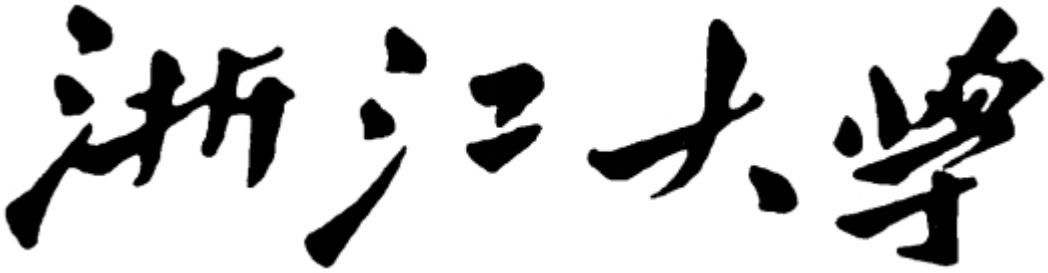


研究报告

课程名称:	**操作系统**
姓 名:	臧可
学 院:	计算机科学与技术学院
系:	计算机科学与技术系
专 业:	计算机科学与技术学院
学 号:	3180102095
指导教师:	夏莹杰

2020年　12月　8日

Linux进程管理

一、研究问题

1. 分析Linux的task_struct
2. 分析Linux进程创建do_fork函数和进程调度schedule函数

二、内核版本

linux-2.6.38.8

三、研究分析

3.1 task_struct结构体分析

3.1.1 进程控制块PCB

操作系统中存放进程的管理和控制信息的数据结构称为进程控制块PCB(Prorocess Control Block), 它是进程实体的一部分, 是操作系统中最重要的记录性数据结构。每一个进程均有一个 PCB，在创建进程时，建立 PCB，伴随进程运行的全过程，直到进程撤消而撤消。PCB 记录了操作系统所需的，用于描述进程的当前情况以及控制进程运行的全部信息（如打开的文件、挂起的信号、进程状态、地址空间等等）。

Linux内核的进程控制块是task_struct结构体。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。

3.1.2 task_struct 数据结构

`task_struct`代码在./linux/include/linux/sched.h中

> **标示符**：描述本进程的唯一标识符，用来区别其他进程。
>
> **状态**：任务状态，退出代码，退出信号等。
>
> **优先级**：相对于其他进程的优先级。
>
> **程序计数器**：程序中即将被执行的下一条指令的地址。
>
> **内存指针**：包括程序代码和进程相关数据的指针，还有和其他进程共享的内存块的指针。
>
> **上下文数据**：进程执行时处理器的寄存器中的数据。
>
> **I/O状态信息**：包括显示的I/O请求,分配给进程的I/O设备和被进程使用的文件列表。
>
> **记账信息**：可能包括处理器时间总和，使用的时钟数总和，时间限制，记账号等。

3.1.3 task_struct进程状态

task_struct结构中定义

```
volatile long state; /*表示进程的可运行性*/
int exit_state; /*表示进程退出时的状态*/
```

state的可能取值为：

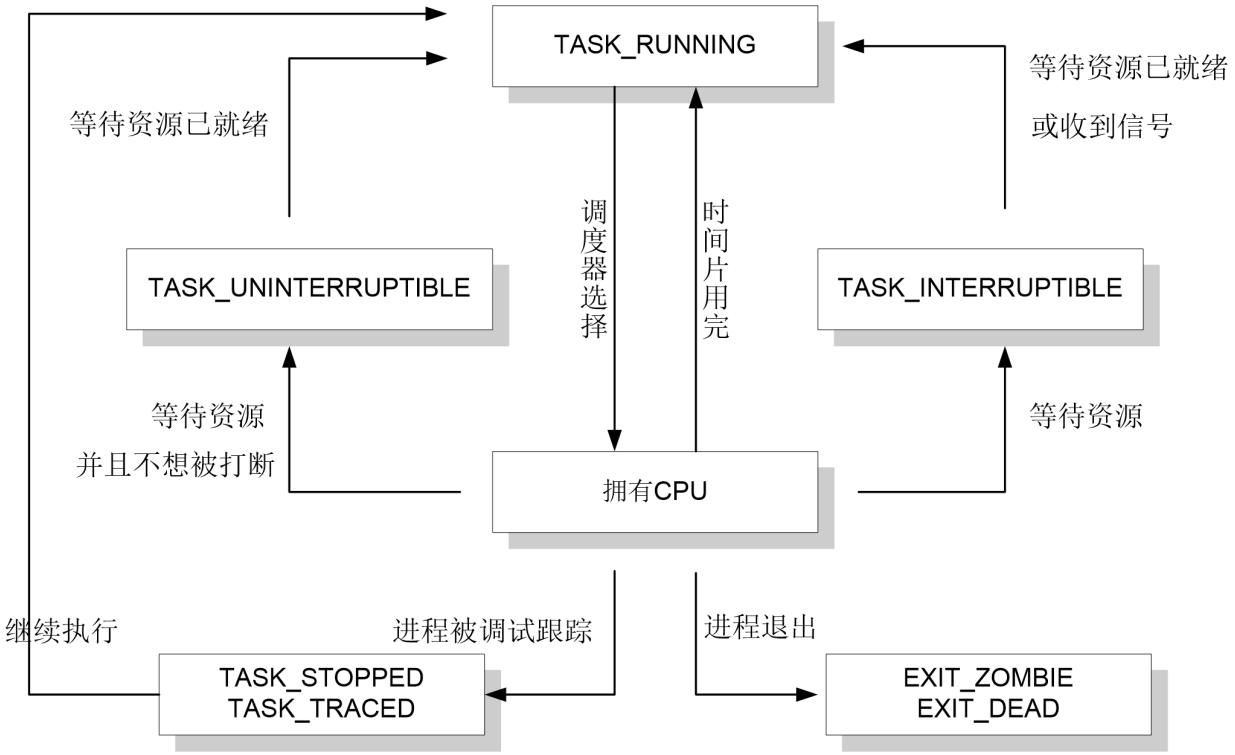
```
#define TASK_RUNNING         0 /*正在运行的进程，即系统的当前进程或准备运行的进程即在Running队列中的进程。只有处于该状态的进程才实际参与进程调度*/
#define TASK_INTERRUPTIBLE  1 /*处于等待资源状态中的进程，当等待的资源有效时被唤醒，也可以被其他进程或内核用信号、中断唤醒后进入就绪状态*/
#define TASK_UNINTERRUPTIBLE    2 /*处于等待资源状态中的进程，当等待的资源有效时被唤醒，不可以被其它进程或内核通过信号、中断唤醒*/
#define __TASK_STOPPED       4 /*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*/
#define __TASK_TRACED        8 /*进程被跟踪，一般在调试的时候用到*/

/* in task->exit_state 有关任务退出*/
```

```
#define EXIT_ZOMBIE      16 /*正在终止的僵尸状态的进程，表示进程被终止，但是父进程还没有获取它的终止信息，比如进程有没有执行完等信息。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）*/
#define EXIT_DEAD        32 /*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*/
/* in tsk->state again */

/* in tsk->state again */
#define TASK_DEAD        64 /*死亡*/
#define TASK_WAKEKILL        128 /*唤醒并杀死的进程*/
#define TASK_WAKING      256 /*唤醒进程*/
```

Linux进程状态转换



3.1.4 task_struct进程的标识(PID)

Linux系统每一个进程都有一些属性，包括所有者的ID、进程名、进程ID(PID)、进程状态、父进程的ID、进程已执行时间等。从用户角度来看，其中最有用的属性是PID，很多进程控制命令用它作为参数。PID是大多数操作系统的内核用于唯一标识进程的一个数值

```
pid_t pid;/*进程的唯一标识*/
pid_t tgid;/*线程组的领头线程的pid成员的值*/
```

在Linux系统中，一个线程组中的所有线程使用和该线程组的领头线程（该组中的第一个轻量级进程）相同的PID，并被存放在tgid成员中。只有线程组的领头线程的pid成员才会被设置为与tgid相同的值。注意，getpid()系统调用返回的是当前进程的tgid值而不是pid值。

3.1.5 task_struct进程的标记(flags)

反应进程状态的信息，但不是运行状态，用于内核识别进程当前的状态，以备下一步操作

```
unsigned int flags; /*进程当前的状态标志*/
```

flags定义了大量的指示器，指示进程是否正在创建(PF_STARTING)或退出(PF_EXITING)，进程当前是否正在分配内存(PF_MEMALLOC)。可执行文件的名称(不包括路径)占用comm(command)字段。

3.1.6 task_struct进程之间的亲属关系

在Linux系统中，所有进程之间都有着直接或间接地联系，每个进程都有其父进程，也可能有零个或多个子进程。拥有同一父进程的所有进程具有兄弟关系。

```
/* pointers to (original) parent process, youngest child, younger
sibling,older sibling, respectively. (p->father can be replaced with p-
>real_parent->pid)*/

struct task_struct *real_parent; /* real_parent指向其父进程，如果创建它的父进程不再存在，则指向PID为1的init进程 */
struct task_struct *parent; /* reportsparent指向其父进程，当它终止时，必须向它的父进程发送信号。它的值通常与real_parent相同，SIGCHLD的接收者，wait4()报告 */

/*children/sibling构成了natural children的链表*/
struct list_head children; /* children表示链表的头部，链表中的所有元素都是它的子进程（进程的子进程链表）*/
struct list_head sibling; /* 链接在我的父母的子女名单，sibling用于把当前进程插入到兄弟链表中（进程的兄弟链表）*/
struct task_struct *group_leader; /* threadgroup leader */
```

3.1.7 task_struct进程调度信息

```
int prio; /*保存动态优先级*/
int static_prio; /*保存静态优先级，可以通过nice系统调用来进行修改*/
int normal_prio; /*值取决于静态优先级和调度策略(进程的调度策略有：先来先服务，短作业优先、时间片轮转、高响应比优先等等的调度算法)*/
unsigned int rt_priority; /*保存实时优先级，一经设定在运行时不变，作为其动态优先级*/
const struct sched_class *sched_class;
struct sched_entity se;
struct sched_rt_entity rt;
unsigned int policy; /*表示进程的调度策略*/
```

每个进程都有一个优先级(称为static_prio)，但是进程的实际优先级是根据加载和其他因素动态确定的。优先级值越低，其实际优先级越高。

1. sleep_avg
    - 进程的平均等待时间
    - 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
2. run_list
    - 串联在优先级队列中
    - 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_array中找到相应的run_list，从而找到其宿主结构task_struct

3. time_slice
    - 进程的运行时间片剩余大小
    - 进程的默认创建时间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
4. 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**
5. 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。
6. unsigned long rt_priority
    - 实时进程的优先级
    - sys_sched_setschedule()
7. 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

3.1.8 task_struct ptrace系统调用

ptrace 提供了一种父进程可以控制子进程运行，并可以检查和改变它的核心image。

它主要用于实现断点调试。一个被跟踪的进程运行中，直到发生一个信号。则进程被中止，并且通知其父进程。在进程中止的状态下，进程的内存空间可以被读写。父进程还可以使子进程继续执行，并选择是否忽略引起中止的信号。

```
unsigned int ptrace;
struct list_head ptraced;
struct list_head ptrace_entry;
unsigned long ptrace_message;
siginfo_t *last_siginfo; /* For ptrace use. */
ifdef CONFIG_HAVE_HW_BREAKPOINT
atomic_t ptrace_bp_refcnt;
```

成员ptrace被设置为0时表示不需要被跟踪。

3.1.9 task_struct时间数据成员

一个进程从创建到终止叫做该进程的生存期，进程在其生存期内使用CPU时间，内核都需要进行记录，进程耗费的时间分为两部分，一部分是用户模式下耗费的时间，一部分是在系统模式下耗费的时间。

```
cputime_t utime, stime; /*用于记录进程在用户态/内核态下所经过的节拍数（定时器）*/
ctime_t utimescaled, stimescaled; /*用于记录进程在用户态/内核态的运行时间，但它们以处理器的频率为刻度*/
cputime_t gtime; /*以节拍计数的虚拟机运行时间（guest time）*/
cputime_t prev_utime, prev_stime;/*记录当前的运行时间（用户态和内核态）*/
unsigned long nvcsw, nivcsw; /*自愿/非自愿上下文切换计数*/
struct timespec start_time; /*进程的开始执行时间*/
struct timespec real_start_time; /*进程真正的开始执行时间*/
unsigned long min_flt, maj_flt;
struct task_cputime cputime_expires;/*cpu执行的有效时间*/
struct list_head cpu_timers[3];/*用来统计进程或进程组被处理器追踪的时间*/
struct list_head run_list;
unsigned long timeout;/*当前已使用的时间（与开始时间的差值）*/
unsigned int time_slice;/*进程的时间片的大小*/
int nr_cpus_allowed;
unsigned long last_switch_count; /*nvcsw和nivcsw的总和*/
struct task_cputime cputime_expires; /*用来统计进程或进程组被跟踪的处理器时间，其中的三个成员对应着cpu_timers[3]的三个链表*/
```

3.1.10 task_struct信号处理信息

信号在最早的Unix系统中即被引入，用于用户态进程间通信，内核也可用信号通知进程系统所发生的时间。

```
struct signal_struct *signal;/*指向进程信号描述符*/
struct sighand_struct *sighand;/*指向进程信号处理程序描述符*/
sigset_t blocked, real_blocked;/*阻塞信号的掩码*/
sigset_t saved_sigmask; /* restored if set_restore_sigmask() was used */
struct sigpending pending;/*进程上还需要处理的信号*/
unsigned long sas_ss_sp;/*信号处理程序备用堆栈的地址*/
size_t sas_ss_size;
```

3.1.11 文件系统信息

```
/* filesystem information */
    struct fs_struct *fs;/*文件系统的信息的指针*/
/* open file information */
    struct files_struct *files;/*打开文件的信息指针*/
```

3.2 do_fork()函数

在类unix系统中，使用fork()系统调用来创建新的进程。调用fork()的进程是父进程，新创建的进程是它的子进程。

```
int main()
{
    int a=50;
    pid_t process;
    process= fork();
    if(process==0)
    printf("Child process\n");
    else
    printf("Parent process\n");
}
```

fork()系统调用之后发生的事情很有意思，因为现在有两个进程，它们必须有单独的地址空间。但是在Linux中使用了写时拷贝（copy on write），因此父进程和子进程共享进程地址空间的单一副本。

如果这两个进程中的某一个进程试图写入进程地址空间，那么将为该进程创建一个单独的副本，使它们不再共享相同的地址空间。这种技术防止了子进程多次使用exec()系统调用创建完全不同的进程时复制大量数据的浪费。

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。

在Linux中，fork()所做的是实现clone()系统调用，这个调用使用flags决定哪些资源必须在父元素和子元素之间共享。clone()系统调用反过来调用do fork()。

do_fork()在kernel/fork.c中定义。_do_fork以调用copy_process开始，后者执行生成新的进程的实际工作，并根据指定的标志复制父进程的数据。

do_fork() 函数生成一个新的进程，大致分为以下三个步骤。

3.2.1 建立进程控制结构并赋初值，使其成为进程映像

1. 在内存中分配一个 task_struct 数据结构，以代表即将产生的新进程。

```
p = alloc_task_struct();    // 分配内存建立新进程的 task_struct 结构
```

2. 把父进程 task_struct的内容拷贝到子进程的 task_struct 中。

```
 *p = *current ;  //将当前进程的 task_struct 结构的内容复制给新进程的 PCB结构
```

3. 为新进程分配一个唯一的进程标识号 PID 和 user_struct 结构。然后检查用户具有执行一个新进程所必须具有的资源。
4. 重新设置子进程task_struct 结构中那些与父进程值不同的数据成员。

```
//下面代码对父、子进程 task_struct 结构中不同值的数据成员进行赋值

if ( atomic_read ( &p->user->processes ) >= p->rlim[RLIMIT_NPROC].rlim_cur
    && !capable( CAP_SYS_ADMIN ) && !capable( CAP_SYS_RESOURCE ))
    goto bad_fork_free;

atomic_inc ( &p->user->__count);   //count 计数器加 1
atomic_inc ( &p->user->processes); //进程数加 1

if ( nr_threads >= max_threads )
    goto bad_fork_cleanup_count ;

get_exec_domain( p->exec_domain );

if ( p->binfmt && p->binfmt->module )
    __MOD_INC_USE_COUNT( p->binfmt->module ); //可执行文件 binfmt 结构共享计数 + 1
p->did_exec = 0 ;                                   //进程未执行
p->swappable = 0 ;                                  //进程不可换出
p->state = TASK_UNINTERRUPTIBLE ;                   //重置进程状态
copy_flags( clone_flags,p );                        //拷贝进程标志位
p->pid = get_pid( clone_flags );                    //为新进程分配进程标志号
p->run_list.next = NULL ;
p->run_list.prev = NULL ;
p->run_list.cptr = NULL ;

init_waitqueue_head( &p->wait_childexit );          //初始化 wait_childexit 队列

p->vfork_done  = NULL ;

if ( clone_flags & CLONE_VFORK ) {
    p->vfork_done = &vfork ;
    init_completion(&vfork) ;
}

spin_lock_init( &p->alloc_lock );

p->sigpending = 0 ;

init_sigpending( &p->pending );
p->it_real_value = p->it_virt_value = p->it_prof_value = 0 ; //初始化时间数据成员
p->it_real_incr = p->it_virt_incr = p->it_prof_incr = 0 ;    //初始化定时器结构
init_timer( &p->real_timer );
p->real_timer.data = (unsigned long)p;
p->leader = 0 ;
p->tty_old_pgrp = 0 ;
```

```
p->times.tms_utime = p->times.tms_stime = 0 ;                    //初始化进程的各种运行时间
p->times.tms_cutime = p->times.tms_cstime = 0 ;

p->lock_depth = -1 ;          // 注意：这里 -1 代表 no ,表示在上下文切换时，内核不上锁
p->start_time = jiffies ;     // 设置进程的起始时间
```

5. 设置进程管理信息，根据所提供的 clone_flags 参数值，决定是否对父进程 task_struct 中的指针 fs 、files 指针等所选择的部分进行拷贝，如果 clone_flags 参数指明的是共享而不是拷贝，则将其计数器 count 的值加 1 ，否则就拷贝新进程所需要的相关信息内容 PCB 。这个地方是区分 sys_fork() 还是 sys_clone() 。

```
if ( copy_files ( clone_flags , p ))       //拷贝父进程的 files 指针，共享父进程已打开的文件
    goto bad_fork_cleanup ;

if ( copy_fs ( clone_flags , p ))          //拷贝父进程的 fs 指针，共享父进程文件系统
    goto bad_fork_cleanup_files ;

if ( copy_sighand ( clone_flags , p ))     //子进程共享父进程的信号处理函数指针
    goto bad_fork_cleanup_fs ;

if ( copy_mm ( clone_flags , p ))
    goto bad_fork_cleanup_mm ;         //拷贝父进程的 mm 信息，共享存储管理信息

retval = copy_thread( 0 , clone_flags , stack_start, stack_size , p regs );
//初始化 TSS、LDT以及GDT项

if ( retval )
    goto bad_fork_cleanup_mm ;
```

6. 把子进程的counter设为父进程的counter值的一半

```
p->counter = (current->counter + 1) >> 1 ;//进程动态优先级，这里设置成父进程的一半,应注意的是，这里是采用位操作来实现的。
```

7. 把子进程加入到可运行队列中

```
wake_up_process(p);          //把新进程加入运行队列，并启动调度程序重新调度，使新进程获得运行机会
```

8. 结束do_fork()函数返回PID值

```
} else {
nr = PTR_ERR(p);
}
return nr;
```

3.2.2 为新进程的执行设置跟踪进程执行情况的相关内核数据结构。包括任务数组、自由时间列表 tarray_freelist 以及 pidhash[] 数组。

1. 把新进程加入到进程链表中。
2. 把新进程加入到 pidhash 散列表中，并增加任务计数值。
3. 通过拷贝父进程的上、下文来初始化硬件的上下文（TSS段、LDT以及 GDT）。

3.2.3 启动调度程序，使子进程获得运行的机会

1. 设置新的就绪队列状态 TASK_RUNING , 并将新进程挂到就绪队列中，并重新启动调度程序使其运行。
2. 向父进程返回子进程的 PID，设置子进程从 do_fork() 返回 0 值。

3.3 schedule()函数

3.3.1 Linux进程调度概述

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。

在内核中的许多地方, 如果要将CPU分配给与当前活动进程不同的另一个进程, 都会直接调用主调度器函数schedule, 从系统调用返回后, 内核也会检查当前进程是否设置了重调度标志TLF_NEDD_RESCHED

3.3.schedule主框架

schedule就是主调度器的函数, 在内核中的许多地方, 如果要将CPU分配给与当前活动进程不同的另一个进程, 都会直接调用主调度器函数schedule。

```
struct task_struct *tsk = current;
sched_submit_work(tsk);
__schedule();
```

该函数完成如下工作

1. 确定当前就绪队列, 并在保存一个指向当前(仍然)活动进程的task_struct指针
2. 检查死锁, 关闭内核抢占后调用__schedule完成内核调度
3. 恢复内核抢占, 然后检查当前进程是否设置了重调度标志TLF_NEDD_RESCHED, 如果该进程被其他进程设置了TIF_NEED_RESCHED标志, 则函数重新执行进行调度

调用函数__schedule()，schedule函数的功能全部被封装在了这个函数中；

1. 创建一些局部变量

```
asmlinkage void __sched schedule(void) {
 struct task_struct *prev, *next; //当前进程和一下个进程的进程结构体
 unsigned long *switch_count; //进程切换次数
 struct rq *rq; //就绪队列
 int cpu;
```

2. 进程替换前，schedule做的事情是用某一个进程替换当前进程
   - 关闭内核抢占，初始化一些局部变量

```
need_resched:
preempt_disable();//关闭内核抢占
cpu = smp_processor_id();
rq = cpu_rq(cpu);//与CPU相关的runqueue保存在rq中
rcu_note_context_switch(cpu);
prev = rq->curr;//将runqueue当前的值赋给prev
```

当前进程current被保存在prev，和当前CPU相关的runqueue的地址保存在rq中

   - 释放大内核锁

```
release_kernel_lock(prev);//释放大内核锁
```

   - 不允许抢占，得到当前的需要调度的cpu（多核cpu），得到该cpu的就绪队列。rcu切换，时间调试检查和统计，记录当前进程为上一个进程（此时的当前进程还没有被替换），调试当前进程是不是通过do_exit来触发调度的（这种情况下的调度需要是一个原子操作）。使用自旋锁之前做一些优化，调用自旋锁锁住运行队列，同时关中断。

```
switch_count = &prev->nivcsw; //把当prev进程切换计数的地址赋给switch_count
 if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
 //PREEMPT_ACTIVE表示此时是否正在进行内核抢占
 if (unlikely(signal_pending_state(prev->state, prev))) {
 prev->state = TASK_RUNNING; //TASK_RUNNING的定义是0
 } else {
 /*如果一个worker要睡觉，通知并询问workqueue是否要唤醒一个任务来保持并发性。如果是这样，唤醒任务。*/
 if (prev->flags & PF_WQ_WORKER) {
 struct task_struct *to_wakeup;
 to_wakeup = wq_worker_sleeping(prev, cpu);
 if (to_wakeup)
 try_to_wake_up_local(to_wakeup);
 }
 deactivate_task(rq, prev, DEQUEUE_SLEEP);
 }
 switch_count = &prev->nvcsw;
```

   - 在2.6以后内核允许cpu抢占，但是不能在任何时候都可以抢占比如终端处理的时候或者调度的时候，所以在thread_info有一个字段preempt_count，表示是否允许内核抢占，为0允许，非0不允许；

- 如果prev进程的状态是TAAK_RUNNING，则直接跳到下一个代码片。这里要解释一下为什么要判断这个状态，如果prev进程不是自愿让出自己的cpu（被内核抢占，抢占的时候prev的状态会被修改为别的），调度程序为了公平一点会再给prev进程一次运行机会（修改prev状态为就绪状态）但如果prev进程已经是就绪状态了，那就不用判断了；如果prev进程的状态不是就绪状态，并且是内核抢占导致的进程调度，调用函数signal_pending_state判断prev的状态是不是可中断、被杀死的或者未决信号集为空，则修改prev进程的状态为就绪状态；否则删除prev进程，prev进程的就绪队列存在标志清零；如果prev进程在工作对列中（申请了I/O资源被插入工作对列，那么意味着prev进程要进入睡眠期等待资源来唤醒），prev进程开始沉睡，然后在工作队列唤醒一个新进程并插入就绪队列（如果没有进程需要被唤醒则返回NULL）。

```
if (unlikely(!rq->nr_running))
idle_balance(cpu, rq);
```

3. 进程转换时
   - 选择next进程

```
put_prev_task(rq, prev);
 next = pick_next_task(rq); //挑选一个优先级最高的任务排进队列
 clear_tsk_need_resched(prev); //清除prev的TIF_NEED_RESCHED标志
 rq->skip_clock_update = 0;
```

   - 完成进程的调度

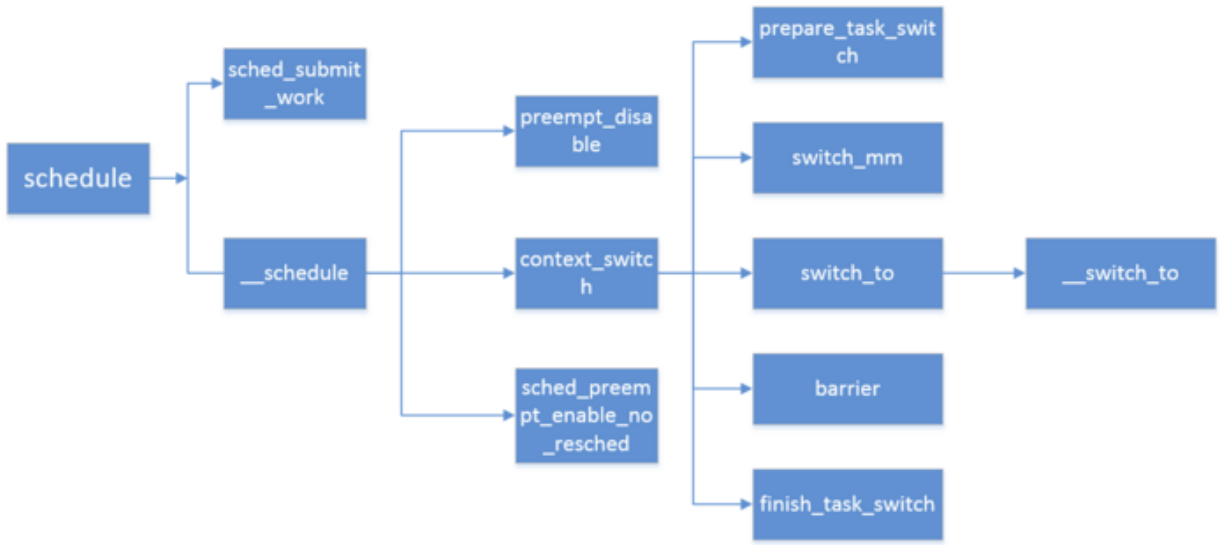
```
if (likely(prev != next)) { //如果prev和next是不同进程
 sched_info_switch(prev, next);
 perf_event_task_sched_out(prev, next);
 rq->nr_switches++; //队列切换次数更新
 rq->curr = next;
 ++*switch_count; //进程切换次数更新
 context_switch(rq, prev, next); /* unlocks the rq，进程上下文的切换 */
 /*上下文切换将堆栈从us之下翻转过来，并恢复了在这个任务调用schedule()时保存的本地变量。
prev == current仍然是正确的，但是可以移动到另一个cpu/rq*/
 cpu = smp_processor_id();
 rq = cpu_rq(cpu);
 } else //如果是同一个进程不需要切换
 raw_spin_unlock_irq(&rq->lock);
```

   - 如果rq里边没有进程（nr_running=0），启动cpu负载均衡。把prev进程放入就绪队列，从就绪队列中选出一个next进程，清除prev需要重新调度的标志，打开允许抢占
   - 如果next进程还是prev进程，解开就绪队列的自旋锁；如果next进程不是prev进程（如果就绪队列里只有一个prev进程，那重新选出来的进程还是prev进程），就绪队列中的进程切换次数加1，就绪队列的current进程换成了next进程，调度程序的进程切换次数加1。进行prev进程和next进程的上下文切换（在上下文切换中就绪队列完成解锁操作）。

```
post_schedule(rq);
 if (unlikely(reacquire_kernel_lock(prev)))
 goto need_resched_nonpreemptible;
 preempt_enable_no_resched();
 if (need_resched())
 goto need_resched;
```

- 查看是否需要调度，如果需要再次执行该程序

整个schedule的执行过程可以用下面的流程图表示：



四、总结

本次研究报告我收获很多。通过对积分内核代码的研读，我对于内核代码有了更加深刻的了解，对于内核进程的创建、管理、调度等流程有了一个较为系统的认知。

在一开始阅读源码的时候感到的些许吃力，于是转变了学习方法，先去研究别人对于函数的理解和这个函数在操作系统中的作用，在根据其作用去看其定义，对于源码的书写感受就更加深刻了。

在研究do_fork源码的时候，发现里面大量使用了goto语句，这令我阅读源码时感到了一丝困惑，因为平时老师是不建议我们在写代码的时候使用goto语句的。于是我去复习了一下goto语句。一般不推荐使用goto是因为goto的使用会造成代码难以理解和维护，调试时难以发现错误。但是当程序有多层嵌套，当处在嵌套内的逻辑判断为真或为假时，需要彻底或者连续跳出几层循环时，一般考虑使用goto，因为break一次只能跳出一层，并且需要跳出多层循环时需要更多的判断逻辑，这种情况下，会考虑使用goto；在大型程序中处理复杂逻辑时，一般也会考虑使用goto。在do_fork源码中，统一对于出错处理使用`goto 错误类型`， 在代码中的一块区域集中进行出错处理，反而使程序逻辑看起来更加清楚。

对于进程的切换，主要有两部分需要理解，一个是进程切换的时机，一个是schedule函数的调用过程。这部分的理解需要结合之前已经研究过的task_struct和do_fork函数来看，才可以更好地理解其原理。

参考文献

1. 《边干边学——LINUX内核指导（第二版）》李善平 季江民 尹康凯 等编著 胡志刚 主审

2. 《Linux-进程描述符 task_struct 详解》

   原文链接：https://www.cnblogs.com/JohnABC/p/9084750.html

3. 《浅析Linux下的task_struct结构体》

   原文链接：https://www.jianshu.com/p/691d02380312

4. Anatomy of Linux process management (Creation, management, scheduling, and destruction) By M. Jones Published December 20, 2008

   原文链接：https://developer.ibm.com/technologies/linux/tutorials/l-linux-process-management/#:~:text=Within%20the%20Linux%20kernel%2C%20a%20process%20is%20represented,maintain%20relationships%20with%20other%20processes%20%28parents%20and%20children%29.

5. 《linux内核 do_fork函数源代码浅析》

   原文链接：https://developer.aliyun.com/article/495906

6. The fork() system call By Pavan Koli

   原文链接：https://www.hackerearth.com/practice/notes/the-fork-system-call/#:~:text=In%20Linux%20what%20fork%20%28%29%20does%20is%20that,system%20call%20in%20turn%20calls%20the%20do_fork%20%28%29.

7. 《Linux进程调度器概述-Linux进程的管理与调度》

   原文链接：https://kernel.blog.csdn.net/article/details/51456569