国嵌

www.enjoylinux.cn

Linux内核开发

谢伟 著

Contents

Contents

定义

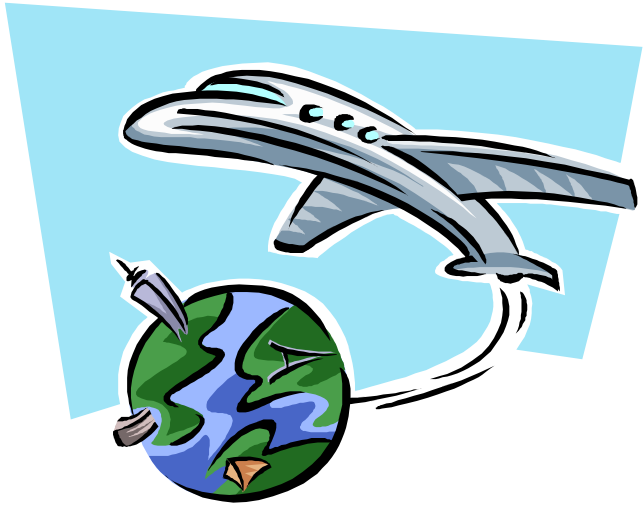
什么是进程？

什么是程序？

进程和程序的区别在哪里？

定义

- 程序是存放在磁盘上的一系列代码和数据的可执行映像，是一个静止的实体。
- 进程是一个执行中的程序。它是动态的实体。

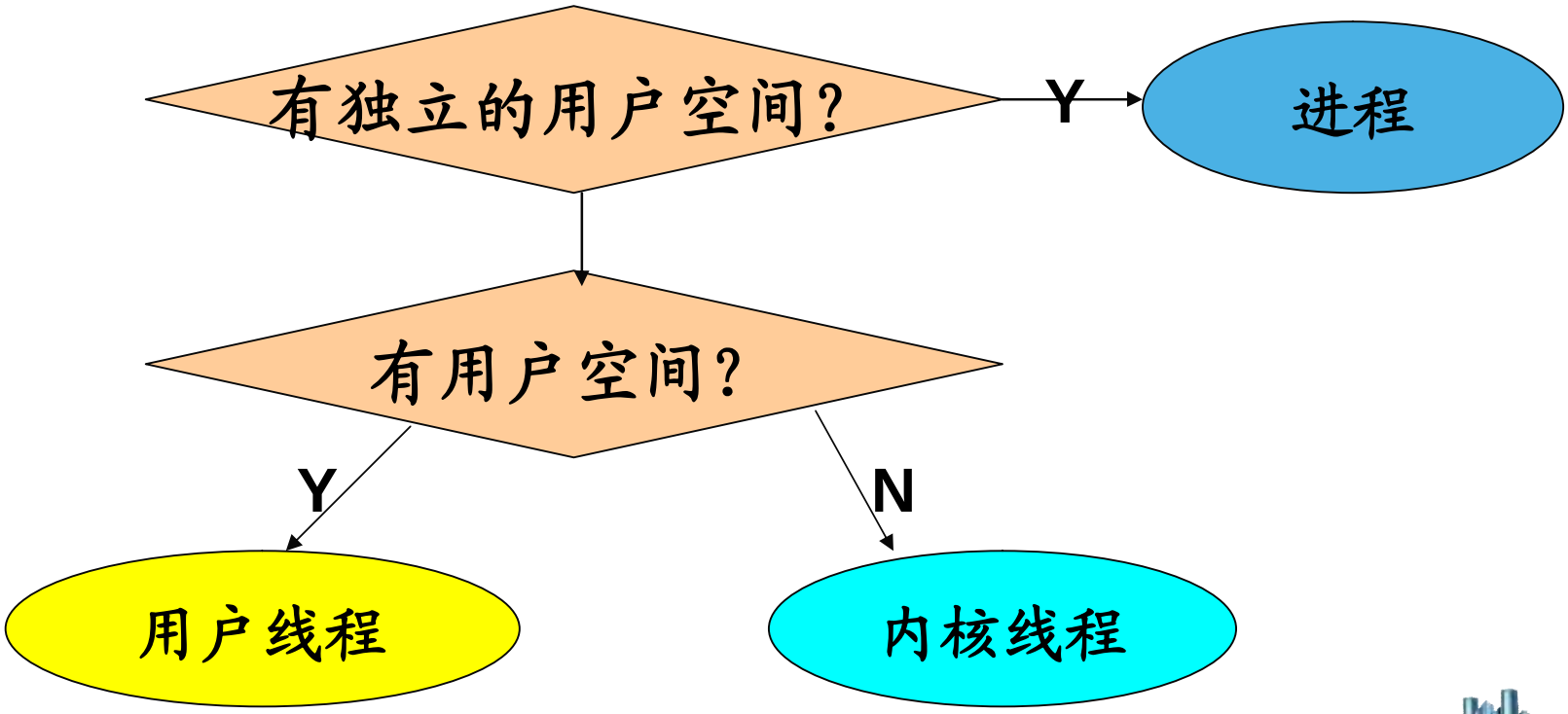


进程四要素

1. 有一段程序供其执行。这段程序不一定是某个进程所专有，可以与其他进程共用。
2. 有进程专用的内核空间堆栈。
3. 在内核中有一个task_struct数据结构，即通常所说的“进程控制块”。有了这个数据结构，进程才能成为内核调度的一个基本单位接受内核的调度。

进程四要素

4. 有独立的用户空间。



进程描述

在Linux中，线程、进程都使用struct task_struct来表示，它包含了大量描述进程/线程的信息，其中比较重要的有：

- pid_t pid;

  进程号，最大值10亿

进程状态

- volatile long state /* 进程状态 */

1. TASK_RUNNING

进程正在被CPU执行，或者已经准备就绪，随时可以执行。当一个进程刚被创建时，就处于TASK_RUNNING状态。

2. TASK_INTERRUPTIBLE

处于等待中的进程，待等待条件为真时被唤醒，也可以被信号或者中断唤醒。

进程状态

3. TASK_UNINTERRUPTIBLE

处于等待中的进程，待资源有效时唤醒，但不可以由其它进程通过信号(signal)或中断唤醒。

4. TASK_STOPPED

进程中止执行。当接收到SIGSTOP和SIGTSTP等信号时，进程进入该状态，接收到SIGCONT信号后，进程重新回到TASK_RUNNING。

进程状态

5. TASK_KILLABLE

Linux2.6.25新引入的进程睡眠状态，原理类似于TASK_UNINTERRUPTIBLE，但是可以被致命信号(SIGKILL)唤醒。

6. TASK_TRACED

正处于被调试状态的进程。

进程状态

7. TASK_DEAD

进程退出时(调用do_exit)，state字段被设置为该状态。

进程状态

- int exit_state /*进程退出时的状态*/

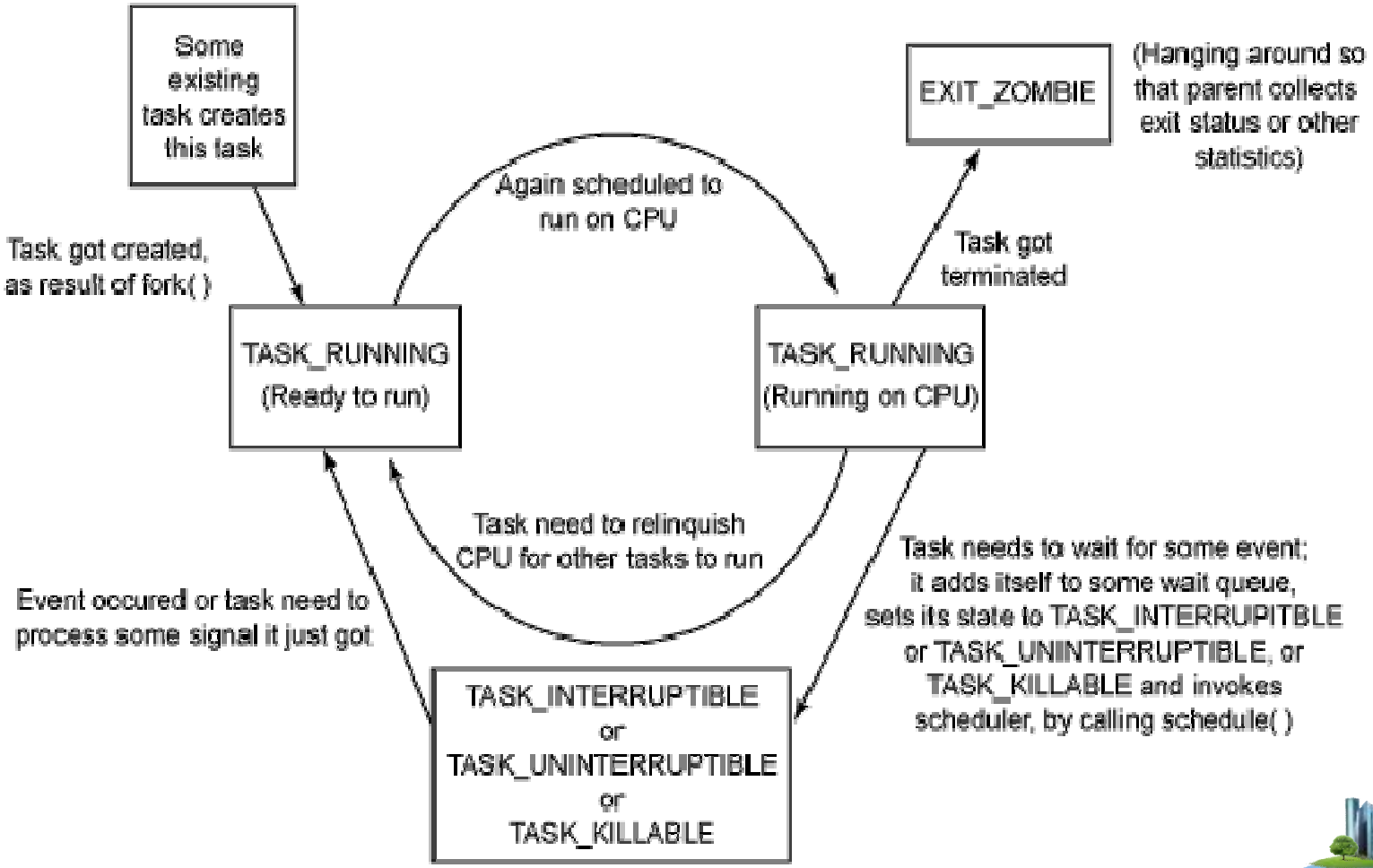
EXIT_ZOMBIE(僵死进程)

表示进程的执行被终止，但是父进程还没有发布waitpid()系统调用来收集有关死亡的进程的信息。

EXIT_DEAD(僵死撤销状态)

表示进程的最终状态。父进程已经使用wait4()或waitpid()系统调用来收集了信息，因此进程将由系统删除。

进程状态



进程描述

- **struct mm_struct *mm**

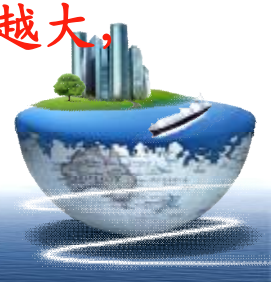
  进程用户空间描述指针,内核线程该指针为空。

- **unsigned int policy**

  该进程的调度策略。

- **int prio**

  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-MX_PRIO-1 属于非实时进程。数值越大，表示进程优先级越小



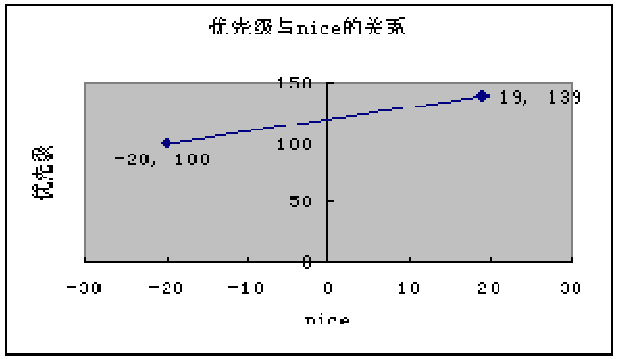
进程描述

- **int static_prio**

静态优先级，与 2.4 的 nice 值意义相同。nice 值仍沿用 Linux 的传统，在 -20 到 19 之间变动，数值越大，进程的优先级越小。nice 是用户可维护的，但仅影响非实时进程的优先级。进程初始时间片的大小仅决定于进程的静态优先级，这一点不论是实时进程还是非实时进程都一样，不过实时进程的static_prio 不参与优先级计算。nice 与 static_prio 的关系如下:

static_prio = MAX_RT_PRIO + nice + 20

内核定义了两个宏用来完成这一转换：PRIO_TO_NICE()、NICE_TO_PRIO



进程描述

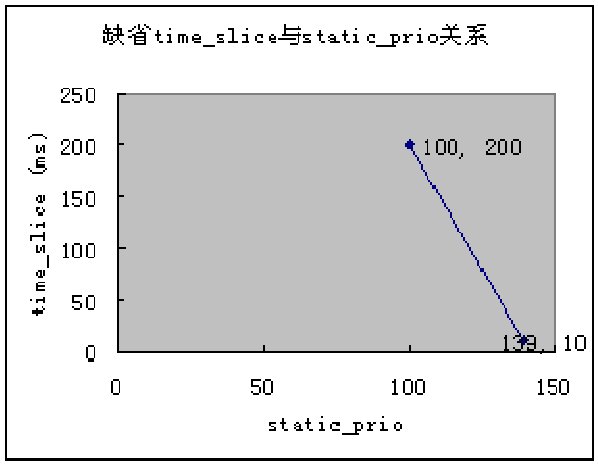
- struct sched_rt_entity rt

  rt->time_slice

  时间片,进程的缺省时间片与进程的静态优先级（在 2.4 中是 nice 值）相关，使用如下公式得出:

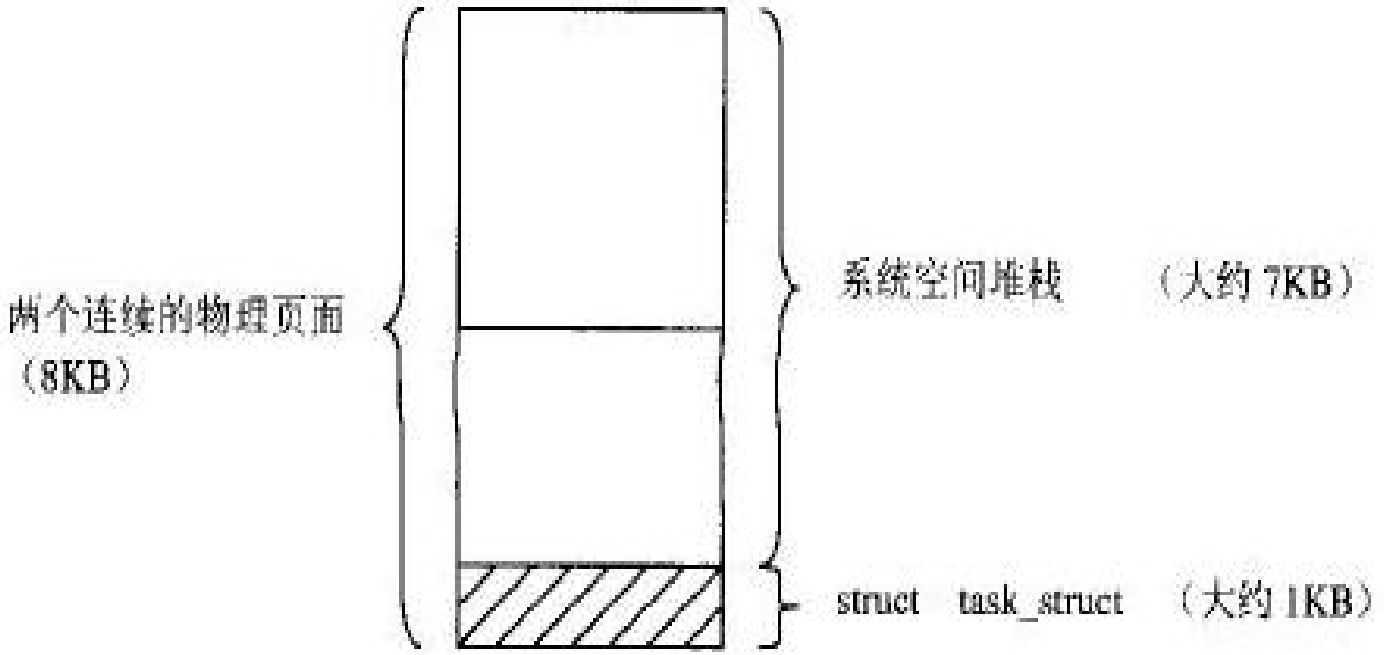
```
MIN_TIMESLICE + ((MAX_TIMESLICE - MIN_TIMESLICE) *
(MAX_PRIO-1 - (p)->static_prio) / (MAX_USER_PRIO-1))
```

进程描述



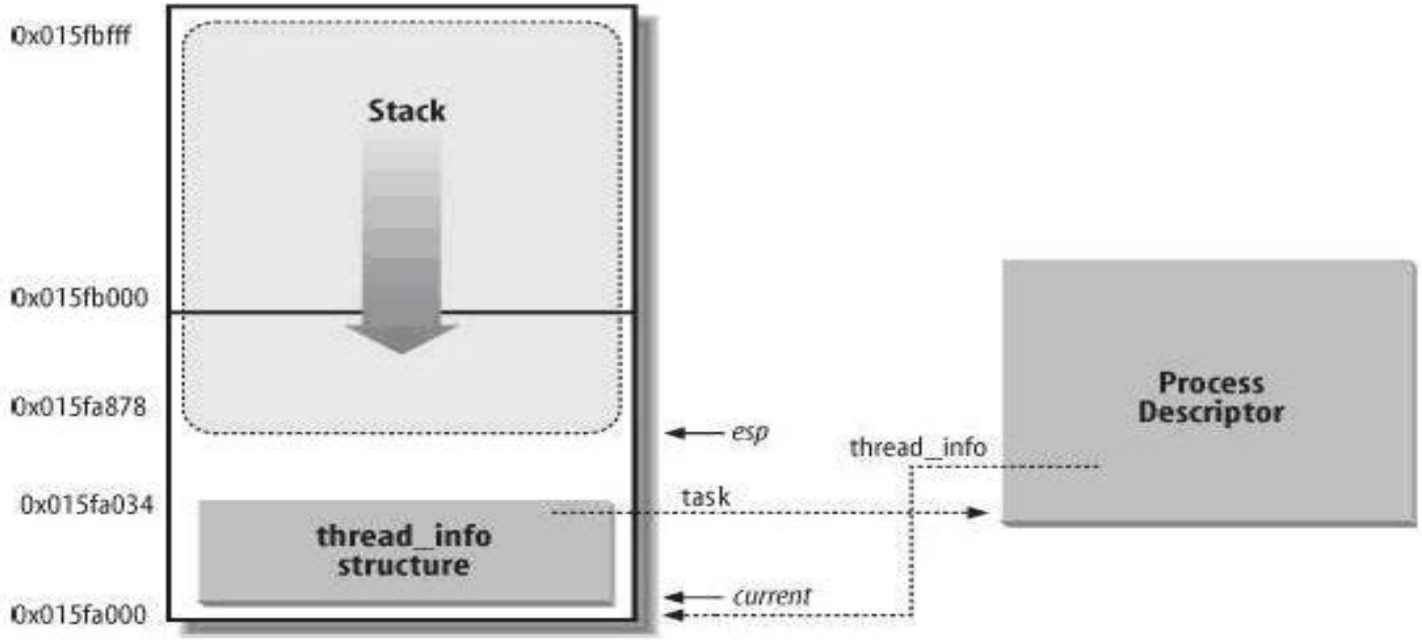
内核将 100-139 的优先级映射到 200ms-10ms 的时间片上去，优先级数值越大，则分配的时间片越小。

Task_struct位置(2.4)

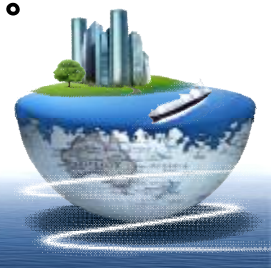


进程系统堆栈示意图

Task_struct位置(2.6)



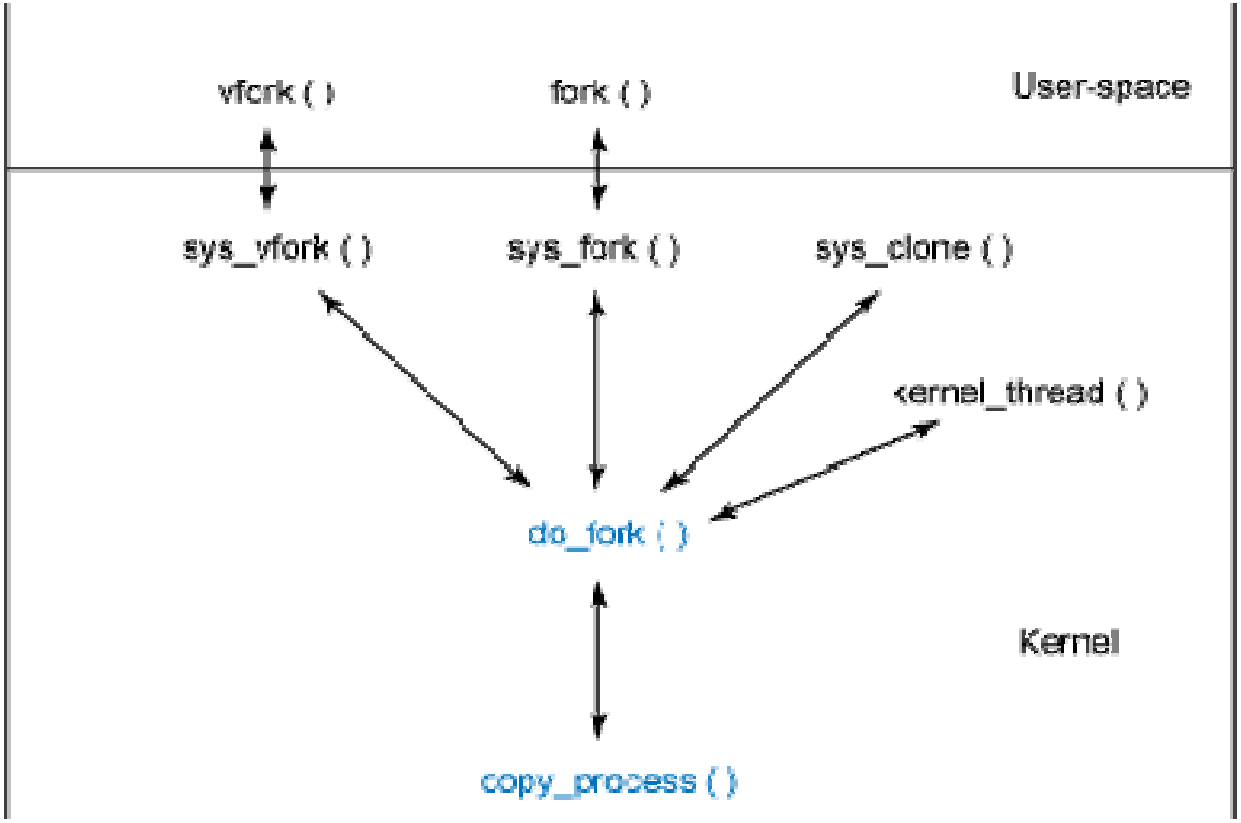
可以是4K字节(1个页面)也可以是8K字节(2个页面)。



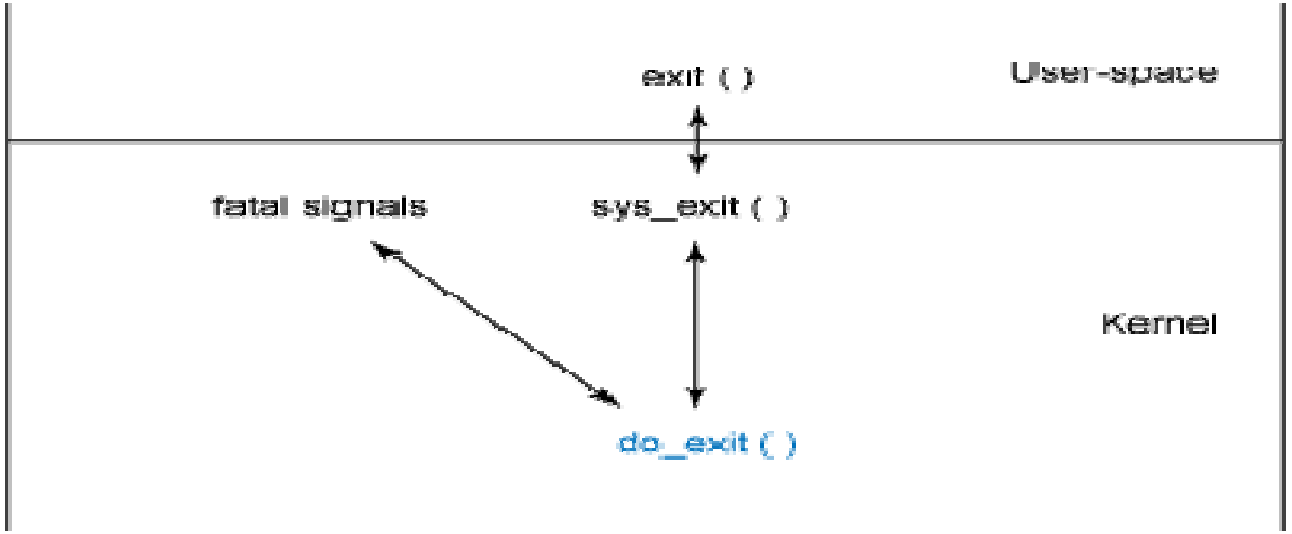
Current

在Linux中用current指针指向当前正在运行的进程的task_struct。

进程创建



进程销毁



进程销毁可以通过几个事件驱动 — 通过正常的进程结束、通过信号或是通过对 exit 函数的调用。不管进程如何退出，进程的结束都要借助对内核函数 do_exit的调用。

Contents

调度

什么是调度？

从就绪的进程中选出最适合的一个来执行。

学习调度需要掌握哪些知识点？

1、调度策略

2、调度时机

3、调度步骤

调度策略

- SCHED_NORMAL(SCHED_OTHER):普通的分时进程
- SCHED_FIFO :先入先出的实时进程
- SCHED_RR: 时间片轮转的实时进程
- SCHED_BATCH:批处理进程
- SCHED_IDLE: 只在系统空闲时才能够被调度执行的进程

调度类

调度类的引入增强了内核调度程序的可扩展性，这些类（调度程序模块）封装了调度策略，并将调度策略模块化。

- **CFS 调度类**（在 kernel/sched_fair.c 中实现）用于以下调度策略：SCHED_NORMAL、SCHED_BATCH 和 SCHED_IDLE。
- **实时调度类**（在 kernel/sched_rt.c 中实现）用于 SCHED_RR 和 SCHED_FIFO 策略。

调度类


```
struct sched_class {
struct sched_class *next;
void (*enqueue_task) (struct rq *rq, struct task_struct *p, int wakeup);
void (*dequeue_task) (struct rq *rq, struct task_struct *p, int sleep);
void (*yield_task) (struct rq *rq, struct task_struct *p);
void (*check_preempt_curr) (struct rq *rq, struct task_struct *p);
struct task_struct * (*pick_next_task) (struct rq *rq);
void (*put_prev_task) (struct rq *rq, struct task_struct *p);
unsigned long (*load_balance) (struct rq *this_rq, int this_cpu, struct rq *busiest, unsigned
    long max_nr_move, unsigned long max_load_move, struct sched_domain *sd, enum
    cpu_idle_type idle, int *all_pinned, int *this_best_prio);
void (*set_curr_task) (struct rq *rq); void (*task_tick) (struct rq *rq, struct task_struct *p);
void (*task_new) (struct rq *rq, struct task_struct *p);
};
```

- **pick_next_task:** 选择下一个要运行的进程

调度时机

调度什么时候发生？即：schedule()函数什么时候被调用？

调度的发生有两种方式：

1、主动式

在内核中直接调用schedule()。当进程需要等待资源等而暂时停止运行时，会把状态置于挂起（睡眠），并主动请求调度，让出CPU。

调度时机

主动放弃cpu例:

1. current->state = TASK_INTERRUPTIBLE;
2. schedule();

调度时机

2、被动式（抢占）

用户抢占（Linux2.4、Linux2.6）

内核抢占(Linux2.6)

用户抢占

用户抢占发生在:

- 从系统调用返回用户空间。
- 从中断处理程序返回用户空间。

内核即将返回用户空间的时候，如果need_resched标志被设置，会导致schedule()被调用，此时就会发生用户抢占。

用户抢占

- ENTRY(ret_from_exception) //异常返回

```
get_thread_info tsk
mov          why, #0
b     ret_to_user
```

- __irq_usr: //在用户态收到中断

```
usr_entry
kuser_cmpxchg_check
.....        .....        .....
b   ret_to_user
```

用户抢占


```
ENTRY(ret_to_user)
ret_slow_syscall:
    disable_irq   @ disable interrupts
    ldr r1, [tsk, #TI_FLAGS]
    tst r1, #_TIF_WORK_MASK
    bne work_pending
```

用户抢占


```
work_pending:
    tst r1, #_TIF_NEED_RESCHED
    bne work_resched

work_resched:
    bl  schedule
```

内核抢占

- 在不支持内核抢占的系统中，进程/线程一旦运行于内核空间，就可以一直执行，直到它主动放弃或时间片耗尽为止。这样一些非常紧急的进程或线程将长时间得不到运行。

- 在支持内核抢占的系统中，更高优先级的进程/线程可以抢占正在内核空间运行的低优先级进程/线程。

内核抢占

在支持内核抢占的系统中，某些特例下是不允许内核抢占的：

- 内核正进行中断处理。进程调度函数schedule()会对此作出判断，如果是在中断中调用，会打印出错信息。
- 内核正在进行中断上下文的Bottom Half(中断的底半部)处理。硬件中断返回前会执行软中断，此时仍然处于中断上下文中。
- 进程正持有spinlock自旋锁、writelock/readlock读写锁等，当持有这些锁时，不应该被抢占，否则由于抢占将导致其他CPU长期不能获得锁而死等。
- 内核正在执行调度程序Scheduler。抢占的原因就是为了进行新的调度，没有理由将调度程序抢占掉再运行调度程序。

内核抢占

为保证Linux内核在以上情况下不会被抢占，抢占式内核使用了一个变量preempt_count，称为内核抢占计数。这一变量被设置在进程的thread_info结构中。每当内核要进入以上几种状态时，变量preempt_count就加1，指示内核不允许抢占。每当内核从以上几种状态退出时，变量preempt_count就减1，同时进行可抢占的判断与调度。

内核抢占

内核抢占可能发生在:

- 中断处理程序完成，返回内核空间之前。
- 当内核代码再一次具有可抢占性的时候，如解锁及使能软中断等。

内核抢占(中断)


```
__irq_svc:          /*内核态接收到中断*/
      svc_entry
。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。
 /*进入中断，抢占计数加1*/
#ifdef CONFIG_PREEMPT
      get_thread_info tsk
      ldr    r8, [tsk, #TI_PREEMPT]          @ get preempt count
      add    r7, r8, #1                 @ increment it
      str    r7, [tsk, #TI_PREEMPT]
#endif
      irq_handler         /*中断处理*/
#ifdef CONFIG_PREEMPT
      str    r8, [tsk, #TI_PREEMPT]          @ restore preempt count
      ldr    r0, [tsk, #TI_FLAGS]       @ get flags
      teq    r8, #0                          @ if preempt count != 0
      movne            r0, #0                     @ force flags to 0
      tst    r0, #_TIF_NEED_RESCHED
      blne   svc_preempt
#endif
```

内核抢占(中断)


```
svc_preempt:
    mov r8, lr
1: bl preempt_schedule_irq  /*调度*/
    ldr       r0, [tsk, #TI_FLAGS]
    tst       r0, #_TIF_NEED_RESCHED
    moveq pc, r8
```

内核抢占（解锁）


```
void __lockfunc _spin_unlock(spinlock_t
   *lock)
{
    spin_release(&lock->dep_map, 1,
   _RET_IP_);
    _raw_spin_unlock(lock);
    preempt_enable();
}
```

内核抢占（解锁）


```
define preempt_enable() \
do { \
    preempt_enable_no_resched(); \
    barrier(); \
    preempt_check_resched(); \
} while (0)
```

内核抢占（解锁）


```
#define preempt_enable_no_resched() \
do { \
    barrier(); \
    dec_preempt_count(); \
  /*抢占计数减一*/
} while (0)
```

内核抢占（解锁）


```
#define preempt_check_resched() \
do { \
if
    (unlikely(test_thread_flag(TIF_NEED_RES
    CHED))) \
    preempt_schedule(); \   /*调度*/
} while (0)
```

调度标志

TIF_NEED_RESCHED

作用：

内核提供了一个need_resched标志来表明是否需要重新执行一次调度。

设置：

当某个进程耗尽它的时间片时，会设置这个标志；

当一个优先级更高的进程进入可执行状态的时候，也会设置这个标志。

调度步骤

Schedule函数工作流程如下：

1). 清理当前运行中的进程；

2). 选择下一个要运行的进程；

（pick_next_task 分析）

3). 设置新进程的运行环境；

4). 进程上下文切换。

Contents

定义

Linux内核中设置了一组用于实现各种系统功能的子程序，称为系统调用。用户可以通过系统调用命令在自己的应用程序中调用它们。

区别

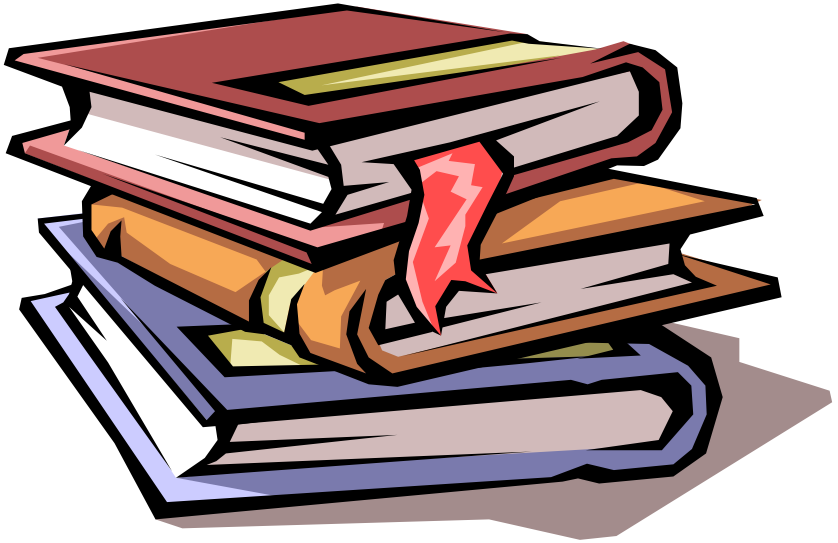
系统调用和普通的函数调用非常相似，区别仅仅在于，系统调用由操作系统内核实现，运行于内核态；而普通的函数调用由函数库或用户自己提供，运行于用户态。

库函数

Linux系统还提供了一些C语言函数库，这些库对系统调用进行了一些包装和扩展，这些库函数与系统调用的关系非常紧密。

系统调用数

在2.6.29 版内核中，共有系统调用332个，可在arch/arm/include/asm/unistd.h中找到它们。



系统调用手册

www.enjoylinux.cn

各系统调用的功能参考

《国嵌系统调用手册》

使用系统调用


```
#include<time.h>
main()
{
        time_t the_time;
        the_time=time((time_t *)0); /*调用time系统调用*/
        printf("The time is %ld\n",the_time);
}
```

/* 从格林尼治时间1970年1月1日0:00开始到现在的秒数。 */

工作原理

一般情况下，用户进程是不能访问内核的。它既不能访问内核所在的内存空间，也不能调用内核中的函数。系统调用是一个例外。其原理是进程先用适当的值填充寄存器，然后调用一个特殊的指令，这个指令会让用户程序跳转到一个事先定义好的内核中的一个位置：

- 在Intel CPU中，这个指令由中断0x80实现。
- 在ARM中，这个指令是SWI。

工作原理

进程可以跳转到的内核位置是ENTRY(vector_swi) <entry-common.S>。这个过程检查系统调用号，这个号码告诉内核进程请求哪种服务。然后，它查看系统调用表(sys_call_table)找到所调用的内核函数入口地址。接着，就调用函数，等返回后，做一些系统检查，最后返回到进程。

工作原理（应用）


```
#define __syscall(name)   "swi\t" __NR_##name "\n\t“

int open( const char * pathname, int flags)
{
。。。。。。
    __syscall(open)；
。。。。。。
}
```

转化为

```
int open( const char * pathname, int flags)
{
。。。。。。
    swi\t  __NR_open
。。。。。。
}
```

工作原理（内核入口）


```
/* arch/arm/kernel/entry-common.S */

ENTRY(vector_swi)

……    ……    ……    ……

adr tbl, sys_call_table              @ load syscall table pointer

……    ……    ……    ……

ldrcc  pc, [tbl, scno, lsl #2]       @ call sys_* routine

……    ……    ……    ……


ENTRY(sys_call_table)

#include "calls.S"
```

工作原理（内核入口）


```
/* arch/arm/kernel/calls.S */
/* 0 */    CALL(sys_restart_syscall)
           CALL(sys_exit)
           CALL(sys_fork_wrapper)
           CALL(sys_read)
           CALL(sys_write)
/* 5 */    CALL(sys_open)
…… …… …… …… …… …… …… …… …… …… …… ……
           CALL(sys_dup3)
           CALL(sys_pipe2)
/* 360 */CALL(sys_inotify_init1)
```

实现系统调用

向内核中添加新的系统调用，需要执行 3 个步骤:

1. 添加新的内核函数
2. 更新头文件 unistd.h
3. 针对这个新函数更新系统调用表 calls.S

实现系统调用

1. 在kernel/sys.c中添加函数:

```
asmlinkage int sysMul(int a, int b)
{
    int c;
    c = a*b;
    return c;
}
```

/* asmlinkage: 使用栈传递参数 */

实现系统调用

2. 在arch/arm/include/asm/unistd.h中添加如下代码:

```
#define __NR_sysMul 361
```

实现系统调用

3.在arch/arm/kernel/calls.S中添加代码，指向新实现的系统调用函数:

CALL(sysMul)

使用系统调用


```
#include <stdio.h>
#include <linux/unistd.h>

    main()
    {
    int result;
    result = syscall(361,1, 2);
    printf("result = ", result);
    }
```

实验

www.enjoylinux.cn

系统调用实现

1.修改内核，实现一个用于加法运算的系统调用

2.实现应用程序，使用该系统调用

Contents

定义

什么是proc文件系统？

实例：通过 /proc/meminfo，查询当前内存使用情况。

结论：proc文件系统是一种在用户态检查内核状态的机制。

Proc文件

子目录/文件名	内容描述
apm	高级电源管理信息
bus	总线以及总线上的设备
devices	可用的设备信息
driver	已经启用的驱动程序
interrupts	中断信息
ioports	端口使用信息
version	内核版本

特点

- 每个文件都规定了严格的权限

  可读？可写？哪个用户可读？哪个用户可写？
- 可以用文本编辑程序读取（more命令，cat命令，vi程序等等）
- 不仅可以有文件，还可以有子目录。
- 可以自己编写程序添加一个/proc目录下的文件。
- 文件的内容都是动态创建的，并不存在于磁盘上。

内核描述

```
struct proc_dir_entry {
{
。。。。。。。。。。。。。。。。。。。
   read_proc_t  *read_proc;
   write_proc_t  *write_proc;
。。。。。。。。。。。。。。。。。。。
}
```

创建文件


```
struct proc_dir_entry* create_proc_entry (const char
    *name,mode_t mode,struct proc_dir_entry *parent)
```

功能:

创建proc文件

参数:

- name :要创建的文件名
- mode :要创建的文件的属性 默认0755
- parent :这个文件的父目录

创建目录


```
struct proc_dir_entry * proc_mkdir (const char *name,struct
    proc_dir_entry *parent)
```

功能:

创建proc目录

参数:

- name :要创建的目录名
- parent :这个目录的父目录

删除目录/文件

void remove_proc_entry (const char *name,struct proc_dir_entry *parent)

功能:

删除proc目录或文件

参数:

- **name** :要删除的文件或目录名
- **parent** :所在的父目录

读写

为了能让用户读写添加的proc文件，需要挂接上读写回调函数:

- read_proc
- write_proc

读操作


```
int read_func (char *buffer,char **stat,off_t
off,int count,int *peof,void *data)
```

参数:

- buffer :把要返回给用户的信息写在buffer里，最大不超过PAGE_SIZE
- stat :一般不使用
- off :偏移量
- count :用户要取的字节数
- peof :读到文件尾时，需要把*peof置1
- data :一般不使用

写操作


```
int write_func (struct file *file,const char
   *buffer,unsigned long count,void *data)
```

参数:

- file :该proc文件对应的file结构，一般忽略。
- buffer :待写的数据所在的位置
- count :待写数据的大小
- data :一般不使用

实现流程

实现一个proc文件的流程:

（1）调用create_proc_entry创建一个struct proc_dir_entry。

（2）对创建的struct proc_dir_entry进行赋值：read_proc，mode，owner，size，write_proc 等等。

实例

Proc.c
Proc1.c

Proc文件实现

编写内核模块，实现一可读可写的Proc文件

Contents

定义

常在河边走，哪能不湿鞋。内核级的程序，总有死机的时候，如果运气好，会看到一些所谓“Oops”信息(在屏幕上或系统日志中)，比如：

```
Unable to handle kernel paging request at virtual address f899b670
printing eip:
c01de48c
*pde = 00737067
Oops: 0002 [#1]
Modules linked in: bluesmoke_e752x bluesmoke_mc md5 ipv6 parport_pc
lp parport nls_cp936 vfat fat dm_mod button battery asus_acpi ac joydev
CPU: 0
EIP: 0060:[] Not tainted VLI
EFLAGS: 00210286 (2.6.9-11.21AXKProbes)
EIP is at kobject_add+0x83/0xd7
。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。
```

定义

Oops 可以看成是内核级的Segmentation Fault。应用程序如果进行了非法内存访问或执行了非法指令，会得到Segfault信号，一般的行为是coredump，应用程序也可以自己截获Segfault信号，自行处理。如果内核自己犯了这样的错误，则会打出Oops信息。

分析步骤

1. 错误原因提示
2. 调用栈（对照反汇编代码）
3. 寄存器

实验

www.enjoylinux.cn

内核异常分析

编写内核模块，产生内核异常，根据OOPS分析异常原因