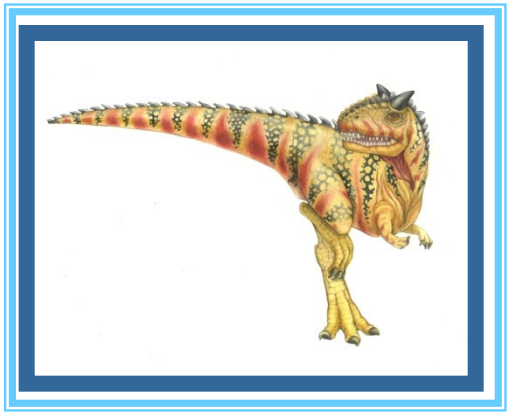
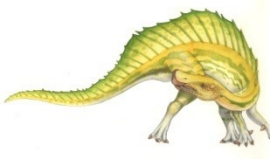
进程管理



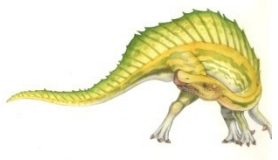
本章内容

- 本章内容：
  - Linux进程概念
  - 进程控制块 task_struct结构
  - Linux进程的创建
  - Linux进程调度



进程概念

- Linux是一个多任务多用户操作系统。
- 一个任务(task)就是一个进程(process)。每一个进程都具有一定的功能和权限，它们都运行在各自独立的虚拟地址空间。
- 在Linux中，进程是系统资源分配的基本单位，也是使用CPU运行的基本调度单位。
- Linux进程与传统UNIX进程的概念没有多大区别。 没有真正意义上的线程概念。但Linux通过clone()系统调用支持轻量级(Lightweight process)进程(线程)，两个轻量级进程可以共享如:地址空间、打开文件等资源;
- Linux还支持内核线程，内核线程永远在内核态运行，没有用户空间。

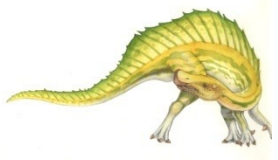


进程概念

- Linux中运行ps命令，你能得到当前系统中进程的列表，比如：

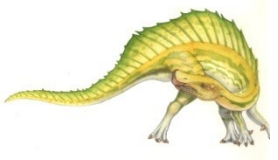
```
$ ps x
  PID TTY      STAT   TIME COMMAND
 1668 tty1     Ss     0:00 -bash
 3201 tty1     S+     0:00 xinit
 3206 tty1     S      0:00 twm
 3209 tty1     R      0:02 xterm
 3211 pts/0    Ss     0:00 bash
 3486 pts/0    S      0:00 xeyes
 4392 ?        Ss     0:00 gvim
 4400 pts/0    R+     0:00 ps x
```

- pstree

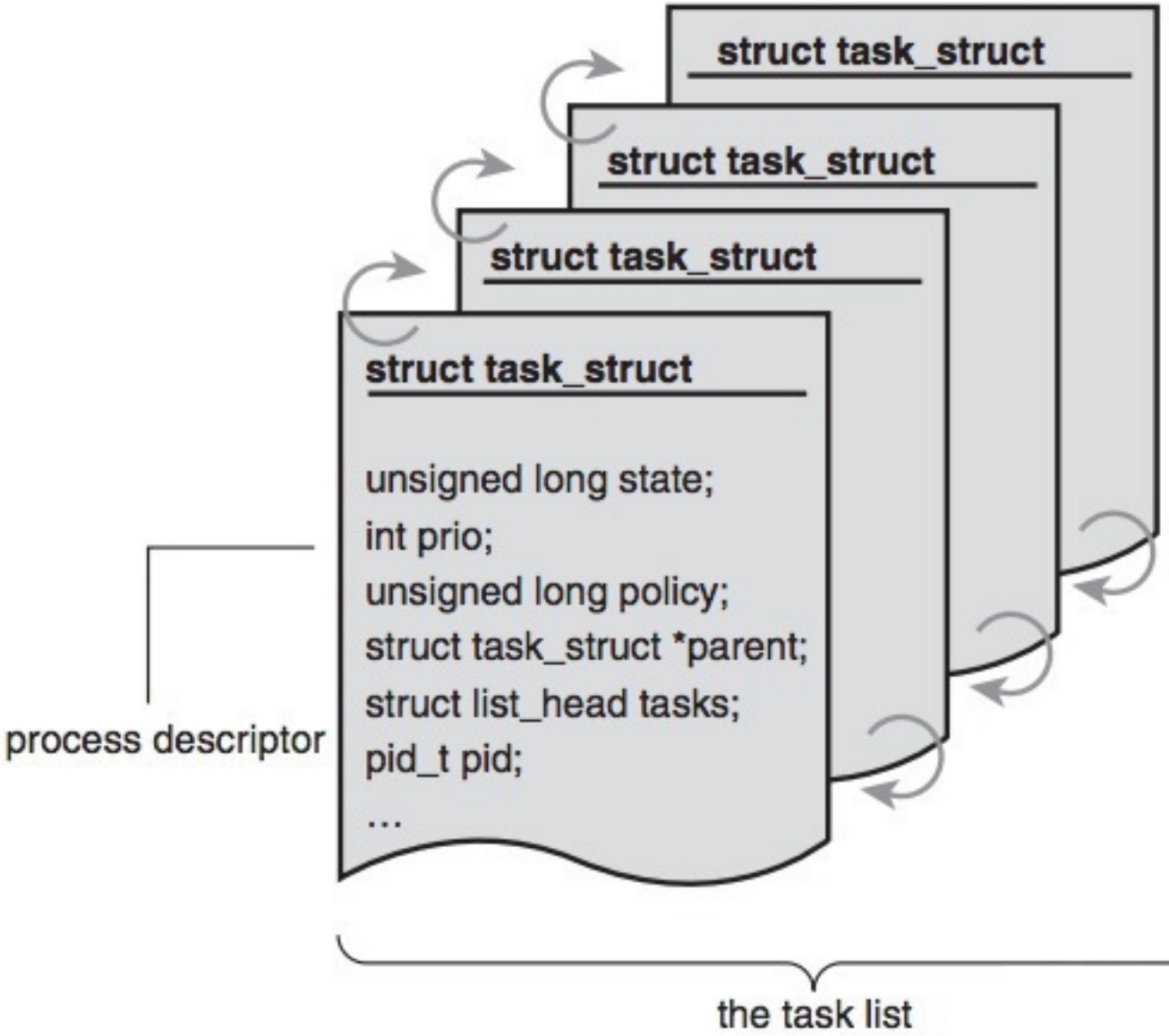


Linux进程的组成

- 存放在磁盘上的可执行文件的代码和数据的集合称为可执行映象(Executable Image)。
- 当一个程序装入系统中运行时，它就形成了一个进程。
- 进程是由正文段(text)、用户数据段(user segment)和系统数据段(system segment)、堆栈段组成的一个动态实体。
  - 正文段中存放着进程要执行的指令代码，具有只读的属性。
  - 用户数据段是进程在运行过程中处理数据的集合，它们是进程直接进行操作的所有数据， 以及进程使用的进程堆栈。
  - 系统数据段存放着进程的控制信息。其中包括进程控制块PCB。



进程数据结构


struct task_struct
struct task_struct
struct task_struct
struct task_struct
unsigned long state;
int prio;
unsigned long policy;
struct task_struct *parent;
struct list_head tasks;
pid_t pid;
…
process descriptor
the task list

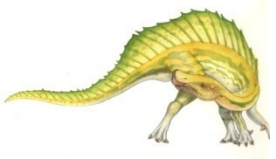
Linux进程状态

- task_struct结构中定义：

  volatile long state;

  long exit_state;

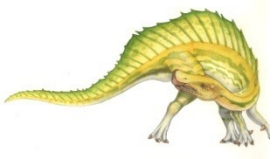
  - 用于表示进程的状态
    - 前者表示用来表征进程的可运行性，
    - 后者表征进程退出时候的状态。



Linux进程状态

- include/linux/sched.h（2.6.15）

```
114 /*
115  * Task state bitmask. NOTE! These bits are also
116  * encoded in fs/proc/array.c: get_task_state().8
117  *
118  * We have two separate sets of flags: task->state
119  * is about runnability, while task->exit_state are
120  * about the task exiting. Confusing, but this way
121  * modifying one set can't modify the other one by
122  * mistake.
123  */
124 #define TASK_RUNNING            0
125 #define TASK_INTERRUPTIBLE      1
126 #define TASK_UNINTERRUPTIBLE    2
127 #define TASK_STOPPED            4
128 #define TASK_TRACED             8
129 /* in tsk->exit_state */
130 #define EXIT_ZOMBIE            16
131 #define EXIT_DEAD              32
132 /* in tsk->state again */
133 #define TASK_NONINTERACTIVE     64
```



include/linux/sched.h（3.12）

```
#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 TASK_WAKEKILL                   128
#define TASK_WAKING                     256
#define TASK_STATE_MAX                  512
#define TASK_STATE_MAX                  1024
 ……
/* Convenience macros for the sake of set_task_state */
#define TASK_KILLABLE                   (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED                    (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED                     (TASK_WAKEKILL | __TASK_TRACED)
/* Convenience macros for the sake of wake_up */
#define TASK_NORMAL                     (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)
#define TASK_ALL                        (TASK_NORMAL | __TASK_STOPPED | __TASK_TRACED)
……
```

Linux进程状态

■**TASK_RUNNING**：正在运行的进程即系统的当前进程或准备运行的进程即在Running队列中的进程。只有处于该状态的进程才实际参与进程调度。

■**TASK_INTERRUPTIBLE**：处于等待资源状态中的进程，当等待的资源有效时被唤醒，也可以被其他进程或内核用信号、中断唤醒后进入就绪状态。

■**TASK_UNINTERRUPTIBLE**：处于等待资源状态中的进程，当等待的资源有效时被唤醒，不可以被其它进程或内核通过信号、中断唤醒。

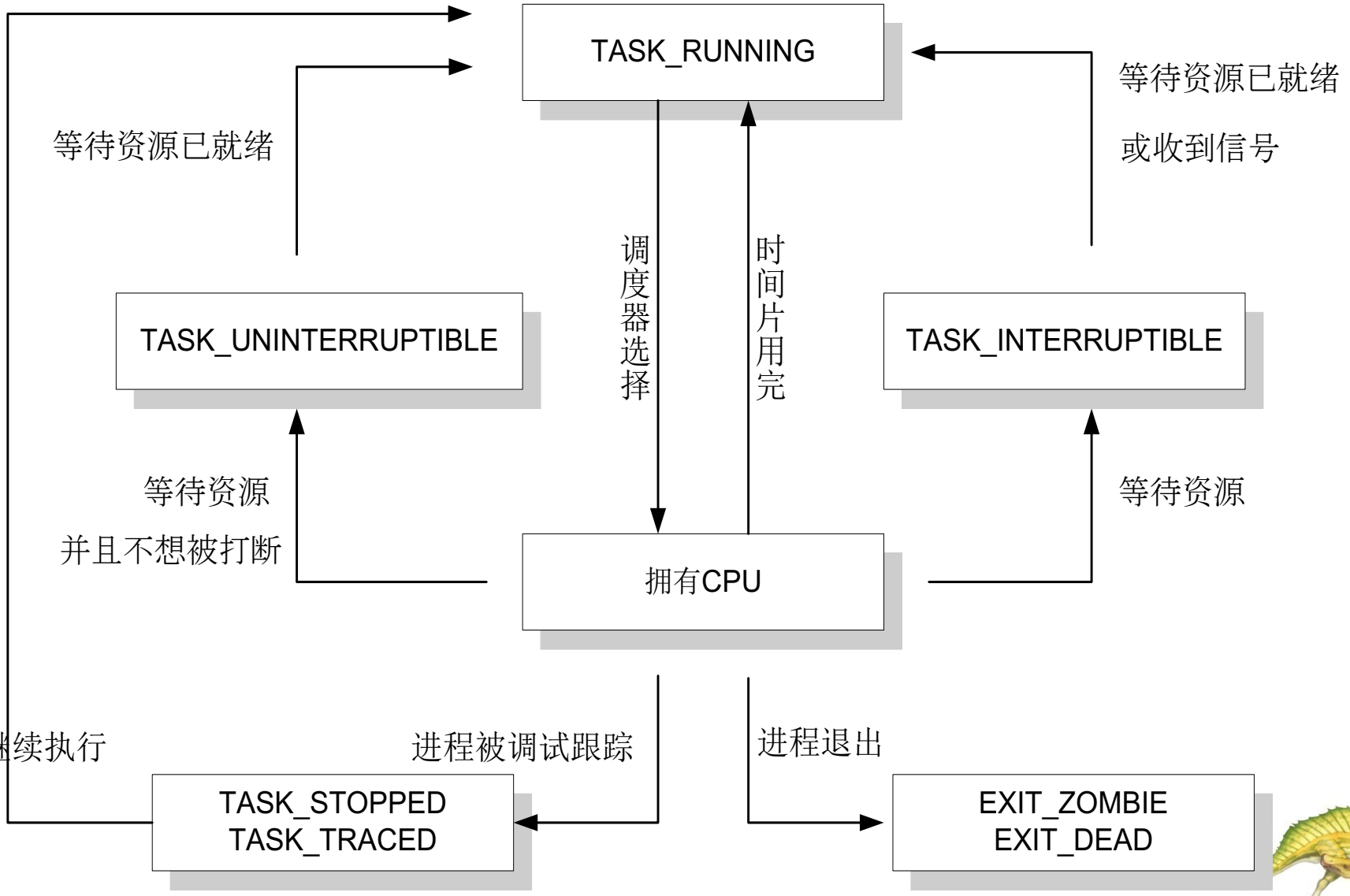
■**TASK_STOPPED**：进程被暂停，一般当进程收到下列信号之一时进入这个状态：SIGSTOP，SIGTSTP，SIGTTIN或者SIGTTOU。通过其它进程的信号才能唤醒。

■**TASK_TRACED**：进程被跟踪，一般在调试的时候用到。

■**EXIT_ZOMBIE**：正在终止的进程，等待父进程调用wait4()或者waitpid()回收信息。是进程结束运行前的一个过度状态（僵死状态）。虽然此时已经释放了内存、文件等资源，但是在内核中仍然保留一些这个进程的数据结构（比如task_struct）等待父进程回收。

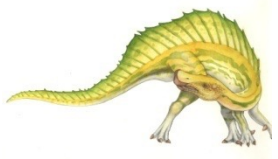
■**EXIT_DEAD**：进程消亡前的最后一个状态，表示父进程已经获得了该进程的记账信息，该进程可以被销毁了。

Linux进程状态转换



Linux进程控制块-task_struct

- 进程控制块PCB是名字为task_struct的数据结构。进程的task_struct是进程存在的唯一标志。
- 当一个进程被创建时，系统就为该进程建立一个task_struct进程控制块。当进程运行结束时，系统撤消该进程的task_struct 。
- Linux在内存空间中开辟了一个专门的区域存放所有进程的task_struct 。系统中的最多进程数目受task 数组大小的限制。 ulimit –u显示用户可以同时执行的最大进程个数
- task_struct结构是一个复杂的结构，32位机器上它大约2k字节，其各个成员用来准确描述进程在各方面的信息.主要有以下几个部分（include/linux/sched.h文件）：



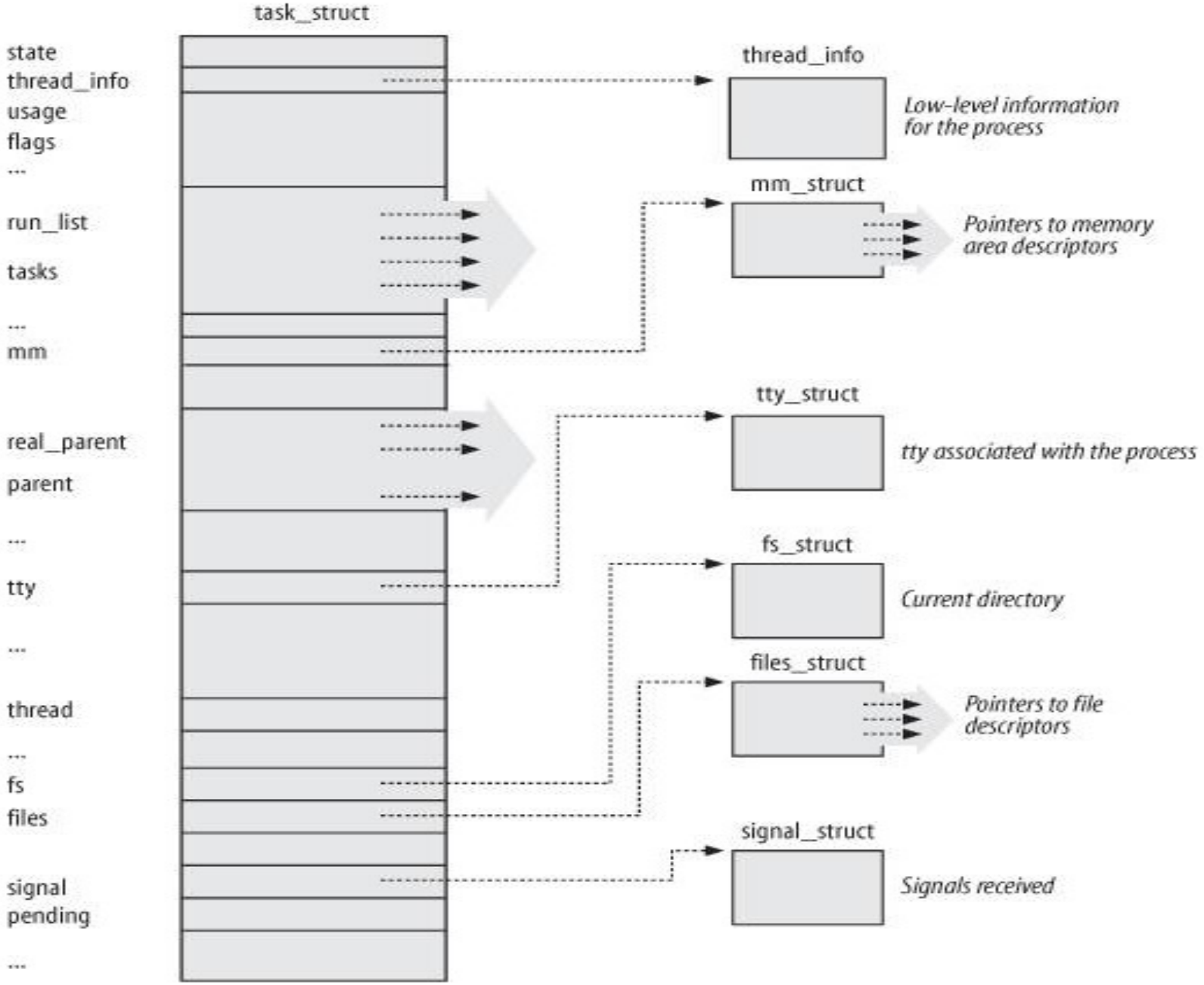
task_struct

include/linux/sched.h, line 1224（3.17.2）

```
1224 struct task_struct {
1225     volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
1226     void *stack;
1227     atomic_t usage;
1228     unsigned int flags; /* per process flags, defined below */
1229     unsigned int ptrace;
1230
1231     #ifdef CONFIG_SMP
1232     struct list_node wake_entry;
1233     int on_cpu;
1234     struct task_struct *last_wakee;
1235     unsigned long wakee_flips;
1236     unsigned long wakee_flip_decay_ts;
1237     ….
```

https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/include/linux/sched.h?id=refs/tags/v3.12.3　1023行开始

The Linux process descriptor

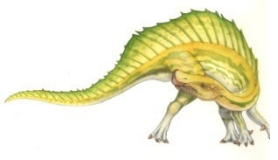


task_struct

- 1. 进程的状态和标志
  - volatile long state //进程的状态
  - long exit_state //退出时状态
  - unsigned long flags //进程的标志

这些标志的含义分别为：

PF_ALIGNWARN	标志打印“对齐”警告信息。
PF_STARTING	进程正被创建。
PF_EXITING	标志进程开始关闭。
PF_DEAD	标志进程已经完成退出。
PF_FORKNOEXEC	进程刚创建，但还没执行。
PF_SUPERPRIV	超级用户特权标志。
PF_DUMPCORE	标志进程是否清空core文件。
PF_SIGNALED	标志进程被信号杀出。
PF_MEMALLOC	进程分配内存标志。
PF_FLUSHER	负责磁盘写回。
PF_USED_MATH	如果没有置位，那么使用fpu之前必须初始化。
PF_FREEZE	由于系统要进入休眠，进程正在被停止。
PF_NOFREEZE	系统睡眠的时候，这个进程不能被停止。
PF_FROZEN	系统要进入睡眠，进程被停止。
PF_FSTRANS	在一个文件系统事务之中。
PF_KSWAPD	kswapd内核线程。
PF_SWAPOFF	在换出页的过程中。
PF_LESS_THROTTLE	尽可能少把我换出。
PF_SYNCWRITE	负责把脏页写回。
PF_BORROWED_MM	内核线程借用进程的mm。
PF_RANDOMIZE	随机虚拟地址空间。
PF_SWAPWRITE	允许被写到swap中去。

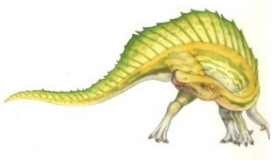


task_struct

2.进程的标识

- int pid //进程标识号

- unsigned short uid, gid //用户标识号，组标识号
- unsigned short euid, egid //用户有效标识号，组有效标识号
- unsigned short suid, sgid //用户备份标识号，组备份标识号
- unsigned short fsuid, fsgid //用户文件标识号，组文件标识号
- .....



task_struct

3.进程的族亲关系

- struct task_struct *p_opptr //指向祖先进程PCB的指针
- struct task_struct *p_pptr //指向父进程PCB的指针
- struct task_struct *p_cptr //指向子进程PCB的指针
- struct task_struct *p_ysptr //指向弟进程PCB的指针
- struct task_struct *p_osptr //指向兄进程PCB的指针

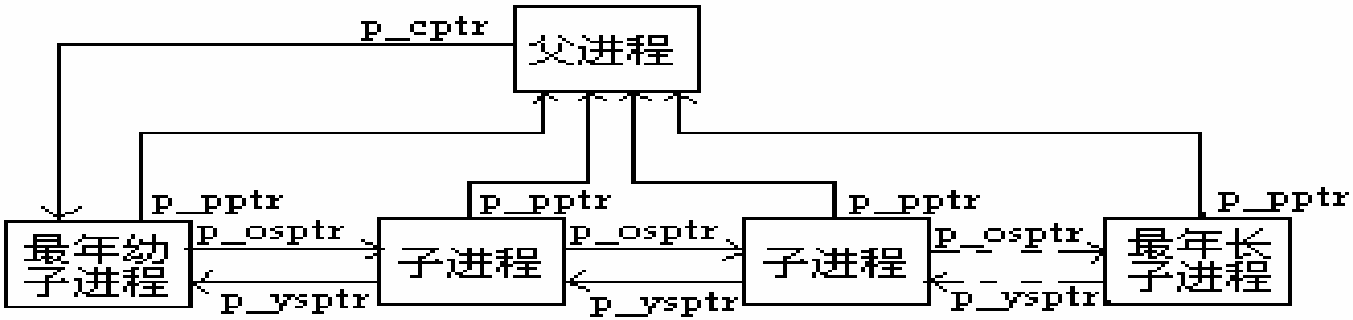
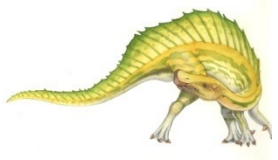


图　　进程的族亲关系

task_struct

- 4. 进程间的链接信息
  - struct task_struct *next_task //指向下一个PCB的指针
  - struct task_struct *prev_task //指向上一个PCB的指针
  - struct task_struct *next_run //指向可运行队列的下一个PCB的指针
  - struct task_struct *prev_run //指向可运行队列的上一个PCB的指针
- 5.进程的调度信息
  - long counter //时间片计数器
  - long nice //进程优先级
  - unsigned long rt_priority //实时进程的优先级
  - unsigned long policy //进程调度策略
  - ……

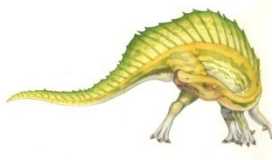


task_struct

- 6.进程的时间信息
  - long start_time //进程创建的时间
  - long utime //进程在用户态下耗费的时间
  - long stime //进程在核心态下耗费的时间
  - long cutime //所有子进程在用户态下耗费的时间
  - long cstime //所有子进程在核心态下耗费的时间
  - unsigned long timeout //进程申请延时
  - ……
- 7.进程的虚存信息
  - struct mm_struct *mm //进程的虚存信息
  - struct desc_struct *ldt //进程的局部描述符表指针
  - unsigned long saved_kernel_stack //核心态下堆栈的指针
  - unsigned long kernel_stack_page //核心态下堆栈的页表指针

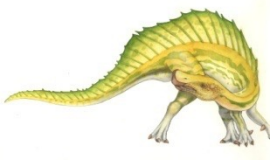
task_struct

- 8. 进程的文件信息
  - struct fs_struct *fs //进程的可执行映象所在的文件系统
  - struct files_struct *files //进程打开的文件
- 9.与进程间通信有关的信息
  - unsigned long signal //进程接收到的信号
  - unsigned long blocked //阻塞信号的掩码
  - struct signal_struct*sig //信号处理函数表的指针
  - int exit_signal //进程终止的信号
  - struct sem_undo *semundo //进程要释放的信号量
  - struct sem_queue *semsleeping //与信号量操作相关的等待队列
- 10. 其它信息
  - int errno //系统调用的出错代码
  - long debugreg[8] //进程的8个调试寄存器
  - char comm[16] //进程接收到的信号



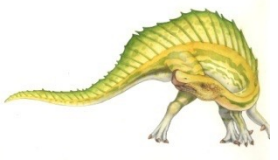
进程标识符PID

- 每个进程都有一个唯一的标识符，内核通过这个标识符来识别不同的进程 。
  - 进程标识符PID（*Process IDentifier*） 也是内核提供给用户程序的接口，用户程序通过PID对进程发号施令；
  - PID是32位的无符号整数，它被顺序编号，最大值为32768。
- 每个进程都属于某个用户、组。
  - task_struct结构中定义有用户标识符UID（*User Identifier*）和组标识符GID（*Group Identifier*）
  - 这两种标识符用于系统的安全控制
  - 系统通过这两种标识符控制进程对系统中文件和设备的访问。



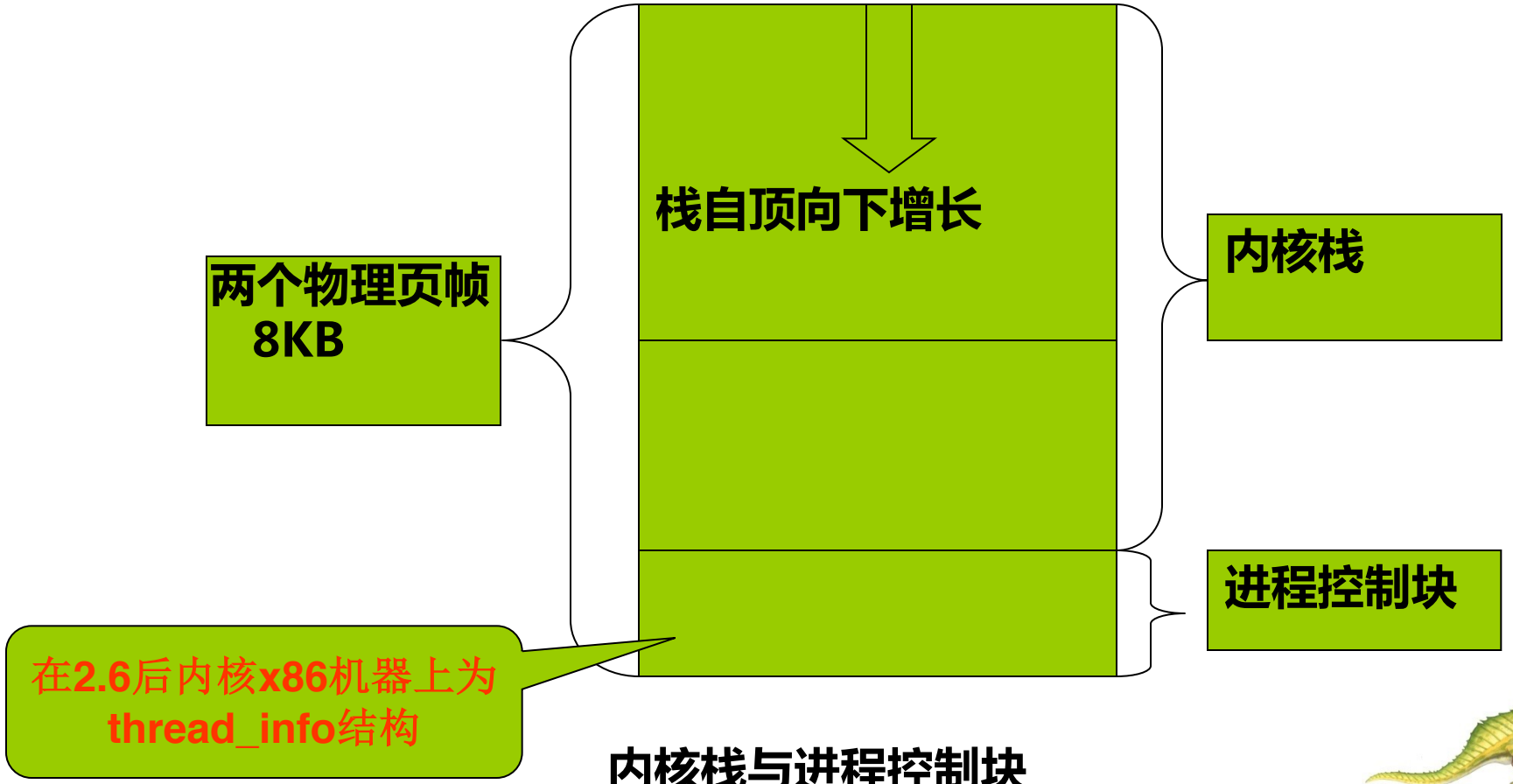
task_struct的存放

- 进程拥有两个栈，用户模式栈与内核模式栈，分别在相应模式下使用。
- 进程控制块和进程内核栈的空间分配在一起(2.4版内核)，内核为它们分配两个连续的物理页面。

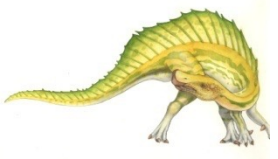


task_struct的存放

- 内核栈与进程控制块如图所示：



内核栈与进程控制块



task_struct的存放

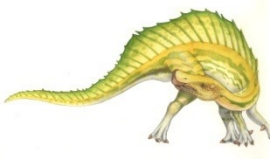
- 把task_struct与内核栈放在一起具有以下好处：
  - (1) 内核可以方便而快速地找到PCB，用伪代码描述如下：

    X86:堆栈段寄存器SS

    p = (struct task_struct *) STACK_POINTER & 0xffffe000

    屏蔽后13位，即8K
  - (2) 避免在创建进程时动态分配额外的内存

- 在Linux内核中，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



进程列表
struct list_head tasks

任务ID

```
pid_t pid;
pid_t tgid;
struct task_struct *group_leader;
```

亲缘关系

```
struct task_struct __rcu *real_parent;
struct task_struct __rcu *parent;
struct list_head children;
struct list_head sibling;
```

任务状态

```
volatile long state;
int exit_state;
unsigned int flags;
```

权限

```
const struct cred __rcu *real_cred;
const struct cred __rcu *cred;
```

运行统计

```
u64 utime;
u64 stime;
unsigned long nvcsw;
unsigned long nivcsw;
u64 start_time;
real_start_time;
```

调度相关

```
int on_rq;
int prio;
int static_prio;
int normal_prio;
const struct sched_class *sched_class;
struct sched_entity se;
unsigned int policy;
```

信号处理

```
struct signal_struct *signal;
struct sighand_struct *sighand;
struct sigpending pending;
```

内存管理

```
struct mm_struct *mm;
struct mm_struct *active_mm;
```

文件与文件系统

```
struct fs_struct *fs;
struct files_struct *files;
```

内核栈

```
struct thread_info thread_info;
void *stack;
```

2.6的内核堆栈和task_struct

- 2.4的设计的隐患也一直被很多内核黑客所讨论：
  - 如果task_struct越来越大怎么办？
  - 如果内核堆栈压得太多（比如函数调用层次太深）怎么办？

- 2.6的内核中采取办法：
  - 把task_struct从这部分空间中移走：在2.6的内核中，抽象出一个thread_info的结构（把最经常被entry.S访问的变量抽出来）。

2.6的内核堆栈和task_struct

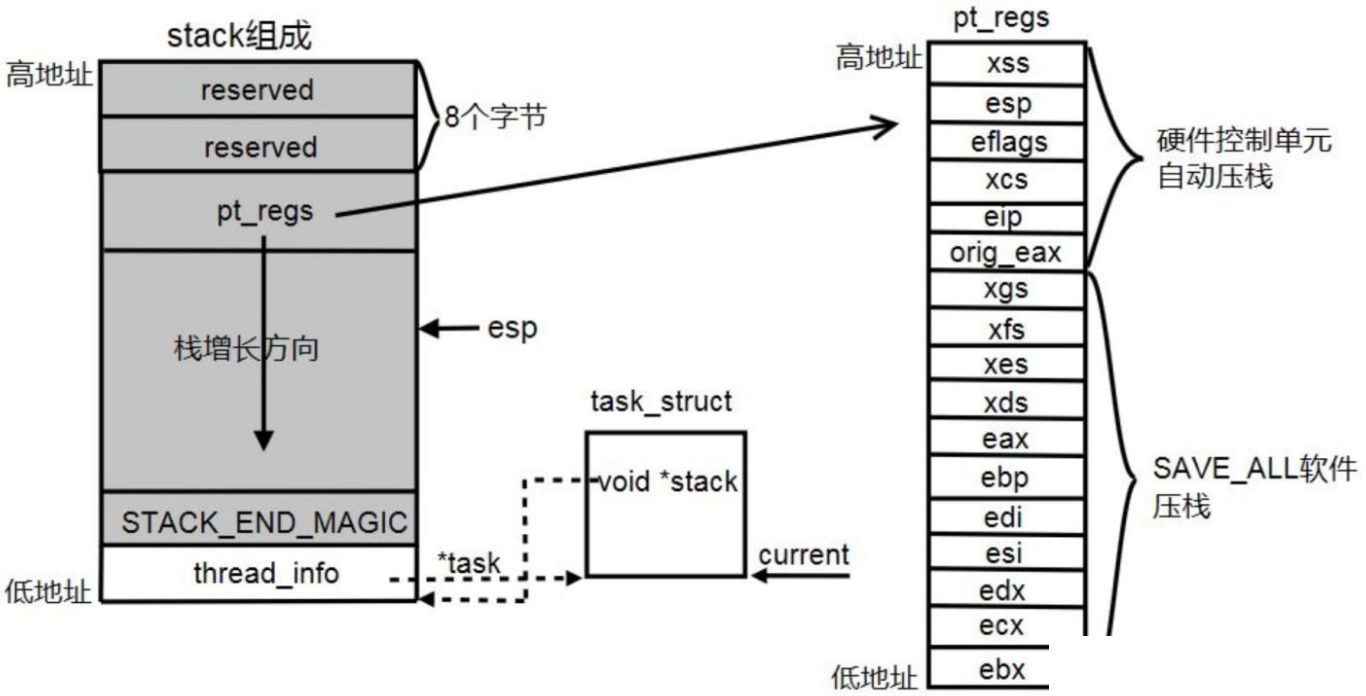
- include/asm-i386/thread_info.h, line 28

```
28 struct thread_info {
29      struct task_struct      *task;          /* main task structure */
30      struct exec_domain      *exec_domain;   /* execution domain */
31      unsigned long           flags;          /* low level flags */
32      unsigned long           status;         /* thread-synchronous flags */
33      __u32                   cpu;            /* current CPU */
34      int                     preempt_count;  /* 0 => preemptable, <0 => BUG */
35
36
37      mm_segment_t            addr_limit;     /* thread address space:
38                                                 0-0xBFFFFFFF for user-thead
39                                                 0-0xFFFFFFFF for kernel-thread
40                                              */
41      void                    *sysenter_return;
42      struct restart_block    restart_block;
43
44      unsigned long           previous_esp;   /* ESP of the previous stack in case
45                                                 of nested (IRQ) stacks
46                                              */
47      __u8                    supervisor_stack[0];
48 };
```

2.6的内核堆栈和task_struct

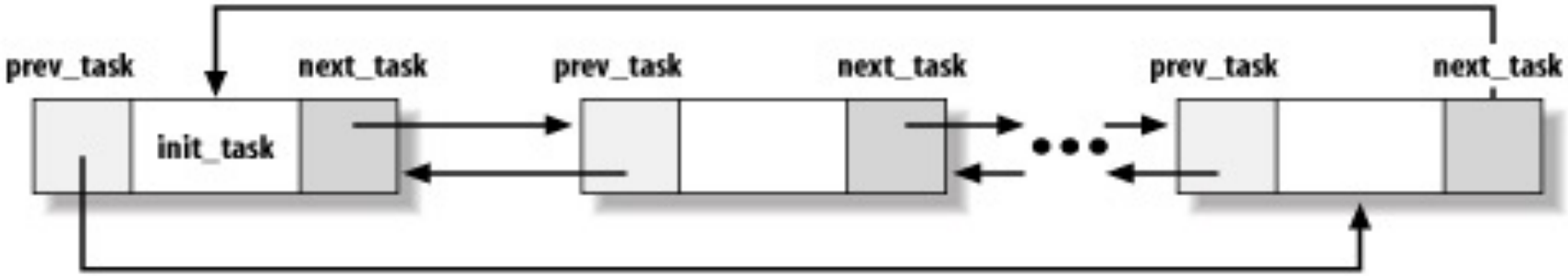
- thread_info代替了原先task_struct的位置，跟内核堆栈放在一块，thread_info中放置一个指向task_struct的指针，如下图。

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



进程的组织方式-进程链表

- A circular doubly linked list



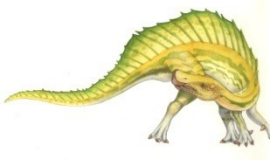
- 宏for_each_task()遍历整个进程链表

```
#define for_each_task(p) \
    for (p = &init_task ; (p = p->next_task) != &init_task ; )
```

PID=0，为swapper空闲进程

- 组成HASH表，task_struct中：

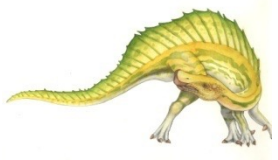
```
/* PID/PID hash table linkage. */
struct pid_link pids[PIDTYPE_MAX];
```



进程的组织方式- 可运行队列

- 把可运行状态的进程组成一个双向循环链表，也叫可运行队列（runqueue）
- 在task_struct结构中定义了两个指针。

  struct task_struct *next_run, *prev_run;
- init_task起链表头的作用
- 在调度程序运行过程中，允许队列中加入新出现的可运行态进程，新出现的可运行态进程插入到队尾

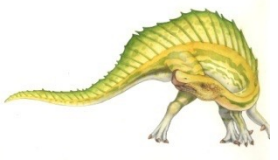


进程的组织方式-等待队列

- 等待队列表示一组睡眠的进程
- 等待队列定义为如下结构：

```
struct wait_queue {
  struct task_struct * task;
  struct wait_queue * next;
};
```

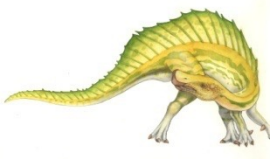
- 如何让正在运行的进程等待某一特定事件？ Linux内核中实现了sleep_on()函数,请给出该函数的实现。
- 如果要让等待的进程唤醒，就调用唤醒函数wake_up（），它让待唤醒的进程进入TASK_RUNNING状态。



Linux进程的创建

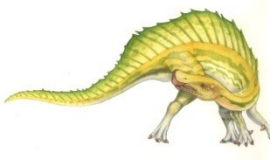
进程创建的原理：

- 系统创建的第1个真正进程是init进程，pid=1。
- 系统中所有的进程都是由进程使用系统调用fork()创建的。
- 子进程被创建后继承了父进程的资源。子进程共享父进程的虚存空间（只读方式）。
- 写时拷贝 (copy on write)：子进程在创建后共享父进程的虚存内存空间，只是在两个进程中某一个进程需要向虚拟内存写入数据时才拷贝相应部分的虚拟内存。
- 子进程在创建后执行的是父进程的程序代码。子进程通过调用exec系列函数执行真正的任务。

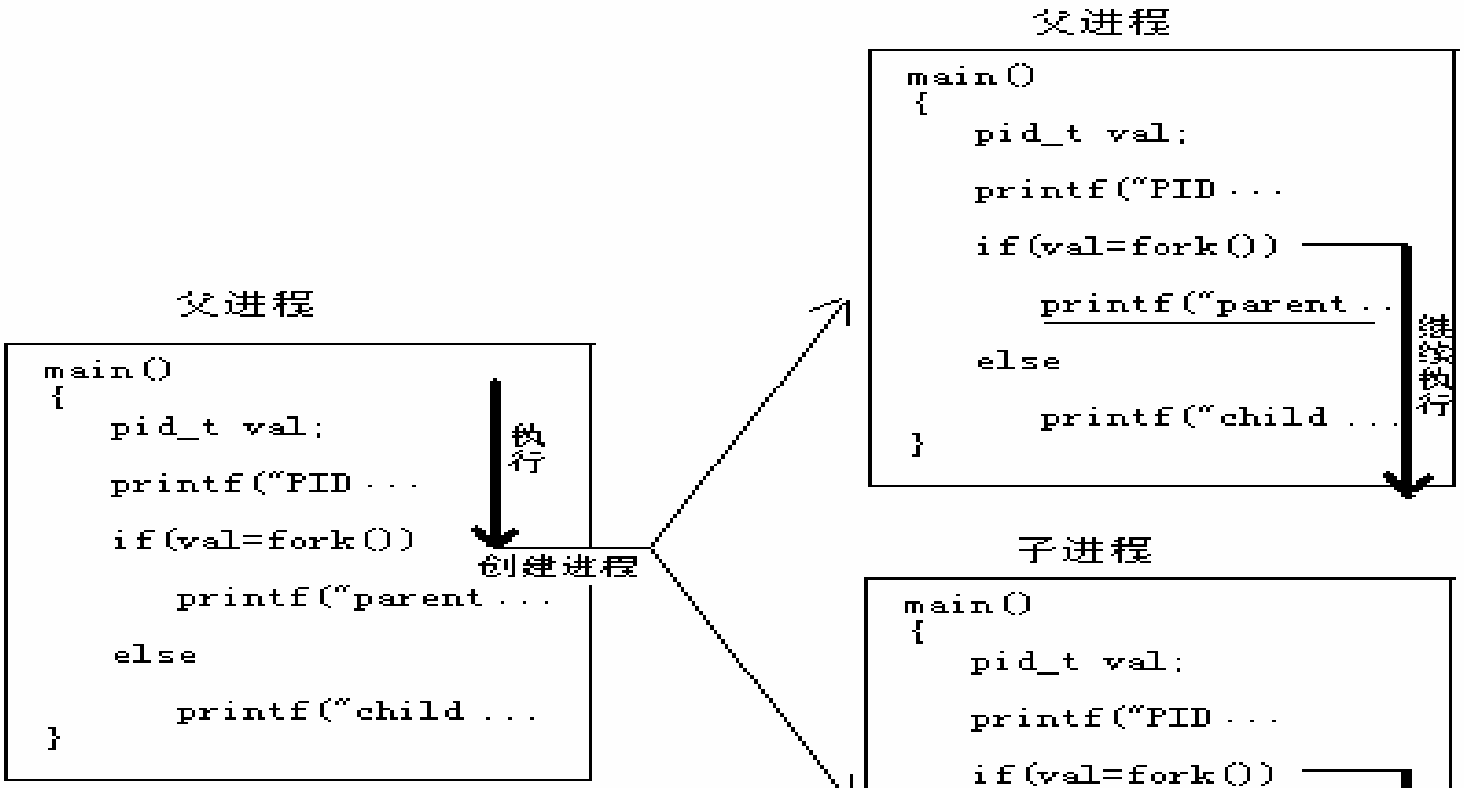


相关系统调用

- **fork()** 通过复制调用进程来建立新的进程，是最基本的进程建立过程
- **exec()** 包括一系列系统调用，它们都是通过用一个新的程序覆盖原来的内存空间，实现进程的转变
- **wait()** 提供初级的进程同步措施，能使一个进程等待，直到另外一个进程结束为止。
- **exit()** 该系统调用用来终止一个进程的运行



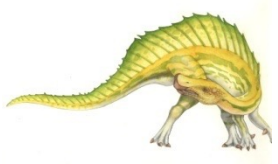
fork实例


父进程
main()
{
pid_t val;
printf("PID ...
if(val=fork())
printf("parent ...
else
printf("child ...
}
执行
创建进程
父进程
main()
{
pid_t val;
printf("PID ...
if(val=fork())
printf("parent ...
else
printf("child ...
}
继续执行
子进程
main()
{
pid_t val;
printf("PID ...
if(val=fork())
printf("parent ...
else
printf("child ...
}
继续执行

fork实例

- 父进程执行fork()返值是子进程的PID值，子进程执行fork()的返值是0。

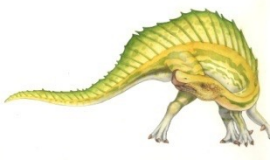
```
#include <sys/type.h>/* 提供类型pid_t的定义,在PC机上与int型相同*/
#include <unistd.h> /* 提供系统调用的定义 */
#include <stdio.h>
main()
{pid_t pid;
 printf("PID before fork():%d\n",(int)getpid());
  pid=fork();
  /*此时已经有两个进程在同时运行*/
 if(pid==0) /*子进程执行进入此部分*/
        printf(" child process PID:%d\n",(int)getpid());
  else   /*父进程*/
        printf(" parent process PID:%d\n",(int)getpid());
}
```



fork

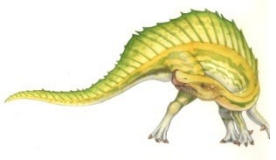
- 该程序的执行结果：
  - PID before fork():2367
  - parent process PID:2367
  - child process PID:2368

再运行一遍，输出结果可能不同。为什么？



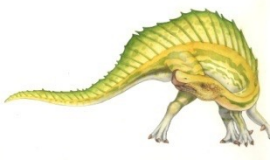
fork()函数

- 进程创建的过程:
  1. 为新进程分配task_struct内存空间;
  2. 把父进程task_struct拷贝到子进程的task_struct;
  3. 为新进程在其虚拟内存建立内核堆栈;
  4. 对子进程task_struct中部分进行初始化设置
  5. 把父进程的有关信息拷贝给子进程，建立共享关系;
  6. 把子进程的counter设为父进程counter值的一半;
  7. 把子进程加入到可运行队列中;
  8. 结束do_fork()函数返回PID值.



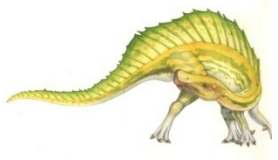
do_fork()

- 三个系统调用sys_clone(),sys_vfork(),sys_fork() 可以实现创建子进程，这三个系统调用最终都会调用do_fork()函数完成主要工作。
- 三个系统调用的源程序在arch/i386/kernel/process.c中
- do_fork()函数的第一个参数clone_flags可由多个标志位组成，常见的标志位有：
  - CLONE_VM 子进程父进程共享进程空间;
  - CLONE_FS 子进程父进程共享文件系统信息;
  - CLONE_FILES 子进程父进程共享打开的文件;
  - CLONE_VFORK 如果父进程想使子进程释放空间时唤醒它，则置该位。
  - CLONE_SIGHAND 父进程和子进程共享信号处理函数表
  - CLONE_PTRACE 如果父进程被跟踪的话，那么子进程也被跟踪。
  - ……



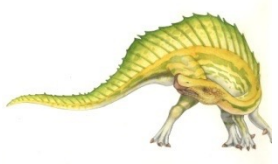
do_fork() (续)

- sys_clone()对应的clone_flags可能是多个标志位的组合，取决于具体情况。
- sys_fork()对应的clone_flags值是SIGCHILD。SIGCHILD的作用是子进程终结或暂停时给父进程发信号。
- sys_vfork()对应的clone_flags值是CLONE_VFORK|CLONE_VM|SIGCHILD。
  - vfork()是一个老的函数调用，子进程共享父进程的空间，包括页表，父进程被挂起直到子进程执行exec系列函数或子进程退出时。在合适的场合，较之fork()时的“写时拷贝” 策略vfork()无疑开销更小。



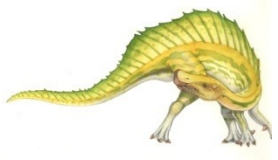
do_fork() (续)

- do_fork()的执行过程(源代码在kernel/fork.c文件中):
  - 1) 调用alloc_task_struct()分配子进程task_struct空间。严格地讲，此时子进程还未生成。
  - 2) 把父进程task_struct的值全部赋给子进程task_struct。
  - 3) 检查是否超过了资源限制，如果是，则结束并返回出错信息。更改一些统计量的信息。
  - 4) 修改子进程task_struct的某些成员的值使其正确反映子进程的状况，如进程状态被置成TASK_UNINTERRUPTIBLE。
  - 5) 调用get_pid( )函数为子进程得到一个pid号。



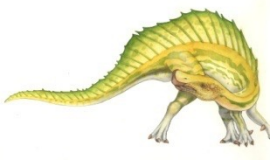
do_fork() (续)

- 6)共享或复制父进程文件处理、信号处理及进程虚拟地址空间等资源。
- 7) 调用copy_thread( )初始化子进程的内核栈,内核栈保存了进程返回用户空间的上文。此处与平台相关，以i386为例，其中很重要的一点是寄存器eax值的位置被置0,这个值就是执行系统调用后子进程的返回值。
- 8) 将父进程的当前的时间配额counter分一半给子进程。
- 9) 利用宏SET_LINKS将子进程插入所有进程都在其中的双向链表。调用hash_pid（）,将子进程加入相应的hash队列。

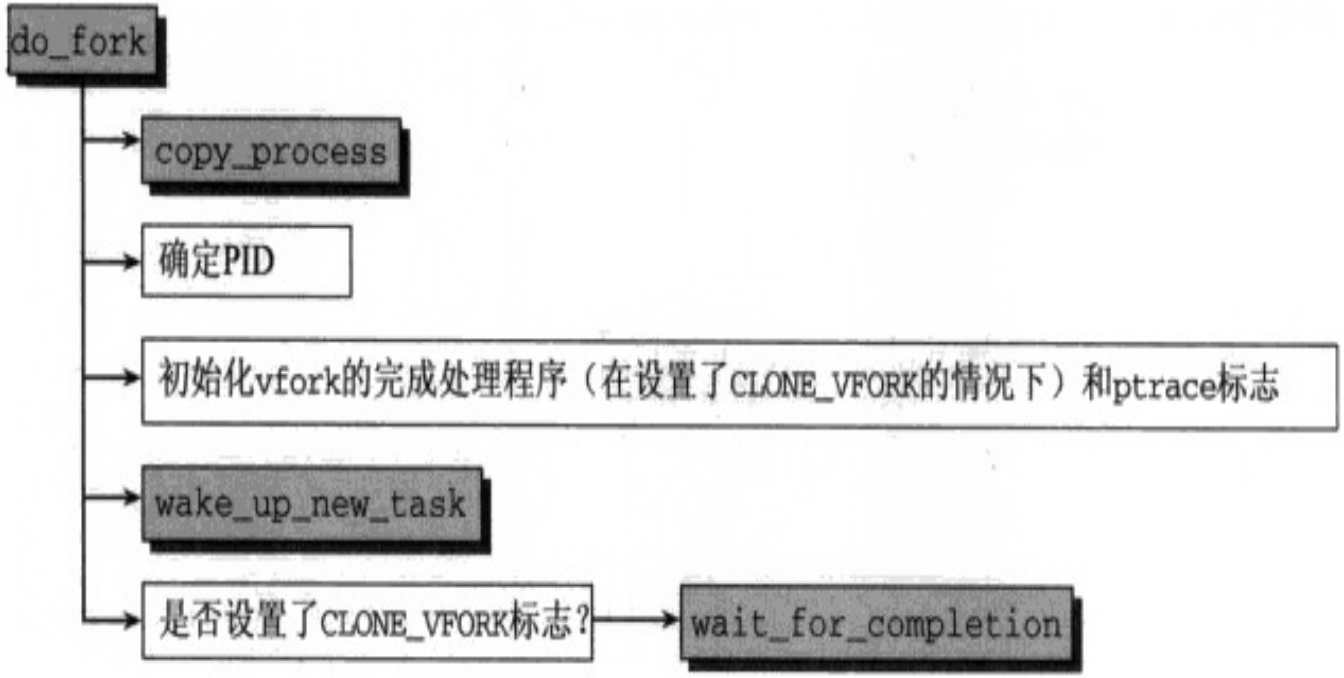


do_fork() (续)

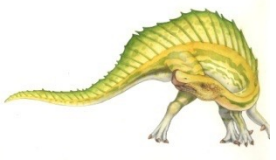
- 10) 调用wake_up_process( ),将该子进程插入可运行队列。至此，子进程创建完毕,并在可运行队列中等待被调度运行。
- 11) 如果clone_flags包含有CLONE_VFORK 标志，则将父进程挂起直到子进程释放进程空间。进程控制块中有一个信号量vfork_sem可以起到将进程挂起的作用。
- 12) 返回子进程的pid值，该值就是系统调用后父进程的返回值。



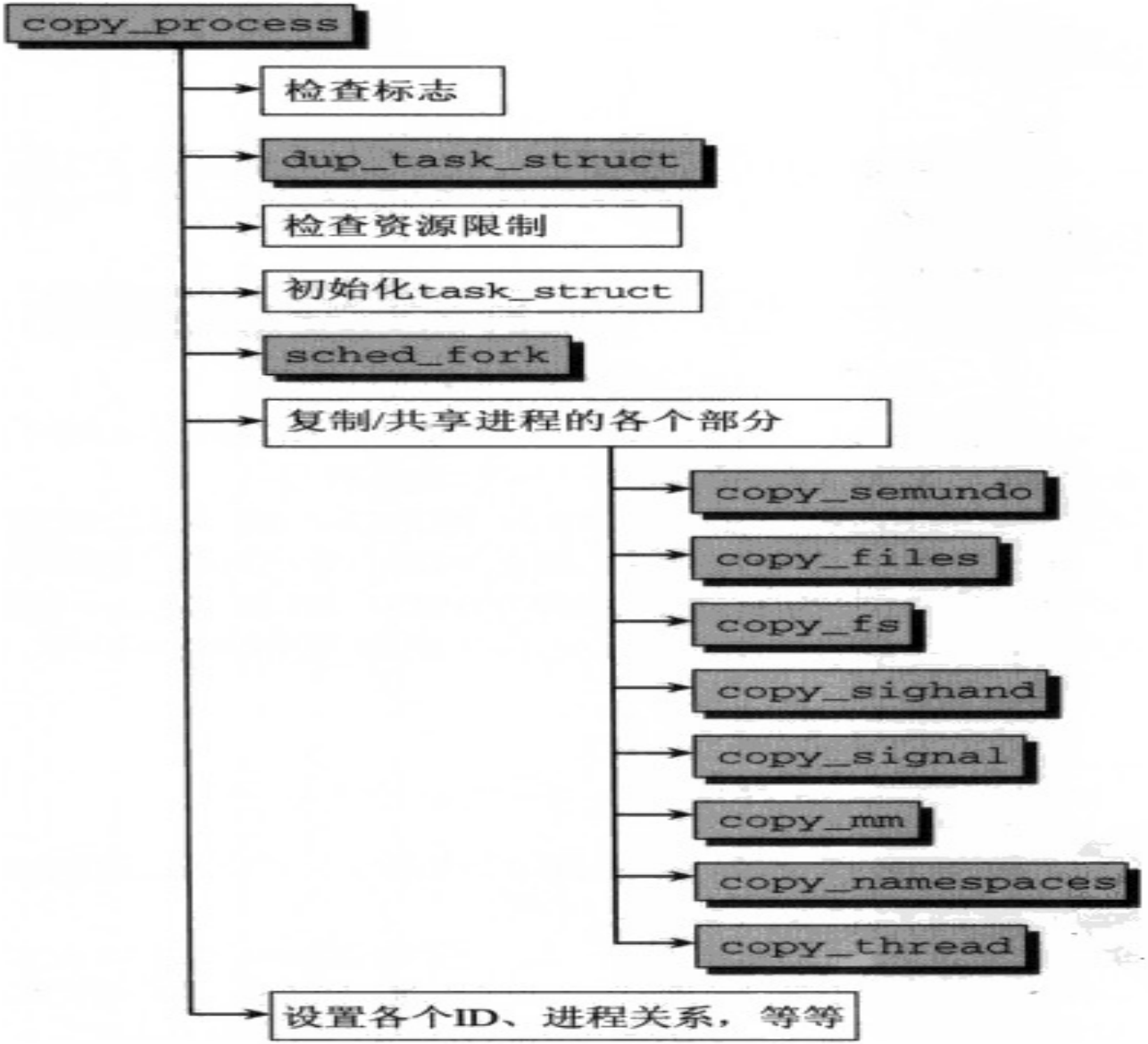
do_fork代码流程

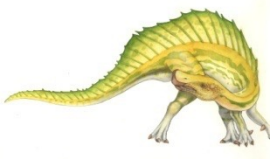


- fork分析请看“边干边学”教材：234-246



copy_process代码流程



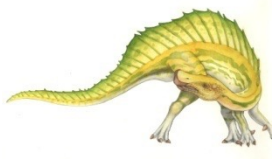


Linux进程的执行

- 在系统中创建一个进程的目的是需要该进程完成一定的任务，需要该进程执行它的程序代码。
- 在Linux系统中，使程序执行的唯一方法是使用系统调用exec()。
- 系统调用exec()有多种使用形式，称为exec()族，它们只是在参数上不同，而功能是相同的。如：

```
int execl(const char * path, const  char *arg0,...,
    const  char *argn,(char*)0)
```

  - path: 要执行的程序文件的完整路径名
  - arg0: 要执行程序的文件名或命令名
  - arg1,...,argn: 执行程序所需的参数，



Linux进程的执行(续)

- exec有一系列的系统调用：

```
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char*path, const char *arg , ..., char *
    const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *filename, char *const argv [], char
    *const envp[]);
```

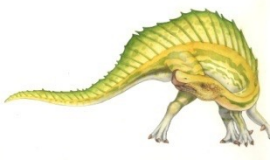
  - 前面几个函数都是通过调用execve来实现的。请看教材中execve实现的分析。

Linux进程的执行(续)

- exec()函数返值：
  - 该系统调用将引起另一个程序的执行，成功调用后并不需要返回。
  - 调用失败时，返值为-1。
- 例：

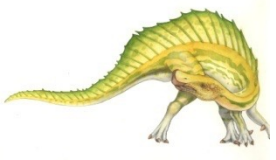
```
#include <stdio.h>
main( )
{
execl("/bin/ls","ls","-l",0);
printf("Can only get here on error/n");
}
```

- 在Linux多进程程序设计中，一个进程使用fork()建立子进程后，让子进程执行另一个程序的方法也是通过使用exec()系统调用。



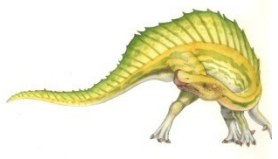
Linux进程的等待和唤醒

- Linux的等待态，又称为睡眠态。等待态的进程又称睡眠进程，它们被加到等待该事件的等待队列中。
- 当等待的事件出现后，睡眠进程被唤醒，然后加入到可运行队列中，
- Linux内核中设置了用于进程睡眠、进程唤醒和对等待队列进行管理的控制机制，它们主要是若干内核函数和数据结构。



等待队列及操作

- 为了便于对等待状态的进程进行管理和控制， Linux设置了等待队列和面向等待队列的操作函数。
- 当运行状态进程因等待某个事件成为等待状态后，由内核函数把它插入到等待队列中。
- 当进程等待的事件发生后，内核函数再把它从等待队列中删除。
- Linux中对应于每种等待事件都有一个等待队列。等待状态进程根据等待事件的不同排在不同的等待队列中.
- 系统中同时存在着多个等待队列。各个等待队列都是使用wait_queue结构体链接而成的循环链表



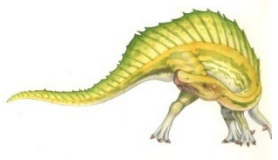
等待队列及操作(续)

```
struct wait_queue {
        struct task_struct *task;
        struct wait-queue *next;
    };
```

- task:指向等待队列中进程的任务结构体
- next:指向队列中下一个wait_queue结构体。

- 系统中每个等待队列都有一个指向该队列的指针，它被定义为静态全局指针变量，并被初始化为NULL，表示空队列。
- 例如，某个event等待队列的指针定义如下：

```
static struct wait_queue
* event_queue = NULL;
```



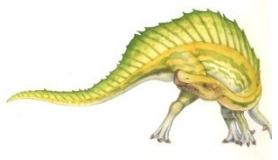
等待队列及操作(续)

- 对等待队列的操作的两个内核函数：

```
extern inline void
    add_wait_queue(
    struct wait_queue ** p,
    struct wait_queue * wait)
extern inline void
    remove_wait_queue(
    struct wait_queue ** p,
    struct wait_queue * wait)
```

p：指向要操作的等待队列。wait：指向要加入或移出的wait_queue结构体的指针。

- 在函数中对等待队列进行操作时禁止中断（关中断），使对等待队列的加入或删除操作可以互斥地进行。

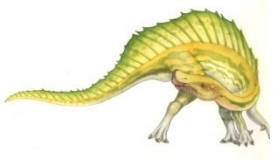


进程等待的机制

- Linux提供了多种使进程进入等待态的机制，如：wait、sleep等
- 内核函数sys_wait4()的功能就是使进程进入等待态。
- 当前进程在运行过程中需要等待它的子进程终止时，调用该函数使其状态从运行态转换成可中断的等待态(INTERUPTIBLE)，并加入到等待队列中。
- 当被等待的子进程运行终止后，发出信号通知处于等待态的父进程，把父进程唤醒，使父进程继续运行。

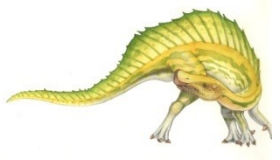
```
int sys_wait4(pid_t pid,unsigned int * stat_addr,
int options,struct rusage * ru)
```

  - **pid**：
    - pid > 0 等待进程标识为pid的子进程。
    - pid = 0 等待与当前进程属于同一个用户组的子进程。
    - pid < -1 等待最早创建的子进程终止。
    - pid = -1 等待所有子进程的终止。



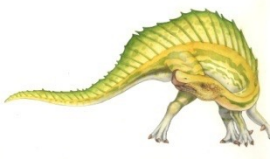
进程等待的机制（续）

- **state_addr**：状态变量的地址：
  - 函数执行时的状态代码存放在该变量中，待函数返回时，可以通过对这个变量的值的检测，了解到子进程终止的原因是正常终止、还是因信号被终止或因信号而暂停。
- **option**：控制选项
  - 用于指定调用该函数的进程中是等待子进程的结束，还是不需等待子进程的结束而继续运行。
- **ru**：传递使用资源结构体。
  - 在执行sys_wait4( )过程中，把进程及其子进程使用资源的有关信息写入该结构体中。在函数返回后，可以从该结构体中得到这些信息。



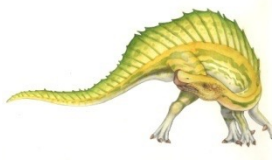
进程的睡眠和唤醒

- 进程的睡眠
  - 当前进程在运行过程中还可以通过系统调用函数sleep_on()和interruptible_sleep_on()使其从运行态转换为等待状态。
  - 把使用这些函数进入的等待态又称为睡眠态。
- 进程的唤醒
  - 在等待队列中处于睡眠状态的进程，在等待的事件发生后需要使用内核函数把它们唤醒。
  - wake_up()：唤醒等待队列中的可中断态和不可中断态的进程
  - wake_up_interruptible()：唤醒可中断态进程。



进程的终止和撤消

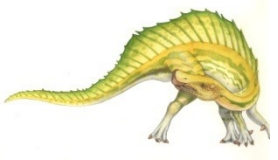
- 进程完成本身的任务，自动终止
- 进程被内核强制终止
- 进程终止：do_exit()
  - (1)设定当前进程的标志
  - (2)释放系统中该进程在各种管理队列中的任务结构体
  - (3)释放进程使用的各种资源
  - (4)把进程的状态转为僵死态
  - (5)把退出码置入任务结构体
  - (6)变更进程族亲关系
  - (7)执行进程，选择下一个使用CPU的进程。



Linux内核线程

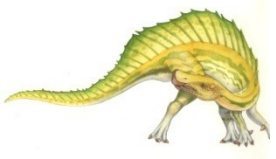
- Linux把线程和进程一视同仁，每个线程拥有唯一属于自己的task_struct结构。不过线程本身拥有的资源少，共享进程的资源，如：共享地址空间、文件系统资源、文件描述符和信号处理程序。
- Linux内核线程— 在内核态下创建、独立执行的一个内核函数：

```
int kernel_thread(int (*fn)(void *), void * arg,
                  unsigned long flags)
{   pid_t p;
    p = clone( 0, flags | CLONE_VM );
    if ( p )        /* 父*/
        return p;
    else {          /* 子*/
        fn(arg);   exit(  );
    }  }
```

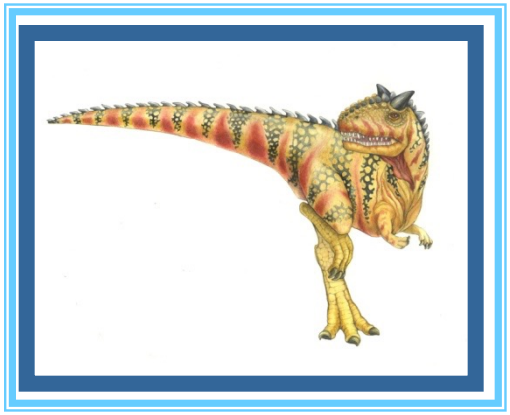


几个特殊身份的内核线程

- swapper（或idle）进程(init_task)，0号进程（*pid=0*）—从无到有诞生的第一个线程，执行cpu_idle()函数。它只有在CPU不能执行其他进程时才执行
- init(*pid=1*)进程，既是内核线程也是1号用户进程的。init进程创建和监控操作系统外层所有进程的活动。
- 还有另外几个内核线程，他们是守护进程（运行于后台的系统进程）：
  - kflushd (即bdflush)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出去。
  - kswapd线程：执行内存回收功能。
- 用命令ps aux参看所有进程，包括守护进程。



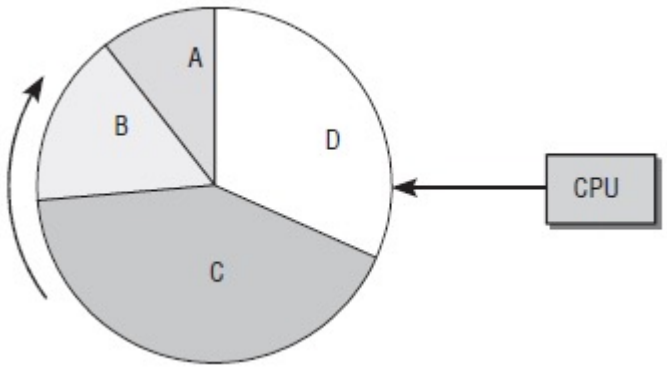
进程调度概述



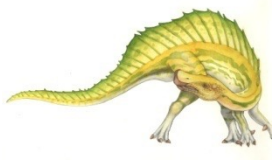
Linux的进程调度

Linux进程调度方式：

- Linux系统采用抢占调度方式。Linux2.6内核是抢占式的，这意味着进程无论是处于内核态还是用户态，都可能被抢占。
- Linux的调度基于分时技术（time-sharing）。对于优先级相同进程进程采用时间片轮转法。
- 根据进程的优先级对它们进行分类。进程的优先级是动态的。



不同的进程分配不同的时间片份额



Linux进程调度策略

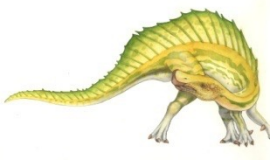
- task_struct中与进程调度相关的一些变量有：

  unsigned long policy: 进程调度策略

- Linux中现在有四种类型的调度策略：

include/linux/sched.h, （3.11.4）

```
#define SCHED_NORMAL          0
#define SCHED_FIFO            1
#define SCHED_RR              2
#define SCHED_BATCH           3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE            5  //最不重要的进程
/* Can be ORed in to make sure the process is reverted back to
    SCHED_NORMAL on fork */
#define SCHED_RESET_ON_FORK   0x40000000
```

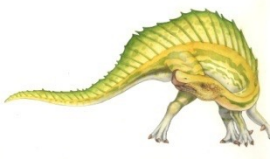


Linux进程调度策略

- Linux的进程分为普通进程和实时进程，实时进程的优先级高于普通进程
- Linux进程的策略*policy*有：

符号常量	意义
SCHED_NORMAL	普通进程的时间片轮转算法
SCHED_FIFO	实时进程的先进先出算法
SCHED_RR	实时进程的时间片轮转算法
SCHED_BATCH	后台处理进程

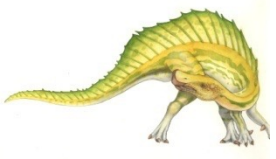
- 普通进程按照SCHED_NORMAL调度策略进行进程调度。
- 实时进程按照SCHED_FIFO或SCHED_RR策略进行调度。
- SCHED_BATCH是2.6新加入的调度策略，这种类型的进程一般都是后台处理进程，总是倾向于跑完自己的时间片，没有交互性。所以对于这种调度策略的进程，调度器一般给的优先级比较低，这样系统就能在没什么事情做的时候运行这些进程，而一旦有交互性的进程需要运行，则立刻切换到交互性的进程，从用户的角度来看，系统的响应性/交互性就很好。



进程的调度算法

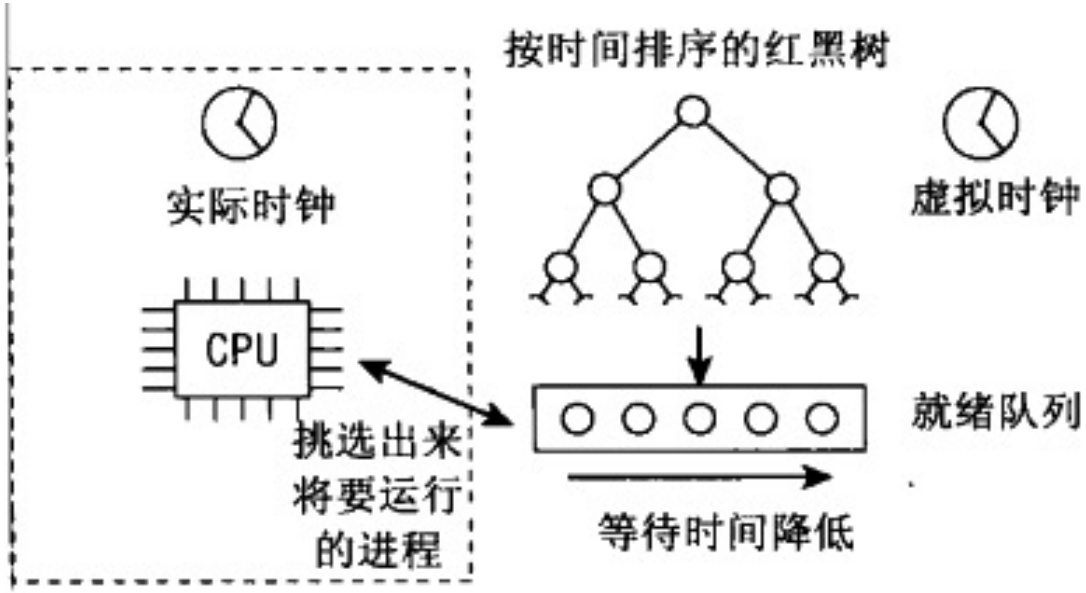
- 调度程序源代码在kernel/sched.c文件中。

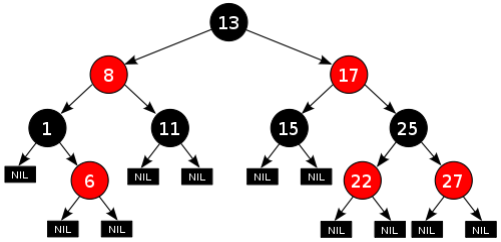
- Linux 2.4的调度算法：
  - 遍历进程可运行队列，算法时间O(n)
  - 时间片，动态优先级
- Linux 2.6.1-2.6.22 调度算法（O(1)算法）：
  - 优先级0-139，使用active和expire两个队列，按照优先级调度，算法时间O(1)
- Linux 2.6.23- 调度算法（CFS算法）：
  - 非实时： CFS (完全公平调度器）进程调度器，使用红黑树选取下一个被调度进程，O(lg N)
  - 实时：优先级队列



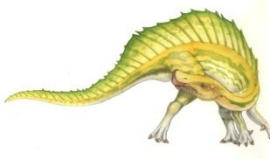
一、CFS调度算法

- 在Linux 2.6.23 内核中发布了一种新的调度算法**CFS算法**（Completely Fair Scheduler 完全公平调度算法）。





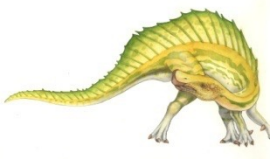
调度器通过将进程在红黑树中排序，跟踪进程的等待时间



二、Linux 2.6 O(1)进程调度

- 特性：
  - O(1)调度算法，调度器开销恒定（与当前系统负载无关），实时性能更好
  - 高可扩展性
  - 新设计的 SMP 亲合方法
  - 优化计算密集型的批处理作业的调度
  - 重载条件下调度器工作更平滑
  - 子进程先于父进程运行等其他改进

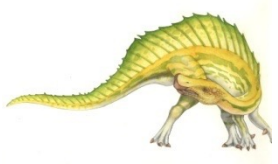
- Linux 2.6 调度系统分析O(1)



Linux 2.6 O(1)进程调度

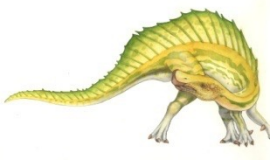
- Linux 2.6的进程设置140个优先级。实时进程优先级为0-99，普通进程优先级100-139的数。0为最高优先权，139为最低优先权。
- 优先级分为静态优先级和动态优先级
- 调度程序根据动态优先级来选择新进程运行。
- 基本时间片：
  - 静态优先权本质上决定了进程的基本时间片，即进程用完了以前的时间片时，系统分配给进程的时间片长度。静态优先权和基本时间片的关系用下列公式确定：

    base time quantum （ms）：
    - （140-static_priority）*20　if static_priority <120
    - （140-static_priority）*5　if static_priority >=120



task_struct 相关域

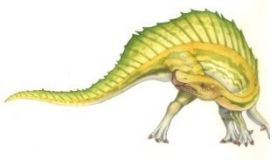
- **sleep_avg**
  - 进程的平均等待时间（nanosecond）
  - 反映交互式进程优先与分时系统的公平共享
  - 值越大，计算出来的进程优先级也越高
- **run_list**
  - 串连在优先级队列中
  - 优先级数组prio_array中按顺序排列了各个优先级下的所有进程
  - 调度器在prio_arrary中找到相应的run_list，从而找到其宿主结构task_struct



task_struct 相关域

time_slice

- 进程的运行时间片剩余大小
- 进程的默认时间片与进程的静态优先级相关
- 进程创建时，与父进程平分时间片
- 运行过程中递减，一旦归零，则重置时间片，并请求调度
- 递减和重置在时钟中断中进行（scheduler_tick()）
- 进程退出时，如果自身并未被重新分配时间片，则将自己剩余的时间片返还给父进程



task_struct 相关域

static_prio静态优先级

- 与2.4版本中的nice值意义相同，但取值区间不同，是用户可影响的优先级
- 通过set_user_nice()来改变
- static_prio= MAX_RT_PRIO + nice + 20
  - MAX_RT_PRIO 定义为100
- 进程初始时间片的大小仅决定于进程的静态优先级
- 核心将100~139的优先级映射到200ms~10ms的时间片上
- 优先级数值越大，优先级越低，分配的时间片越少
- 实时进程的static_prio不参与优先级prio的计算

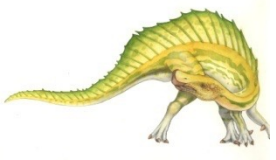
task_struct 相关域

prio 动态优先级

- 相当于 2.4 中 goodness() 的计算结果，在 0~MAX_PRIO-1 之间取值（MAX_PRIO 定义为 140），其中：
  - 0~MAX_RT_PRIO-1 （MAX_RT_PRIO 定义为100）属于实时进程范围；
  - MAX_RT_PRIO~MAX_PRIO-1 属于非实时进程。数值越大，表示进程优先级越小。
- 2.6 中，动态优先级不再统一在调度器中计算和比较，而是独立计算，并存储在进程的 task_struct 中，再通过描述的 priority_array 结构自动排序。
- 普通进程 prio = max (100, min (static priority - bonus + 5, 139))
- Bonus 是范围从 0 到 10 的值，bonus 的值小于 5 表示降低动态优先权以示惩罚，bonus 的值大于 5表示增加动态优先权以示额外奖赏。Bonus 的值依赖于进程过去的情况，说得更准确一些是与进程的平均睡眠时间相关。
- prio 的计算和很多因素相关。
  - 请参阅Understanding The Linux Kernel,3rd Edition

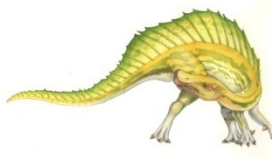
task_struct 相关域

- unsigned long rt_priority
  - 实时进程的优先级
  - sys_sched_setschedule()
  - 一经设定在运行时不变，作为其动态优先级
- prio_array_t *array
  - 记录当前CPU活动的就绪队列
  - 以优先级为序组成数组



runqueue结构（kernel/sched.c）

- runqueue结构是 Linux2.6调度程序最重要的数据结构。系统中的每个 CPU都有它自己的运行队列，所有的 runqueue 结构存放在 runqueues 每 CPU（ per-CPU）变量中。
- 系统中的每个可运行进程属于且只属于一个运行队列。只要可运行进程保持在同一个运行队列中，它就只可能在拥有该运行队列的CPU上执行。但是，可运行进程会从一个运行队列迁移到另一个运行队列。

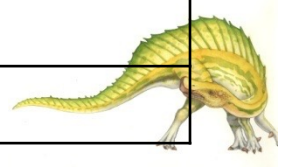


The fields of the runqueue structure

Type	Name	Description
spinlock_t	lock	保护进程链表的自旋锁
unsigned long	nr_running	运行队列链表中可运行进程的数量
unsigned long	cpu_load	基于运行队列中进程的平均数量的CPU负载因子
unsigned long	nr_switches	CPU执行进程切换的次数
unsigned long	nr_uninterruptible	先前在运行队列链表中而现在睡眠在TASK_UNINTERRUPTIBLE 状态的进程的数量（对所有运行队列来说，这些字段的总数才是有意义的）
unsigned long	expired_timestamp	过期队列中最老的进程被插入队列的时间。
unsigned long long	timestamp_last_tick	最近一次定时器中断的时间戳的值
task_t *	curr	当前正在运行进程的进程描述符指针（对本地CPU，它与current相同）
task_t *	idle	当前CPU（this CPU）上交换进程的进程描述符指针。

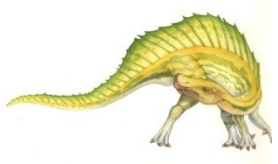
The fields of the runqueue structure

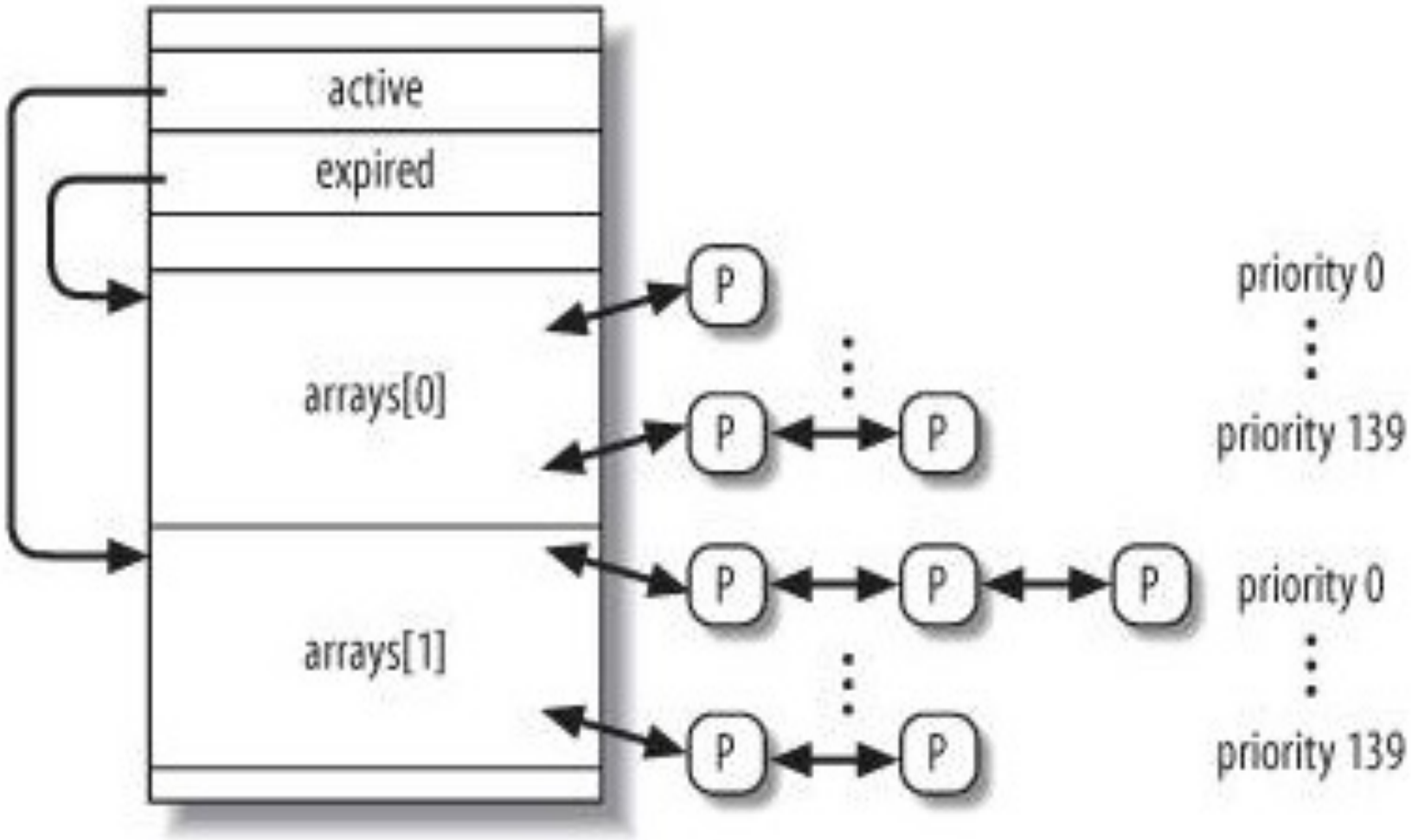
struct mm_struct *	prev_mm	在进程切换期间用来存放被替换进程的内存描述符的地址
prio_array_t *	active	指向活动进程链表的指针
prio_array_t *	expired	指向过期进程链表的指针
prio_array_t [2]	arrays	活动和过期进程的两个集合
int	best_expired_prio	过期进程中静态优先权最高的进程（权值最小）。
atomic_t	nr_iowait	先前在运行队列的链表中而现在正等待磁盘I/O操作结束的进程的数量。
struct sched_domain *	sd	指向当前CPU 的基本调度域
int	active_balance	如果要把一些进程从本地运行队列迁移到另外的运行队列（平衡运行队列），就设置这个标志。
int	push_cpu	未使用
task_t *	migration_thread	迁移内核线程的进程描述符指针。
struct list_head	migration_queue	从运行队列中被删除的进程的链表



The fields of the runqueue structure

- runqueue结构的字段：
  - prio_array_t *active, *expired, arrays[2]
    - 每个CPU均有两个具有优先级的队列，按时间片是否用完分为“活动队列”（active指针所指）和“过期队列”（expired指针所指）。active 指向时间片没用完、当前可被调度的就绪进程，expired 指向时间片已用完的就绪进程。
    - 每一类队列用一个struct prio_array表示（优先级排序数组）
    - 一个任务的时间片用完后，它会被转移到“过期”的队列中
    - 在该队列中，任务仍然是按照优先级排好序的
    - 当活动队列中的任务均被执行完时，就交换两个指针

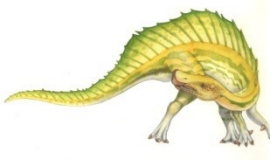



active
expired
arrays[0]
arrays[1]
P
priority 0
priority 139
priority 0
priority 139

prio_array

- 每一类就绪进程都用一个 struct prio_array 的结构表示

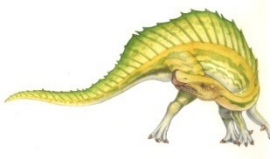
```
struct prio_array{
    unsigned int nr_active; /*本进程组中的进程数*/
    unsigned long bitmap[BITMAP_SIZE];/*加速HASH表访问的
      位图快速定位第一个非空的就绪进程链表*/
    struct list_head queue[MAX_PRIO];/*以优先级为索引的
      HASH表*/
    }
```



schedule()函数

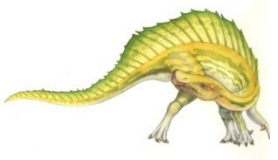
- 进程调度由schedule()函数实现。首先，schedule()利用下面的代码定位优先级最高的就绪进程：

```
prev=current;
array=rq->active;
idx=sched_find_first_bit(array->bitmap);
queue= array- >queue+idx;
next =list_entry(queue->next，struct task_struct, run_list);
```



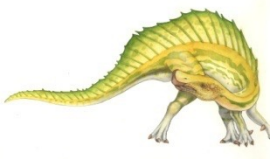
schedule()函数

- schedule()通过调用sched_find_first_bit()函数找到当前CPU就绪进程队列runqueue的active进程数组中第一个非空的就绪进程链表。这个链表中的进程具有最高的优先级，schedule()选择链表中的第一个进程作为调度器下一时刻将要运行的进程。
- 如果prev(当前进程)和next(将要运行的进程)不是同一个进程，schedule()调用context_switch()将CPU切换到next进程运行。



调度程序所使用的函数

- 调度程序依靠几个函数来完成调度工作，其中最重要的函数有：
  - scheduler_tick( )，维持当前最新的 time_slice 计数器
  - try_to_wake_up( )，唤醒睡眠进程
  - recalc_task_prio( )，更新进程的动态优先权
  - schedule( )，选择要被执行的新进程
  - load_balance()，维持多处理器系统中运行队列的平衡。

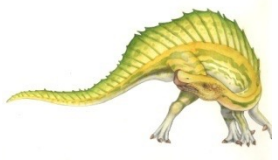


三、Linux进程调度算法（2.4）

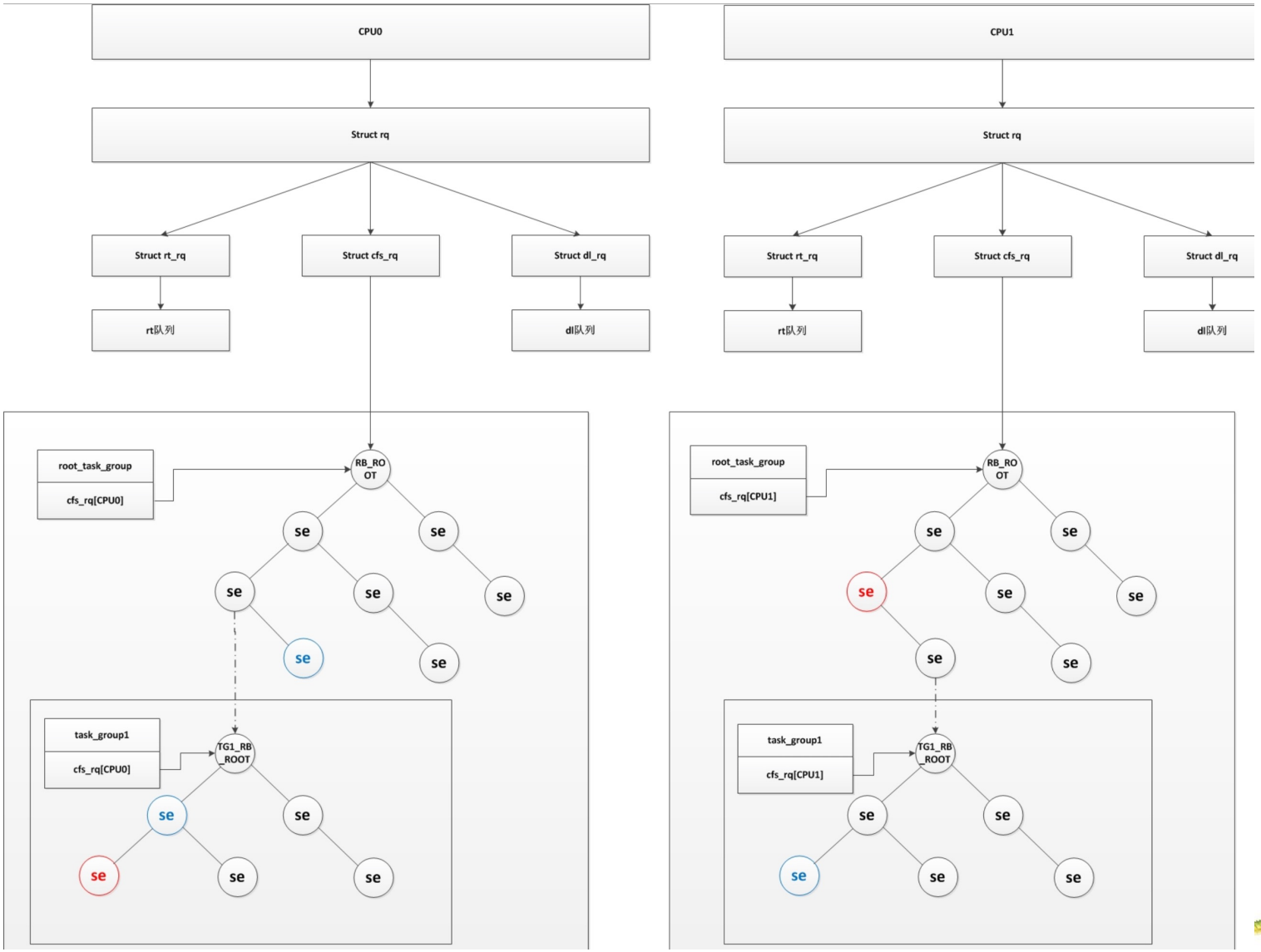
- Linux的进程调度程序是内核中的schedule()函数。schedule()函数需要确定：
  - policy 进程的调度策略
    - SCHED_FIFO、SCHED_RR的是实时进程，
    - SCHED_NORMAL是普通进程。
  - rt_priority 实时进程优先级。Linux2.6默认的变化范围[0,99]。
  - nice 普通进程的优先级，取值[-20,19]，默认0;nice为-20的进程获得时间片最长。2.6为static_prio，静态优先级，不参与优先级计算
  - counter 进程使用CPU运行时间的计数值(保存在jiffies 中)。进程首次运行时为进程优先级的数值，它随时间变化递减，随时钟滴答减1（即时钟中断的间隔时间，为10ms或1ms）。counter=0时，进程的时间片已用完。

Linux进程调度方法（2.4）

- Linux进程调度采用动态优先级法，调度对象是可运行队列，每个处理器有一个可运行队列；可运行队列中优先级大的进程首先得到CPU投入运行。
- 进程在运行中，counter的代表进程的动态优先级。
- Linux采取了加权的方法来保证实时进程优先于普通进程。
- 普通进程的权值就是它的counter的值，而实时进程的权值是它的rt_priority的值加1000 （2.4）。
- 在调度过程中，调度程序检查可运行队列中所有进程的权值，选择其中权值最大的进程做为下一个运行进程。



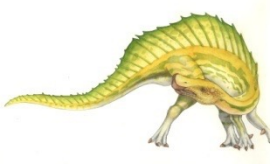
进程的调度：Run-Queue



进程的调度

- Linux-2.6.18/kernel/Sched.c
- **schedule** 函数并没有太多的操作，只是获取当前task，然后加锁，最后都交给了**__schedule()**函数。

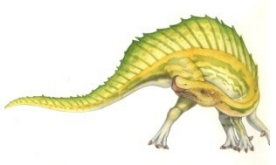
```
asmlinkage void __sched schedule(void)
{
        struct task_struct *tsk = current;

        sched_submit_work(tsk);//获取当前current 并且加锁。
        __schedule();
}
```



进程的调度

```
static void __sched __schedule(void)
{
        struct task_struct *prev, *next;
        unsigned long *switch_count;
        struct rq *rq;
        int cpu;

need_resched:
        preempt_disable();
        cpu = smp_processor_id();
        rq = cpu_rq(cpu);//找到当前cpu拿取队列

        prev = rq->curr;//保持此时此刻的任务
.....
        pre_schedule(rq, prev);//做一些预处理

....

        put_prev_task(rq, prev);
        next = pick_next_task(rq);//选取下一个进程
        clear_tsk_need_resched(prev);
        rq->skip_clock_update = 0;
```

```
if (likely(prev != next)) {//选好了，
                rq->nr_switches++;
                rq->curr = next;
                ++*switch_count;//计数加一
...
                context_switch(rq, prev, next); /* unlocks the rq */
...
                cpu = smp_processor_id();
                rq = cpu_rq(cpu);
        } else
                raw_spin_unlock_irq(&rq->lock);

....
        if (need_resched())//是否需要重新调度
                goto need_resched;
}
```

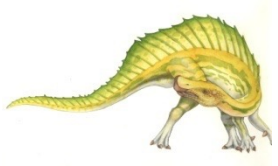


进程的调度

```
static inline struct task_struct * pick_next_task(struct rq *rq)
{
        const struct sched_class *class;
        struct task_struct *p;

        /*
         * Optimization: we know that if all tasks are in
         * the fair class we can call that function directly:
         */
        if (likely(rq->nr_running == rq->cfs.h_nr_running)) {检查当前cpu队列中是否所有的都是cfs，如果是就直接掉用cfs的调度类
                p = fair_sched_class.pick_next_task(rq);
                if (likely(p))
                        return p;
        }

        for_each_class(class) {循环遍历每一个调度类
                p = class->pick_next_task(rq);
                if (p)
                        return p;
        }


}
```

进程的调度

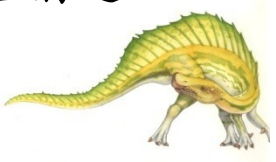
```
static struct task_struct *pick_next_task_fair(struct rq *rq)
{
        struct task_struct *p;
        struct cfs_rq *cfs_rq = &rq->cfs;
        struct sched_entity *se;

        if (!cfs_rq->nr_running)
                return NULL;

        do {
                se = pick_next_entity(cfs_rq);
                set_next_entity(cfs_rq, se);
                cfs_rq = group_cfs_rq(se);
        } while (cfs_rq);

        p = task_of(se);
        if (hrtick_enabled(rq))
                hrtick_start_fair(rq, p);

        return p;
}
```



进程切换时机

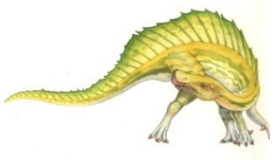
当前进程放弃CPU的情况可以分为两种。

- 进程主动地放弃CPU。大体可以分为两类：
  - 隐式地主动放弃CPU。往往是因为需要的资源目前不能获取，如执行read( )等系统调用的过程中,这种情况下的处理过程如下：
    - 将进程加入合适的等待队列。
    - 把当前进程的状态改为TASK_INTERRUTIBLE或TASK_UNINTERRUTIBLE。
    - 调用schedule( )函数，该函数的执行结果往往是当前进程放弃CPU。
    - 检查资源是否可用，如果不可用，则跳转到第2)步.
    - 资源已可用，将该进程从等待队列中移去。
  - 进程显式地主动放弃CPU，如系统调用sched_yield( ),sched_setscheduler( ),pause( ) 及nanosleep（）均会导致当前进程让出CPU。



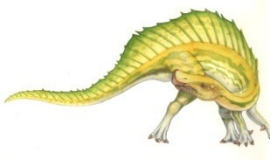
进程切换时机

- 进程被动放弃CPU。下面两种情形均会导致当前进程控制块的need_resched被置1。
  - 当前进程的时间片已经用完；
  - 刚被唤醒进程的优先级别高于当前进程。



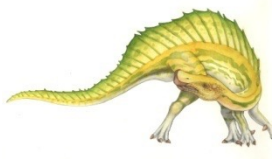
进程的调度时机

- 进程的调度时机分成两种情形：
  - 直接调用schedule()调度函数，例如进程主动放弃CPU的第一类情形；
  - 间接调用schedule()。
- 例如，进程被动放弃CPU的情形。当进程控制块的need_resched被置1时，并不立即直接调用schedule()调度函数。而是在随后的某个时刻，当进程从内核态返回用户态之前检查need_resched是否为1，如果为1，则调用schedule()调度函数。



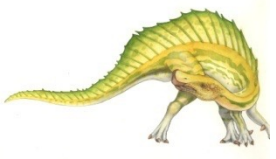
进程的调度算法(2.4)

- 调度程序源代码见kernel/sched.c文件。调度程序的算法和相关的代码大部分都在2.5版内核的开发版中重写了。
- 核心函数是schedule( ),该函数的任务是选出一个可运行的进程。
- 对于普通进程来讲CPU时间的分配是典型的时间片策略。
- 在某个时刻，运行队列中的每个进程都有一个counter值，当运行队列中进程的counter值都变为0以后，表明一轮已经结束，每个进程的counter根据其nice重新赋值，开始新的一轮执行过程。拥有CPU的进程每次“时钟滴答” counter值减一。
- Linux 2.4内核设定“时钟滴答”是10ms，普通进程counter初值为21（计算公式：20-(nice))+1），所以默认的时间片为210ms。



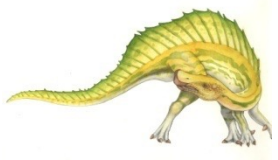
schedule()函数执行过程：

- 1) 检查是否有软中断服务请求，如果有，则先执行这些请求。
- 2) 若当前进程调度策略是SCHED_RR， 且 counter为0，则将该进程移到可执行进程队列的尾部并对counter重新赋值。
- 3) 检查当前进程的状态，如为 TASK_INTERRUPTIBLE，且该进程有信号接收，则将进程状态置为TASK_RUNNING。
- 4) 当前进程的状态不是TASK_RUNNING，则将其从可执行进程队列中移出，然后将当前进程控制块的need_resched恢复成0。



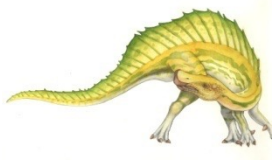
schedule()函数执行过程

- 5) 进入函数的核心部分。可运行进程队列的每个进程都将被计算出一个权值，主要是利用goodness()函数。最终最大的权值保存在变量c中，与之对应的进程控制块保存在变量next中。
- 6) 检查c是否为0。若为0则表明所有可执行进程的时间配额都已用完，因而对所有进程的counter重新计算（赋值）。
- 7) 如果next进程就是当前进程,则结束shedule()的运行。否则进行进程切换，CPU改由next进程占据。



goodness()函数

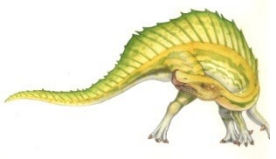
- goodness()函数计算进程的当前权值。该函数的第一个参数是待估进程的控制块。
  - 如果该进程是实时进程，它的权值为 1000＋rt_priority，1000是普通进程权值无法到达的数字，因而实时进程总可以优先得到执行。
  - 对于普通进程，它的权值为counter，如果其又是内核线程，由于无需切换用户空间，则将权值加一作为奖励。



goodness()

```
static inline int goodness(struct task_struct * p, int this_cpu, struct mm_task
    *this_mm)
{   int weight;
    if (p->policy & SCHED_YIELD)
        goto out      //如果设置的是SCHED_YIELD标志，则p自愿放弃CPU
   if (p->policy == SCHED_NORMAL){
                      //对于非实时进程，则返回当前的counter值
     weight = p->counter;
     if(!weight)
        goto out; //如果权值weight为0，则进程的时间片已用完，直接返回。
     ……              //对称多处理机代码
     if(p->mm==this_mm || !p->mm)
                              //对于使用同一struct mm_task 结构的进程
          weight+=1;    //将增加少许权值，因为他们使用同一页表，
          goto out;       //减少在MMU上进行的上下文切换所花费的时间开销
     }
    weight= 1000 + p->rt_priority;
                      //如果p是实时进程，则权值的weight基值为1000。
    out:
    return weight; }
```

普通进程counter初值为21（计算公式：20-(nice))+1）



与调度相关的其他系统调用

系统调用	描述
nice( )	改变一个普通进程的优先级
getpriority( )	取得一组普通进程的最大优先级
setpriority( )	设置一组普通进程的优先级
sched_getscheduler( )	取得一个进程的调度策略
sched_setscheduler( )	设置一个进程的调度策略和优先级
sched_getparam( )	取得一个进程的调度优先级
sched_setparam( )	设置一个进程的优先级
sched_yield( )	不阻塞的情况下自愿放弃处理机
sched_get_ priority_min( )	取得某种策略的最小优先级
sched_get_ priority_max( )	取得某种策略的最大优先级