

0.1 仿真内容

以 linux0.11 中进程、内存、系统调用相关的代码为核心，描述了某个简单程序在 linux0.11 运行时的操作序列。通过图形化的方式展示整个过程，会体现出进程的 fork、execve、schedule、release，内存的缺页和写保护异常等过程，也会涉及到文件系统相关的内容。

Linux 操作系统为背景，涉及

- 操作系统原理 (80%)
- 计算机组装 (30%)
- 计算机体系结构 (30%)
- 数据结构 (30%)
- 汇编语言 (50%) 等基本知识和方法

1

内核运行数据输出方法

数据提取工具为 gdb 调试工具，在 linux0.11 源代码中打断点，从而提取出内核运行相关的信息。data 文件夹中的 ff.c 文件是需要在 linux0.11 中运行的文件，断点用于监测该程序运行过程中 linux 内核所干的事情。采用虚拟机中提供的 edit_rootfs_hd 函数，将 ff.c 文件移动到 usr/root 文件夹下。在设置断点之前，用虚拟机提供的函数 run_gdb_script_hd_console_livedisplay 将 ff.c 文件通过 gcc 编译成可执行文件 fff。之后从 data 文件夹下找到 new 1 与 new 2 文件，这两个文件都是断点文件。将两个断点文件的内容输入 gdb_script_beforeboot 文件中。运行虚拟机中的函数 run_gdb_script_hd_console_redirect，在命令行输入 ./fff，待程序运行结束后退出 linux0.11。此时虚拟机桌面上得到 gdb_output.txt 文件，该文件中为提取出的初始数据。

将 gdb_output.txt 文件更名为 debug.txt。在 debug 文件中找到进程 4 创建新进程的系统调用，在该系统调用前一行加上“————begin”字段。同样在 debug 文件中找到 release 最后一个进程（一般为进程 6）的系统调用的结束处（一般为某个 ret_from_sys_call 结束处），空出一行后，加上“————end”字段。用于人工标记考虑的数据的起始和终止位置。如下图所示：

```

***pop old esp
***pop old ss

-----begin
Breakpoint 8, system_call () at system_call.s:82
82          ja bad_sys_call
---in system_call
__pid
$15409 = 4
***had push ss
***had push esp
***had push eflags
***had push cs
***had push eip
***cur register ss
ss      0x10      16
***cur register esp

```

图 1.1: begin 位置

```

Breakpoint 27, ret_from_sys_call () at system_call.s:114
114          je 3f
---in ret_from_sys_call
__pid
$16962 = 4
***not ecx
***and ebx ecx
***bsf ecx ecx
***cur register ecx
ecx      0x0      0

-----end
Breakpoint 8, system_call () at system_call.s:82|
82          ja bad_sys_call
---in system_call
__pid
$16963 = 4

```

图 1.2: end 位置

从 data 文件夹中取出 transferOS.java 文件, 将 transferOS 文件放在某个 java package 中, 在文件首加上“package 包名;”, 使得该 java 代码可以在当前包下运行。将 debug 文件放在 D 盘下, 运行 transferOS.java, 该程序将会输出中间文件 temp_data.txt, real_datan.txt, 以及目标文件 input.txt。将 input.txt 文件移动到 data 文件夹下。检查 input.txt 文件, 保证每两个数据区之间的空行为一个 (生成的文件中可能数据区间隔为多行), 同时也要将包含 “-in copy_page_tables”的相邻断点区域合为一个, 即删除其中靠后的断点区域的前 4 行, 之后再删除两个断点区域之间的空行。

处理好 `input.txt` 后,用 visual studio 打开 `WinOSTest.sln` 文件并运行,即可生成动画。文件存放位置结构如下:

Z > source > repos > WinsOSTest				
名称	修改日期	类型	大小	
.vs	2020/10/9 20:54	文件夹		
data	2020/12/16 0:27	文件夹		
WinsOSTest	2020/12/15 23:28	文件夹		
WinsOSTest.sln	2020/10/9 20:54	Visual Studio Sol...	2 KB	

图 1.3: `winsOSTest` 文件夹下

Z > source > repos > WinsOSTest > data				
名称	修改日期	类型	大小	
debug.txt	2020/11/9 16:45	文本文档	3,121 KB	
ff.c	2020/9/26 15:08	C Source	1 KB	
input.txt	2020/12/15 23:11	文本文档	107 KB	
new 1	2020/11/8 12:50	文件	19 KB	
new 2	2020/11/9 16:44	文件	6 KB	
transferOS.java	2020/12/16 0:04	JAVA 文件	19 KB	

图 1.4: `data` 文件夹下

2

可视化方法的描述

2.1 使用的工具

采用 `winform` 编写整个程序，使用了 `winform` 的基础控件，在基础控件的基础上利用 c# 自带的 `drawing` 库完成图像绘制。

2.2 工作原理

将从 `linux0.11` 内核中提取出的数据文件 `debug.txt` 作为 `java` 程序 `transferOS.java` 的输入，完成对初始数据的过滤和规范化，输出可视化程序的输入文件 `input.txt`。可视化程序 `WinOSTest` 将读入 `input.txt` 文件，根据文件中的内容生成动画。`debug.txt` 与 `input.txt` 都在 `data` 文件夹下，而 `data` 文件夹位于 `zip` 包的 `winOSTest` 文件夹下。

2.3 实现功能

根据读入的数据，决定当前正在展示的过程，以及接下来将要展示的过程。即动画展示的操作的顺序和内容都是数据驱动的。程序中有四个绘图 `pictureBox`（可视为绘图窗口），分别负责展示进程信息，展示寄存器作用，展示堆栈状态和小区域绘图，大区域绘图。绘图窗口之间会根据需要互

相切换显示。

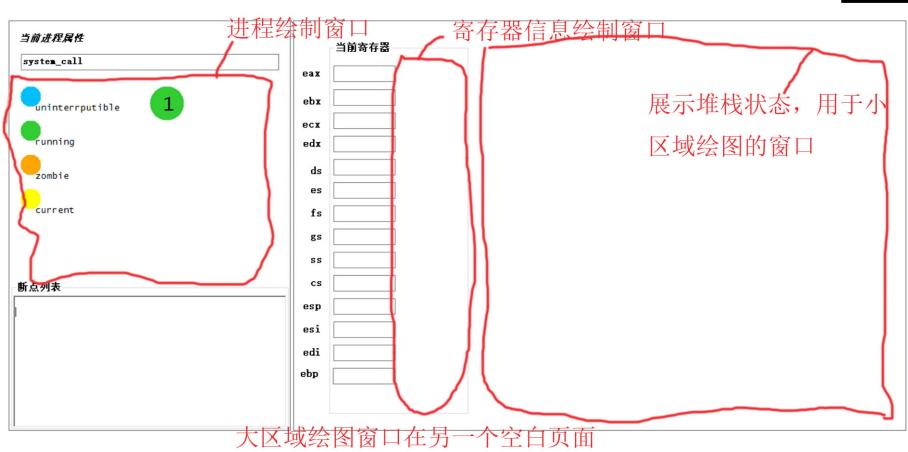


图 2.1：各个绘图窗口

展示内存相关内容时，表现的内存区中和当前过程相关的地址会被展示。展示的内存区的大小、比例会根据读入数据中按比例展示。例如：展示 `copy_pahc_tables` 时候，会根据页目录项在页目录中的地址和 1024 的比值，以及页目录绘制的起点，得到页目录项在屏幕上应当绘制的位置，从而绘制出页目录项。即展示的数据是数据驱动的，绘制出的图像的比例也是数据驱动的。

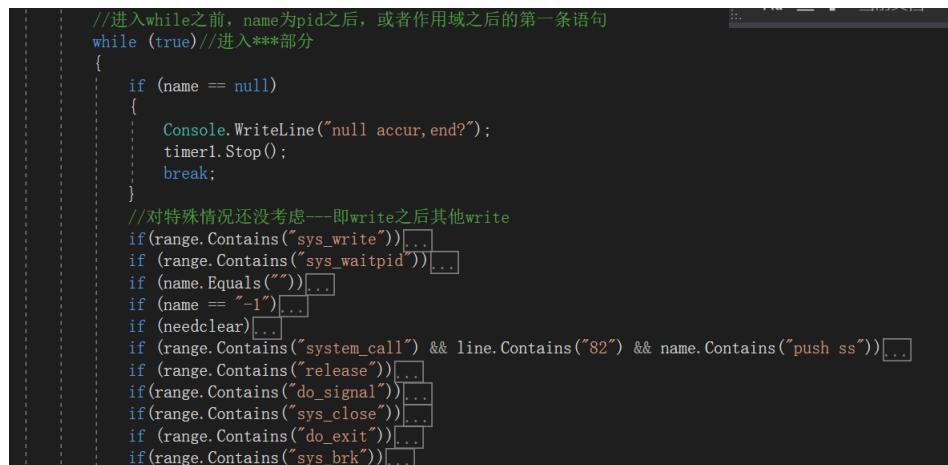
展示过程大部分采用动画的形式，辅以一些固定要展示相关数据（譬如寄存器的值）。对于各个过程会采用逐步推出的方式展示每一步应当做的动作，使得整个展示的信息量更加丰富，也使得展示更加直观。对于一些固定会展示的数据：采用圆和箭头的形式实时展示出相关进程之间的关系和所处状态。显示当前过程所在的函数，以及当前各个寄存器的值和堆栈的状态。

在过程相关动画展示结束之后，会输出进程的线性地址空间中哪些部分发生了缺页异常，哪些部分发生了写保护异常。展示了一个较为完整的程序运行过程，可以发现出现缺页异常和写保护异常的规律，以及各个操作之间的顺序关系。

2.4 程序架构

2.4.1 程序主体

该程序主体为 timer1_Tick 函数，该函数和一个 Timer 挂钩，每隔 1s 就会调用一次该函数。一个 timer1 过程会完成一个断点区域的读入（即 input 文件中两个空格之间的断点区域，可能包含一个或多个断点）。timer1_Tick 函数会进行一系列判断，例如是否需要重绘窗口（当从另一个窗口返回时需要重绘），是否有上一个断点区域多读的数据存储在缓冲区中等。timer1_Tick 在第一次读入数据时，根据读入 __pid 字段的值，绘制初始的进程节点。之后，该函数会进入循环，不断从 input 文件逐行读入数据，根据读入数据的范围（即所在的 linux 函数）和当前数据标识符，选择处理该数据的方式，进行处理；直到读空，该断点区域结束。下图为循环的部分代码：



```
//进入while之前, name为pid之后, 或者作用域之后的第一条语句
while (true)//进入***部分
{
    if (name == null)
    {
        Console.WriteLine("null accr, end?");
        timer1.Stop();
        break;
    }
    //对特殊情况还没考虑---即write之后其他write
    if(range.Contains("sys_write"))[...]
    if (range.Contains("sys_waitpid"))[...]
    if (name.Equals(""))[...]
    if (name == "-1")[...]
    if (needclear)[...]
    if (range.Contains("system_call") && line.Contains("82") && name.Contains("push ss"))[...]
    if (range.Contains("release"))[...]
    if (range.Contains("do_signal"))[...]
    if (range.Contains("sys_close"))[...]
    if (range.Contains("do_exit"))[...]
    if (range.Contains("sys_brk"))[...]
```

图 2.2：循环部分代码

上图循环中多通过对 range 所包含的字符串进行判断从而进行分支处理。循环中选择的处理方式可能直接在循环中完成处理，也可能转到其他子函数去解决。

2.4.2 主体架构

timer1_tick 在循环中读入 range，发现进入一个较为完整的大过程时，会转入对应的子函数处理。具体架构如下：

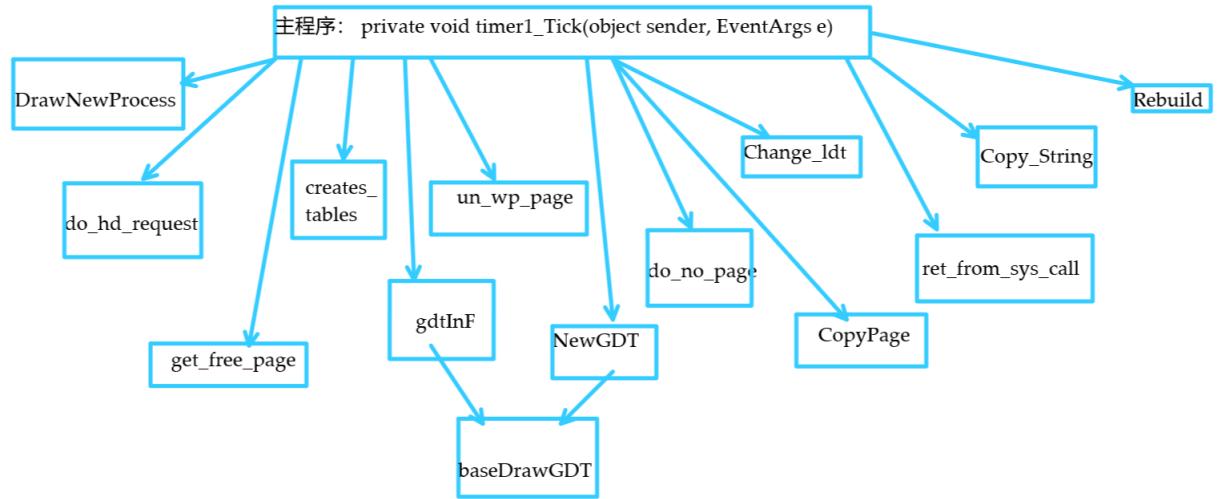


图 2.3：主体架构

上图中各个模块的名称对应于 linux0.11 元代码中的一个过程，而这个模块就是用于处理和对应的过程相关的读入数据，完成图像的绘制。例如图中 CopyPage 就是用于完成 linux 函数 copy_page_tables 过程的动画绘制。un_wp_page 就是用于绘制写保护异常时的操作过程。此外该程序中还包含一些被引用较多次数的工具函数。

2.4.3 工具函数

`drawArrow()` 函数封装了 `drawing` 中的绘图函数，能够绘制不同方向和长度的箭头。`DrawStack` 函数根据输入参数绘制不同长度的空栈。`clearStack` 函数通过调用 `DrawStack` 采用重绘的方式清空堆栈。`stacksmaller` 和 `stackLonger` 用于调整堆栈长度，`pushStack` 和 `popStack` 用于对显示的堆栈进行 `push` 和 `pop`。`buildNewNode` 用于绘制新的进程节点，`registerShow` 用于根据不同的寄存器信息，在屏幕

上不同的寄存器窗口显示数值。`clearAll` 用于清空除了显示进程的所有绘图区域，以及所有的寄存器窗口。还有一些上述子模块的功能性子函数。

2.4.4 全局参数

程序中的各个函数通过一些全局参数进行控制和数据交流，各个绘图窗口的切换状态也是通过全局参数的值进行判断。由于展示的过程涉及多个进程，展示某个过程时可能需要用到从之前数据读入的数据值。故而程序代码以全局参数的形式一定程度上模拟了 `task_struct`，记录从数据中提取出的进程相关信息。

```
string curpid;
string father;
string childpid;

String[] pids = { null, null, null, null };//假设总共只展示四个进程
int[] states = { -1, -1, -1, -1 };//0 运行1 gray 2 un 3die 4cur run
String[] structPosi = { null, null, null, null };//task struct 分配到的地址
int pidcounter = 1;//当前出现了几个进程展示
```

图 2.4: 模拟 `task_struct`

```
struct Node
{
    public string id;
    public string nr;
    public string data_base;
    public bool isFather1;
}
Node process1;//程序中创建的第一个进程
Node process2;//创建的第二个进程
```

图 2.5: 模拟 `task_struct`

`rebuildpic1` 参数用于判断是否重绘展示进程相关信息的窗口，`rebuildpic2` 用于判断是否重绘堆栈。重绘堆栈时需要借助全局栈 `drawStack` 记录的值来进行重绘。即该程序中也模拟了内核栈。

```
if (rebuildpic1)
{
    init();
    Rebuild();
    rebuildpic1 = false;
}
if (rebuildpic2)
{
    DrawStack();
    Stack<string> tempStack = new Stack<string>();
    int cc = drawStack.Count;
    for (int i = 0; i < cc; i++)
    {
        tempStack.Push(drawStack.Peek()); drawStack.Pop();
    }

    for (int i = 0; i < cc; i++)
    {
        pushStack(tempStack.Peek()); tempStack.Pop();
    }
    rebuildpic2 = false;
    Thread.Sleep(1000);
}
```

图 2.6：重绘窗口

2.5 最终效果以及相关截图

进程相关关系和状态展示窗口效果：用颜色表示进程的状态，箭头表示进程间的父子关系，箭头的 `from` 端表示父进程，`to` 端表示子进程。

当前进程属性

ret_from_sys_call

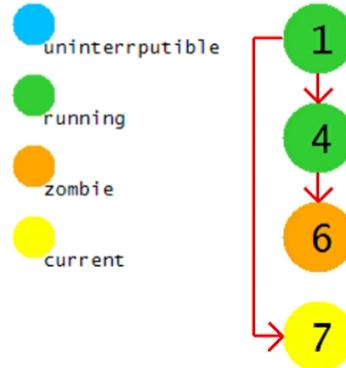


图 2.7: 进程窗口

堆栈绘制和小范围动画效果：堆栈长度和显示会随当前系统调用过程而改变，堆栈右侧区域用于显示展示当前过程的动画。

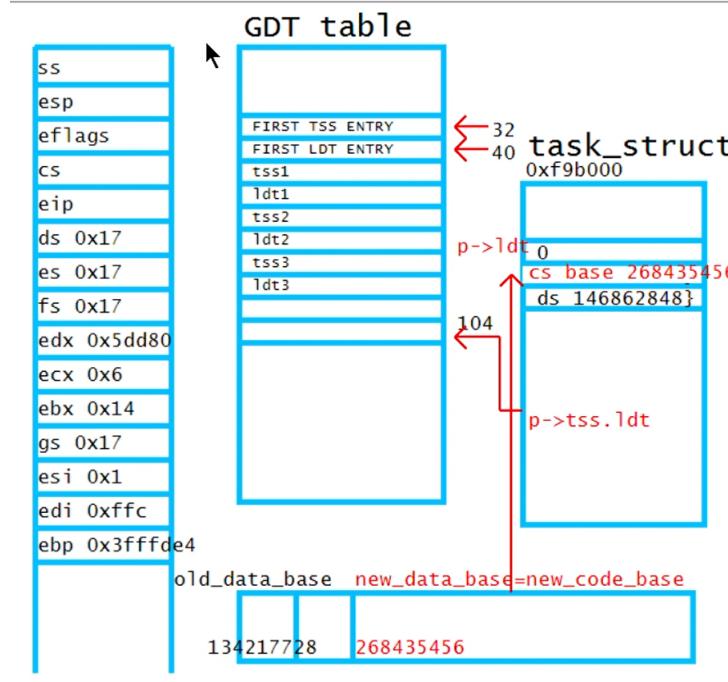


图 2.8

展示寄存器信息的绘图窗口效果：在对应寄存器的右侧以不同的颜色显示当前寄存器内容的含义。



图 2.9

大绘图区域的展示效果：将整个显示窗口用于绘图

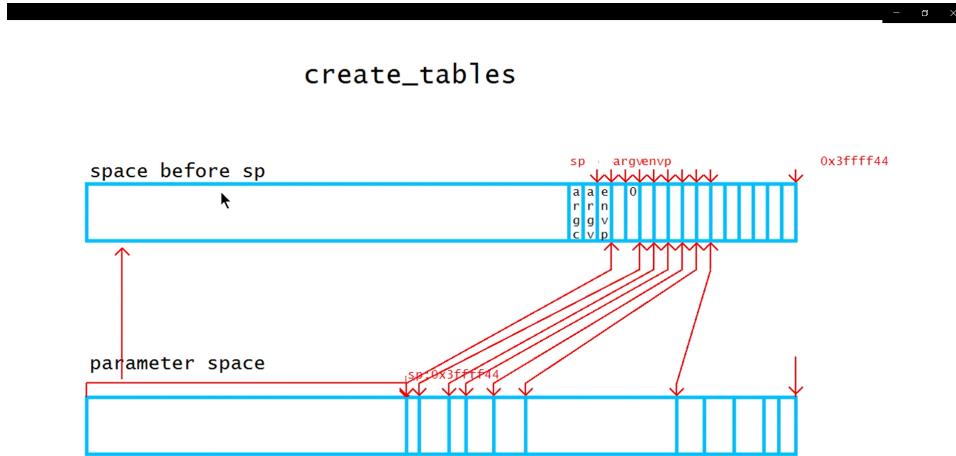


图 2.10

整体展示效果：上述大绘图区域和下面的整体区域之间的显示关系是切换的。

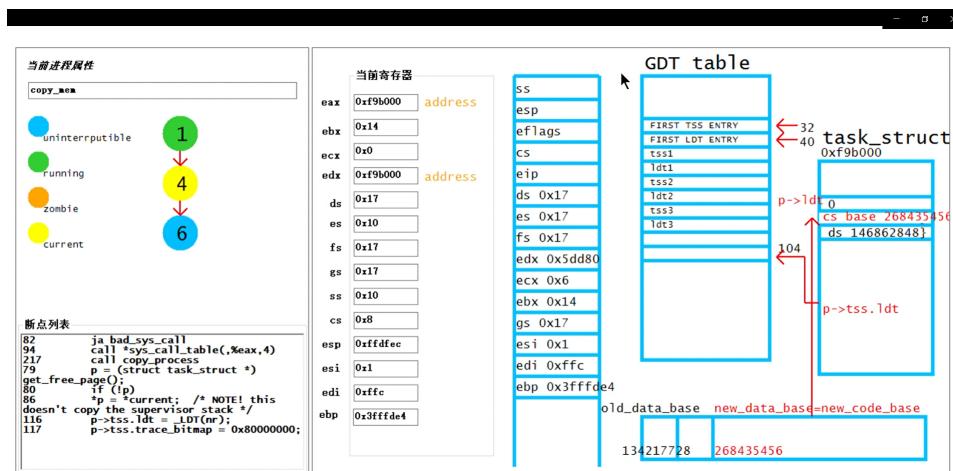


图 2.11

进程的内存展示窗口：用于展示出现缺页异常和写保护异常的页面的位置

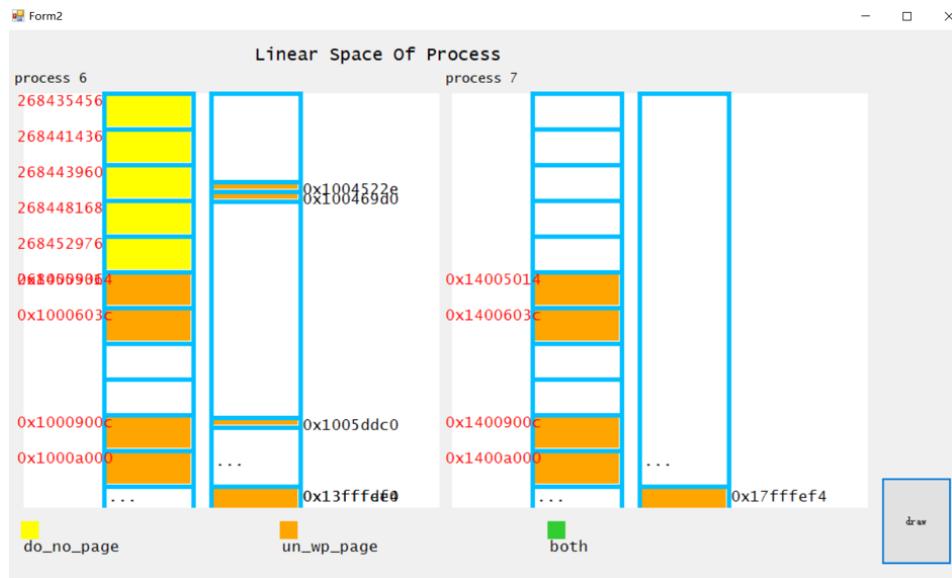


图 2.12

2.6 编译、配置、运行的具体方法

编译时使用 visual studio, 直接进入代码压缩包中的 Win0STest.sln 文件, 运行该文件。运行时应当保证 input.txt 输入文件在 Win0STest/data 文件夹下。并且需要保证 input.txt 文件中各个数据区之间的间隔为一个空行 (不能为多个)。

3

系统运行过程的形式化描述方法

3.1 描述

过程主要展示了运行一个用户程序时的进程控制和调度的具体操作，以及与它们相关的内存变化。同时也会包含页异常处理，文件系统相关的内容和涉及的系统调用的展示。假设登陆 Shell 运行在进程 4，则在命令行输入程序并开始运行时，会创建新的进程 6，进程 4 在继续运行的过程中会产生写保护异常。进程 4 通过 schedule 切换到进程 6，进程 6 发生写保护异常，进程 6 设置一些信号的句柄，进程 6 execve 开始真正执行用户程序。进程 6 发生缺页异常，之后打开文件，打开文件后又发生缺页异常。缺页异常与重设数据段末尾交替进行，后尝试对文件输入输出进行控制。进程 6 创建进程 7，进程 6 发生写保护异常，之后尝试输入输出控制。输入输出设置成功后，进程 6 开始调用 sys_write 向终端写入字符。第一次 sys_write 会引发写保护异常，之后进行用户循环中要求的次数的 sys_write。在最后一次向终端写入字符后，sys_write 中调用 file_write，进行文件写入。进程 6 关闭打开的文件并 exit，将进程 7 挂在进程 1 下，schedule 到进程 7。刚进入进程 7 时，进程 7 发生写保护异常，之后进行输入输出设置。输入输出设置之后进行 sys_write，具体过程与进程 6 时相同。进程 7 退出之后，schedule 到进程 1，进

程 1 中彻底释放进程 7, 进程 1 调用 `sys_waitpid`, `schedule` 到进
程 4, 进程 4 收到 `SIGCHLD`, 发生写保护异常, 释放进程 6.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
    FILE *fp=fopen("test", "a+");
    int pid;
    if(!(pid=fork()))
    {
        int i=0;
        while(1)
        {
            printf("input B\n");
            fputs("B", fp);
            i++;
            if(i>10)
            {
                break;
            }
        }
        if(pid>0)
        {
            int i=0;
            while(1)
            {
                printf("input A\n");
                fputs("A", fp);
                i++;
                if(i>10)
                {
                    break;
                }
            }
            fclose(fp);
            return 0;
        }
    }
}
```

图 3.1: 用户程序

3.2 细节

编号	动作	事件	完 成 度
(1)	调用系统调用, 将用户栈的寄存器 压入内核栈	进程由用户态进入内核态	√
(2)	压入刚进入系统调用时候的寄存 器的值保存	进程压栈	√

(3)	Ds,ss,es 都指向内核段, fs 指向原数据段	修改段寄存器	√
(4)	通过 find_empty_process 找到空闲的任务号与 pid, 压入堆栈 copy_process 需要的参数	进程调用 sys_fork	√
(5)	调用 get_free_page 从主内存区中获取空闲物理页面。复制原页面的任务数据结构到新内存页中	进程获得新的内存页	√
(6)	Task_struct 更改, 得到新任务在 GDT 表中的 LDT 段描述符的选择符值。	修改新进程的任务数据结构, 新进程进入不可中断等待状态	√
(7)	根据 64MB* 任务号得到新进程在线性空间中的基地址。由此设置局部描述符表中的基地址。	设置新进程的位置	√
(8)	得到 from_dir 与 to_dir, from_page_table, 为每一个要复制的页表 to_page_table 用 get_free_page 分配内存页	新进程复制页目录项和页表项	√
(9)	根据 TSS 与 LDT 在 GDT 表中的位置以及自身的基地址设置描述符的值并放入 GDT 表	在 GDT 表中设置新的 TSS 与 LDT	√
(10)	State=TASK_RUNNING, 弹出内核堆栈的内容: addl \$20,%esp	新进程建立完成	√
(11)	压入系统调用返回值	返回原进程进行进程创建的后续工作	√
(12)	获取当前进程的状态和时间片, 判断是进行 reschedule 还是进行 ret_from_sys_call	system_call 后续处理	√
(13)	将 ret_from_sys_call 压入堆栈, 长跳转到 schedule 执行	重新调度	√

(14)	将当前任务指针放入 eax 通过 eax 得到当前进程的相关状态，判断进程是否为 task0.	执行 ret_from_sys_call	√
(15)	判断任务是否是用户任务	执行 ret_from_sys_call	√
(16)	得到信号位图和阻塞码，得到要发出的信号。信号值入栈，执行 do_signal	执行 ret_from_sys_call	√
(17)	弹出内核栈软件压入和硬件压入的内容。	从系统调用返回调用程序	√
(18)	根据 signr 得到信号结构，判断该信号是否需要处理。	对某个进程的信号进行处理	部分完成
(19)	将 eip 指向信号处理句柄	对某个进程的信号进行处理	部分完成
(20)	将原调用程序的用户堆栈向下拓展，并通过 verify_area 进行检查。若检查成功，则向用户堆栈放入数据。	对某个进程的信号进行处理	部分完成
(21)	根据页边界对齐开始地址，对每个页面执行 write_verify 操作	Verify_area 内存验证	部分完成
(22)	取出对应页目录项，判断页是否存在	写页面验证	部分完成
(23)	得到页表地址和页表中的偏移，从而得到页表项指针，判断标志位，若页面不可写且存在则执行写时复制	写页面验证	部分完成
(24)	查询所有任务的 alarm 值，对过期信号发送 SIGALARM 信号。	进程调度	√
(25)	循环比较所有任务的运行时间 counter，选择 counter 最大的任务。若 counter 为 0 则更新所有任务的 counter 再进行计算。	进程调度	√
(26)	switch_to 调度查询到的任务执行	进程调度	√
(27)	遍历任务数组判断是否有处于可中断状态的具有非阻塞信号的任务	进程调度	新增

(28)	通过 <code>system_call</code> 进入系统调用，再调用 <code>_sys_execve</code> 。将指向 <code>eip</code> 的指针入栈新进程调用 <code>execve</code> 调用 <code>_do_execve</code> 。		√
(29)	检测原进程的 <code>CS</code> 值，设置参数环境空间指针，初始化参数空间，根据要执行的程序的名称得到 <code>inode</code> 节点。	Execve 从命令行得到要执行的程序的名称和参数	√
(30)	检测文件是否为常规文件	Execve 检测文件的类型	√
(31)	判断 <code>inode</code> 节点中是否设置了 <code>S_ISUID</code> 与 <code>S_ISGID</code>	Execve 判断是否改变当前权限	√
(32)	根据 <code>inode</code> 节点得到文件的 <code>uid</code> 和 <code>gid</code> ，与当前进程的 <code>uid</code> 和 <code>gid</code> 比较。	Execve 检测执行文件的权限	√
(33)	调用 <code>copy_strings</code> 将参数字符串和环境字符串从高到低存放。若参数环境空间中没有内存页面，则要申请内存页面	Execve 设置参数和环境空间	√
(34)	释放代码段和数据段原来占用的页表和内存	Execve 为执行程序修改进程状态	√
(35)	修改 <code>ldt</code> 表中的描述符，将参数和环境空间页面存放在数据段末端	Execve 为执行程序修改进程状态	√
(36)	在栈空间中创建环境和参数变量指针表。修改进程各个字段和新执行文件有关的信息	Execve 为执行程序修改进程状态	√
(37)	修改内核栈上的返回地址为新执行程序的入口，替换栈指针。返回 <code>_sys_execve</code> ，丢弃内核栈中的内容，执行系统调用的返回	Execve 执行程序	√
(38)	进程调用 <code>sys_alarm</code> 设置报警时间。调用 <code>sigsuspend()</code> 挂起进程，直到有信号到达	进程通过 <code>sleep()</code> 进入睡眠	✗

(39)	进程调度时检查所有的 alarm, 当 alarm 到达时, 向对应任务发送 SIGALARM, 使得 sleep() 的进程被唤醒。	进程通过 sleep() 进入睡眠	✗
(40)	调用 _page_fault, 修改段寄存器的值, 取出引起页异常的线性地址, 根据标志位判断引起异常的原因。	发生页错误	✓
(41)	申请物理内存, 计算缺页所在的文件数据块号, 用 bread_page 将逻辑块读入物理页面中。将物理页面映射到指定的线性地址处。	缺页处理	✓
(42)	调用 put_page, 根据线性地址计算它的页表地址。若页表项无效, 则申请空闲页面。在页表中设置相关页表项内容。	将物理页面映射 到线性地址	✓
(43)	弹出之前压入堆栈的参数, 退出中断。	缺页处理结束	✓
(44)	page_fault 调用 do_wp_page 进行写保护异常处理。在地址不位于代码空间时调用 un_wp_page 进行处理。	写保护异常	新增
(45)	判断线性地址所在页面的属性, 若为共享页面则申请一页新的内存, 使线性地址对应页表项指向新页面。	写保护异常	新增
(46)	复制原页面到新页面	写保护异常	新增
(47)	搜索文件结构指针数组, 找到空项作为打开文件的句柄	打开文件	✓
(48)	为打开文件在文件表中寻找一个空闲结构项	打开文件	✓
(49)	得到文件的 inode, 根据 inode 设置文件表指针	打开文件	✓
(50)	调用 sys_write 系统调用, 完成 system_call 中对堆栈的一系列操作	写入文件	✓
(51)	取出文件 inode 节点, 调用设备写入函数	写入文件	✓

(52)	Printf 调用 write,write 调用 rw_char 最终调用 con_write, 根据输入字符类型的不同, 控制终端上的光标的变	写入文件	√
(53)	释放当前进程代码段和数据段占有的内存页	进程退出	√
(54)	设置其子进程的状态, 进程 1 发送信号	进程退出	√
(55)	改变当前进程的状态并通知父进程。执行调度。	进程退出	√
(56)	扫描任务数组, 并用 free_page 释放任务相关页面	进程释放	√
(57)	根据请求想的命令对硬盘发送读写命令 , 调用 hd_out 发送信息。	do_hd_request 执行 硬盘读写请求操作	√
(58)	检查参数有效性	硬盘控制器发送操作命令	√
(58)	设置硬盘中断应当调用的 c 函数指针 do_hd。	硬盘控制器发送操作命令	√
(59)	向控制器端口发送 7 字节的参数命令块	硬盘控制器发送操作命令	√
(60)	进入内核态堆栈, 压入寄存器内容, 修 改 ss,es,fs 等段寄存器的内容。	硬盘中断处理程序	√
(61)	将 do_hd 指针放入 edx 寄存器, 根据 edx 的内容调用函数	硬盘中断处理程序	√
(62)	判断写命令操作是否出错, 将欲写扇区数减 1。若扇区数据写完, 调用 end_request() 处理结束事宜。	写扇区中断调用函数	√
(63)	sys_brk 对给定参数 end_data_seg 进行判断, 在数值合理情况下设置 task_struct 中的 brk 字段	数据段末尾设置	新增
(64)	通过 sys_ioctl 进行输入输出之前的 相关设置。	输入输出控制	新增
(65)	通过 sys_signal 设置给定信号的句柄	信号设置	新增

3.3 界面与操作方法

可视化界面支持 alt+f4 关闭，由于该界面的主要目的是绘制一个连贯的动画，所以并没有太强的交互性。程序会展示多个绘图界面，支持用户在主界面内通过断点列表内的滑块，查看之前已经出现的断点。

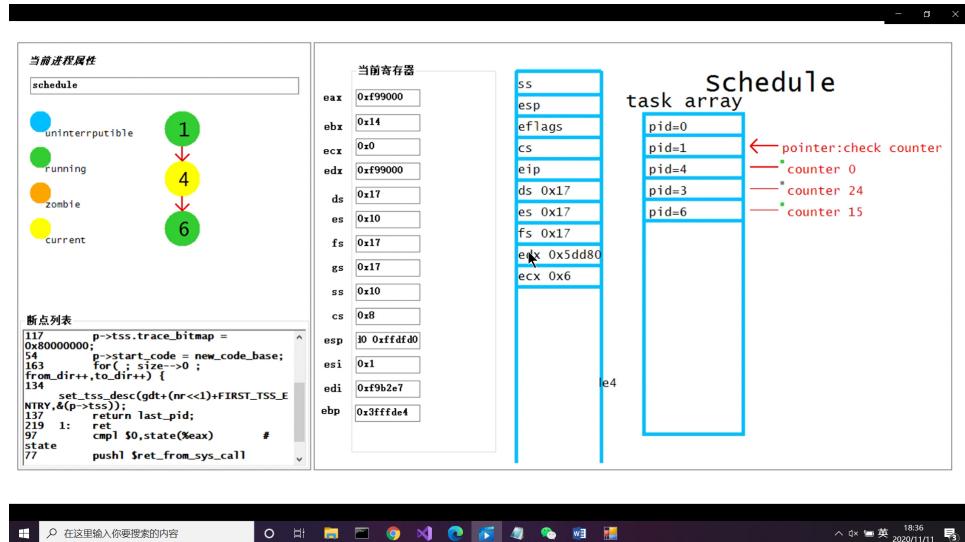


图 3.2： 用户界面

当出现进程线性空间界面时，可以按下 draw 按钮，绘制如下进程内存相关的信息。

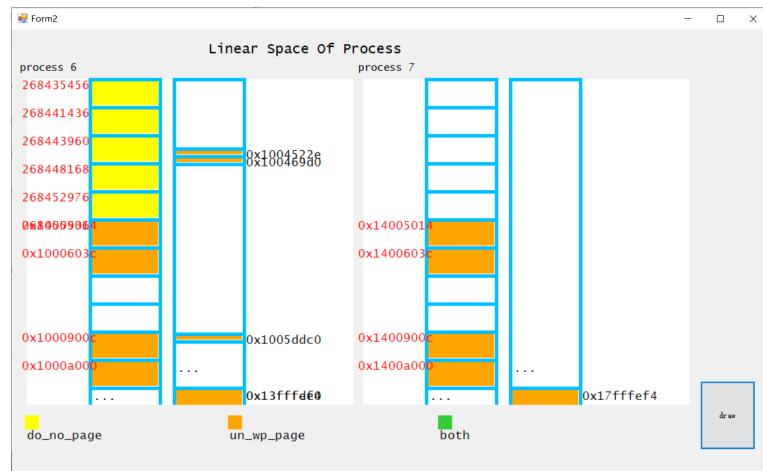


图 3.3：内存界面

(此处描述你的可视化界面支持什么操作，比如空格暂停、alt+f4 关闭等等，如果有鼠标操作，可附加截图说明)

4

系统运行过程实例

4.1 进程创建

4.1.1 get_free_page

要寻找一个新的物理页面存放 copy 的 task_struct.get_free_page 在 mem_map 中找到一个为 0 的物理页面，将对应内存映像的比特置为 1。通过 ecx 计算出页面实际地址，用 edx 指向这个物理页面。设置 ecx 为 1024，将 edi 指向页面末端，从 edi 开始清零内存。最后将页面起始地址存入 eax.

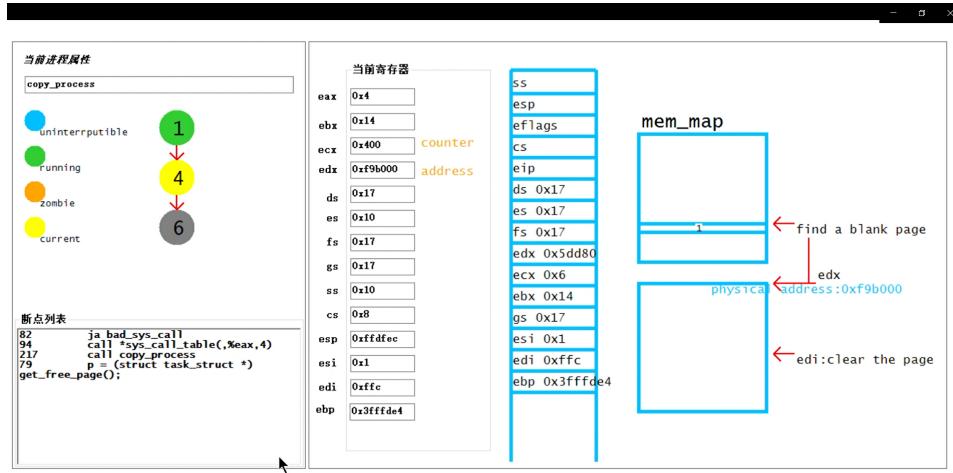


图 4.1: 找空闲物理页面

4.1.2 修改 task_struct

修改复制的 `task_struct` 中的内容，此处仅展示下 `esp0` 指向的位置，即为当前物理页面的最后。正好为 `0xf9000+4096` 的结果。

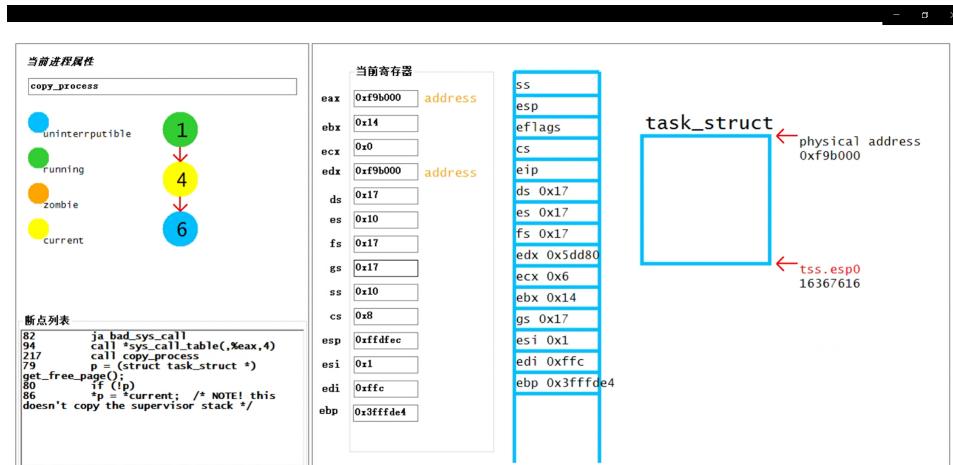


图 4.2: 修改 task_struct

4.1.3 得到 GDT 表中存放当前 LDT 的位置，更改线性基地址

改变 task_struct 中 tss.ldt 的指向，更改原来的线性基地址为当前进程的线性基地址，修改 ldt 表中的内容。

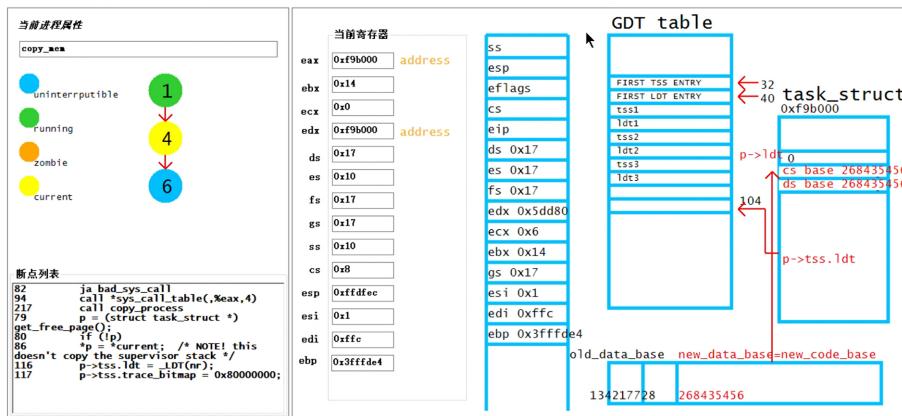


图 4.3：展示 GDT 和 LDT 表

4.1.4 copy_page_tables

左侧模拟页目录，根据线性地址可以得到左侧的 from 与 to,128 与 256 为在页目录中的索引。copy 页表中的内容，让新页表中对应的内存页面只读，若页表所指的物理页面地址在 1MB 以上，则设置源页表项只读。Copy 时候，只复制有效的源目录（即指定的页表存在）项指向的页表。

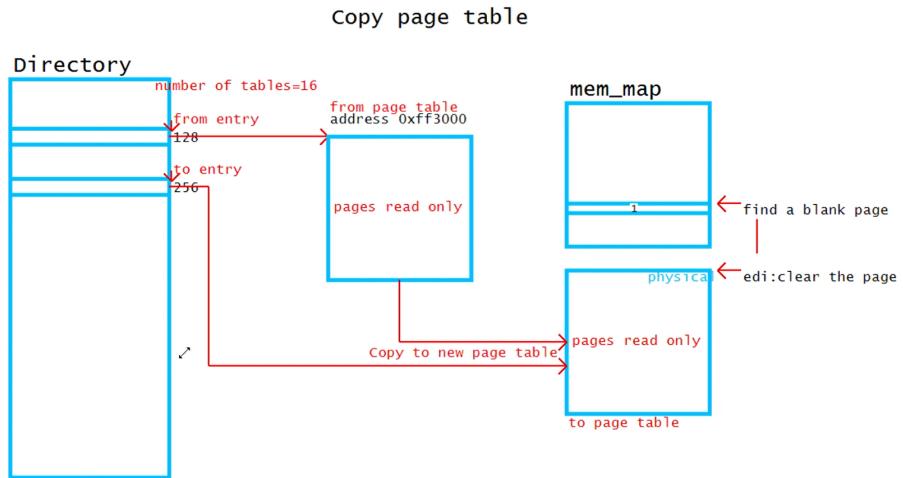


图 4.4: 复制进程页表

4.1.5 修改 GDT 表

设置 GDT 表中新建进程的 tss 段描述符与 ldt 段描述符。当前的 LDT 表和 TSS 段都在之前申请的 task_struct 中。且 LDT table 和 TSS section 之间间隔 24byte，又由于 LDT 项为 8byte，则 LDT 表和 TSS 段之间正好间隔 3 个 LDT 表项。

Change GDT Table

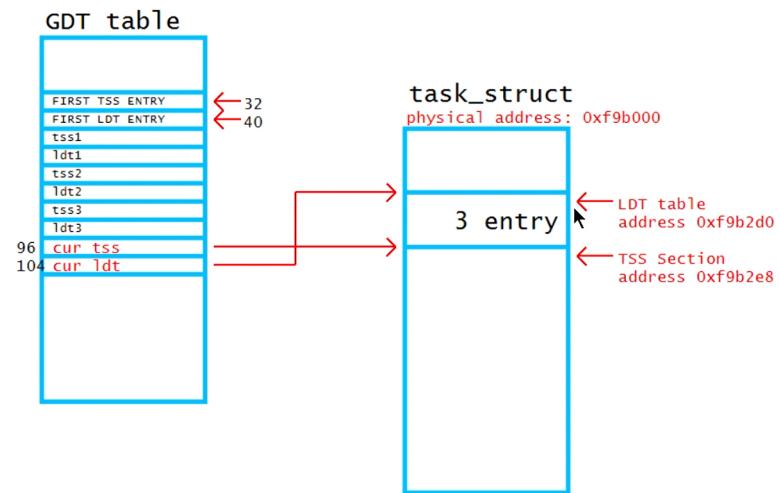


图 4.5: 设置 GDT 表

4.2 进程调度

4.2.1 查找是否有过期的 alarm

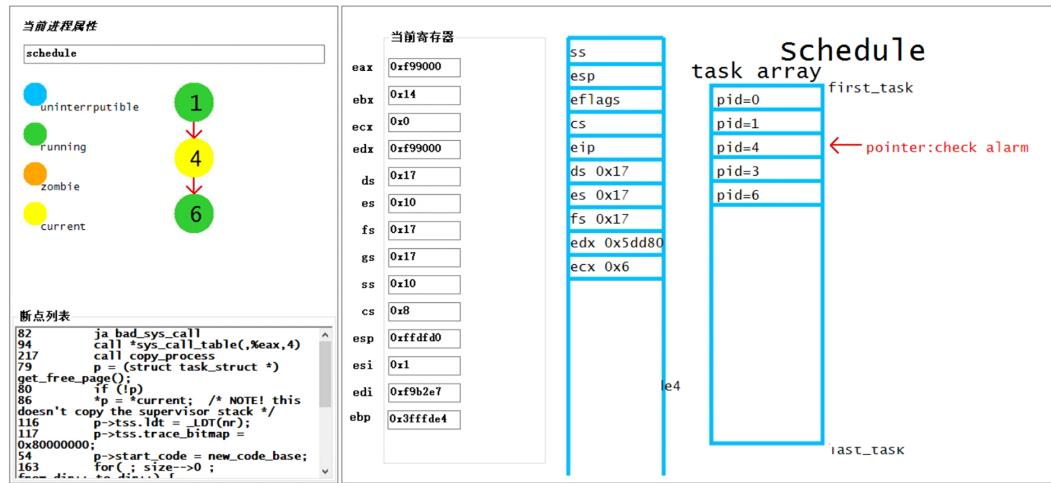


图 4.6: 遍历 alarm

4.2.2 找到 counter 值最大的就绪状态任务

循环遍历任务，找到 counter 值最大的就绪状态任务并将循环运行出的结果表现在右侧。小方块的颜色代表了当前进程的状态，绿色为运行状态，灰色为其他状态

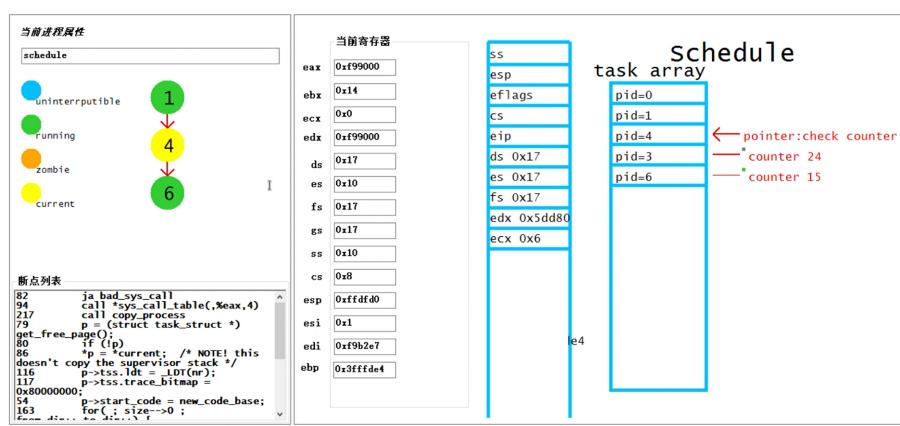


图 4.7: 遍历 counter

4.2.3 更新 counter

在找不到 counter 值不为 0 的就绪状态任务时，根据 priority 更新所有任务的 counter。

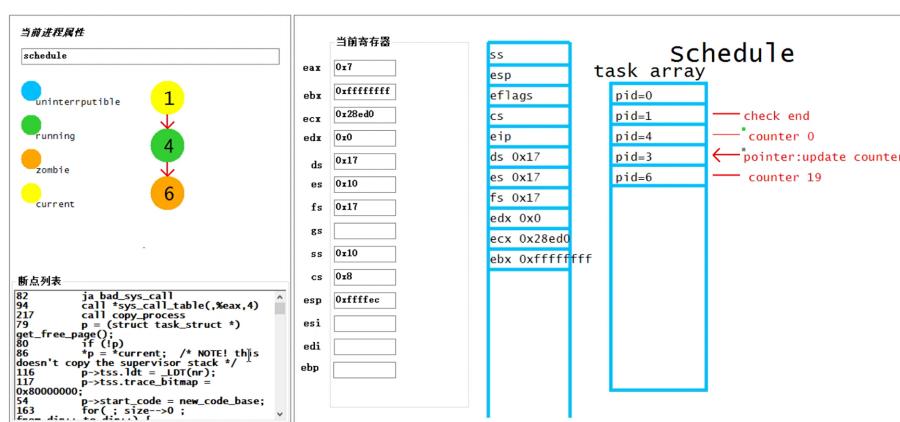


图 4.8: 遍历 counter

4.3 进程 execve

4.3.1 通过名字得到 inode 节点

采用硬盘中断，取出执行文件的 i 节点

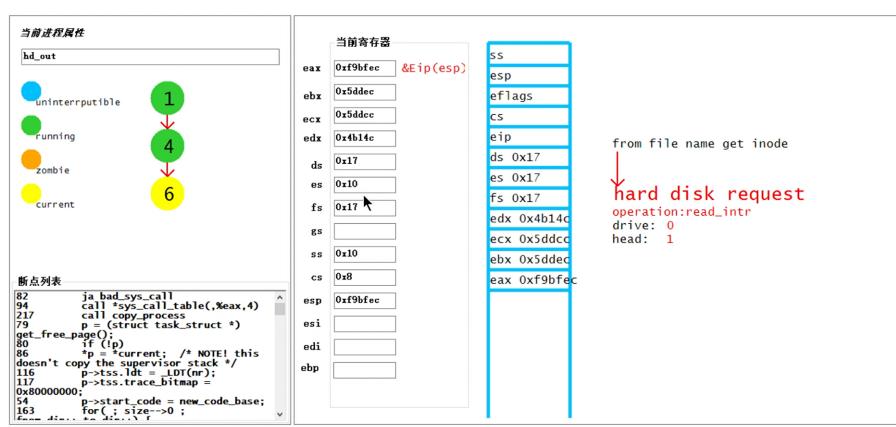


图 4.9: 遍历 counter

4.3.2 权限判断

从 task_struct 中得到 euid 和 egid，判断是否改变 euid 与 egid。根据 euid 与 egid 的关系右移 imode，右移后的 imode 低三位作为权限。

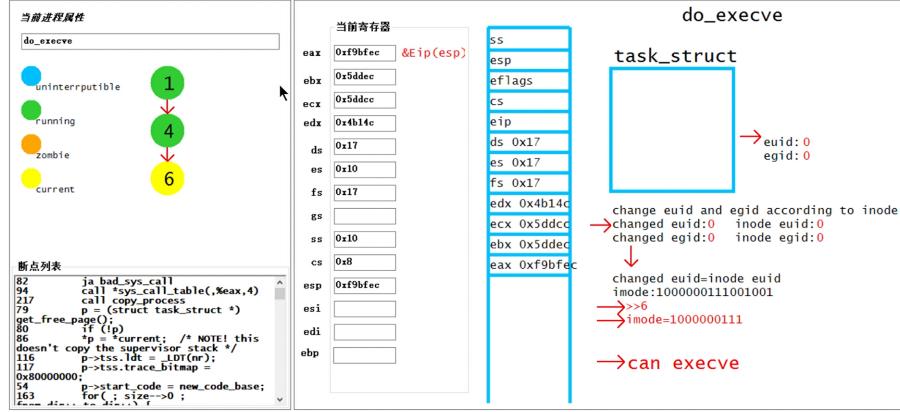


图 4.10：权限判断

4.3.3 复制字符串到参数环境空间

参数空间位于进程逻辑地址的后 128kB，参数空间的填写从后往前进行。第一个长方形表示的是整个当前进程的线性地址空间，将参数空间部分放大得到图中所示的 parameter space。其中 parameter pointer 最开始指向参数空间末端。Tmp 为当前需要复制的字符串指针，逆向逐字符将 tmp 指向的字符串复制到参数和环境空间末端。若 p 指针指向的线性地址处没有分配页面，则为其申请一页物理内存。将 tmp 的内容复制到 offset 处。此时复制字符串是复制到新分配的物理页面。

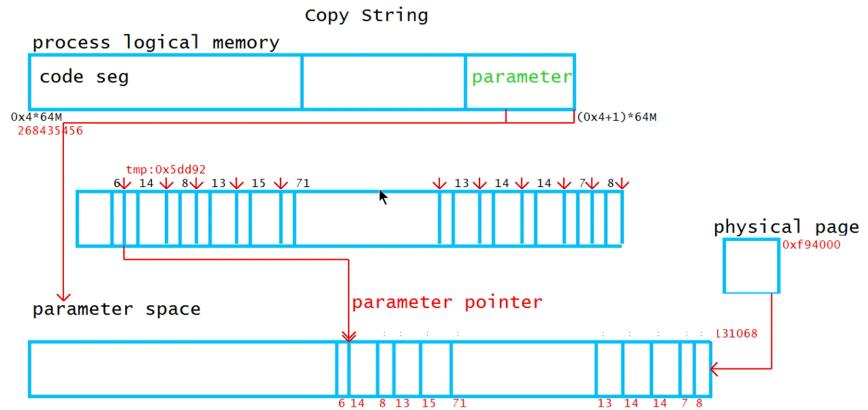


图 4.11: `copy_string`

4.3.4 修改代码段限长并映射物理页面

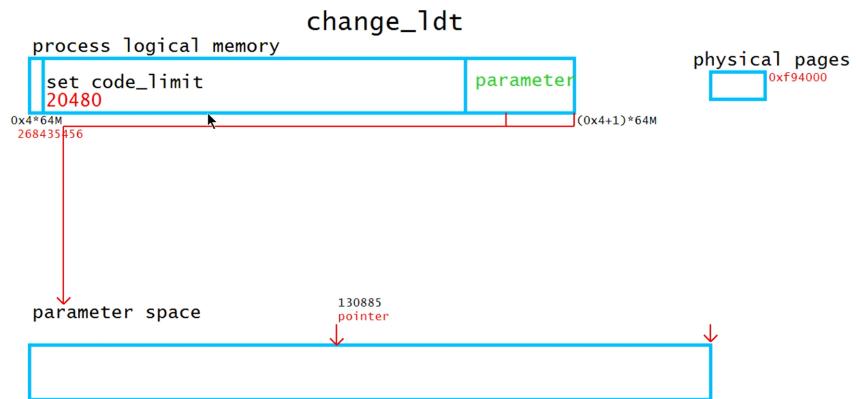


图 4.12: 修改段限长

将参数和环境空间中已经存放数据的页面放到数据段末端。从进程线性地址空间末端一页一页映射。映射方法为修改线性地址所指向的页表项的内容，使之指向物理页面。

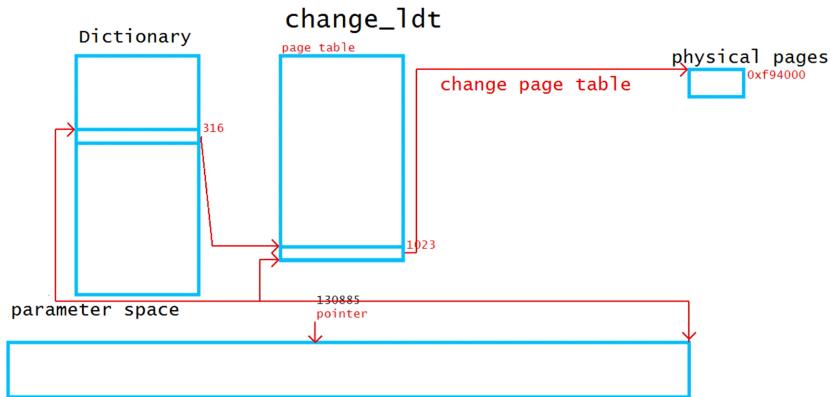


图 4.13：映射物理页面到参数地址空间

4.3.5 在参数环境空间中设置指针表

根据 `copy_string` 时的数据表示来划分参数和环境空间。参数环境空间设置完成后，修改 `eip[0]` 将原来的返回地址改为执行程序入口，`eip[3]` 指向新的 `esp` 的位置。

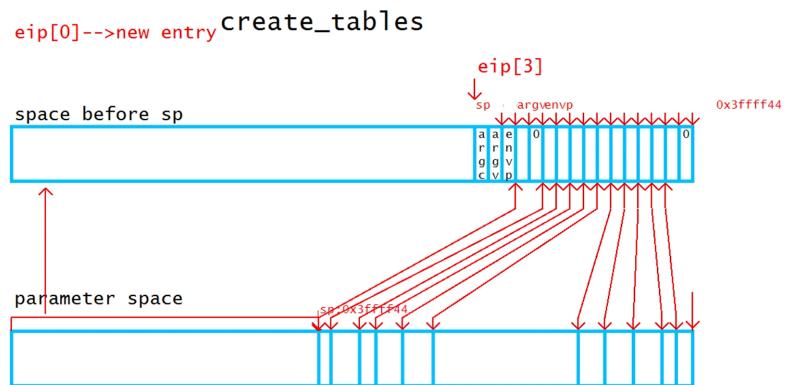


图 4.14：创建指针表

4.4 进程退出

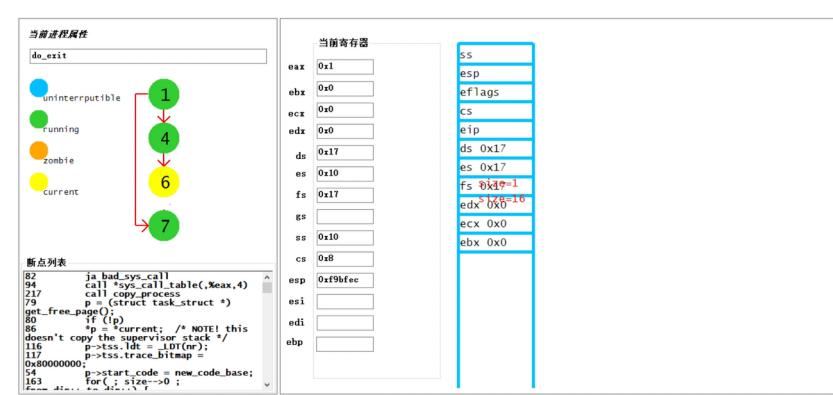


图 4.15：子进程挂到进程 1 下

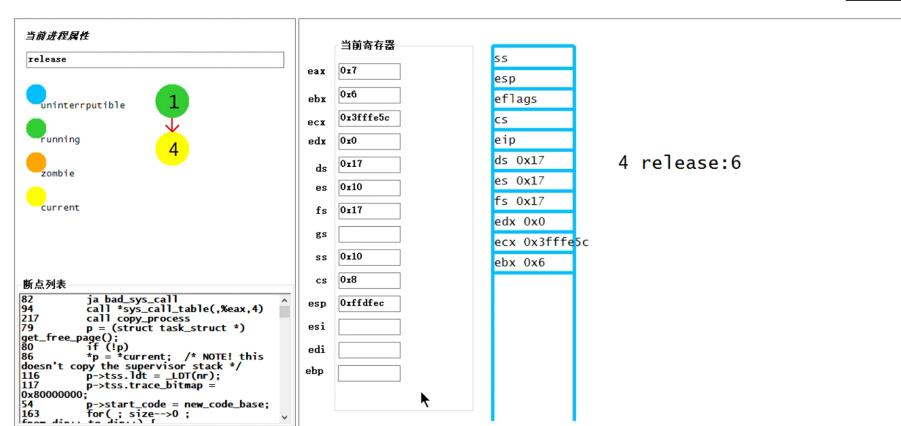


图 4.16：进程被父进程释放

4.4.1 写保护异常

根据线性地址找到物理页面

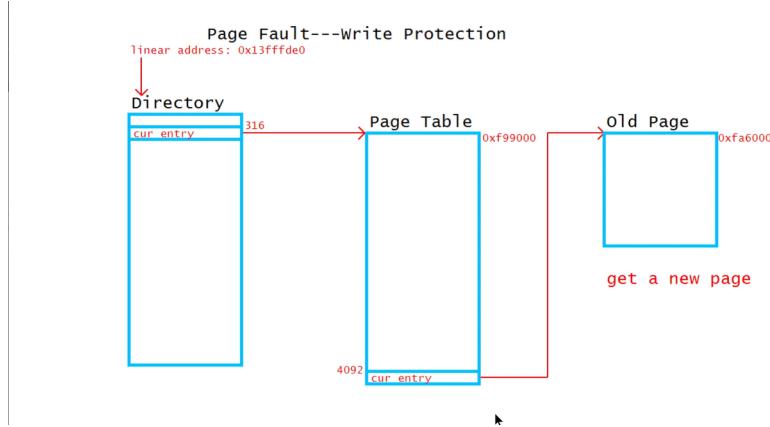


图 4.17： 展示原物理页面

创建一个新的物理页面，让当前线性地址下页表项指向新页面。并设置该页表项中可读写标志。将原页面的内容复制到新页面上。

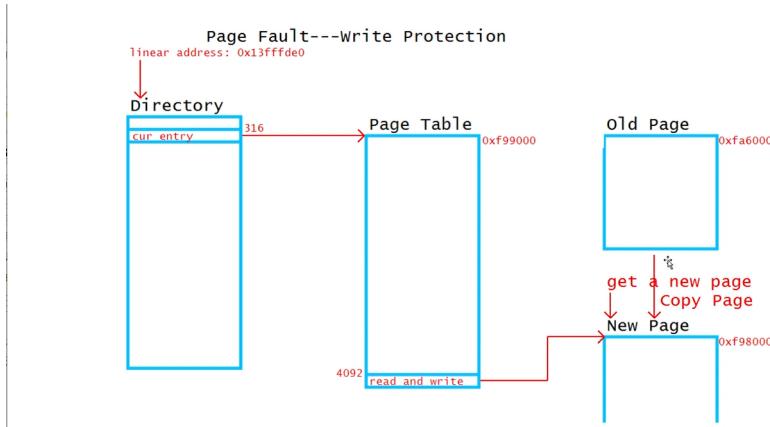


图 4.18： 复制到新的页面

4.4.2 缺页异常

左侧 linear address 列展示发生缺页异常的线性地址（会列出当前进程所有发生缺页异常的线性地址）。通过 get_free_page 得到一个新的物理页面，等待存入从硬盘读入数据。根据偏移地址，计算得到文件系统中的首个块号。

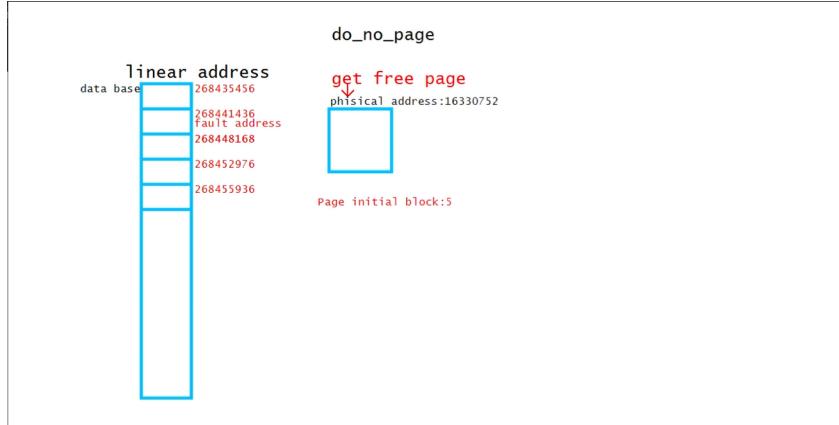


图 4.19： 创建新页面

根据首个块号通过 `bmap` 得到对应的磁盘中的四个块号。右侧展示了当前读入的磁盘块号。通过 `do_hd_request` 和 `hd_out` 从磁盘读入数据。箭头表示读取该块时的设备号与磁头号。

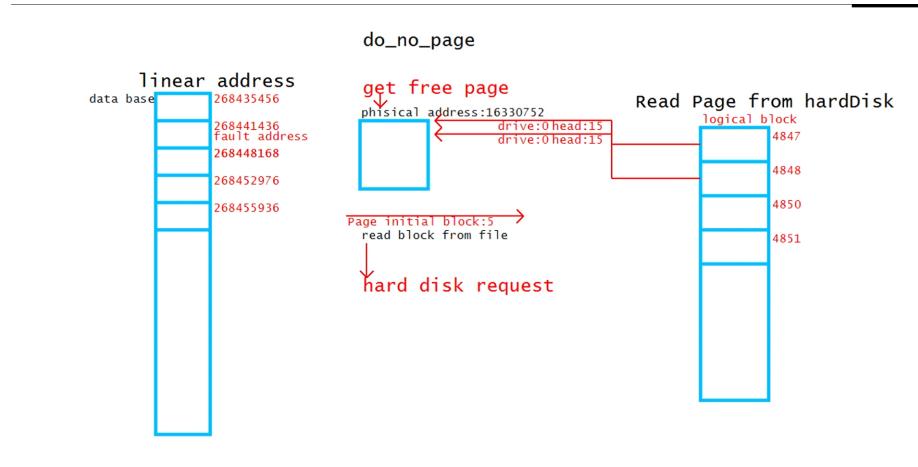


图 4.20

四个数据块读入物理页面后，将线性地址对应的页表项指向该物理页面。

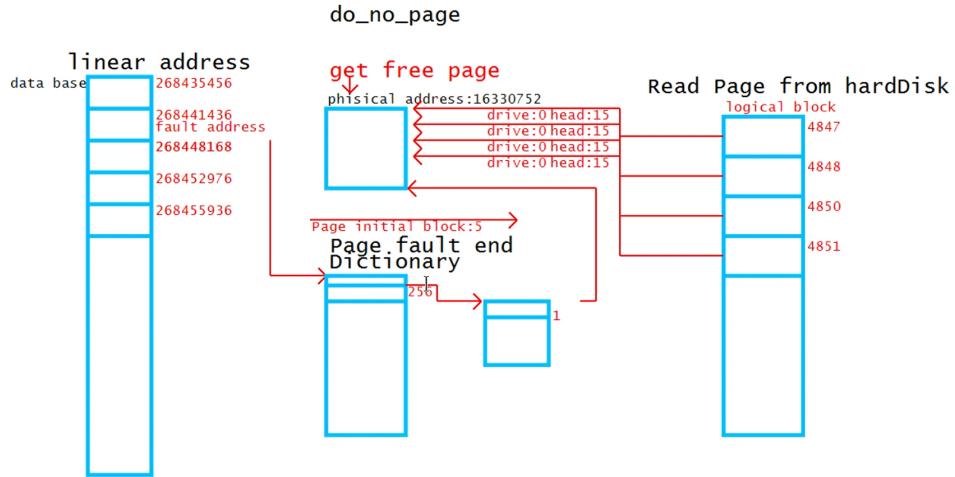


图 4.21

4.5 系统调用

进入系统调用时，向栈中先压入 `ss, esp, eflags, cs, eip` 等值，进入 `system_call` 之后，压入其他段寄存器和通用寄存器的值。

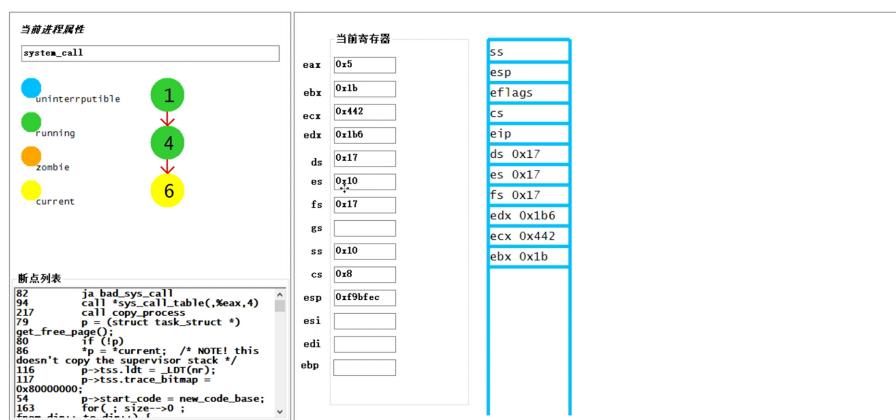


图 4.22：系统调用开始

系统调用结束后查看当前进程的状态和时间片剩余情况，判断是继续进行 `ret_from_sys_call` 还是 `reschedule`。

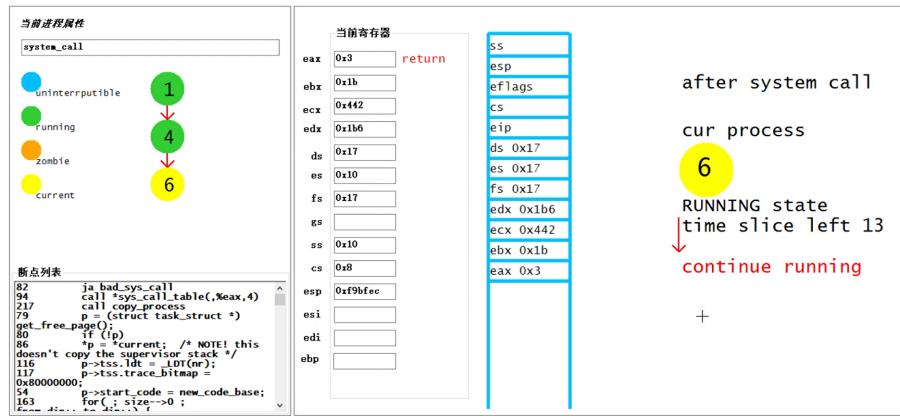


图 4.23：系统调用结束

进行 `ret_from_sys_call` 判断当前任务是否为任务 0，以及是否是用户任务。判断完成后取信号位图入 ebx，阻塞位图入 ecx，对 ebx 和 ecx 进行一系列操作，最后 ecx 中存储为 1 的位的偏移值。

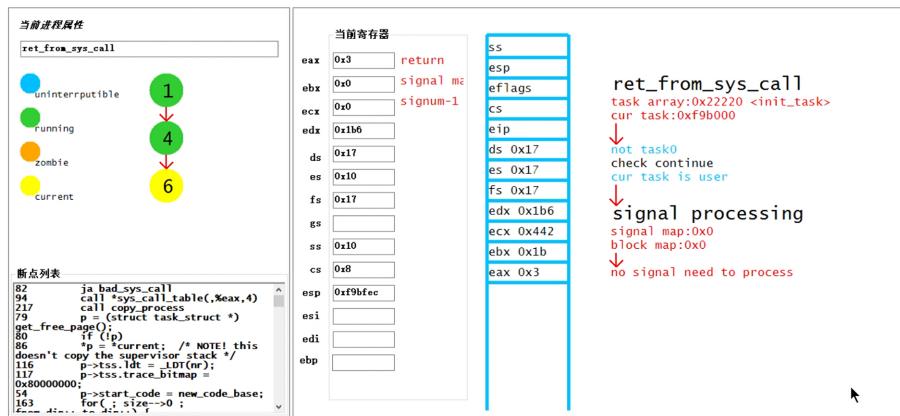


图 4.24：`ret_from_sys_call`

4.6 信号设置

在栈右侧区域显示通过 `sys_signal` 系统调用设置句柄的信号的名称。

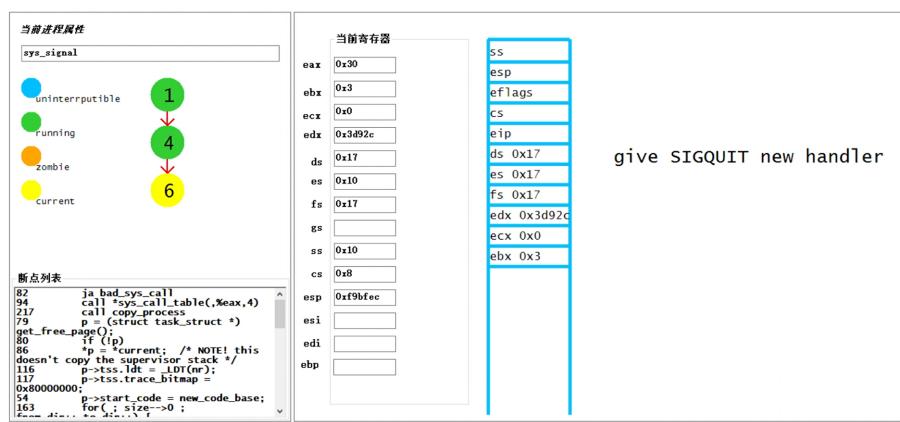


图 4.25：信号设置

4.7 输入输出控制

展示输入输出控制时候的判断过程。

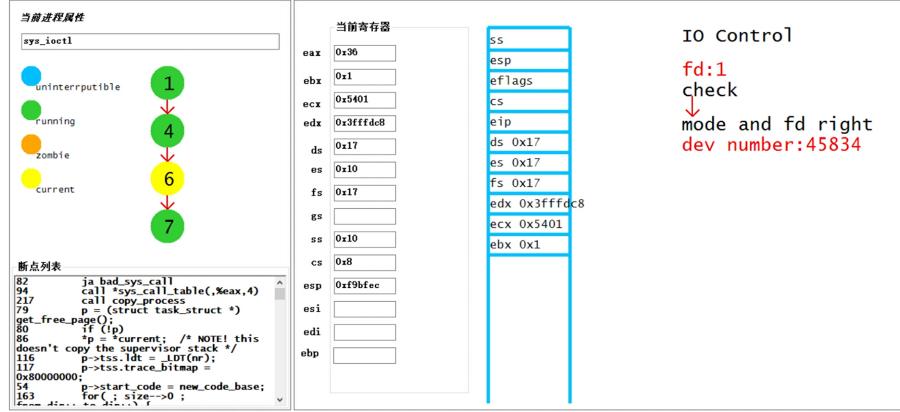


图 4.26： 输入输出控制

4.8 写操作

展示当前写操作调用的函数（根据调用的函数了解执行的具体操作）

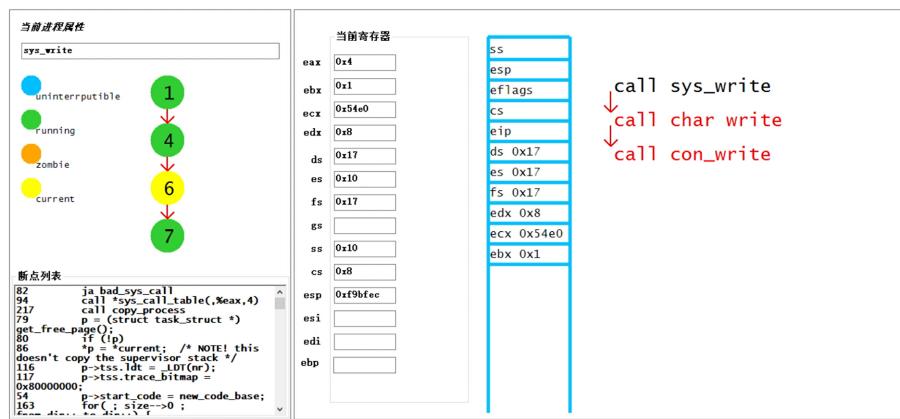


图 4.27： 写终端

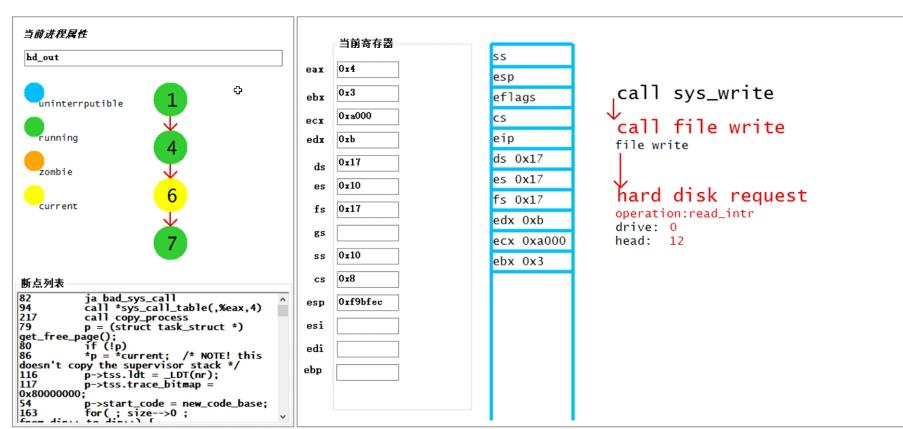


图 4.28：写文件

4.9 打开文件

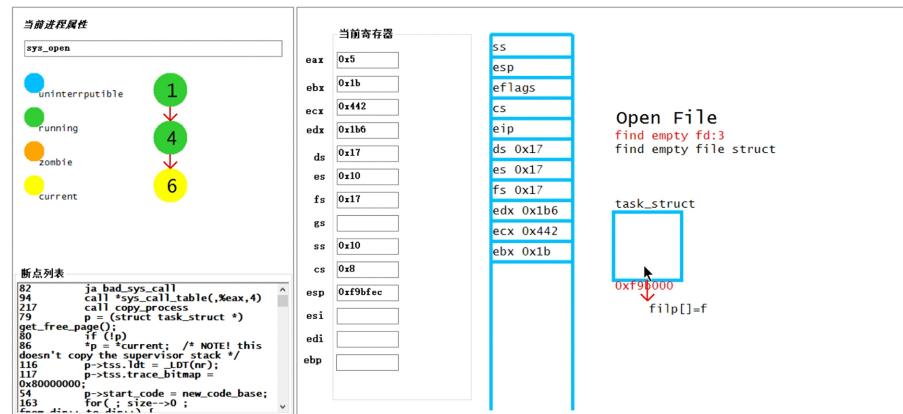


图 4.29：打开文件

4.10 数据段末尾设置

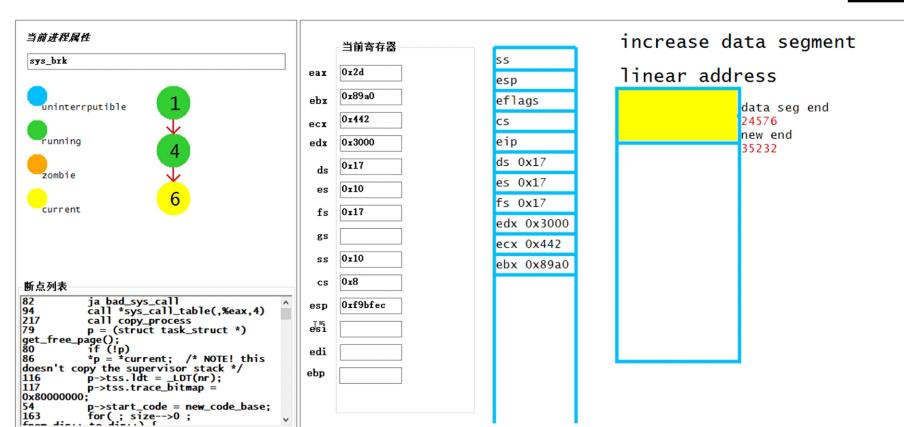


图 4.30：数据段末尾设置

5

Linux0.11 系统源代码分析

5.1 系统调用

系统调用是用户程序和操作系统内核交互的接口，可以通过观察程序进行了哪些系统调用，了解操作系统为程序做了哪些事情。

系统调用的核心为 `system_call.s` 程序，每个系统调用发生时都会先进入 `_system_call` 过程，将段寄存器和一些通用寄存器的值入栈，然后通过 `_sys_call_table` 间接调用指定的某个功能的函数。不同的处理函数可能进一步调用不同的函数来完成系统调用的功能。而每个系统调用又会返回 `_system_call` 过程进行结束或者 `reschedule` 处理。可以观察每次进入 `_system_call` 的功能号，而判断操作系统接下来要进行的操作。通过 `call _sys_call_table` 调用返回判断一个系统调用的结束，而后了解系统调用结束时，操作系统需要做的事情。

本实验展示的过程中涉及 `sys_fork`, `sys_signal`, `sys_execve`, `sys_open`, `sys_ioctl`, `sys_brk`, `sys_write`, `sys_exit`, `sys_close`, `sys_waitpid` 系统调用。

5.2 内存管理

进程控制和内存管理紧密相关，进程创建的过程中会通过内存管理得到存放 `task_struct` 的页面，会由于写保护异常而需要内存管理处理；进程运行过程中会产生缺页异常；进程 `execve` 时，需要内存管理来复制页面，释放页表等。

内存管理的核心代码位于 `memory.c` 中，涉及对物理地址空间和线性地址空间的操作。`page.s` 调用 `memory.c` 中的代码处理缺页异常和写保护异常。`get_free_page` 通过 `mem_map` 找到一个值为 0 的项，从而找到对应的物理页面，清零这个物理页面，将物理页面起始地址作为返回值。`copy_page_tables` 完成对页表的复制，输入 `from` 基地址和 `to` 基地址，根据线性地址的结构得到对应的页目录项，和页目录项指向的页表；为 `to` 的页表项创建分配物理页面存储页表，复制 `from` 页目录项指向的页表到新页表中，并设置内存页面的只读性质。

5.2.1 写保护异常

`do_wp_page` 通过调用 `un_wp_page` 处理写保护异常。`un_wp_page` 判断产生异常的页面是否被共享，为共享的页面申请一个空的物理页面，使页表指向新的物理页面，并调用 `copy_page` 将原页面复制到新的物理页面上。

5.2.2 缺页异常

发生缺页异常时，`page_fault` 会调用 `do_no_page` 处理，根据缺页的线性地址得到在线性地址空间中相对于进程基址的偏移，从而判断缺页是否在当前进程范围内。若在范围内则判断是否为共享页面，若不是则申请一页新的物理页面。从文件中读入四个数据块到当前物理页面中，然后调用 `put_page` 将新的物理页面映射到缺页线性地址处。`put_page` 根据线性地址计算处所在的页目录项，得到页表地址和对应的页表项，修改页表项的内容。

5.3 进程控制

进程控制主要包含进程创建，进程 execve，进程 exit 三个部分。这三个操作各自对应一个系统调用，且这三个过程都和内存密切相关。每个进程根据其任务号在线性地址空间中对应一连续的内存空间，进程控制时常需要对这个空间进行操作，进程运行时的代码和数据也都来源于内存。在本实验中选取的展示过程中包含了这三个部分，希望能够采用可视化的手段展示出进程控制和内存管理之间的关系。

5.3.1 进程创建

进程创建的核心代码为 fork.c 中的函数，fork 时需要先分配一个任务号和 pid，再复制父进程的 task_struct 到一个新的物理页面，并根据子进程的状况对复制的 tsak_struct 进行一定的修改。利用 copy_page_tables 复制父进程的页表到子进程处，修改 GDT 表，加入新的 TSS 和 LDT 段描述符。最后将子进程的状态设置为 TASK_RUNNING。

5.3.2 进程 execve

进程 execve 的核心代码为 exec.c 中的 do_execve。首先判断是否有访问文件的权限，然后读出执行文件头部的数据，根据数据进行判断，从而选择应当执行的操作。当执行文件不是脚本文件时，do_execve 需要对文件格式，文件大小，是否有重定位部分，是否从页边界开始进行判断。若经判断没有错误，则继续进行对参数环境空间的操作。

参数环境空间位于当前进程对应的线性地址空间的后 128kb 内，对参数环境空间的操作涉及内存管理。调用 copy_strings 将指定个数的参数字串复制到参数环境空间中。对某个字符串复制的方法为：根据参数环境空间指针 p，得到 p 所在的页在线性空间中的位置，判断该页是否存在，若不存在则为该线性地址空间中的页面分配一个空的物理页面。将字符串中的字符逐个逆向复制到物理页面上 p 所指向的位置，并在复制的过程

中不断更新 `p` 的值。循环整个过程，逆向复制各个参数到指定的偏移地址处。

对 `task_struct` 进行一系列设置操作后，通过 `free_page_tables` 释放原来的页和页表指向的物理页面。调用 `change_ldt` 修改代码段限长，将之前 `copy_strings` 被复制了信息的物理页面通过 `put_page` 映射到对应的线性地址处，即真正将参数环境空间放在了数据段末端。此时 `p` 也变为相对于数据段起始处的指针。

调用 `create_tables` 在线性地址空间中为环境变量和参数字符串创建指针。根据 `p` 指针和参数个数，环境变量个数，得到存放指针的空间。之后将 `envp, argv, argc` 压入后续空间中。最后将各个参数环境变量指针放入之前得到的指针空间中。

其后完成对 `task_struct` 中的值进行修改等操作后，将 `create_table` 中 `envp, argv, argc` 压入后的起始处设置为 `esp`。将代码指针替换为新的执行程序入口。

5.3.3 进程 exit

进程 `exit` 的核心代码为 `exit.c` 中的 `do_exit` 函数。该函数主要做的事情就是释放当前晋城市 `uo` 占用的一系列资源，包括内存相关资源

首先调用 `free_page_tables` 释放进程页与页表。将当前进程的子进程挂在 `init` 进程下，并在子进程僵死时，向进程 1 发送那个 `SIGCHLD` 信号。完成对文件系统资源的释放，根据在会话中的低位释放资源。设置当前进程僵死并设置退出码。通过 `tell_father` 告知父进程当前进程将停止。最后调用 `schedule()`。进程最终被父进程 `release` 之后才是真正被完全释放掉。

5.3.4 进程调度

进程调度的原因多种多样，最常见原因为：时间片用尽，进程 `exit` 时的 `schedule()`，进程 `release` 时的 `schedule()`。进程调度的核心代码为 `sched.c` 中的 `schedule` 函数，`schedule` 函数的功能主要是

选择下一个将要转到的状态。

`schedule` 通过遍历任务数组，检测是否有 `alarm` 过时的进程，若有则向任务发送 `SIGALARM` 信号；检测是否在任务处于可中断状态时有非阻塞信号，若有则将进程状态变为就绪。之后循环检测每个就绪状态任务的 `counter` 值，找到最大且不为 0 的 `counter` 值，转到该值对应的进程。若没有非 0 状态，则根据优先级和当前 `counter` 重设每个任务的 `counter` 值，之后再济宁比较，找到应当转到的状态。

5.4 其他操作

硬盘操作

硬盘是计算机的外存，操作系统的内存资源有限，故而常常需要从硬盘读取数据。从硬盘读取数据时会调用 `hd.c` 中的 `do_hd_request` 处理硬盘当前请求。`do_hd_request` 用于检查请求项的合法性，根据当前的设备号和扇区号信息计算出在对应硬盘上的位置。根据当前请求的类型，通过 `hd_out` 向硬盘控制器发送读或者写命令。`hd_out` 用于向硬盘控制器发送命令块，其中会设置 `do_hd` 函数指针，该函数会在中断程序中被调用，该函数指针的值不是固定的，是 `hd_out` 的输入参数。可以根据 `do_hd` 指针的类型，了解要对硬盘进行什么操作。

信号设置

`sys_signal` 位于 `signal.c` 中，用于为指定的信号安装信号句柄。先根据 `signum` 判断信号值是否有效，若有效则根据参数组件新的 `sigcation` 结构。取出原信号处理句柄作为返回值，将新的句柄设置到 `task_struct` 的 `sigcation` 数组中的对应位置。

写文件

写文件时调用 `read_write.c` 中的 `sys_write` 系统调用。该系统调用先判断文件句柄的有效性，若有效则根据 `i` 节点，调用不同文件类型对应的读操作函数。

数据段末尾设置

系统调用处理函数 `sys_brk` 位于 `sys.c` 中用于对给定的参数 `end_data_seg` 进行判断，在数值合理的情况下更新 `task_struct` 中的 `brk` 字段为 `end_data_seg`。

输入输出控制

`sys_ioctl` 位于 `ioctl.c` 程序中，用于进行输入输出控制。在进程进行输入输出之前需要调用该函数先进行相关设置。该函数对文件描述符有效性和文件类型进行判断，若都可行则从文件的 `i` 节点得到设备号，查表找到该设备对应到函数指针并调用。

5.5 与现在的 linux 的比较

`linux0.11` 的内存中为了简洁，同一进程的代码段和数据段基地址限制为相同的，而在现在的 `linux` 中并没有这个限制。`linux0.11` 中查找页表是直接从内存中的目录项中读取页表所在的物理地址，从而查找页表，而现在的 `linux` 中往往先查找 TLB 表，若没有命中，再查内存中的页表。`linux0.11` 的内存管理是比较低效的，例如 `get_free_page` 的时候需要遍历整个内存位图。现在的 `linux` 为内存管理开发了比较高效和灵活的方式，出现了 slab 分配器，slob，slub 等。`linux0.11` 的进程调度是通过 `schedule()` 中比较每个任务的 counter，而现在的 `linux` 系统有更多样的调度算法，包括短作业优先，多级反馈队列调度算法等。