Linux-0.11 可视化实验报告

姓名：陆宇霄
学号：201600301264
学院：泰山学堂
同组组员：张童 张延慈

一、 Linux-0.11 介绍

正如 Linux 系统的创始人 Linus 所说，要理解一个系统的真正运行机制，一定要阅读其源代码。系统本身是一个整体，具有很多看似不重要的细节。只有在详细阅读过完整的内核源代码之后，才会对整个系统的运作过程有深刻的理解，以后再选择较新的的内核源代码进行学习是，也不会碰到大问题，基本上都能顺利额理解新代码的内容。
Linux-0.11 目前的 linux 内核基本功能较为相近，又非常短小。0.11 版本的内核源代码只有一万四千行左右，其中包括的内容基本上都是 Linux 系统的精髓。

二、 实验目的

阅读 Linux-0.11 源代码，了解各个部分的工作原理，清楚进程创建和销毁的内在逻辑，实际关键帧，提取进程创建和销毁中的数据并将整个过程可视化

三、 实验环境

Linux-0.11 需要基于虚拟的环境运行，有三种虚拟的方案：VMware 公司的 VMware Workstation 软件，Connectix 公司的 Virtual PC 和开放源代码的 Bochs。不过它们有一定的区别：

- Bochs 仿真了 X86 的硬件环境（CPU 指令）以及外围设备
- VMware Workstation 仿真了一些 I/O 功能，其他部分则是在 X86 实时硬件上执行的
- Virtual PC 仿真了 X86 的大部分指令，其他部分则是采用虚拟技术实现。

我们组的实验环境基于 15 级学长搭建的实验环境，用的 Bochs 仿真方式。

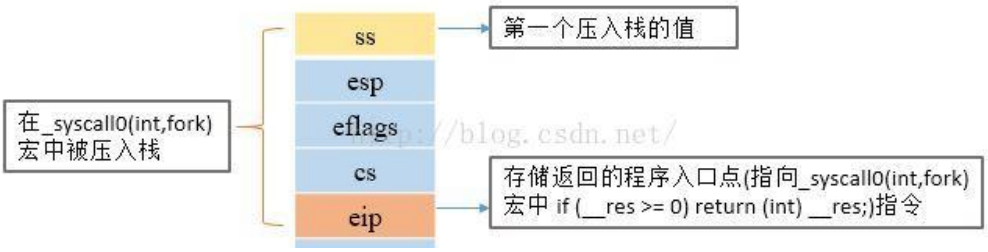
四、 linux-0.11 的进程创建

为了分析 fork，可以从它定义处开始一步一步的分析它执行的过程以及堆栈内容的变化。下面从 syscall0(int,fork)展开后的结果：

```
static inline int fork(void)
{
    long __res;
    __asm__ volatile ("int $0x80" \        //调用系统中断 0x80
    : "=a" (__res) \                       //__res 用来承载中断返回值
    : "0" (__NR_fork)); \                  //输入为系统中断调用号__NR_fork ( = 2)
if (__res >= 0) \
```

```
    return (int) __res; \          //如果返回值>=0，则直接返回该值。
errno = -__res; \                  //否则置出错号
return -1; \                       //并返回-1
}
```

从上面第 2 行代码可知，fork 执行过程的起点为“int $0x80”，通过调用系统中断 0x80 从而跳转到_system_call 中去执行。返回值__res 从 eax 寄存器中得到，当__res >= 0 时返回__res 值，否则报错并返回-1 。在调用系统中断 0x80 时，CPU 保存现场，自动把一些寄存器的值按顺序压入栈，此时栈内的内容如下图所示，把 ss、esp、eflags、cs 寄存器入栈，在调用 system_call 函数时把返回程序的入口地址也入栈。



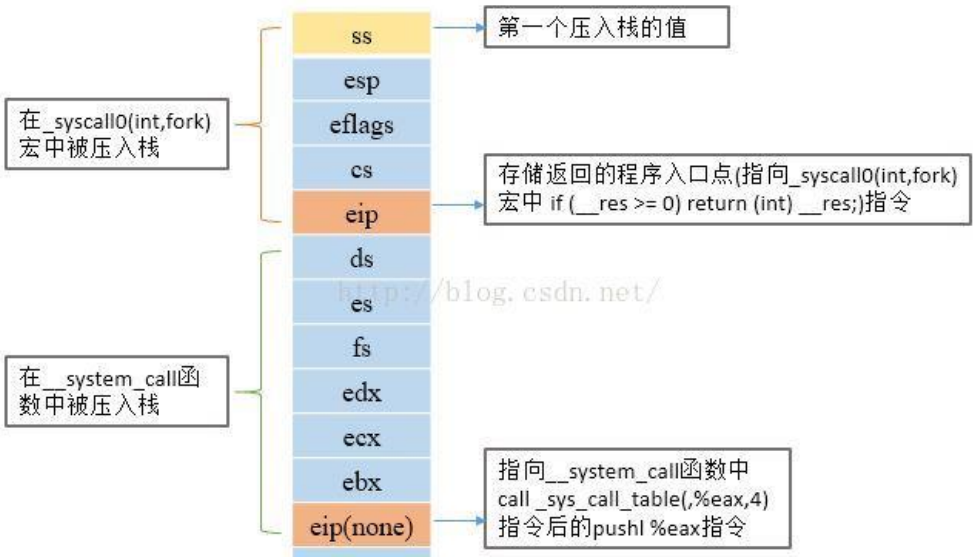
接下来，程序跳转进入/kernel/system_call.S 文件中 system_call 系统调用入口函数 _system_call 处执行。

```
.align 2
reschedule:
        pushl $ret_from_sys_call
        jmp _schedule
.align 2
_system_call:
        cmpl $nr_system_calls-1,%eax
        ja bad_sys_call
        push %ds
        push %es
        push %fs
        pushl %edx
        pushl %ecx          # push %ebx,%ecx,%edx as parameters
        pushl %ebx          # to the system call
        movl $0x10,%edx         # set up ds,es to kernel space
        mov %dx,%ds
        mov %dx,%es
        movl $0x17,%edx         # fs points to local data space
        mov %dx,%fs
        call _sys_call_table(,%eax,4)
        pushl %eax                      //%eax 刚好是_sys_fork:中 call _copy_process 的返
```

回值----last_pid，即 子进程号

```
        movl _current,%eax
        cmpl $0,state(%eax)         # state
        jne reschedule
        cmpl $0,counter(%eax)       # counter
        je reschedule
ret_from_sys_call:
        movl _current,%eax          # task[0] cannot have signals
        cmpl _task,%eax
        je 3f
        cmpw $0x0f,CS(%esp)         # was old code segment supervisor ?
        jne 3f
        cmpw $0x17,OLDSS(%esp)      # was stack segment = 0x17 ?
        jne 3f
        movl signal(%eax),%ebx
        movl blocked(%eax),%ecx
        notl %ecx
        andl %ebx,%ecx
        bsfl %ecx,%ecx
        je 3f
        btrl %ecx,%ebx
        movl %ebx,signal(%eax)
        incl %ecx
        pushl %ecx
        call _do_signal
        popl %eax
3:      popl %eax
        popl %ebx
        popl %ecx
        popl %edx
        pop %fs
        pop %es
        pop %ds
        iret
```

在__system_call 中，先把 ds,es,fs,edx,ecx,ebx 压入栈，再把 call _sys_call_table(,%eax,4)指令的返回入口地址入栈，然后调转到 sys_fork 函数处执行，此时栈内的内容为：



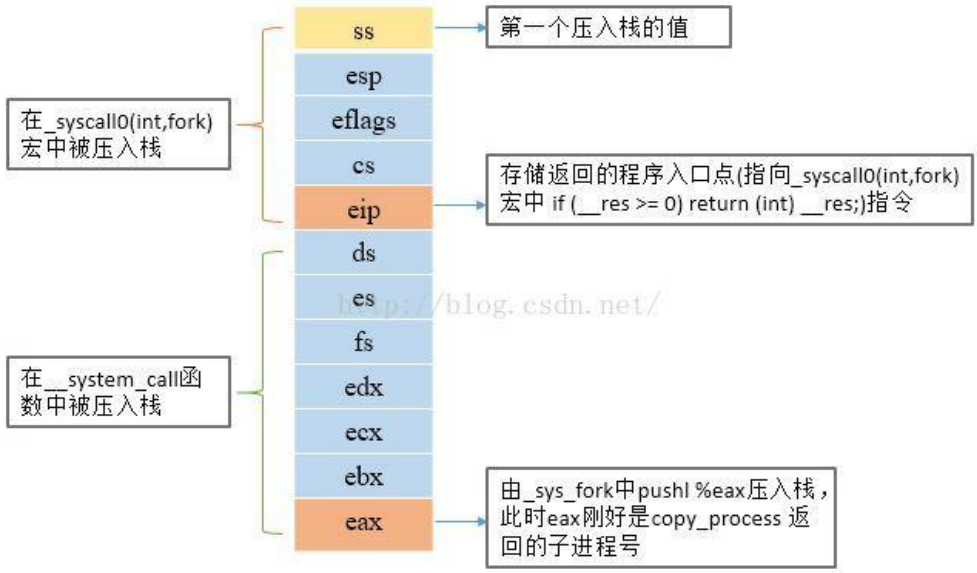
在 copy_process 函数中有如下两行代码对于 fork 生成的子进程很关键：

p->tss.eip = eip; //新的进程 b 的 TSS 里头的 eip 指向 syscall0 中的 （if (__res >= 0) return (int) __res;）指令

...

p->tss.eax = 0; //新的进程 b 的 TSS 里头的 eax 赋值为 0，当调度新进程运行时，新进程的 syscall0 中返回值__res = p->tss.eax = 0（即在新进程中 fork 返回的进程号为 0）

执行完 copy_process 函数后，返回 addl $20,%esp 指令处，把栈顶指针 esp 上移 20(栈内的 5 个项* 4 字节 = 20)，刚好把 gs,esi,edi,ebp,eax(nr)的空间忽略掉。然后使用 ret 指令返回 eip(none)中的地址，即__system_call 函数中的 pushl %eax 指令处，把 eax 寄存器的值(为 copy_process 返回的子进程号)入栈。此时栈内的内容变为：



此时，程序返回到_system_call 中，接下来判断是否进行进程调度：

```
        movl _current,%eax
        cmpl $0,state(%eax)         # state
        jne reschedule
        cmpl $0,counter(%eax)       # counter
    je reschedule
```

这些代码的意思是：先比较当前 current，即子进程的状态是否可以运行，如果当前进程不再就绪状态就去执行调度程序，如果该任务在就绪状态但时间片用完了，也就执行调度程序。所有后续的情况是，我们无法确定进程 0 或者进程 1 先执行，但是返回值已经明显确定了。

a. 对于进程 0(父进程)而言，接下来会执行 ret_from_sys_call 后的指令，进行 system_call 系统调用的退出和信号处理，其中 call _do_signal 后面的 popl %eax 表示把 子进程号 出栈存到 eax 中，返回到 syscall0 时传递给__res，表示进程 0(父进程)的 fork 返回的子进程号。

b. 对于进程 1(子进程)而言，在 schedule 函数中调度到子进程运行时，由前面 copy_process 函数可知，子进程会返回到 syscall0 中 if(__res >= 0) return (int) __res;)指令处，__res 为 0，即子进程的 fork 返回 0 ，其过程如下：

```
    .align 2
    reschedule:
        pushl $ret_from_sys_call
    jmp _schedule
```

先把_ret_from_sys_call 的地址压入栈，再跳转到 schedule 函数执行，在 schedule 函数最后会调用 switch_to 宏：

```
    #define switch_to(n) {\
    struct {long a,b;} __tmp; \
    __asm__("cmpl %%ecx,_current\n\t" \   // 比较当前任务 current 和要切换到的任务 task[n]
        "je 1f\n\t" \                     // 如果要切换到的任务是当前任务，则跳到标号 1，即结束，什么也不做，否则继续执行下面的代码
        "movw %%dx,%1\n\t" \              // 把新任务的 TSS 选择符_TSS(n) 赋值给__tmp.b 的低 16 位
        "xchgl %%ecx,_current\n\t" \   // 交换两个操作数的值，相当于 C 代码的：current = task[n] ，ecx = 被切换出去的任务（原任务）；
        "ljmp %0\n\t" \                   // 长跳转到地址&__tmp.a 中包含的 48bit 逻辑地址处：__tmp.a 即为该逻辑地址的 offset 部分，
                                          // __tmp.b 的低 16bit 为 seg_selector(高 16bit 无用)部分， 即切换到选择符_TSS(n)指定的的任务

        "cmpl %%ecx,_last_task_used_math\n\t" \   // 返回原进程后开始执行指令的地方。
```

```
        "jne 1f\n\t" \
        "clts\n" \
        "1:" \                                  // 返回_ret_from_sys_call 处
        ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
        "d" (_TSS(n)),"c" ((long) task[n])); \
}
```

switch_to 宏的核心是"ljmp %0\n\t"指令，它实现任务的切换。当它切换到子进程时，由于子进程的 tss.eip 指向 syscall0 中 if(__res >= 0) return (int) __res;)指令处，且 tss.eax = 0，所以子进程中 fork 会返回 0 值。

五、 Linux-0.11 的进程销毁

目前在看的 linux 0.11 版本的内核里面的 exit()系统调用里面就是调用了 do_exit()，它的源码如下：

```
int do_exit(long code)
{
    int i;

    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++)
        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);
        }
    for (i=0 ; i<NR_OPEN ; i++)
        if (current->filp[i])
            sys_close(i);
    iput(current->pwd);
    current->pwd=NULL;
    iput(current->root);
    current->root=NULL;
    iput(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;
    if (current->leader)
        kill_session();
    current->state = TASK_ZOMBIE;
```

```
        current->exit_code = code;
        tell_father(current->father);
        schedule();
        return (-1);    /* just to suppress warnings */
}
```

首先，调用 free_page_tables 去释放代码段和数据段，get_base 通过 ldt[1]和 ldt[2]这两个描述符来找到代码段和数据段的起始位，熟悉 LDT 的话应该知道这点。get_limit 通过两个段选择符来找到描述符进而确定代码段和数据段的段界限。free_page_tables()即要去释放页表指向的物理内存空间，也要去释放页表本身占据的内存空间，这里的释放跟内存管理有关，先不去管。

紧接着，进入一个循环，如果某个进程的父进程是当前进程，则将当前进程的父进程设置为 init 进程，在设置完之后，再去查看子进程的状态是否在僵死状态，如果是，则向该子进程的父进程发送信号，这里的父进程必定是 init 进程了，这样就可以通过 init 进程去回收该僵死进程。

之后又进入一个循环，来关闭所有该进程所打开的文件。

之后放回各个 i 结点并分别置空。

紧接着去判断该进程是否是回话头领并且占据着终端控制权，若果是的话要将终端释放。还要去终止该回话中的其他相关进程。最后要把自己设置为僵死状态，设置退出号并通向其父进程发送信号表明自己已经结束。之后运行调度程序进行进程切换。

在这里可以发现进程调用 do_exit()之后释放了绝大多数占用的资源，但是进程描述符表和内核堆栈共用的那块地址空间怎么没有释放？其实这两个过程应该是分开的，我的理解是假如子进程直接把所有的东西的释放完了，那父进程无从去得到子进程的相关信息。这就要关注另一段代码，waitpid()系统调用，其源码如下：

```
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) {
                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:
                                if (!(options & WUNTRACED))
                                        continue;
                                put_fs_long(0x7f,stat_addr);
                                return (*p)->pid;
                        case TASK_ZOMBIE:
                                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=1;
                                continue;
                }
        }
        if (flag) {
                if (options & WNOHANG)
                        return 0;
                current->state=TASK_INTERRUPTIBLE;
                schedule();
                if (!(current->signal &= ~(1<<(SIGCHLD-1))))
                        goto repeat;
                else
                        return -EINTR;
        }
        return -ECHILD;
}
```

分析：首先要明确参数对这个系统调用的影响！当 pid>0 表示等待回收该 pid 的子进程；pid=0 时，回收进程组号等于当前进程号的子进程；pid<-1 时回收进程号为-pid 的子进程；pid=-1，回收任何子进程。option 可以使该系统调用直接返回或者陷入阻塞等待；stat_addr 用来保存状态信息。

下面看分析下代码，首先遍历所有进程排除不符合条件的进程，包括空的，进程号等于自己的，不是当前进程子进程的进程，接下来的进程就都是当前进程的子进程了（也就是经过上述筛选后的进程都符合 pid=-1 时的调用了，所以后面不需要排除 pid=-1 里面的不符合条件的进程了）。接下来分析 pid。

①pid>0 时，即要等待指定的进程，所以排除掉进程号不等于该 pid 的子进程

②pid=0，即要回收进程组号等于该进程号的子进程，所以排除掉进程组号不符合的子进程

③pid<-1，排除掉进程号不等于-pid 的子进程

做完这一切选择之后，都是符合条件的子进程了。

case1：如果 WUNTRACE 置位了，也就是说不追踪，这样的话对暂停状态的信号就直接返回 pid 退出，否则继续扫描。

case2：子进程处于僵死状态，把它在用户态和内核态运行的时间累加到当前进程中，取得子进程的 pid 和退出码，这里也可以看出为什么子进程调用 do_exit()的时候为什么不直接释放掉进程描述符这个结构，因为父进程还要获取子进程描述符中上述信息。之后就可以调用 release()去释放描述符（这个函数里面会调用 schedule()函数，书上写着觉得没必要，好像是没什么必要，每回收一次描述符就要调度一次），并把调用 put_fs_long()将状态信息也就是退出码写到用户态堆栈上面，之后返回 flag 也就是子进程 pid。

default：找到过子进程，但它不出在以上两个状态，就置为 1。

执行到 if(flag)时，说明该子进程并不符合僵死或暂停，也就是执行了默认操作，假如设置了 WNOHANG，也就是不符合条件直接返回，否则，则是阻塞调用，它将自己设置为可中断等待状态，然后执行任务切换。等下次切换回来，如果还没收到子进程结束的信号，则继续遍历！

六、 可视化工具的选择

目前来说，processing 是对设计师最友好的编程软件。如果你是设计师，Processing 能用最简单的方式，最鲜活的视觉形式让你的想法通过编程实现，其功能强大，简单易学，插件很多，扩展性高，近期在编程语言排行榜的位置越来越靠前。

Processing 诞生于美国麻省理工学员媒体实验室，有旗下美学与运算小组成员的 Ben Fry 和 Casey Resa 创作完成，它直观、易上手、开源、功能强大

如果你想要实现你脑海中天马行空的创意，如果现有的工具满足不了你，也许接触下 Processing 会有新的发现。就比如，如果你想记录你看到的风景，你可以选择去学画画，也可以选择学摄影。如果你想画出各种奇特的图案，你可以选择用笔和纸，也可以选择用编程软件。摄影永远不能完全代替绘画，编程也是，但是它们都是帮助想法突破现状束缚的工具

七、 可视化过程

进程的创建与销毁中的一个个关键帧时间在代码中都是通过时间进行管理的，确定每个函数应该执行的时间段，利用 draw 函数不断的重画整张画布的特点，不断的刷新画布来实现动画的形式例如下图：

```
if(0 < millis() - start && millis() - start < 45000){
    if(millis() - start > 500){
       strokeWeight(3);
       stroke(0, (millis() - start - 500) * 255 / 1000);
       fill(0, (millis() - start - 500) * 255 / 1000);
       textFont(createFont("Arial", 24));
       text("task[1]", 115, 105);
       text("task[2]", 215, 105);
       text("task[3]", 315, 105);
       text("task[4]", 415, 105);
       text("task[5]", 515, 105);
       text("task[60]", 1015, 105);
       text("task[61]", 1115, 105);
       text("task[62]", 1215, 105);
       text("task[63]", 1315, 105);
       text("task[64]", 1415, 105);

  if(millis() - start > 1500){
    if(millis() - start < 2500){
      triangle(150, 900 - (millis() - start - 1500) * 750 / 1000, 14
      line(150, 910 - (millis() - start - 1500) * 750 / 1000, 150, 9
    }
    else{
      triangle(150, 150, 144.23, 160, 155.77, 160);
      line(150, 160, 150, 180);
      textFont(createFont("Arial", 20));
      text("avaliable", 85, 200);
    }
  }
  if(millis() - start > 2500){
    text("father's task_struct",190, 245);
    task_struct(50.0, 250.0, 500.0, 850.0, start + 2500);
  }
  if(millis() - start > 3000){
    strokeWeight(3);
    if(millis() - start < 4000){
       extend_line(180, 150, 180, 200, 1000, start + 3000);
    }
    else{
      line(180, 150, 180, 200);
      if(millis() - start < 6000){
        extend line(180, 200, 880, 200, 2000, start + 4000);
```

创建的过程主要是更新 task_struct 和 tss 两张表，因此将两张表的创建和更新数据独立成函数：

```
void update_tss(int x1, int y1, float spacing,
                int start, int last_pid, int pid, int priority,
                int jiffies, 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){
    move_rect(color(152,251,152), x1 + 350, y1 + spacing * 25, 350, spacing, 2000, start);
    move_text("back_link: " + str(0), x1+5 + 350, y1-5 + spacing * 26, 2000, start + 1000);
    move_rect(color(152,251,152), x1, y1 + spacing * 24, 700, spacing, 2000, start + 1000);
    move_text("PAGE_SIZE + (long)p", x1+5, y1-5 + spacing * 25, 2000, start + 2000);
    move_rect(color(152,251,152), x1 + 350, y1 + spacing * 23, 350, spacing, 2000, start + 2000);
    move_text("ss0: 0x10", x1+5 + 350, y1-5 + spacing * 24, 2000, start + 3000);
    move_rect(color(152,251,152), x1, y1 + spacing * 17, 700, spacing, 2000, start + 3000);
    move_text("eip: " + str(eip), x1+5, y1-5 + spacing * 18, 2000, start + 4000);
    move_rect(color(152,251,152), x1, y1 + spacing * 16, 700, spacing, 2000, start + 4000);
    move_text("eflags: " + str(eflags), x1+5, y1-5 + spacing * 17, 2000, start + 5000);
    move_rect(color(152,251,152), x1, y1 + spacing * 15, 700, spacing, 2000, start + 5000);
    move_text("eax: " + str(0), x1+5, y1-5 + spacing * 16, 2000, start + 6000);
    move_rect(color(152,251,152), x1, y1 + spacing * 14, 700, spacing, 2000, start + 6000);
    move_text("ecx: " + str(ecx), x1+5, y1-5 + spacing * 15, 2000, start + 7000);
    move_rect(color(152,251,152), x1, y1 + spacing * 13, 700, spacing, 2000, start + 7000);
    move_text("edx: " + str(edx), x1+5, y1-5 + spacing * 14, 2000, start + 8000);
    move_rect(color(152,251,152), x1, y1 + spacing * 12, 700, spacing, 2000, start + 8000);
    move_text("ebx: " + str(ebx), x1+5, y1-5 + spacing * 13, 2000, start + 9000);
    move_rect(color(152,251,152), x1, y1 + spacing * 11, 700, spacing, 2000, start + 9000);
    move_text("esp: " + str(esp), x1+5, y1-5 + spacing * 12, 2000, start + 10000);
    move_rect(color(152,251,152), x1, y1 + spacing * 10, 700, spacing, 2000, start + 10000);
    move_text("ebp: " + str(ebp), x1+5, y1-5 + spacing * 11, 2000, start + 11000);
    move_rect(color(152,251,152), x1, y1 + spacing * 9, 700, spacing, 2000, start + 11000);
    move_text("esi: " + str(esi), x1+5, y1-5 + spacing * 10, 2000, start + 12000);
    move_rect(color(152,251,152), x1, y1 + spacing * 8, 700, spacing, 2000, start + 12000);
    move_text("edi: " + str(edi), x1+5, y1-5 + spacing * 9, 2000, start + 13000);
```

```
void update_task_struct(int x1, int y1, float spacing,
                int start, int last_pid, int pid, int priority,
                int jiffies, 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){
    move_rect(color(152,251,152), x1, y1, 450, spacing, 2000, start);
    move_text("TASK_UNINTERRUPTIBLE", x1+5, y1-5 + spacing, 2000, start + 2000);
    move_rect(color(152,251,152), x1, y1 + spacing * 8, 450, spacing, 2000, start + 4000);
    move_text("pid:" + str(last_pid) + " father_pid:" + str(pid) + " leader:" + str(0), x1+5, y1-5 + spac
    move_rect(color(152,251,152), x1, y1 + spacing * 1, 450, spacing, 2000, start + 8000);
    move_text("counter:" + str(priority), x1+5, y1-5 + spacing * 2, 2000, start + 10000);
    move_rect(color(152,251,152), x1, y1 + spacing * 3, 450, spacing, 2000, start + 12000);
    move_text("signal:" + str(0), x1+5, y1-5 + spacing * 4, 2000, start + 14000);
    move_rect(color(152,251,152), x1, y1 + spacing * 11, 450, spacing, 2000, start + 16000);
    move_text("alarm:" + str(0), x1+5, y1-5 + spacing * 12, 2000, start + 18000);
    move_rect(color(152,251,152), x1, y1 + spacing * 12, 450, spacing, 2000, start + 20000);
    move_text("utime:" + str(0) + " stime:" + str(0) + " cutime:" + str(0) + " cstime:" + str(0) + " star
}
```

```
void task_struct(float x1, float y1, float x2, float y2, int start){
    int alpha = (millis() - start) / 10;
    float spacing = (y2 - y1) / 23;
    noFill();
    strokeWeight(1);
    for(int i=0; i<23; i++){
      rect(x1, y1 + i * spacing, x2 - x1, spacing);
    }
    textFont(createFont("Arial", 15));
    fill(0);
    text("long state", x1 + 10, y1 + spacing - 5);
    text("long counter", x1 + 10, y1 + 2 * spacing - 5);
    text("long priority", x1 + 10, y1 + 3 * spacing - 5);
    text("long signal", x1 + 10, y1 + 4 * spacing - 5);
    text("struct sigaction sigcation[32]", x1 + 10, y1 + 5 * spacing - 5);
    text("long blocked", x1 + 10, y1 + 6 * spacing - 5);
    text("int exit_code", x1 + 10, y1 + 7 * spacing - 5);
    text("unsigned long start_code, end_code, end_ data, brk, start_stack", x1 + 10, y1 + 8
    text("long pid, father, pgrp, session, leader", x1 + 10, y1 + 9 * spacing - 5);
    text("unsigned short uid, euid, suid", x1 + 10, y1 + 10 * spacing - 5);
    text("unsigned short git, egid, sgid", x1 + 10, y1 + 11 * spacing - 5);
    text("long alarm", x1 + 10, y1 + 12 * spacing - 5);
    text("long utime, stime, cutime, cstime, start_time", x1 + 10, y1 + 13 * spacing - 5);
    text("unsigned short used_math", x1 + 10, y1 + 14 * spacing - 5);
    text("int tty", x1 + 10, y1 + 15 * spacing - 5);
    text("unsigned short used_math", x1 + 10, y1 + 16 * spacing - 5);
    text("stuct m_inode * pwd", x1 + 10, y1 + 17 * spacing - 5);
    text("stuct m_inode * root", x1 + 10, y1 + 18 * spacing - 5);
    text("stuct m_inode * executable", x1 + 10, y1 + 19 * spacing - 5);
    text("unsigned long close_on_exec", x1 + 10, y1 + 20 * spacing - 5);
    text("struct file * filp[NR_OPEN]", x1 + 10, y1 + 21 * spacing - 5);
    text("struct desc_struct ldt[3]", x1 + 10, y1 + 22 * spacing - 5);
```

```
void tss(float x1, float y1, float x2, float y2, int start){
    int alpha = (millis() - start) / 10;
    float half_width = (x2 - x1) / 2;
    float spacing = (y2 - y1) / 26;
    if(millis() - start < 2000){
       noStroke();
    }
    else{
       strokeWeight(1);
    }

    fill(255, 192, 203, alpha);
    rect(x1, y1, half_width, spacing);
    fill(200, alpha);
    rect(x1 + half_width, y1, half_width, spacing);

    fill(200, alpha);
    rect(x1, y1 + spacing, half_width, spacing);
    fill(255, 192, 203, alpha);
    rect(x1 + half_width, y1 + spacing, half_width, spacing);

    for(int i=2; i<=7; i++){
      fill(200, alpha);
      rect(x1, y1 + spacing * float(i), half_width, spacing);
      fill(152, 251, 152, alpha);
      rect(x1 + half_width, y1 + spacing * float(i), half_width, spacing);
    }
    for(int i=8; i<=17; i++){
       fill(152, 251, 152, alpha);
       rect(x1, y1 + spacing * float(i), x2 - x1, spacing);
    }
    fill(255, 192, 203, alpha);
    rect(x1, y1 + spacing * 18, x2 - x1, spacing);
```

八、 心得体会

通过阅读源码，了解了 linux-0.11 的进程创建和销毁的过程，并且熟悉了汇编代码。在可视化的过程中，原本计划用 Qt 进行可视化，但是 Qt 的优势在于实时的与用户进行交互。但是由于系统的限制，并不能实时的提取数据，因此选择了 processing 这种可视化的工具。