

# 操作系统实验三

## 前言

这次实验提示还是很多，不过要看的東西很多，最后要知道Linux 0.11需要在多少个地方进行状态转移，最后向文件输出，还要写一个程序调用fork等等程序进行分析，还要输出分析log，查看分析结果，需要做的工作非常多，但是难度不算很大，只是进程转换状态的时候情况有些多，需要花很长时间。

## 实验三 进程运行轨迹的跟踪与统计

本次实验主要是通过输出一个log文件来分析进程运行轨迹，其实结果是跟踪，但是为了能够完成实验实际上需要将整个进程的流程都看一遍，变相的知道了操作系统内部进程是怎么建立的，怎么进入不同状态的，最后又是怎么消亡的，还要理解操作系统的调度算法，好在Linux 0.11的调度算法比较初级简单易懂。

## 进程是怎么表示的

实际上进程在操作系统里面大致是这么一个应该是文件，是个结构体，在sched.h文件中有详细的定义：

```
struct task_struct
{
    /* these are hardcoded - don't touch */
    long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    long counter;
    long priority;
    long signal;
    struct sigaction sigaction[32];
    long blocked; /* bitmap of masked signals */
    /* various fields */
    int exit_code;
    unsigned long start_code, end_code, end_data, brk, start_stack;
    long pid, father, pgrp, session, leader;
    unsigned short uid, euid, suid;
    unsigned short gid, egid, sgid;
    long alarm;
    long utime, stime, cutime, cstime, start_time;
    unsigned short used_math;
    /* file system info */
    int tty; /* -1 if no tty, so it must be signed */
    unsigned short umask;
    struct m_inode *pwd;
    struct m_inode *root;
    struct m_inode *executable;
```

```

unsigned long close_on_exec;
struct file *filp[NR_OPEN];
/* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
struct desc_struct ldt[3];
/* tss for this task */
struct tss_struct tss;
};

```

这里先有几个部分，比如state，这里就是进程状态的意思，之后counter就是时间片的大小，priority，优先级，在读了之后我认为这里的优先级实际上就是时间片，在Linux 0.11中，初始化的时候实际上这个优先级就是15个系统滴答时间，一个滴答时间是10ms，也就是150ms，signal是信号的意思，之后会详细的写，而blocked是阻塞信号，注意这里不是指操作系统的进程状态的信号，而是说这个进程“屏蔽”的操作系统信号，exit\_code退出代码，alarm定时器，utime,stime,cutime,start\_time都是来表示时间，应该是用户态执行时间，内核态执行时间（Linux将内核态都叫做超管状态或者是监视状态差不多就是s开头）cutime，子进程是就是child，也就是说这里是子进程在用户态执行的时间，cstime也就是子进程在内核态执行的时间，pid进程编号用来唯一的表示编号，uid用户编号，father父进程进程号，filp指使用到的文件的文件结构指针表等等。

这差不多其实就是操作系统上说的PCB，上下文的概念，fork创建进程等等操作都基于这个进行，其中还有几个关键问题解决，到底有多少状态。

如果粗略的就照课本简单的说，操作系统有五种状态，新建进程，进程就绪，进程运行，进程阻塞，进程退出，这实际上和实际情况比较起来，还是有出入的，大体上思想还是这五种但是实际上操作系统的实现和状态是不大一样的，例如：

```

#define TASK_RUNNING      0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_ZOMBIE      3
#define TASK_STOPPED      4

```

上图是实验用的Linux 0.11的内核中表述的进程状态，有五种似乎和课本上说的一样，但是实际上不是这样简单，首先在Linux 0.11 TASK\_STOPPED这个状态还没有实现其状态转换，它设计出来的目的也不是表示进程被停止，而是在调试的时候，表示暂停状态，设计是为了这个，而且还在这里没有被实现，而所谓的新建进程和就绪进程，实际上都是靠0也就是TASK\_RUNNING表示的，在Linux 0.11中，估计是单处理机的原因，有一个全局的current指针来表示当前正在运行的进程，所以实际上创建和就绪都是这个进程状态代码，只不过是current指针指不指到它的问题，之后1和2，粗略都可以交阻塞的概念，但是实际上还不是这样，这两个直译是可以打断的状态和不可以打断的状态，这两个也有着严格的区分。至于ZOMBIE倒是这次就是终止状态，不过不是就这么消失了，除了第一个系统进程0进程是系统“手工创建”以外，其他都是fork出来的现场，也就是都有父进程，那么到ZOMBIE状态之后，还需要父进程获取信息之后彻底死去，所以进程的状态转换就算是再Linux 0.11这么老的内核中，也不是课本那样简单。

至于现在，我专门下载了Linux 5.49当前最新稳定版本的Linux，看了看其的进程状态表发现：

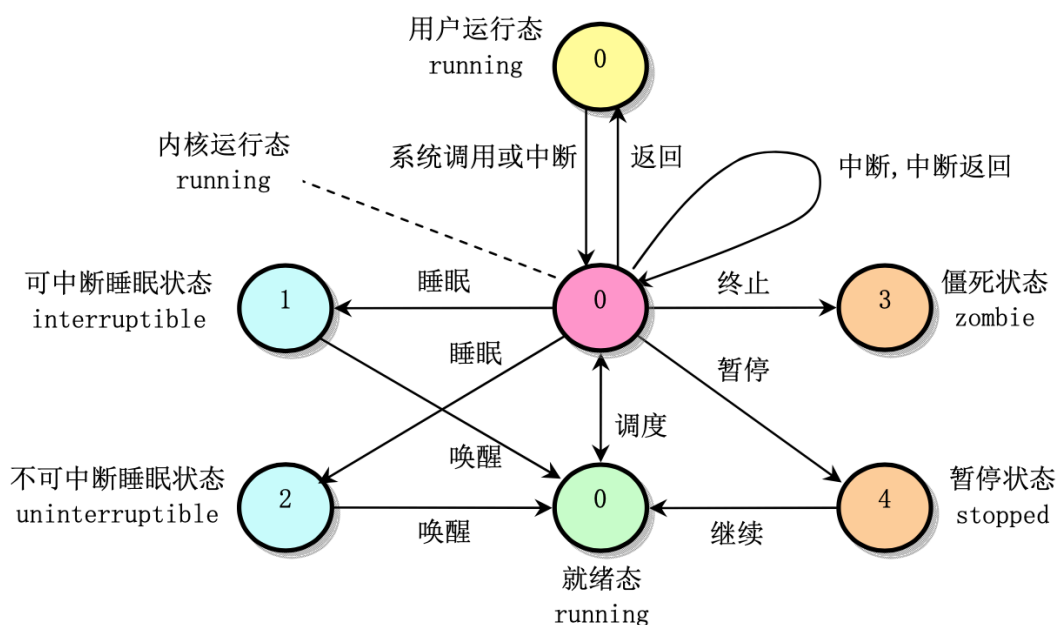
```

/* Used in tsk->state: */
#define TASK_RUNNING                0x0000
#define TASK_INTERRUPTIBLE          0x0001
#define TASK_UNINTERRUPTIBLE        0x0002
#define __TASK_STOPPED              0x0004
#define __TASK_TRACED              0x0008
/* Used in tsk->exit_state: */
#define EXIT_DEAD                   0x0010
#define EXIT_ZOMBIE                 0x0020
#define EXIT_TRACE                   (EXIT_ZOMBIE | EXIT_DEAD)
/* Used in tsk->state again: */
#define TASK_PARKED                 0x0040
#define TASK_DEAD                   0x0080
#define TASK_WAKEKILL               0x0100
#define TASK_WAKING                 0x0200
#define TASK_NLOAD                  0x0400
#define TASK_NEW                    0x0800
#define TASK_STATE_MAX              0x1000

```

更不是简单的5个可以说清楚的了。

至于进程在Linux 0.11中是如何转化的，那本书中有这样一幅图：



这一堆转化也很头大了，想新的Linux中不知道还是有多么复杂的操作。

## fork是怎么工作的

fork实现就在内核中的fork.c中实现，主要是copy\_process这个函数：

```

/*
 * Ok, this is the main fork-routine. It copies the system process
 * information (task[nr]) and sets up the necessary registers. It
 * also copies the data segment in it's entirety.
 */
int copy_process(int nr, long ebp, long edi, long esi, long gs, long none,

```

```

        long ebx, long ecx, long edx,
        long fs, long es, long ds,
        long eip, long cs, long eflags, long esp, long ss)
{
    struct task_struct *p;
    int i;
    struct file *f;

    //首先系统要在任务数组中找到一个还没有被任何进程使用的空槽, Linux 0.11这里最多允许64个线程
    p = (struct task_struct *)get_free_page();
    if (!p)
        return -EAGAIN; //如果找不到就要报错, 没有空槽
    //申请一页内存
    task[nr] = p;
    //下面复制当前进程任务数据作为新进程的模版
    *p = *current; /* NOTE! this doesn't copy the supervisor stack */
    /*防止这个时候还没有被完全新建的进程被进程调度程序被调度程序执行
    /*设置为不能中断状态, 相当于现在是创建状态
    p->state = TASK_UNINTERRUPTIBLE;
    p->pid = last_pid;
    p->father = current->pid; /*设置当前进程设置为新建进程的父进程
    p->counter = p->priority;
    p->signal = 0;
    p->alarm = 0;
    p->leader = 0; /* process leadership doesn't inherit */
    p->utime = p->stime = 0;
    p->cutime = p->cstime = 0;
    p->start_time = jiffies; //当前时钟滴答数
    /* 这里已经设置好了时钟了, 就开始创建了
    fprintf(3, "%ld\t%c\t%ld\n", p->pid, 'N', jiffies); /*N创建状态
    p->tss.back_link = 0;
    p->tss.esp0 = PAGE_SIZE + (long)p;
    p->tss.ss0 = 0x10;
    p->tss.eip = eip;
    p->tss.eflags = eflags;
    p->tss.eax = 0;
    p->tss.ecx = ecx;
    p->tss.edx = edx;
    p->tss.ebx = ebx;
    p->tss.esp = esp;
    p->tss.ebp = ebp;
    p->tss.esi = esi;
    p->tss.edi = edi;
    p->tss.es = es & 0xffff;
    p->tss.cs = cs & 0xffff;
    p->tss.ss = ss & 0xffff;
    p->tss.ds = ds & 0xffff;
    p->tss.fs = fs & 0xffff;

```

```

p->tss.gs = gs & 0xffff;
p->tss.ldt = _LDT(nr);
p->tss.trace_bitmap = 0x80000000;
if (last_task_used_math == current)
    __asm__("clts ; fnsave %0" :: "m"(p->tss.i387));
if (copy_mem(nr, p))
{
    task[nr] = NULL;
    free_page((long)p);
    return -EAGAIN;
}
for (i = 0; i < NR_OPEN; i++)
    if ((f = p->filp[i]))
        f->f_count++;
if (current->pwd)
    current->pwd->i_count++;
if (current->root)
    current->root->i_count++;
if (current->executable)
    current->executable->i_count++;
set_tss_desc(gdt + (nr << 1) + FIRST_TSS_ENTRY, &(p->tss));
set_ldt_desc(gdt + (nr << 1) + FIRST_LDT_ENTRY, &(p->ldt));
/*就绪
p->state = TASK_RUNNING;                /* do this last, just in case */
fprintf(3, "%ld\t%c\t%ld\n", p->pid, 'J', jiffies); /*就绪状态
return last_pid;
}

```

这里给出的代码我已经有写注释和加入这次实验要求的fprintf了，这里先不说，而是看其的工作流程，差不多都写在上面的注释中了，可以看到其基础是需要current当前的进程作为父进程，当创建时间确定了之后就相当于创建状态，之后最后进入就绪状态，很有趣的是这个地方，最开始：

```

/*防止这个时候还没有被完全新建的进程被进程调度程序被调度程序执行
/*设置为不能中断状态，相当于现在是创建状态
p->state = TASK_UNINTERRUPTIBLE;

```

这也从侧面表述了这个TASK\_UNINTERRUPTIBLE到底是什么状态，而且也可以发现，这个进程是在父进程的基础上拷贝之后修改得到。在Linux 0.11中最多有64个进程，那么最早的进程呢，实际上在main函数中最先得系统去创建一个0进程，这是最早的进程，这就没个fork的空间了，之后系统会调用fork，这样系统中除了第一个进程，其他的进程都是fork出来的。

## 进程是怎么调度的

这个地方实际是非常核心的部分，没有调度的话，没有进程之间的切换的话，那就只能一个程序执行到死了，所以需要在进程之间切换调度，这些差不多都在sched中实现了，其中很重要的函数是schedule函数，这个函数我写了很多注释，看了较长时间的书：

```

/*

```

```

* 'schedule()' is the scheduler function. This is GOOD CODE! There
* probably won't be any reason to change this, as it should work well
* in all circumstances (ie gives IO-bound processes good response etc).
* The one thing you might take a look at is the signal-handler code here.
*
* NOTE!! Task 0 is the 'idle' task, which gets called when no other
* tasks can run. It can not be killed, and it cannot sleep. The 'state'
* information in task[0] is never used.
*
* 调度函数
*/
void schedule(void)
{
    int i, next, c;
    struct task_struct **p;
    /*
    * 2020 Zhang Xuenan
    * try to generate a new about next pcb => log
    * hope will work :-)
    * --viewvos append
    */
    struct task_struct **tmp;

    /* check alarm, wake up any interruptible tasks that have got a signal */
    /*
    * 检查现在的alarm, alarm是进程的报警定时器, 如果进程使用系统调用alarm()设置的字段值
    * (alarm函数会把秒数换成滴答数, 加上现在系统的滴答数字存在这里) 之后当系统的滴答数大于
    * 这个alarm值的时候, 内核就会想这个进程发送一个SIGALRM (14) 信号, 默认这个信号会终止
    * 程序的执行, 也可以使用信号捕捉函数 (signal或者sigaction) 来捕捉信号之后进行指定操作
    * --viewvos commit
    * signal 字段是当前收到信号的位图, 共有32位, 每个位置都代表一种信号
    * 信号值 = (位偏移值+1) 这里Linux 0.11内核最多便有32种信号
    * 那么代码中的实际上就是算位偏移值之后或到当前信号位置上去, 这样就能让信号位图变成这个
    SIGALRM
    * --viewvos commit
    * blocked字段是进程当前不想处理的信号的阻塞位图, 这就明白为什么要用位图了, 与signal类似
    * 一位代表一个需要阻塞的状态
    * --viewvos commit
    * 这里的LAST_TASK实际上就是task[NR_TASKS-1], 而 NR_TASKS就是最大进程数64, 这就是最后
    一个
    * 进程, 而 FIRST_TASK 就是第一个 (不知道为啥要这么写) 之后就设置进行alarm操作
    * ---viewvos commit
    */
    for (p = &LAST_TASK; p > &FIRST_TASK; --p)
        if (*p)
        {
            if ((*p)->alarm && (*p)->alarm < jiffies)
            {
                (*p)->signal |= (1 << (SIGALRM - 1));
            }
        }
    }

```



```

    (*p)->alarm = 0;
}
if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
    (*p)->state == TASK_INTERRUPTIBLE)
{
    (*p)->state = TASK_RUNNING;
    fprintf(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);
}
}

/* this is the scheduler proper: */
/*
 * 扫描任务数组，比较每个就绪态任务允许时间递减，滴答计数counter的值来确定当前那个进程
 * 运行时间最少，也就是那个值最大，就表示这个进程运行时间还补偿，就选中这个进程，之后
 * 使用任务切换宏函数切换到这个进程执行。
 */
while (1)
{
    c = -1;
    next = 0;
    i = NR_TASKS;
    p = &task[NR_TASKS];
    while (--i)
    {
        if (!*--p)
            continue;
        if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
            c = (*p)->counter, next = i; /* 经典操作求最大
    }
    if (c)
        break; /*找到c了
    /*
    * 不巧的是处在TASK_RUNNING进程对时间片都用完了，
    * 这个时候内核会根据优先级重新计算每个任务需要的时间片值
    * 会对系统中所有进程（注意所有），包括睡眠中的进程重新计算counter，公式是：
    * counter = (counter/2)+priority 其实就是先 >> 1 之后 +
    * 这样的操作就对正在睡眠对进程有较高对counter值，看见最前面对while(1)了吗？
    * 说明这样对操作最后一定要找到一个进程，最后对switch_to函数来调度切换
    * --viewvos commit
    * 这是一种非常原始对调度算法，其时间复杂度是O(n)的，这样的时间复杂度是没有办法
    * 提升到很快素的速度，如果有时间我可以尝试一下简单的fix，类似桶排序的优化算了
    */
    for (p = &LAST_TASK; p > &FIRST_TASK; --p)
        if (*p)
            (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
    }
    tmp = &task[next];
    if ((*tmp)->pid != current->pid)
    {

```

```

    if (current->state == TASK_RUNNING)
    {
        fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'J', jiffies); /*R运行状态
    }
    fprintf(3, "%ld\t%c\t%ld\n", (*tmp)->pid, 'R', jiffies); /*R运行状态
}
switch_to(next);
}

```

这个函数实现了进程的调度，大致上的流程我在上面注释都写了，这里我也在需要的地方加上了需要输出到log表示系统状态切换的地方，虽然这个函数很短，但是实际上有很多可以讨论的地方，其中我们可以看到这里的进程调度函数，这里的进程调度函数可以说是最原始的了，就是unix调度函数，由于这里每个进程实际上最早创建的时候，这里的优先级其实设置的是15个系统滴答（也就是150ms）（后面修改时间片的时候会说道）所以这里的调度算法就是想办法找counter最大的，那么其剩下的运行时间也就最小，内核就会切换到这个进程，这里因为系统的内核只支持最多是64，所以这里每次都扫64个进程还算可以接受，这本质上是一个O(n)时间复杂度的算法，如果进程多的话，多扫描几次，这个部分反而非常耗时间，所以后来的Linux好几次选择新的调度算法，这实际上是非常复杂的一个话题，我本来想大体上修改一下，修改成类似Linux 2.4中的O(1)调度算法，也即是说先指定优先级，之后类似桶排序的思想，每次进程都会被丢到一个优先级桶数组位置的队列中去，这样只需要一次就可以找到当前优先级最大的进程之后调度，但是后来发现其实很复杂，最后还是只简单的修改了时间片值和加上nice值来探讨问题。

## 进程是怎么等待和唤醒的

这里很复杂，感觉都是很有技巧的函数，实际上Linux这里内部在变相的维护了一个等待队列，之后根据队列顺序依次唤醒，这是很有技巧的一个事情，也就是在下面的几个函数中：

```

int sys_pause(void)
{
    current->state = TASK_INTERRUPTIBLE;
    if (current->pid != 0)
    {
        /*这个地方主要是吧，应该把0也显示出来，可是0状态这里没事等待对状态实在是太多了
        /*一不小心十万多行出去了，所以这里屏蔽了0状态默认刷新系统
        fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
    }
    schedule();
    return 0;
}
/*
* 这个函数短但是难理解，sleep_on其实类似阻塞，当有一个任务请求当资源
* 正忙，这个时候就需要让这个进程从内存转移出去等待一段时间，之后切换回来执行
* 而放入等待队列是依赖函数中当tmp指针作为正在等待任务当联系
* 首先p是等待队列当头指针，tmp是函数堆栈上当临时指针，current是当前进程指针
* 有点类似交换两个数字需要中间变量
*/
void sleep_on(struct task_struct **p)
{

```



```

/**进入函数p指向等待队列中等待的任务结构
struct task_struct *tmp;

/**当等待队列当p没有等待队伍当时候，就不需要在这里return
if (!p)
    return;
/** Really funny! I need Add a ! ZXXN
if (current == &(init_task.task))
    panic("task[0] trying to sleep!");
tmp = *p;    /**保存原来等待当任务，其实也是为了保留p指针
*p = current; /**之后让p指向当前新的需要等待的任务
/**防调度程序调度走这个current
current->state = TASK_UNINTERRUPTIBLE;
fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
/** 开始调度
schedule();
if (tmp)
{
    tmp->state = 0;
    fprintf(3, "%ld\t%c\t%ld\n", tmp->pid, 'J', jiffies);
}
}

void interruptible_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;
repeat:
    current->state = TASK_INTERRUPTIBLE;
    fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
    schedule();
    if (*p && *p != current)
    {
        (**p).state = 0;
        fprintf(3, "%ld\t%c\t%ld\n", (**p).pid, 'J', jiffies);
        goto repeat;
    }
    *p = NULL;
    if (tmp)
    {
        fprintf(3, "%ld\t%c\t%ld\n", tmp->pid, 'J', jiffies);
        tmp->state = 0;
    }
}

```

```

}

void wake_up(struct task_struct **p)
{
    /*把等待可用资源的指定任务只设为就绪状态*/
    if (p && *p)
    {
        (**p).state = 0;
        fprintf(3, "%ld\t%c\t%ld\n", (**p).pid, 'J', jiffies);
        *p = NULL;
    }
}

```

sys\_pause其实是0也就是初始的系统进程在当现在没有其他进程的时候也会执行这个，变相的等待状态，这里屏蔽掉了，因为这个系统也不是我们工作的系统有那么多进程在工作，之后最后输出的log中大部分几百万都是0进程在等待，实在是恐怖，所以这里就屏蔽掉，这里也把需要插入的地方都加上了需要输出到log的文件中去了。

## 进程是怎么退出的

主要是在exit中实现退出，当进程的时间片用完，或者是进程主动要求停止，就会进行这个流程，每次的释放实际上是这样，先子进程要退出了，把自己标记成为僵死状态，之后找父进程状态，告诉父进程我要终止了，之后父进程被唤醒，这个时间内其实父进程都在wait函数，父函数得到这个消息之后就会进行一些操作之后退出进程，下面一个函数一个函数的分析：

```

int do_exit(long code)
{
    int i;
    /* 首先释放进程代码段占用对内存空间 */
    free_page_tables(get_base(current->ldt[1]), get_limit(0x0f));
    free_page_tables(get_base(current->ldt[2]), get_limit(0x17));
    /* 如果进程有子进程，那么就需要让1进程也就是init进程作为其所有子进程对父进程 */
    for (i = 0; i < NR_TASKS; i++)
        if (task[i] && task[i]->father == current->pid)
        {
            task[i]->father = 1;
            if (task[i]->state == TASK_ZOMBIE)
                /* assumption task[1] is always init */
                (void)send_sig(SIGCHLD, task[1], 1);
        }
    /*
     * filp是指文件指针，这里是文件结构指针表，最多32项，表项号就是文件描述符对值
     * 所以这里是要关闭进程所打开对文件，书上是这样写的，但是这里对N_OPEN是20
     * --viewvos
     */
    for (i = 0; i < NR_OPEN; i++)
        if (current->filp[i])
            sys_close(i);
}

```

```

input(current->pwd);
current->pwd = NULL;
input(current->root);
current->root = NULL;
input(current->executable);
current->executable = NULL;
if (current->leader && current->tty >= 0)
    tty_table[current->tty].pgrp = 0;
if (last_task_used_math == current)
    last_task_used_math = NULL;
/*如果进程是一个会话头进程，并且有控制中断，这里就要结束释放终端，发送SIGCHUP信号
if (current->leader)
    kill_session();
/*将当前进程置为僵死状态，这里还会保存当前信息，父进程还需要，但是已经要停止了
current->state = TASK_ZOMBIE;
fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'E', jiffies);
current->exit_code = code;
/* 通知父进程“我”终止了
tell_father(current->father);
/*前往调度函数
schedule();
return (-1); /* just to suppress warnings */
}

```

实际上他们调用的是这个函数，这个过程就是这个流程，不是直接死掉，而是标记自己为僵死，之后告诉父进程，之后这个tell\_father也在上面：

```

static void tell_father(int pid)
{
    int i;
    /*寻找父进程
    if (pid)
        for (i = 0; i < NR_TASKS; i++)
        {
            if (!task[i])
                continue;
            if (task[i]->pid != pid)
                continue;
            task[i]->signal |= (1 << (SIGCHLD - 1)); /*向父进程发送SIGCHLD信号
            return;
        }
    /* if we don't find any fathers, we just release ourselves */
    /* This is not really OK. Must change it to make father 1 */
    /*没找到，就把自己释放了
    printk("BAD BAD - no father found\n\r");
    release(current);
}

```

这里Linux当时也说了，这样不好要是找不到父亲就把自己释放了。进程数量是固定的，其实还是靠在

task数组中遍历寻找父进程。

实际上父进程在等待，靠的是下面的一个函数：

```
int sys_waitpid(pid_t pid, unsigned long *stat_addr, int options)
{
    /*
     * 其实wait调用的也是这个，参数是这样
     * return waitpid(-1,wait_stat,0);
     */
    int flag, code;
    struct task_struct **p;

    verify_area(stat_addr, 4);
repeat:
    flag = 0;
    for (p = &LAST_TASK; p > &FIRST_TASK; --p)
    {
        /*这里也是在寻找子进程
        if (!*p || *p == current)
            continue;
        if ((*p)->father != current->pid)
            continue;
        if (pid > 0)
        {
            if ((*p)->pid != pid)
                continue;
        }
        else if (!pid)
        {
            if ((*p)->pgrp != current->pgrp)
                continue;
        }
        else if (pid != -1)
        {
            if ((*p)->pgrp != -pid)
                continue;
        }
        switch ((*p)->state)
        {
        case TASK_STOPPED:
            /*
             * 这个状态交暂停状态，实际上在Linux 0.11中还没有实现对这个状态的转换处理
             * 理论上是进程收到信号SIGSTOP、SIGTSTP、或者SIGTTOU信号的时候就会进入状态
             * 之后收到SIGCONT能转移到可运行状态，主要是在调试的时候收到任何信号就会被看作
             * 进入这个状态，但是因为这里没实现，这个就被看作进程终止被处理。
             */
            if (!(options & WUNTRACED))
                continue;
```

```

    put_fs_long(0x7f, stat_addr);
    return (*p)->pid;
case TASK_ZOMBIE:
    /*
     * 当父进程发现这个子进程已经处在僵死状态，这个时候父进程会把运行时间加在自己进程中
     * 最终释放已经终止的子任务的数据结构所占用的内存页面，之后置空子进程在任务数组中
     */
    current->cutime += (*p)->utime;
    current->cstime += (*p)->stime;
    flag = (*p)->pid;
    code = (*p)->exit_code;
    release(*p); /*释放子进程，其实就是把数组那个地方置为NULL
    put_fs_long(code, stat_addr);
    return flag;
default:
    flag = 1;
    continue;
}
}
if (flag)
{
    if (options & WNOHANG)
        return 0;
    /*
     * 找到了子进程，但是子进程没有被停止，也没有僵死
     * 也就是说，子程序在睡眠或者在运行，那么父进程就要处理一下
     */
    fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
    current->state = TASK_INTERRUPTIBLE;
    schedule();
    if (!(current->signal &= ~(1 << (SIGCHLD - 1))))
        goto repeat;
    else
        return -EINTR;
}
return -ECHILD;
}

```

这时候在某些时候父进程是需要被唤醒处理一下的，在这个过程中，实际上父进程一直在等待。

这样差不多整个进程的调度流程就明晰了，实际上这里在写实验的时候又个技巧，在vscode中直接搜索->state，差不多就可以发现所有进程状态切换的位置了，之后分析，就可以理解，到底Linux是怎么实现进程调度的。

## 时间片在那里定义的

其实还是在sched.h里面写的：

```

#define INIT_TASK

/* state etc */ {

    0, 15, 15,

    /* signals */ 0, {

        {},

    },

    0, /* ec,brk... */ 0, 0, 0, 0, 0, 0, /* pid etc.. */ 0, -1, 0, 0, 0, /*
uid etc */ 0, 0, 0, 0, 0, 0, /* alarm */ 0, 0, 0, 0, 0, 0, /* math */ 0, /* fs
info */ -1, 0022, NULL, NULL, NULL, 0, /* filp */ { \

    NULL,

},

{

    {0, 0},

    /* ldt */ {0x9f, 0xc0fa00},

    {0x9f, 0xc0f200},

},

    /*tss*/ {0, PAGE_SIZE + (long)&init_task, 0x10, 0, 0, 0, 0,
(long)&pg_dir, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x17, 0x17, 0x17, 0x17, 0x17,
0x17, _LDT(0), 0x80000000, {}},

}

```

就看最开始的几行0,15,15，实际上就是制定了默认的时间片15个滴答，优先级15个滴答，所以实验最后修改时间片就是在这里修改。

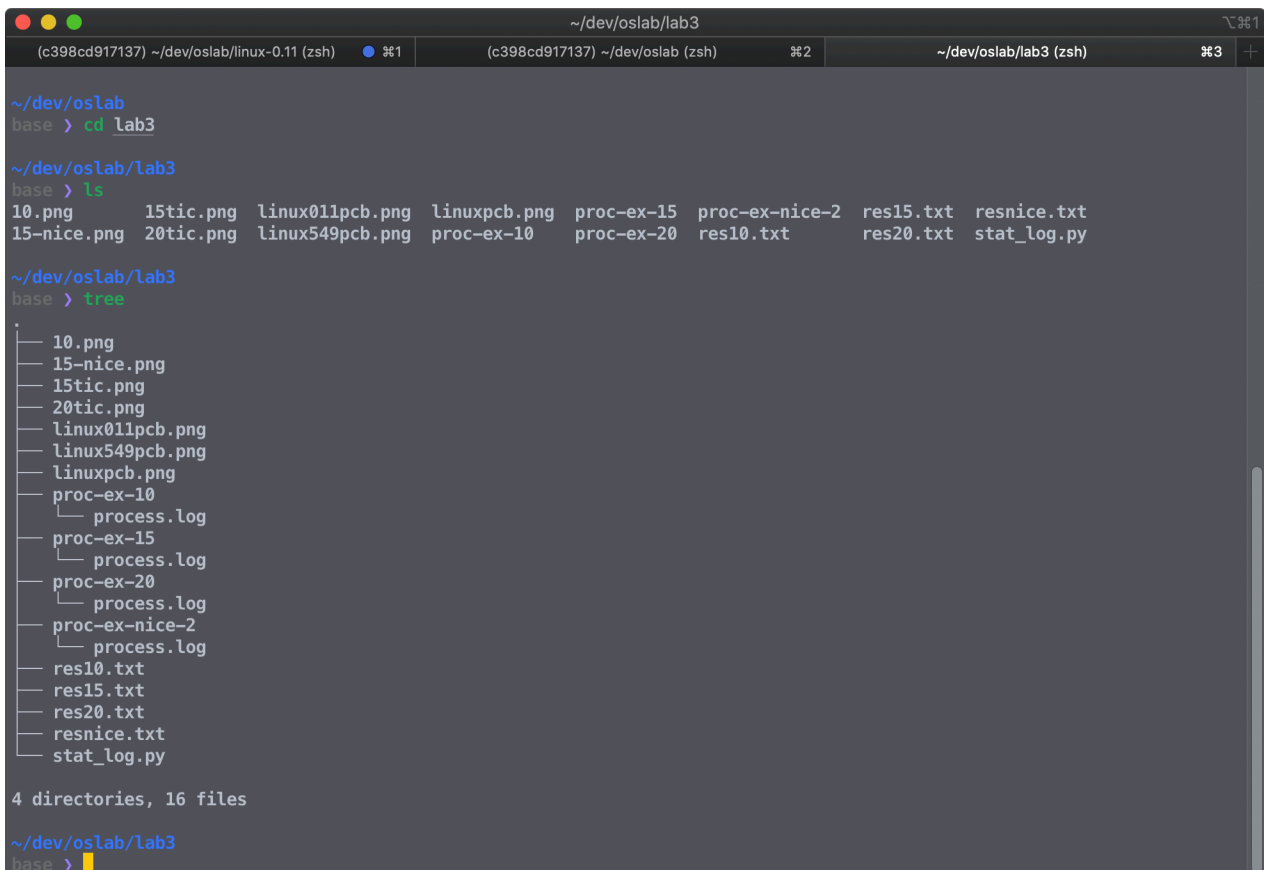


# 实验验证

这里fprintk函数，也就是内核态下的fprintf，在实验手册中已经给出，加入的方法非常简单，比第二个实验加系统调用还要简单，就是像在建议中的一样在printk.c中加入fprintk函数，之后在kernel.h中加入函数声明重新编译就成功了，和普通的写C语言的函数没啥大区别，makefile都不怎么需要改动，这样就实现了打印到log文件中去。

process.c文件简单写了一下，新建了10个进程，每个进程运行10s，其中CPU时间从第一个到0到最后一个的10，IO也是这样，其实这个地方是要考虑IO和CPU的不同性质的问题，选择不同的调度算法实际上是和亲IO还是亲CPU有关的（不知道学术上叫啥），比如上面说到的2.4内核中的O(1)调度算法，人们在之后的2.6就更换为了使用红黑树实现的完全公平调度程序，CFS，虽然算法变成了O(logn)但是其对IO支持图形化等等支持的更好，所以内核的调度是个很大的问题，不是简单的课本上概念性质的说一句，背后实现还是很复杂的，这红黑树我其实就不怎么懂，只知道是一个完全平衡二叉树罢了，的确是logn的查询速度，那这么像我用其他树，比如什么伸缩树之类的实现，总之就是很重要的算法，这里process才设计成IO和CPU时间。

这里的stat\_py文件是基于python2的，这里我简单修改了一下变成python3的，这样就可以在现在python2逐渐式微的今天正常跑了，最后结果如下：



```
~/dev/oslab/lab3
base > cd lab3

~/dev/oslab/lab3
base > ls
10.png          15tic.png      linux011pcb.png  linuxpcb.png    proc-ex-15      proc-ex-nice-2  res15.txt       resnice.txt
15-nice.png     20tic.png      linux549pcb.png  proc-ex-10      proc-ex-20      res10.txt       res20.txt       stat_log.py

~/dev/oslab/lab3
base > tree
.
├── 10.png
├── 15-nice.png
├── 15tic.png
├── 20tic.png
├── linux011pcb.png
├── linux549pcb.png
├── linuxpcb.png
├── proc-ex-10
│   └── process.log
├── proc-ex-15
│   └── process.log
├── proc-ex-20
│   └── process.log
├── proc-ex-nice-2
│   └── process.log
├── res10.txt
├── res15.txt
├── res20.txt
├── resnice.txt
└── stat_log.py

4 directories, 16 files

~/dev/oslab/lab3
base >
```

文件结构

首先是第一次实验，默认系统15时间片，最后生成的log文件在proc-ex-15中，部分截图如下：

1	1	N	48
2	1	J	48
3	0	J	48
4	1	R	48
5	2	N	49
6	2	J	49
7	1	W	49
8	2	R	49
9	3	N	64
10	3	J	64
11	2	J	64
12	3	R	64
13	3	W	68
14	2	R	68
15	2	E	74

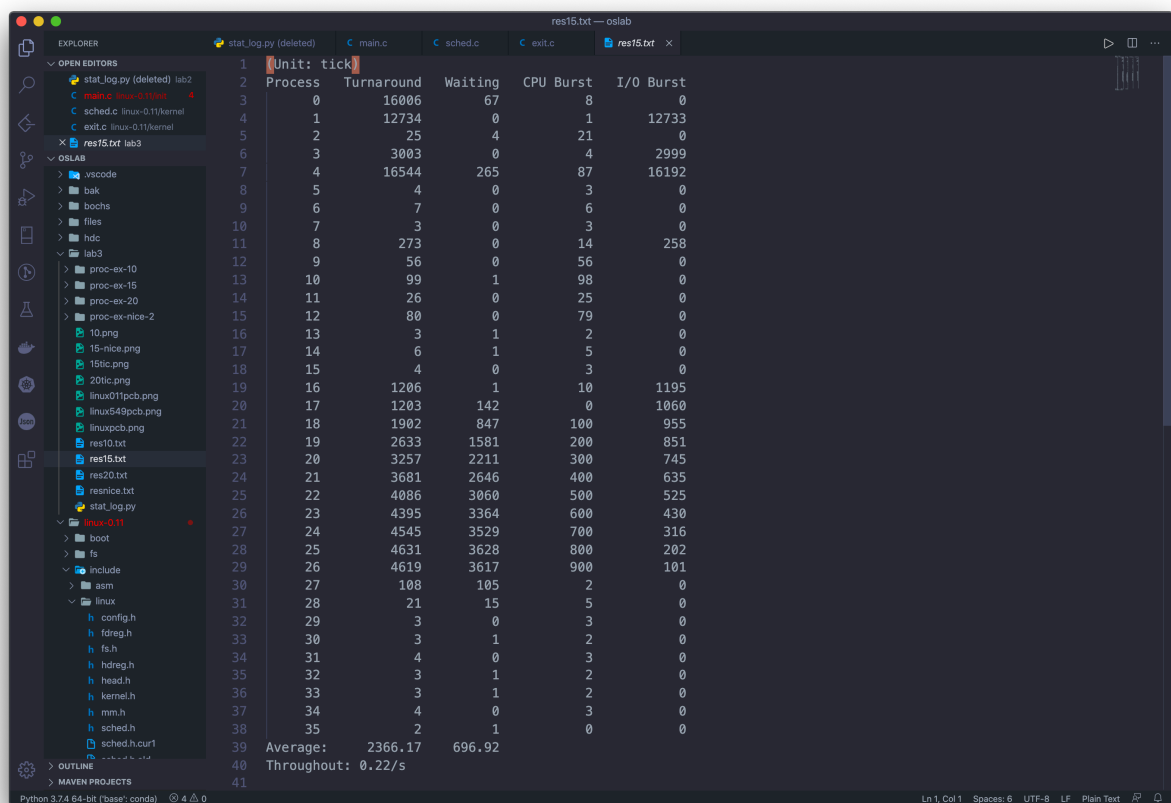
这里简单分析一下，一开始0进程创建不是根据fork创建的，那个时候感觉是没办法fprintf来写入log中的，之后0进程会创建1进程，1进程就是创建状态N，之后1进程就绪进入J状态，这个时候0进程进入就绪状态给1进程让路，之后1进程正式开始执行，进入R状态运行，之后1进程会新建2进程，之后2进程就绪，这个时候1进程阻塞等待2进程，我记得好像2是个终端创建还是什么都，反正相当于调用IO，也就类似阻塞状态，终端也是IO资源，之后2进程开始运行，创建3进程，这里我一开始是把其父进程也写打破log里面的，可是后来发现手册给的统计python程序应该只能确定这种格式，最后就没有打印父进程了，所以会混乱一点，3进程进入J就绪状态，2进程也就进入就绪状态作为父进程等待，之后3进程开始运行，3进程运行之后不知道什么原因进入了等待类似阻塞，这个时候其父进程上线，开始运行，运行之后可能由于时间片用完了，就终止状态E了，之后1状态作为其父进程就绪之后开始运行，之后创建了4状态。4状态是建立shell根据手册，之后4状态就绪，这个时候1状态就进入等待状态，shell是资源，之后4状态运行，创建5状态.....

这就是跟踪到的系统的进程调度和执行，这时候抛开这一堆分析，突然发现，每次都其实只有一个进程在运行，没有两个进程同时运行的，也就是说在这个单处理机上，一次只有一个current，进一步说明了进程这个问题。

```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
A: B: CD
USER Copy Paste Snapshot CONFIG Reset Suspend Power
README      hello.o      memtest      process-s.c  stat_log.py
gcclib140    iam.c        miao.sh      process.c    testlab2.c
hello        linux-0.00    mtools.howto shoe        testlab2.sh
hello.c      linux0.tgz    process      shoelace.tar.Z whoami.c
[/usr/root]# ./process
Child PID: 17
Child PID: 18
Child PID: 19
Child PID: 20
Child PID: 21
Child PID: 22
Child PID: 23
Child PID: 24
Child PID: 25
Child PID: 26
[/usr/root]# ls
README      hello.o      memtest      process-s.c  stat_log.py
gcclib140    iam.c        miao.sh      process.c    testlab2.c
hello        linux-0.00    mtools.howto shoe        testlab2.sh
hello.c      linux0.tgz    process      shoelace.tar.Z whoami.c
[/usr/root]# iam viewvos
[/usr/root]# whoami
viewvos[/usr/root]#
[/usr/root]# sync
[/usr/root]#
```

process的运行画面

这既是默认时间片下面的实验，而对其使用python程序进行统计之后结果放在了res15里面：



Process	Turnaround	Waiting	CPU	Burst	I/O	Burst
0	16006	67	8	0		
1	12734	0	1	12733		
2	25	4	21	0		
3	3003	0	4	2999		
4	16544	265	87	16192		
5	4	0	3	0		
6	7	0	6	0		
7	3	0	3	0		
8	273	0	14	258		
9	56	0	56	0		
10	99	1	98	0		
11	26	0	25	0		
12	80	0	79	0		
13	3	1	2	0		
14	6	1	5	0		
15	4	0	3	0		
16	1206	1	10	1195		
17	1203	142	0	1060		
18	1902	847	100	955		
19	2633	1581	200	851		
20	3257	2211	300	745		
21	3681	2646	400	635		
22	4086	3060	500	525		
23	4395	3364	600	430		
24	4545	3529	700	316		
25	4631	3628	800	202		
26	4619	3617	900	101		
27	108	105	2	0		
28	21	15	5	0		
29	3	0	3	0		
30	3	1	2	0		
31	4	0	3	0		
32	3	1	2	0		
33	3	1	2	0		
34	4	0	3	0		
35	3	1	2	0		
36	3	1	2	0		
37	4	0	3	0		
38	2	1	0	0		
Average:	2366.17	696.92				
Throughout:	0.22/s					

之后修改了时间片为20，log文件存放在了proc-ex-20文件夹下面，而分析结果存放在了res20.txt文件中，同理修改时间片为10，log文件放在了proc-ex-10文件夹下面，下面来说一下这个nice

在Linux 0.11中使用的是传统unix调度算法，公式是这样：

$$prio = USER + \frac{p_cpu}{4} + 2 \times nice$$

其实在系统函数有这么一个东西 `sys_nice(long increment)`，也就是调节nice的，但其实实际上这里只能减nice，不能减到负数的，而在Linux 0.11系统中这个计算是这样的：

```
(*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
```

其实就是：

$$counter = \frac{counter}{2} + priority$$

加上nice函数就可以实现了，所以我在这里就强行测试一次，直接在这个函数后面+2，相当于强行加了nice，之后log文件就是proc-ex-nice-2下面的文件夹，分析结果是resnice.txt

最后我们发现了问题，时间片变小了，就会导致调度次数变多最后等待时间还是长了，但是时间片要是长了，进程因中断或者睡眠进入的进程调度次数也变多了，最后等待时间还是长了，所以选择合适的时间片是很重要的，提升系统的调度函数也是很重要的。