控制台字符设备文件 “/dev/console”
 - prepare_namespace(): 挂载根文件系统，后面装载init程序时需要从存储设备上的文件系统中读文件
 - free_initmem():释放初始化代码和数据占用的内存
 - 装载init程序，从内核线程转换成用户空间的init线程



1号进程

- ◆ init启动配置文件是 “/etc/inittab” ,用于指定要执行的程序以及程序执行级别
 - “/etc/inittab” 配置行格式为: id:runlevels:action: process
 - id是配置行标识符, runlevel是运行级别, action是要执行的动作, process是要执行的程序

级别	动作
0	关机
1	单用户系统
6	重启
2	没有联网的多用户模式
3	联网且使用命令行界面的多用户模式
5	联网且使用图形用户界面的多用户模式



其它特殊内核线程

- ◆ kflushd (即bdf flush)线程：刷新“脏”缓冲区中的内容到磁盘以归还内存
- ◆ kupdate线程：刷新旧的“脏”缓冲区中的内容到磁盘以减少文件系统不一致的风险
- ◆ kpiod线程：把属于共享内存映射的页面交换
- ◆ kswapd线程：执行内存回收功能



Part 6 Linux进程相关系统调用



进程的生命周期

◆ 生命周期

- `fork()`: 进程复制, 执行`fork()`之后, 两个进程并发执行
- `exec()`: 新进程脱胎换骨, 离家独立, 开始独立工作的职业生涯
- `exit()`: 进程退出
- `wait()`: 阻塞自己或对僵死的子进程进行善后处理



进程退出

◆ 子进程退出

- 如果父进程关注子进程退出事件，则进程终止时释放各种资源，只留下一个空的进程描述符，变成僵尸进程，并发送信号SIGCHLD(CHLD是child的缩写) 通知父进程，父进程查询退出原因后回收子进程的进程描述符
- 如果父进程不关注退出事件，则进程退出时直接释放各种资源和进程描述符



进程退出

◆ 父进程退出

- 父进程退出，其子进程变成“孤儿进程”
- 如果父进程属于一个进程组，且组内还有其他进程，则任意选择一个进程当“领养者”
- 选择最亲近的充当“领养者”
- 选择1号进程当领养者



其它进程相关调用

名称	功能
<code>nice()</code>	改变一个普通进程的优先级
<code>getpriority()</code>	取得一组普通进程的最大优先级
<code>setpriority()</code>	设置一组普通进程的优先级
<code>sched_getscheduler()</code>	取得一个进程的调度策略
<code>sched_setscheduler()</code>	设置一个进程的调度策略和优先级
<code>sched_getparam()</code>	取得一个进程的调度优先级
<code>sched_setparam()</code>	设置一个进程的优先级
<code>sched_get_priority_min()</code>	取得某种策略的最小优先级
<code>sched_get_priority_max()</code>	取得某种策略的最大优先级