第3节 从0，1，2号进程到系统调度

第一部分：Linus-0.11中的0，1，2号进程

注意，以下仅仅只是简单说明操作系统如何开始真正能够运行起来，也就是变成一个可以根据用户命令来执行程序的一个状态。下面涉及到的部分代码并不是完完全全原来的代码，部分被修改简化过的，只是为了更方便的理解，因此在代码逻辑上会存在一些错误，不过不影响整体理解。（仅适用于初学者）我自己的学习思路就是，先搞懂操作系统的一个设计思路，整体框架，它是如何设计让进程运行和调度起来的，然后如果有时间的话，再去详细一一击破，搞懂每一块的细节，这样做的原因主要是时间有限，无法做到学一个模块，就花一两周死磕这个模块的所有细节。（当然，操作系统不是按照面向对象的方式做出来的，而是面向过程，因此每一个模块和其他模块联系都比较紧密，也没法做到光学这个模块就能把这个模块死磕清楚的）

main程序

参考链接：https://zhuge.blog.csdn.net/article/details/79646843

```
void main(void)
{
    // 各种初始化工作
    move_to_user_mode();
    if (!fork()){                                                    //创建进程1，

            init();                                                  //如果fork是从进程1中返回，那么执行init，因此init可以看成进程1
        }

    while(1) pause();                                                 // 第一次创建完进程1，初始进程0就会进入阻塞状态，然后去调度进程1执行
}

#define move_to_user_mode() \
__asm__ ("movl %%esp,%%eax\n\t" \
    "pushl $0x17\n\t" \
    "pushl %%eax\n\t" \
    "pushfl\n\t" \
    "pushl $0x0f\n\t" \
    "pushl $1f\n\t" \
    "iret\n" \
    "1:\tmovl $0x17,%%eax\n\t" \
    "movw %%ax,%%ds\n\t" \
    "movw %%ax,%%es\n\t" \
    "movw %%ax,%%fs\n\t" \
    "movw %%ax,%%gs" \
    :::"ax")
```

首先我们从main出发，不清楚main之前做了什么事的可以回顾之前的课程。

main首先做了一堆初始化的工作，这些工作为后面文件读写，设备IO，进程管理，内存管理做准备的，暂时不用管，学到的时候自然会回过头来详细剖析每个部分的初始化工作。

然后main做的第二件事就是创建0号进程，并切换到0号进程的用户态进行执行。

主要流程如下（注意都是在main中做的工作）：

step 1：

首先创建0号进程的数据结构PCB（init_task）是一个4k页，这个PCB中我们暂时主要关心进程0的入口地址（即cs：ip）,以及进程0的用户栈和内核栈，我现在直接讲结果，分析过程后面有时间就分析，没时间网上教程也有，首先这个init_task的ldt表项和内核使用的gdt表项几乎完全一样，线性地址就是物理地址，从0开始，只不过限长不一样，这就导致0号进程看上去似乎就是内核程序。然后cs和ss（固定的索引，0x0f=1，0x17=2）对应的虚拟地址也就是物理地址了，在这种物理地址下，接下来就需要指定0号进程的内核栈和用户栈栈顶位置了。内核栈栈顶设置为4k页末尾，用户栈随便定义一个数组即可。最后就是0号进程的代码执行入口，也就是IP地址，这里IP地址设的方式比较怪，就是后面有一段move_to_user_mode的汇编代码，里面有个标号1，这个IP地址就是标号1的偏移地址，这个偏移地址其实也就是物理地址，因此1号进程的入口地址就是标号1开始的后面一些列代码。

step2：

当然，上面只是创建了这么一个0号进程的数据结构，运行进程0的时候需要硬件直接使用这个数据结构（比如寻址用到ldt表），还需要填写gdt表和关联cpu中相关寄存器后面才能让进程0正确work并使用我们创建的init_task，这个填写gdt表的工作是在`sched_init()`函数中完成的，通过执行`set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss))`宏函数来设置GDT表中任务0的TSS描述符表项，通过执行`set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt))`宏函数来设置GDT表中的任务0的LDT描述符表项。实际上就是吧tss和ldt的地址填入gdt表中第0号进程对应的tss项和ldt项。然后设置关联寄存器的工作是通过两个宏汇编ltr()和lldt()来设置的，实际上就是设置选择符（索引）,0号进程就设置0即可。这样关联好了之后，后面如果切换到了用户态执行某些代码，那么其寻址的ldt表就是0号进程的ldt表，tss也是一样。

注意，这里修改tr寄存器和ldtr寄存器的步骤，不属于准备PCB数据结构的过程，而是属于切换到这个进程的一个操作，因为是对cpu的操作。因为接下来就是要运行进程0，因此这里可以先把这两个寄存器修改了，后面再修改别的寄存器。在switch_to()中，通过输入部分的汇编代码，修改TR寄存器值，然后再通过ljmp（这个指令正好直接利用刚刚修改的TR寄存器找到tss）指令把tss中包括ldtr寄存器在内的所有寄存器值写入cpu。

step3：

准备工作全部搞定后，接下来就是想办法切换到用户态执行标号1开始的代码。

这个切换过程完全模仿int0x80的工作方式，就是int0x80到内核态后，会维护内核栈，当使用iret返回时，会使用这个内核栈切换回用户态，因此，我们这里要做的就是手动把这个内核栈依葫芦画瓢做出来，然后使用iret切到用户态模式，以用户态去执行从标号1开始的代码。

容易混淆的几点，这里重复强调一遍：

1. 0号进程是在main里面的一段代码，从标号1开始
2. 0号进程的ldt表，页表，和内核使用的gdt表，页表几乎一样
3. 0号进程的内核栈栈顶就是init_task数据结构末尾，用户栈就是内核main程序随手创建的数组末尾
4. main最后为了进入进程0，所做的工作就是初始化数据结构，关联TR，LDTR寄存器，模仿int0x80进入到进程0的用户态，执行进程0的代码

init进程

```
void init(void)
{
    while (1) {                                                  // 进入主循环
        if (!fork()) {                                          // 新创建的子进程将要执行的
内容
                execve("/bin/sh",argv,envp);                  // 执行shell进程
        }
    wait();                                                    // 等待子进程结束
    }
}
```

主要逻辑是上面这样的，实际的init前半部分是第一次fork并等待，后半部分就是循环fork等待的过程。这里的简洁版就是一个死循环，不断创建子进程壳子，然后用这个壳子执行shell进程，具体过程大致如下：

1. fork出1号进程，然后执行wait进入阻塞状态
2. 调度1号进程，1号进程从fork返回的地方开始执行，成功进入if条件分支加载shell程序执行
3. 只要不手动终止shell进程，1号进程永远处于阻塞状态，永远不会被再次调度

然后再看0号进程，自从pause进入阻塞状态后，只要1-63号进程槽非空，那么0号进程永远再也不会被调度执行，但是如果手动终止包括0号进程和1号进程在内的所有进程，那么调度程序就会调度回到0号进程（0号进程的调度并不是看它是不是阻塞状态，而是进程数组1-63为空时，强制switch_to(0)），然后0号进程进入了死循环，不停执行pause，然后不停被调度。如果没有其他方式创建新的进程，此时系统进入死机状态。

shell程序

参考链接：https://blog.csdn.net/m0_53157173/article/details/127789474

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int getcmd(char *buf, size_t size) {
    printf("Shell> ");
    return fgets(buf, size, stdin) ? 0 : -1;
}

void runcmd(char *cmd) {
    int ret = system(cmd); // 简单执行命令的示例，实际应用中可以根据需求修改
    if (ret == -1) {
        perror("system");
        exit(1);
    }
}

int main(void) {
    int pid;
    static char buf[100];
```

```
    while (getcmd(buf, sizeof(buf)) >= 0) {
        // 创建新进程
        pid = fork();
        if (pid < 0) {
            perror("fork failed");
            exit(1);
        } else if (pid == 0) {
            // 在子进程中执行命令
            runcmd(buf);
            exit(0);
        } else {
            // 在父进程中等待子进程退出
            int status;
            waitpid(pid, &status, 0);
            // 可以根据子进程的退出状态进行适当的处理
        }
    }
    return 0;
}
```

这个是从xv6里面截取的shell程序里面的主要代码片段，shell 程序就是个死循环，它永远不会自己退出，除非我们手动终止了这个 shell 进程。

在死循环里面，shell 就是不断读取（**getcmd**）我们用户输入的命令，如果读不到命令，那么就循环读，啥也不做，如果读到了一个命令，则创建一个新的进程（**fork**），在新进程里执行（**runcmd**）刚刚读取到的命令，最后等待（**wait**）进程退出，再次进入读取下一条命令的循环中。

由此可见，shell进程是一个死循环，永远不会退出。前面main和init进程将永远处于阻塞状态，不会被调度执行。

另外，所有进程正常退出时，会执行exit（其实就是do_exit）变成僵尸状态，并回收内存资源，释放文件，然后由于这些进程是父进程shell创建的，它们退出时会给父进程发送一个信号，然后调度程序启动父进程的时候，父进程会使用waitpid（sys_waitpid）来彻底销毁子进程，形象来说就是收尸。

上面的shell只能创建一个进程，然后进入等待，下面修改，使得能够启动多个进程，然后

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int getcmd(char *buf, size_t size) {
    printf("Shell> ");
    return fgets(buf, size, stdin) ? 0 : -1;
}

void runcmd(char *cmd) {
    system(cmd); // 简单执行命令的示例，实际应用中可以根据需求修改
}

int main(void) {
    int pid;
    static char buf[100];
    while (1) {
        while (getcmd(buf, sizeof(buf)) >= 0) {
```

```
        // 创建新进程
        pid = fork();
        if (pid < 0) {
            perror("fork failed");
            exit(1);
            }
        else if (pid == 0) {
            // 在子进程中执行命令
            runcmd(buf);
            exit(0);
            }
        }
        int status;
        waitpid(-1, &status, WNOHANG);
        }
    }

    return 0;
}
```

在上面提供的代码中，第一个 `while` 循环是个死循环，永远不退出，因此shell进程永远不退出。

这个shell循环等待用户输入命令，然后执行子进程，无论子进程是否执行结束，父进程都不会阻塞，继续按顺序执行，也就是waitpid啥也不做就结束了，然后进入下一次循环。

如果父进程执行的时候某个子进程结束了，在父进程被调度执行的时候，waitpid会自动检测到这个子进程僵尸，然后销毁。

第二部分：时钟

时钟参考链接：

7-3：定时器*哔哩哔哩*bilibili

1.13.时钟与中断*哔哩哔哩*bilibili

Linux 内核学习笔记：时钟中断 | Max's Blog (xiehongfeng100.github.io)

对于Linux0.11内核版本调度与睡眠机制的一些见解_内核sleep调度-CSDN博客

Linux之时钟中断-阿里云开发者社区 (aliyun.com)

Linux内核时钟机制及调度算法（硬件时钟及运作机制、内核时间系统基准源码分析、时钟中断/程序调度算法）*哔哩哔哩*bilibili

https://www.bilibili.com/video/BV1si4y1N7iR/?spm_id_from=333.788.recommend_more_video.-1&vd_source=5c649fe6538c49c0c8260759f4ebfcff

每个操作系统中会维护一个时钟对象，在linux0.11中简单来说就是由时钟中断维护一个全局变量jiffies，就是说有个时钟芯片会定时给cpu发送一个脉冲信号，这个脉冲信号会让cpu自动执行时钟中断函数do_timer，这个时钟中断函数do_timer里面会自动让jiffies加1，这样就达到了记录时间的效果（其实还会从CMOS芯片中读取具体时间，然后配合jiffies更新系统时间）。有了时间，就可以定义更多和时间相关的工具，比如定时器，原理就是先记录当前的jiffies，然后根据时钟频率计算出一段时间后的

jiffies，然后在需要定时执行的程序里面会有一个判断，判断当前的jiffiers有没有达到这个数值，如果达到了，就会执行后面需要执行的代码，比如关机。

上面只是简单对时钟有个大概的认识，具体可以参考上面的链接。不过我们这里并不是要讲关于时钟的各种东西，只讲do_timer，也就是操作系统每隔一段时间自动执行的这个中断函数，这个函数会执行调度算法，让操作系统有条不紊的执行各个进程。

do_timer()

这里核心的部分主要就是减少当前进程的时间片counter值，然后判断当前进程时间片是否用完了，如果用完了，启动调度执行其他进程。

```
// kenel/sched.c
void do_timer(long cpl)
{
    extern int beepcount;
    extern void sysbeepstop(void);

    if (beepcount)
        if (!--beepcount)
            sysbeepstop();

    if (cpl)    // 一般用户程序
        current->utime++;
    else    // 内核程序
        current->stime++;

    if (next_timer) {
        next_timer->jiffies--;
        while (next_timer && next_timer->jiffies <= 0) {
            void (*fn)(void);

            fn = next_timer->fn;
            next_timer->fn = NULL;
            next_timer = next_timer->next;
            (fn)();
        }
    }
    if (current_DOR & 0xf0)
        do_floppy_timer();
    if ((--current->counter)>0) return; // 自动减少当前进程的时间片counter值，如果时间片没有用完，则返回
    current->counter=0;
    if (!cpl) return;   // 对于内核程序，不依赖 counter 值进行调度
    schedule(); // 进程调度
}

#define TIME_REQUESTS 64
static struct timer_list {
    long jiffies;
    void (*fn)();
    struct timer_list * next;
} timer_list[TIME_REQUESTS], * next_timer = NULL;
```

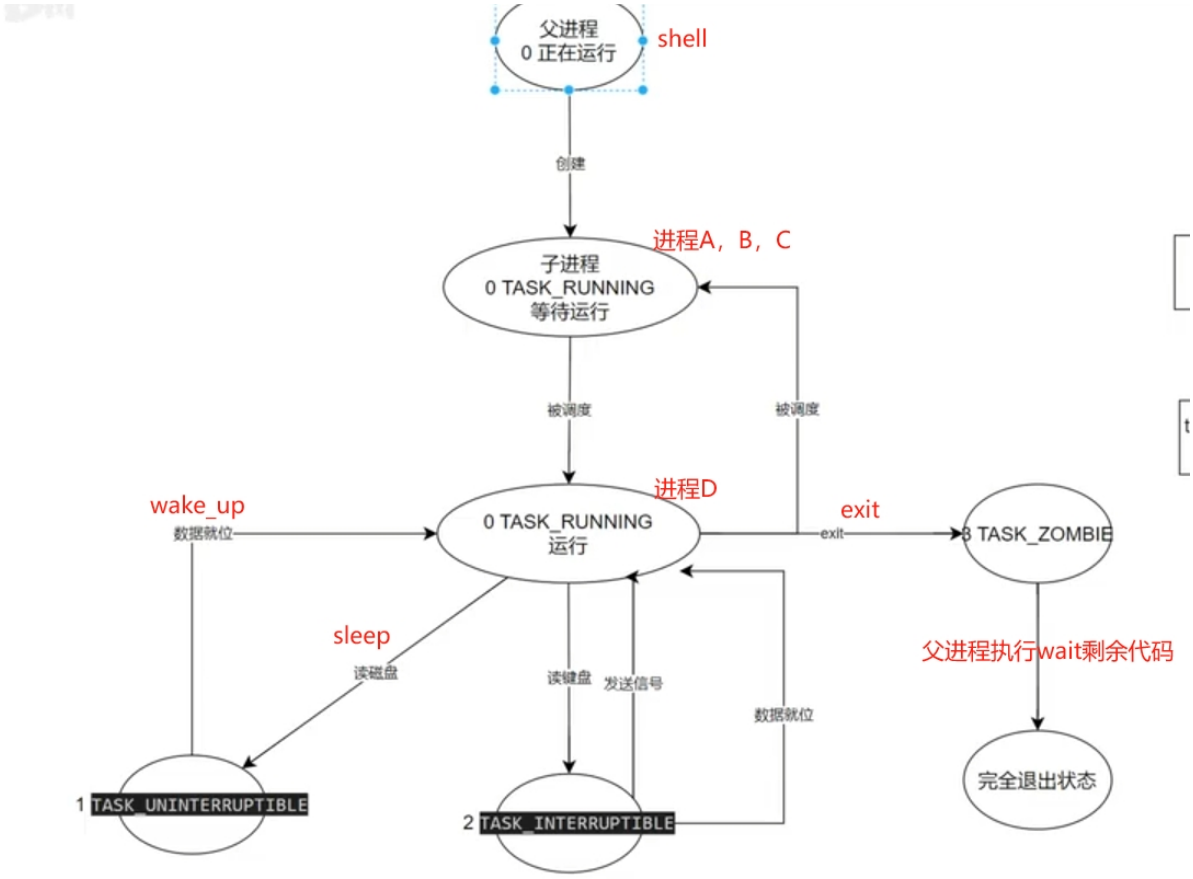
由此可见，do_timer不仅可以实时更新进程的时间片counter并调度进程，还能够更新进程的运行实际，也就是utime和stime，这样每个进程里面就包含了自己的运行时间，并且，为了方便获取进程运行时间，还提供了一个接口times，通过这个接口可以方便获得当前运行的进程时间。

参考：Linux下有关时间的函数：time,times,clock,gettimeofday等_times函数-CSDN博客

下面这个链接讲解了时钟中断如何更新系统时间和进程时间。（获取系统时间可以通过time接口）

参考：Linux之时钟中断 - Yungyu - 博客园 (cnblogs.com)

第三部分：进程调度



关于进程调度的函数，主要就是创建进程，调度进程，睡眠，唤醒，结束进程，这些函数就是改变一个进程的状态，需要把这些函数嵌入到合适的位置才能让进程在操作系统中有条不紊的运行起来。fork主要放在shell进程中使用，不停根据用户输入命令创建一个进程，调度进程主要放在do_timer里面，用于间接性的调度不同的进程给cpu运行，然后睡眠和唤醒主要发生在文件读写等过程中，让进程临时进入等待状态，当文件读写结束后，DMA控制器会发送信号给cpu，执行中断函数唤醒进程，并调度（唤醒的时候counter会很大，会被优先调度执行）。然后结束进程也会放在合适的地方执行，回收资源并销毁。

由于每个函数需要嵌入到其他地方才能讲清楚进程在干什么事的时候发生了转变，然后什么时候再次回到运行状态，因此调度程序需要结合具体的一个事情当中才能讲清楚这个函数的作用。

schdule

```
void schedule(void)
{
    int i,next,c;
    struct task_struct ** p;
```

```
/* check alarm, wake up any interruptible tasks that have got a signal */
    // 处理进程的信号和状态
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
        if (*p) {
            /*
                alarm的值是调用alarm函数设置的，见alarm函数，进程可以调用alarm函数，设置一个时间，
                然后到期后会触发alram信号，alarm < jiffies说明过期了。设置alarm信号
            */
            if ((*p)->alarm && (*p)->alarm < jiffies) {
                    (*p)->signal |= (1<<(SIGALRM-1));
                    (*p)->alarm = 0;
                }
            /*
                _BLOCKABLE为可以阻塞的信号集合，blocked为当前进程设置的阻塞集合，相与
                得到进程当前阻塞的集合，即排除进程阻塞了不能阻塞的信号，然后取反得到可以接收的
                信号集合，再和signal相与，得到当前进程当前收到的信号。如果进程处于挂起状态，则改成可执行
            */
            if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
            (*p)->state==TASK_INTERRUPTIBLE)
                (*p)->state=TASK_RUNNING;
        }

/* this is the scheduler proper: */
    // 开始调度，选择合适的进程执行
    while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        while (--i) {
            if (!*--p)
                continue;
                // 找出时间片最大的进程，说明他执行的时间最短
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }
         /*
         如果没有进程可以执行，则c等于-1，会执行进程0.如果，如果c不等于-1，
         说明有进程可以执行，但是c可能等于0或者等于1，等于0说明，进程时间片执行完了，
              则执行下面的代码重新计算时间片，c等于0则说明有进程可以执行，则进行切换
        */
        if (c) break;
        // 没有break说明c等于0，即所有的进程时间片已经执行完，需要重新设置
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                // 优先级越高，执行的时间越长，被选中执行的机会越大
                (*p)->counter = ((*p)->counter >> 1) +
                        (*p)->priority;
    }
    // 切换进程
    switch_to(next);
}
```

```
#define switch_to(n) {\
struct {long a,b;} __tmp; \
// ecx是第n个进程对应的pcb首地址，判断切换的下一个进程是不是就是当前执行的进程，是就不需要切换了
__asm__("cmpl %%ecx,_current\n\t" \
    "je 1f\n\t" \
    // 把第n个进程的tss选择子复制到__tmp.b
    "movw %%dx,%1\n\t" \
    // 更新current变量，使current变量执行ecx，ecx指向task[n]
    "xchgl %%ecx,_current\n\t" \
    // ljmp 跟一个tss选择子实现进程切换
    "ljmp %0\n\t" \
    // 忽略
    "cmpl %%ecx,_last_task_used_math\n\t" \
    "jne 1f\n\t" \
    "clts\n" \
    "1:" \
    ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
    "d" (_TSS(n)),"c" ((long) task[n])); \
}

int sys_alarm(long seconds)
{
    int old = current->alarm;

    if (old)
        old = (old - jiffies) / HZ;
    // 1秒等于100个jiffies
    current->alarm = (seconds>0)?(jiffies+HZ*seconds):0;
    return (old);
}

// 修改进程执行的优先级,满足条件的情况下increment越大优先权越低
int sys_nice(long increment)
{
    if (current->priority-increment>0)
        current->priority -= increment;
    return 0;
}
```

sleep_on()与wake_up()

sleep_on()就是一个排队睡眠函数，当某些资源不够用的时候，一堆进程相继排队睡眠，用tmp连接，然后维护一个全局头指针，当唤醒对头的进程后，

这个进程回到调度的地方，会唤醒下一个进程，然后产生连锁反应，依次唤醒每个进程。

wake_up就是单纯唤醒一个进程的函数，唤醒队列其他进程的任务在sleep_on的schdule后面实现了。

sleep_on经常在文件读写的时候用到，

- 一个是当发出请求队列完成之后，使用wait_on_buffer(里面调用sleep_on(&bh->b_wait))等待磁盘读写完成，这里出现了b_wait队列，就是一个排队睡眠的队列。

- 另一个就是在发送请求队列的时候，请求队列已经满了，这时候需要排队睡眠，等请求队列空出位置，然后调用调用wake_up(&wait_for_request)唤醒队头

下面就第二种情形进行讲解，提供的参考也是针对第二种情形，对于第一种情形，详细请见文件读写的全流程解析部分。

[linux0.11进程睡眠sleep_on函数和唤醒wake_up函数分析_linux实现ksleep和kwakeup进程同步函数-CSDN博客]

[linux0.11中的sleep_on、wake_up函数与任务等待队列的调度_linux延时函数sleep进程任务调度-CSDN博客]

```
void sleep_on(struct task_struct **p)
{
    struct task_struct *tmp;

    if (!p)
        return;
    if (current == &(init_task.task))
        panic("task[0] trying to sleep");
    tmp = *p;
    *p = current;
    current->state = TASK_UNINTERRUPTIBLE;
    schedule();
    if (tmp)
        tmp->state=0;
}
```

```
void wake_up(struct task_struct **p)
{
    if (p && *p) {
        (**p).state=0;
        *p=NULL;
    }
}
```

我们以块设备请求项队列的同步操作作为例子来分析。

网络上的详细解释如下：

首先我们已经知道进程在访问块设备的时候是通过一个中间层去调用相应的驱动程序完成块设备的读写操作，这个中间层也即块设备请求项队列request，linux0.11一共为这个队列分配了32项。但是如果系统中有大量块设备访问请求的时候，32项也是不够用的，也就是说再有进程访问块设备的话，该进程就必须睡眠等待，直到请求项队列中出现空项。因此，linux在内核中定义了一个等待请求项的任务结构指针wait_for_request，该指针指向因request资源不足而睡眠等待的任务。

为了更加清晰的说明问题，这里假定request队列已满，并且此时还有2个进程也要请求块设备操作。首先是进程A在请求过程中，发现request队列已满，则该进程就会进入睡眠状态，具体会调用sleep_on(&wait_for_request)函数。在sleep函数中首先断言p值和current的有效性，*然后语句*tmp=*p会将局部指针tmp指向wait_for_request（首次调用为NULL），语句*p=current会将wait_for_request指向当前进程，最后将当前进行状态置为不可中断状态并schedule，到此进程A将被挂起。注意进程A挂起时的代码位置，下次恢复执行后将从schedule语句的下一条语句继续执行，也即tmp判断语句处。到这里，sleep_on函数使得进程A被挂起之外，还改变两个任务结构指针变量的值，一个是全局的等待任务

指针wait_for_request，它现在指向任务A的进程结构体，另一个是进程A的局部指针tmp，它指向为NULL。

分析到这里可能还是不能看出更多的端倪，好继续往下分析。任务A一直因申请不到资源处于等待睡眠状态，而这时又有一个进程B执行块设备请求，由于请求项队列处于满状态，因此进程B同样需要调用sleep_on函数进行睡眠等待，由于任务A已经修改了wait_for_request指针的值（指向进程A），所以进程B在执行到schedul语句的时候，其tmp指针将指向进程A，而wait_for_request指针此时将指向进程B。

到这里很明显可以推出：如果某个进程因无法请求到request资源而进入进程睡眠状态时，全局指针wait_for_request始终指向最后一个进入睡眠的进程，而每个进程中的局部指针tmp将指向上一个睡眠的进程，其中第一个进程的tmp指向NULL。

另外一个非常值得注意的地方是：在sleep_on函数中，进程在调用schedule函数切换到其它进程后，而该函数并未返回，也就是说其局部指针变量tmp仍然保留在该进程的堆栈上，当该进程恢复的时候将从schedule后面一条语句继续执行。

到此进程A和进程B因无法申请到request资源而被挂起，而进程C这时完成了块设备的访问操作，它将调用end_request函数释放其占用的request资源，同时将唤醒因无法申请到request资源的进程，具体将调用wake_up(&wait_for_request)。由于进程B比进程A后进入睡眠，所以wait_for_request指针指向进程B，因此很明显wake_up函数的(**p).state=0语句会将进程B唤醒，除此之外还将wait_for_request置为NULL（初始状态）。我们知道进程B恢复之后将继续执行未完的sleep_on函数，也即判断tmp指针项是否为NULL，前面已分析进程B的tmp指向进程A，因此tmp->state=0语句将会唤醒进程A。也就是说进程B帮助进程A获得了运行的资格，另外由于进程A的tmp指针指向NULL，所以进程A直接从sleep_on函数返回。

到这里也很明显的推出：如果某一个进程调用wake_up(&wait_for_request)函数，那么所有因请求request资源而进入睡眠的进程将反向依次恢复运行（最后一个睡眠的进程先恢复），最后一个睡眠的进程由wait_for_request指针进行恢复，后面睡眠进程由前一个睡眠进程的tmp指针进行恢复，直到恢复第一个睡眠的进程为止。这个过程就像多米诺骨牌一样，一个进程恢复所有进程都恢复，而这个引线就是局部指针tmp。至于恢复后的进程是否能够再次请求到request资源就不再管了，如果还是请求不到，那继续sleep_on就行了。

而赵博士认为每次调用wake_up仅恢复wait_for_request指针指向的最后睡眠的那个任务（在tmp判断前应加上*p=tmp），所以认为linus对于sleep_on函数的实现存在bug，而经过以上的分析，可以看出linus的这种实现是可行的。

小结如下：

就是进程A块设备访问请求时，会有一个判断，看块设备访问请求队列request有没有满，如果满了，这时候就会调用sleep_on()，让全局变量wait_for_request指向当前进程，然后当前进程A的函数栈中的tmp变量指向上一个进程。然后进入阻塞状态，调度其他进程执行，注意此时当前进程还处于调用sleep_on()没有执行完的状态，这个进程的函数栈依旧还在，tmp变量依旧还在，当执行到另一个进程的时候，假如还是进行块设备访问请求，还是request满了，那么进程B也会调用sleep_on()函数，让全局变量wait_for_request指向进程B，然后进程B函数栈中的变量tmp指向上一个进程A。经过这样，只要request是满的，这些需要访问块设备的进程都会调用sleep_on()，然后维护一个队列，头指针是wait_for_request，指向最后一个访问块设备的进程，然后最后一个进程的函数栈里面的tmp指向上一个进程，依次类推。

如果某个实际，一个进程完成了块设备的访问操作，它将调用end_request函数释放其占用的request资源，同时将唤醒因无法申请到request资源的进程，具体将调用wake_up(&wait_for_request)，就是直接将wait_for_request指向的进程的阻塞状态修改为就绪状态（state=0），然后这个阻塞队列头的进程就又能够被调度了，再次被调度的时候，就会接着sleep_on()中schdule()后面的代码继续执行，它后面的代码的作用就是唤醒阻塞队列后面的一个进程，然后后面进程就又能够被调度了，然后它再次被调度

的时候，又继续唤醒后面的进程。唤醒后并执行完sleep_on()的进程，会回到调用sleep_on()的地方，再次发送块设备请求，然后进行后面要做的事情。

do_exit

简单来说就是回收进程的资源，但PCB还在，PCB需要靠父进程回收。

回收完资源，并设置进程为僵死状态后，给父进程发送一个信号，然后在调度中会重新唤醒父进程，随后父进程执行wait收尸。

```
int do_exit(long code)
{
    int i;
    // 释放代码段和数据段页表,页目录，物理地址
    free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
    free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
    for (i=0 ; i<NR_TASKS ; i++)
        // 找出当前进程的子进程
        if (task[i] && task[i]->father == current->pid) {
            // 子进程的新父进程是进程id为1的进程
            task[i]->father = 1;
            /*
             父进程没有调wait，子进程退出了，然后父进程也退出了。没人回收子进程的pcb，给init
进程发
            */
            if (task[i]->state == TASK_ZOMBIE)
                /* assumption task[1] is always init */
                (void) send_sig(SIGCHLD, task[1], 1);
        }
    // 关闭文件
    for (i=0 ; i<NR_OPEN ; i++)
        if (current->filp[i])
            sys_close(i);
    // 回写inode到硬盘
    iput(current->pwd);
    current->pwd=NULL;
    iput(current->root);
    current->root=NULL;
    iput(current->executable);
    current->executable=NULL;
    // 是会话首进程并打开了终端
    if (current->leader && current->tty >= 0)
        tty_table[current->tty].pgrp = 0;
    if (last_task_used_math == current)
        last_task_used_math = NULL;
    // 是会话首进程，则通知会话里的所有进程会话结束
    if (current->leader)
        kill_session();
    // 更新状态
    current->state = TASK_ZOMBIE;
    current->exit_code = code;
    // 通知父进程
    tell_father(current->father);
    // 重新调度进程（tell_father里已经调度过了）
```

```
    schedule();
    return (-1);    /* just to suppress warnings */
}
```

sys_wait

参考链接：

https://zhuanlan.zhihu.com/p/105610390

https://www.bilibili.com/video/BV13M411g7v2

https://www.bilibili.com/video/BV1vc411V7Cp

这个函数是父进程用来回收子进程PCB的，简单来说就是如下执行逻辑：

1. 检查需要回收的子进程，直接销毁
2. 如果检查不到，父进程进入休眠被挂起，然后调度执行其他进程
3. 当其他进程do_exit销毁时，会给父进程一个信号然后通过schdule唤醒，此时父进程重新会参与调度，当父进程再次被调度的时候，会执行repeat重新销毁子进程

```
// 等待pid进程退出，并且把退出码写到stat_addr变量
int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
{
    int flag, code;
    struct task_struct ** p;

    verify_area(stat_addr,4);
repeat:
    flag=0;
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
        // 过滤不符合条件的
        if (!*p || *p == current)
            continue;
        // 不是当前进程的子进程则跳过
        if ((*p)->father != current->pid)
            continue;
        // pid大于0说明等待某一个子进程
        if (pid>0) {
            // 不是等待的子进程则跳过
            if ((*p)->pid != pid)
                continue;
        } else if (!pid) {
            // pid等于0则等待进程组中的进程，不是当前进程组的进程则跳过
            if ((*p)->pgrp != current->pgrp)
                continue;
        } else if (pid != -1) {
            // 不等于-1说明是等待某一个组的，但不是当前进程的组，组id是-pid的组，不是该组则
跳过
            if ((*p)->pgrp != -pid)
                continue;
        }
        // else {
        //  等待所有进程
```

```
        // }
        // 找到了一个符合条件的进程
        switch ((*p)->state) {
            // 子进程已经退出,这个版本没有这个状态
            case TASK_STOPPED:
                if (!(options & WUNTRACED))
                    continue;
                put_fs_long(0x7f,stat_addr);
                return (*p)->pid;
            case TASK_ZOMBIE:
                // 子进程已经退出，则返回父进程
                current->cutime += (*p)->utime;
                current->cstime += (*p)->stime;
                flag = (*p)->pid;
                code = (*p)->exit_code;
                release(*p);
                put_fs_long(code,stat_addr);
                return flag;
            default:
                // flag等于1说明子进程还没有退出
                flag=1;
                continue;
        }
    }
    // 还没有退出的进程
    if (flag) {
        // 设置了非阻塞则返回
        if (options & WNOHANG)
            return 0;
        // 否则父进程挂起
        current->state=TASK_INTERRUPTIBLE;
        // 重新调度
        schedule();
        /*
            在schedule函数里，如果当前进程收到了信号，会变成running状态，
            如果current->signal &= ~(1<<(SIGCHLD-1)))为0，即...0000000100000... &
...111111110111111...
            说明当前需要处理的信号是SIGCHLD，因为signal不可能为全0，否则进程不可能被唤醒，
            即有子进程退出，跳到repeat找到该退出的进程，否则说明是其他信号导致了进程变成可执
行状态，
            阻塞的进程被信号唤醒，返回EINTR
        */
        if (!(current->signal &= ~(1<<(SIGCHLD-1))))
            goto repeat;
        else
            return -EINTR;
    }
    return -ECHILD;
}
```