

操作系统课程设计报告

张芮睿 201800301072

2020 年 12 月 18 日

目录

I 要求回顾	4
1 实验内容与任务	5
1.1 任务	5
1.2 工作内容	6
2 实验过程及要求	7
2.1 实验前一学期	7
2.2 实验学期	7
2.2.1 第 1-4 周, 16 课时	7
2.2.2 第 5-6 周, 8 课时	7
2.2.3 第 7 周, 4 课时	7
2.2.4 第 8-11 周, 16 课时	8
2.2.5 第 12-14 周, 12 课时	8
2.2.6 15-16 周, 8 课时	8
3 相关知识及背景	9
4 教学目的	10
5 实验原理及方案	11
5.1 实验的总体思路	11
5.2 实验可能采用的关键技术	12
6 实验报告要求	13
7 考核要求与方法	14

8 参考文献	15
II 实验报告	16
9 Linux0.11 系统源代码分析	17
9.1 用户进程与内存管理	17
9.1.1 进程创建及加载	17
9.1.2 进程退出	20
9.2 文件操作	20
9.2.1 打开文件	20
9.2.2 读取文件	20
9.2.3 写文件	21
9.2.4 关闭文件	21
10 系统运行过程的形式化描述方法	25
10.1 描述	25
10.2 细节	26
10.3 界面与操作方法	32
11 可视化方法的描述	34
11.1 使用的工具	34
11.2 实现功能和原理	34
11.3 程序架构及详细介绍	35
11.4 最终效果以及相关截图	35
11.5 编译、配置、运行的具体方法	35
12 内核运行数据输出方法	37
12.1 使用的工具	37
12.2 提取目标	37
12.3 提取数据的具体方法	39
13 系统运行过程实例	41
13.1 例 1：创建进程	41
13.1.1 shell 进程调用 sys_fork() 创建 str1 进程	41
13.1.2 调用 copy_mem() 取得父进程段限长和段基址	42

13.2 例 2: str1 进程的加载运行	42
13.2.1 调用 do_execve 等函数	42
13.2.2 str1 运行	43
13.2.3 进程退出	44
13.3 filee 运行以及相关文件操作	44
13.3.1 filee 源代码	44
13.3.2 相关文件操作以及截图	46

I

要求回顾

1

实验内容与任务

1.1 任务

该实验以 Linux0.11 为例帮助学生探索操作系统的结构、方法和运行过程，理解计算机软件和硬件协同工作的机制。学生需要完成 4 项任务：

- (1) 分析 Linux0.11 系统源代码，了解操作系统的结构和方法。
- (2) 通过调试、输出运行过程中关键状态数据等方式，观察、探究 Linux 系统的运行过程。
- (3) 建立合适的数据结构，描述 Linux0.11 系统运行过程中的关键状态和操作，记录系统中的这些关键运行数据，形成系统运行日志。
- (4) 用图形表示计算机系统中的各种软、硬件对象，如内存、CPU、驱动程序、键盘、中断事件等等。根据已经产生的系统运行日志，以动画的动态演示系统的运行过程。

1.2 工作内容

将整个系统的运行过程可视化需要付出巨大的工作量，一个学期内难以完成。在全面分析源代码的基础上，学生可以根据自身的能力和兴趣在不同层次、规模、难度上完成本项实验：

- (1) 学生可以探究系统某个模块的某个过程，如文件系统的读操作、键盘的输入、CPU 的调度等。
- (2) 学生可以选择组成大小不等的团队参与实验。
- (3) 在可视化处理上，学生也可以做适当的简化。

2

实验过程及要求

2.1 实验前一学期

在操作系统原理课程中，教师介绍 Linux0.11 源码结构及相关资料，并公布下一学期操作系统课程设计的任务。学生具备了自己分析源代码的基础。

2.2 实验学期

2.2.1 第 1-4 周，16 课时

将 Linux0.11 源代码分成基础模块和选读模块，学生必须分析基础模块，然后从选读模块中选择感兴趣的模块重点分析。

2.2.2 第 5-6 周，8 课时

学生自由组合成团队，提出设计方案，每个团队说明感兴趣的系统运行过程。

2.2.3 第 7 周，4 课时

讨论、评估设计方案。

2.2.4 第 8-11 周, 16 课时

从感兴趣的系统运行过程中提取系统运行的状态数据，并生成系统运行日志。

2.2.5 第 12-14 周, 12 课时

根据日志实现运行过程的可视化。

2.2.6 15-16 周, 8 课时

学生演示运行结果，评定成绩。

3

相关知识及背景

实验以 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%)
- C 语言 (80%)
- 数据结构 (30%) 等课程中的基本知识和方法

通过实验学生的如下方面的能力将得到训练和发展

- (1) 代码分析能力
- (2) 编程能力
- (3) 计算机系统能力
- (4) 沟通协作能力
- (5) 表达能力

4

教学目的

- (1) 将操作系统原理与具体实现相结合，加深对理论知识的理解
- (2) 掌握计算机系统的硬软件整体架构，培养学生的全局观和系统能力
- (3) 理解运行中的系统，锻炼学生解决实际问题的能力

5

实验原理及方案

该实验是一个综合性很强的课程设计，包含了计算机系统中硬件和软件设计的多项基本原理：

- CPU 结构
- CPU 管理
- 内存管理
- 外设控制
- 文件系统

5.1 实验的总体思路

- (1) 在代码级别上理解系统的运行过程；
- (2) 获取系统运行过程的数据；
- (3) 演示系统运行的过程。该实验的结果是开放的，解决具体问题的思路依赖于学生各自的设计目标。

5.2 实验可能采用的关键技术

内核编程技术，用以从内核中输出运行时数据，需要修改内核。其难点在于新增加的代码应该保持内核运行时原有的样子，而且内核编程与普通编程相比，受到更多的限制。

可视化技术，即如何利用图形图像表达系统的运行过程。

6

实验报告要求

- (1) Linux0.11 系统源代码分析报告，说明学生本人分析代码的体会及重点分析的代码描述
- (2) 系统运行过程的形式化描述方法
- (3) 内核运行数据输出方法
- (4) 可视化方法的描述
- (5) 系统运行过程实例（图文描述）

7

考核要求与方法

节点	课序	标准	考核方法
设计方案	28	创新性；完整性；可行性。30%	课堂报告，教师打分
系统运行 过程描述	44	系统状态的形式化描述； 状态数据的获取技术。30%	课堂报告，教师打分
演示	60	表现力；流畅性；界面。30%	课堂报告，教师打分
实验报告	64	规范性；文字表达。10%	教师打分

8

参考文献

- Linux 内核完全解析 0.11
 - 赵炯
- Linux 内核设计的艺术
 - 新设计团队

II

实验报告

9

Linux0.11 系统源代码分析

9.1 用户进程与内存管理

9.1.1 进程创建及加载

此处的创建进程是指 shell 进程创建用户进程的过程。以下为大致调用函数的过程，具体的函数调用以及涉及到的文件见下几张图

创建进程由 shell 进程调用 sys_fork() 创建，过程中设置了新的 task_struct 管理结构（其中设置了 tss 和 ldt），并复制了 shell 进程的页表给其共享。

准备工作由 shell 进程调用 do_execv() 函数完成，主要完成读取可执行文件，释放之前共享的页表，重新设置段限长，最终设置 eip 和 esp 跳转到用户程序执行。

在 do_execve() 中释放了之前的页表，页目录项也随之清零，而 LDT 中代码段基址没有变，所以 do_execve() 返回后，CPU 还是从 LDT 中找到代码段基址然后找到对应的页目录项，此时页目录项为 0，从而引发了缺页中断。

在 copy_strings() 中申请的了新的页面放到了 page[p/4096] 中，然后将参数放入新页面，change_ldt() 中又找到 page[p/4096]，也就是找到这个新页面，然后将其映射到进程空间末尾。假设数据段基址为 0x40000000，参数的地址被放到了 page[31] 中，然后就会将 0x40000000 + 0x40000000 - 4K = 0x7FFF000 这个线性地址与 page[31] 映射起

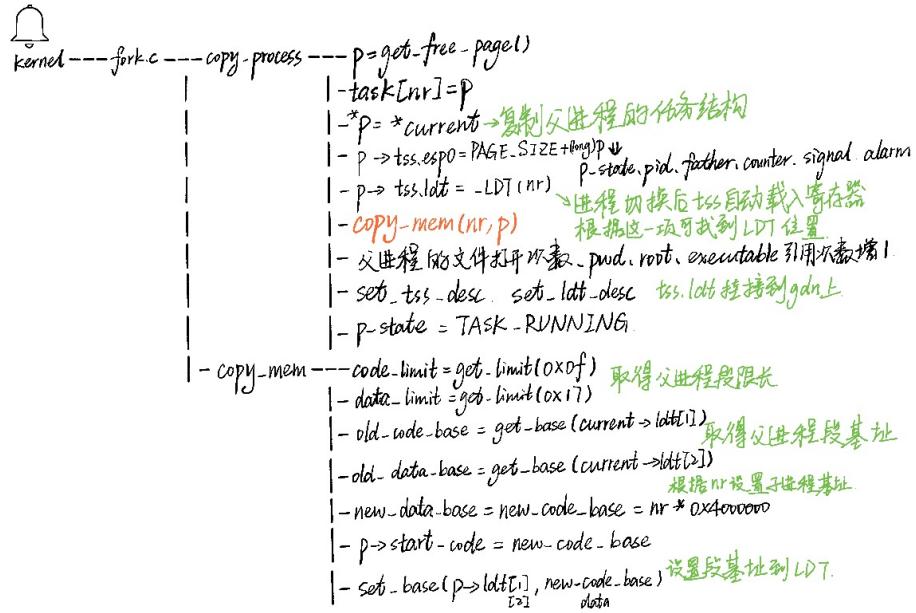


图 9.1：创建用户进程涉及函数及其所在文件

来。也就是说第 31 个页目录项，页表偏移 1023 的页表项里存放的是 $page[31]$ 。

这里 eip 和 esp 的值为段内偏移量，猜测是与段基址结合生成线性地址，如：代码段基址是 $0x4000000$ ，假设 $eip = 1024 = 0x400$ ，则程序的入口地址为 $0x4000400$ ，对应第 16 个页目录项，页表偏移 0，页内偏移 1024。数据段基址等于代码段基址，也是 $0x4000000$ ， $esp = p = 64M - x - 4$ ，假设 $x = 2K$ ， $esp = 0x3FFF7FC$ ，栈指针的线性地址为 $0x7FFF7FC$ ，对应第 31 个页目录项，页表偏移 1023，页内偏移 2044。第 31 个页目录项，页表偏移 1023 的页表项已经完成映射，对应 $page[31] + 2044$ 的物理地址，所以此时 esp 不会发生缺页中断

```

fs --- exec.c --- do_execve --- p=PAGE_SIZE*MAX_ARG_PAGES-4
| - page_size = PAGE_SIZE
| - file = fopen(argv[1], "r");
| - execve(argv[1], &argv[2], &envp[2]);
| - if (fread(&buf, 1, 1024, file) < 0)
|   把程序放入内存的动态空间中并检测代码,数据,堆栈是否超过4GB
|   返回错误
| - current->executables = node
| - current->current = node
| - current->current->current = current
| - free_page_table(&base, current->current->current->current->current->current->current->current);
| - free_page_table(&base, current->current->current->current->current->current->current->current);
| - free_page_table(&base, current->current->current->current->current->current->current->current);
| - pr = change_llt(&a.text, page, MAX_ARG_PAGES * PAGE_SIZE)
| - 设置 llt_struct
| - llt->start = page
| - llt->end = page + MAX_ARG_PAGES * PAGE_SIZE
| - llt->size = page - page
| - llt->offset = page - page
| - llt->page = page
| - llt->page_size = PAGE_SIZE
| - llt->page_offset = page - page
| - llt->page_end = page + PAGE_SIZE - 1
| - llt->page_start = page - PAGE_SIZE
| - code_limit = text_start + PAGE_SIZE - 1 代码段很长为代码长度+4KB-1
| - code_limit = 0xFFFFF000
| - data_limit = 0x00000000 - 4MB
| - code_base = get_base(current + llt->start)
| - data_base = code_base 基址不变→为0x40000整数倍
| - set_base(...), set_limit...
| - data_base + data_limit
| - for(i=MAX_ARG_PAGES-1; i>0; i--)
| - {
| -   data_base += PAGE_SIZE;
| -   if (page[i])
| -     find_page(page[i], data_base); 找到设置参数的画面
| - }
| - free_page(base, 64MB)

```

图 9.2：进程加载和运行前的准备工作

释放了共享的页面后，页目录项为 0，进入缺页中断。

`do_no_page` 中新申请了一个页，然后将文件节点中一页的数据读入内存中，并完成与线性地址的映射。注意这里的 `address` 是发生缺页中断的线性地址。如 `0x4000400`，中断返回后，由于页表已经映射完毕，就可以进入 `0x4000400` 对应的物理内存执行应用程序。

```

mm --- pages --- page-fault() --- do_no_page()
| - memory.c --- do_no_page(address) --- address &= 0xfffffff000 → 清除页内偏移，方便进行块操作。
| - tmp = address - current->start_code → 获得在线性地址中的偏移量
| - if (!current->executable || tmp > current->end_data)
| -   get_empty_page(address)
| -   return
| - 程能在与其他进程共享
| - page = get_free_page()
| - block = 1 + tmp / BLCK_SIZE 计算块开始的位置
| - bread_page(page, current->executable->i_dev) 读取一页的信息到 page。
| - put_page(page, address) 将新的页映射到线性地址

```



```

| - get_empty_page(address) --- tmp = get_free_page()
| - put_page(tmp, address)

```

图 9.3：用户进程的运行和加载

9.1.2 进程退出

进程调用 `exit()` 函数进行退出，最终会进入 `sys_exit()` 中去执行。此处退出进程的父进程是 `shell` 进程，这里先将此进程设置为僵死态，然后向 `shell` 进程发送信号，最后调度到 `shell` 进程。`shell` 进程收到信号后，释放掉此进程的任务结构，并解除 `task[64]` 的关系，退出完成。

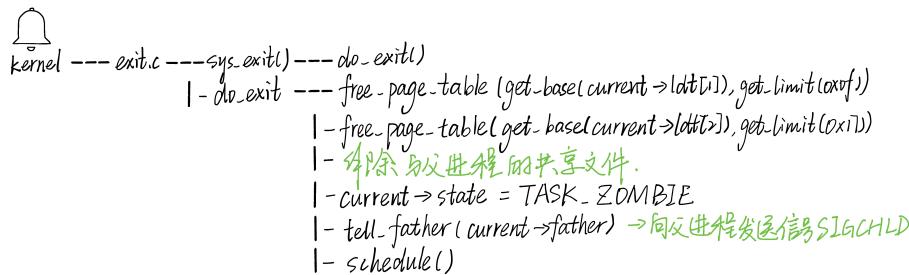


图 9.4: 进程调用 `exit()` 函数进行退出

9.2 文件操作

9.2.1 打开文件

打开文件操作涉及到了文件夹 `fs` 中的 `open.c`, `namei.c` 和 `inode.c`。具体细节如下图：

9.2.2 读取文件

读取文件操作涉及到了文件夹 `fs` 中的 `read_write.c`, `file_dev.c` 和 `inode.c` 文件，由于本实验访问的文件是普通文件 (`txt`)，所以调用的是 `file_read` 函数，具体细节如下图：

9.2.3 写文件

写文件操作是将数据写入缓冲区，其中涉及到了文件夹 `fs` 中的 `open.c`、`file_dev.c` 和 `inode.c` 文件，具体细节如下图：

从缓冲区同步到外设，`sys_sync()` 在 `update` 进程中，一定时间运行一次

9.2.4 关闭文件

关闭文件操作涉及到了文件夹 `fs` 中的 `open.c` 和 `inode.c` 文件，具体细节如下图：



图 9.5： 打开文件涉及到的函数

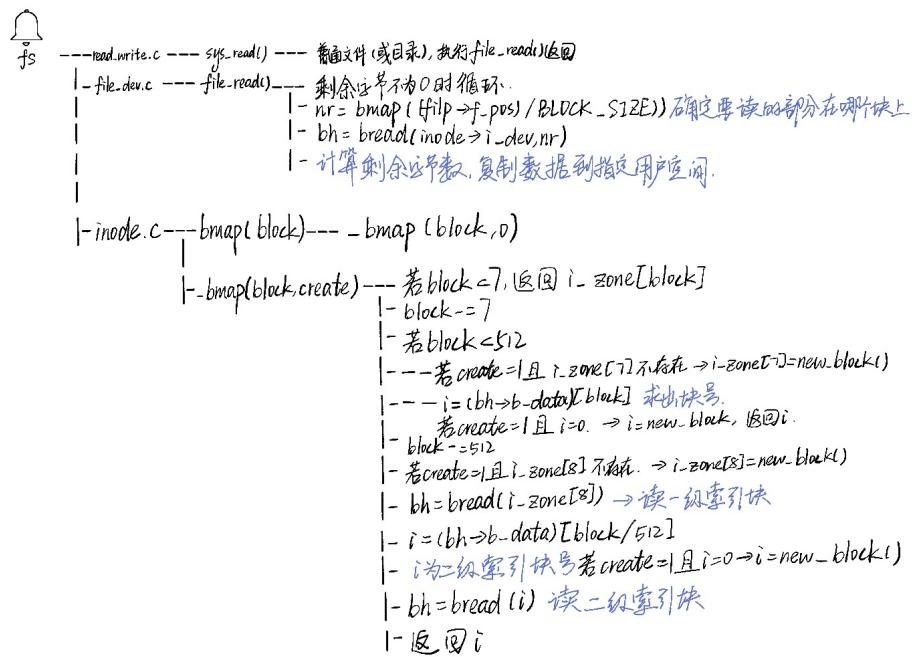


图 9.6: 读取文件涉及到的函数

fs --- open.c --- sys_write --- file_write()
 |- file.dev.c --- file_write() --- B图逻辑写 count 时循环。
 | |- block = create_block(pos/BLOCK_SIZE) 找到需要写入的块号
 | |- bh = bread(inode->i_dev, block)
 | |- 计算 → 搬贝 bh 到缓冲区 → 返回写入字节数

|- inode.c --- create_block(block) --- bmap(block, 1)
 |- bmap(block, create) --- 若 block < 1, 返回 i.zone[block]
 | |- block = 1
 | |- 若 block < 512
 | | --- 若 create=1 且 i.zone[1] 不为空 → i.zone[1]=new_block()
 | | --- i=(bh->b_data)[block] 读块号
 | | --- 若 create=1 且 i.zone[1] 不为空 → i.zone[1]=new_block()
 | | --- bh=bread(i.zone[1]) → 读一级索引块
 | | --- i=(bh->b_data)[block/512]
 | | --- i为二级索引块多若 create=1 且 i=0 → i=new_block()
 | | --- bh=bread(i) 读二级索引块
 | |- 返回 i

|- bimap.c --- new_block() --- sb = get_super(dev)
 | |- 找到以为的逻辑块所在的位置的缓冲区 bh, 由于要新建块, 将缓冲区逻辑块对应的 bit 置 1
 | |- 由位图的信息计算逻辑块号
 | |- bh = getblk(dev, j)
 | |- clear_block(bh->b_data) bh->b_dirt=1
 | |- 返回 j

图 9.7：写文件涉及到的函数

fs --- open.c --- sys_close() --- filp = current->filp[fd]
 | |- current->filp[fd] = NULL 离进程指置空
 | |- filp->f_count file-table 引用计数 -1
 | |- iput(filp->f_inode) inode 引用计数 -1

|- inode.c --- iput(inode) --- 循环开始
 | |- 如果 inode->i_count > 1
 | | --- inode->i_count -1 → 返回
 | |- 如果 inode->i_nlinks = 0
 | | --- truncate(inode)
 | | --- free_inode(inode) → 返回
 | |- 如果 inode->i_dirt = 1
 | | --- write_inode(inode) → 返回
 | |- inode->i_count --
 | |- 返回

图 9.8：关闭文件涉及到的函数

10

系统运行过程的形式化描述方法

10.1 描述

本课程设计以两个预先编制的程序在 Linux0.11 中运行的过程为主线，通过动画展示进程的创建、撤销、文件系统的打开管理以及内存管理调度，可以让观看者对 Linux 的进程和文件系统有一个直观感受。

大致过程如下：首先操作系统启动，展示给用户的是子进程 shell，随后我们通过输入指令 `./str1` 来使 shell 进程执行 str1 程序，在执行 str1 程序之前，shell 进程首先创建子进程 str1，该进程由 shell 进程调用 `sys_fork` 创建，过程中设置了新的 `task_struct` 管理结构并复制了 shell 进程的页表给其共享，后 shell 进程调用 `do_execv` 函数完成读取可执行文件 str1，释放之前共享的页表，最终设置 eip 和 esp 跳转到用户程序执行。释放了共享的页面后，页目录项为 0，进入缺页中断，str1 开始运行和加载，最后 str1 进程调用 `exit` 函数进行退出，最终会进行 `sys_exit`。而第二个程序 filee 也是如此加载运行，不同的是 filee 程序中调用了 `open` 函数打开文件，这使得进程 filee 调用 `sys_open` 获取文件节点并连接到 f 上，并且调用了 `sys_read` 函数对普通文件执行 `file_read` 函数。Filee 程序中同时调用了 `write` 函数对文件进行写入，这使得进程 filee 调用了 `sys_write` 函数对于

文件进行 `file_write` 操作，最后程序调用 `close` 函数关闭文件，这使得进程 `filee` 调用 `sys_close` 函数关闭文件节点，最后程序结束，进程退出。

10.2 细节

编号	动作	事件	完成度
(1)	操作系统启动, <code>init</code> 调用 <code>fork</code> 创建子进程	创建子进程	部分完成
(2)	用户输入指令 <code>./str1</code> 指示 <code>shell</code> 运行可执行文件 <code>str1</code>	程序 <code>shell</code> 执行命令	√
(3)	<code>shell</code> 进程调用 <code>sys_fork()</code> 创建 <code>str1</code> 进程	创建进程	√
(4)	<code>copy_process()</code> 调用 <code>get_free_page()</code> 取空闲页面	为新任务数据结构分配内存	√
(5)	将 <code>get_free_page() [nr]</code> 中	将新任务结构指针放入任务数组中	√
(6)	复制当前进程内容	复制 <code>shell</code> 进程的页表给 <code>str1</code> 进程共享	√
(7)	将此进程 <code>pid</code> 设置为 <code>last_pid</code> 即新进程号	设置 <code>str1</code> 进程任务状态	√
(8)	对此进程 <code>tss</code> 进行设置	设置 <code>str1</code> 任务状态段 <code>TSS</code> 所需的数据	√
(9)	<code>copy_process()</code> 调用 <code>copy_mem()</code> 取得父进程段限长和段基址	设置新任务的代码和数据段基址、限长并复制页表	√
(10)	<code>copy_mem()</code> 根据 <code>nr</code> 设置子进程段基址	设置新任务的代码和数据段基址、限长并复制页表	√
(11)	<code>copy_mem()</code> 设置段基址到 <code>LDT</code>	设置段基址到 <code>LDT</code>	√

(12)	在 GDT 中设置新任务的 TSS 和 LDT 描述符项，数据从 task 结构中取	设置 str1 进程的新数据结构	√
(13)	shell 进程调用 do_execv() 函数	加载 str1 进程	√
(14)	do_execve() 初始化参数和环境串空间的页面指针数组(表)	do_execve() 初始化	√
(15)	do_execve() 取可执行文件的对应 inode 节点号	寻找对应的文件节点号	√
(16)	do_execve() 检查 inode 的 i_mode 和 i_uid 和 i_gid	检查执行文件的文件类型和执行权限	√
(17)	do_execve() 调用 bread() 函数读取指定的数据块保存到 bh 中	读取执行文件的第一块数据到高速缓冲区	√
(18)	do_execve() 操作指向头部份的数据结构的指针 ex, 对执行文件的执行类型进行区分处理	对执行文件的头结构数据进行处理	√
(19)	do_execve() 复制脚本程序文件名、解释程序参数和解释程序名到参数和环境空间中	提供运行参数	√
(20)	do_execve() 取解释程序的 inode 节点，跳转到 restart_interp 处重新处理	重启进程	√
(21)	do_execve() 清复位所有信号处理句柄	信号复位	√
(22)	do_execve() 释放原来程序代码段和数据段所对应的内存页表指定的内存块及页表本身	释放之前共享的页表	√

(23)	do_execve() 调用 change_ldt 重新设置段限长	重新设置段限长	√
(24)	do_execve() 修改当前进程各字段为新执行程序的信息	设置 task_struct	√
(25)	do_execve() 设置 eip 和 esp	设置 eip 和 esp 跳转到用户程序执行	√
(26)	释放共享页面后, str1 进程页目录项为 0, 进入缺页中断	缺页中断	√
(27)	do_no_page() 算出指定线性地址在进程空间中相对于进程基址的偏移长度值 tmp	缺页中断处理	√
(28)	do_no_page() 新申请了一个页, 然后将文件节点中一页的数据读入内存中, 并完成与线性地址的映射	缺页中断处理	√
(29)	运行 str1 程序	程序执行	√
(30)	str1 执行 foo() 函数, 触发缺页中断	缺页中断	√
(31)	do_no_page() 直接调用 get_empty_page(), 申请一页物理内存并映射到指定线性地址处, 返回	缺页中断处理	√
(32)	str1 进程调用 exit() 函数退出, 进入 sys_exit() 执行	程序退出	√
(33)	sys_exit() 调用 do_exit() 函数	程序退出处理开始	√
(34)	do_exit 释放 str1 进程代码段和数据段所占内存页	释放内存页	√

(35)	do_exit 对当前进程的 <code>pwd</code> 、 <code>root</code> 和运行程序的 <code>inode</code> 进行释放，并分别置空	置空操作	√
(36)	do_exit 对当前进程设置为 <code>ZOMBIE</code> 状态并设置退出码	进程设置状态和退出码	√
(37)	do_exit 向 <code>shell</code> 进程发送信号 <code>SIGCHLD</code>	信号传递	√
(38)	<code>shell</code> 进程收到信号后，释放 <code>str1</code> 的任务结构，并解除 <code>task[64]</code> 的关系	退出完成	√
(39)	do_exit 执行 <code>schedule()</code>	重新调度函数运行	√
(40)	用户输出指令 <code>./filee</code>	程序执行命令	√
(41)	程序 <code>filee</code> 被执行	程序执行	√
(42)	程序 <code>filee</code> 中 <code>fopen()</code> 函数打开当前目录文件 <code>a</code>	打开文件	√
(43)	<code>fopen()</code> 函数调用系统接口函数 <code>sys_open</code>	系统调用	√
(44)	<code>sys_open</code> 将用户设置的模式与进程的模式屏蔽码相与	产生许可的文件模式	√
(45)	<code>Sys_open</code> 在 <code>file_table[64]</code> 中找到空闲项 <code>f</code>	寻找空闲项	√
(46)	<code>Current->flip[fd] = f</code> , 进程的对应文件句柄的文件结构指针指向搜索到的文件结构	进程对应文件句柄指向搜索到的文件结构	√
(47)	<code>sys_open</code> 调用 <code>open_namei()</code>	获取文件节点	√
(48)	<code>open_namei()</code> 调用 <code>dir_namei()</code>	获取文件枝梢节点、文件名以及长度	√

(49)	dir_namei() 调用 get_dir() 获取指定路 径名最顶层目录的 inode 节 点	取 inode 节点	√
(50)	get_dir() 将 inode 设 置为根节点, pathname 指向 'root'	get_dir() 操作	√
(51)	get_dir() 判断字符串是否 到末尾, 是则返回 inode	遍历路径名字符串	√
(52)	dir_namei() 获得文件名,计 算其长度	dir_namei() 操作结束	√
(53)	返 open_namei(),open_namei() 调用 find_entry() 从枝梢 节点读取对应的目录项 de	读取对应目录项	√
(54)	find_entry() 读取文件的 第一个数据块 bh, de=bh- >b_data 指向数据块头, 迭代 遍历数据块, 查找名字为 name 长度为 namelen 的目录项,找 到则返回 bh, 否则返回 NULL	读取文件块	√
(55)	open_namei() 获取 de 之 后从目录项中调用 iget() 获 取文件 inode 号	调用 iget() 获取 inode 号	√
(56)	iget 从 inode 表中取出一 空白 inode, 扫描 inode 表, 寻找指定节点的 inode 号	寻找指定节点 inode	√

(57)	<code>iget</code> 如果在 <code>inode</code> 表中找到指定节点 <code>inode</code> , 等待解锁并将该 <code>inode</code> 节点引用计数 +1; 如果没找到, 则查找对应超级块寻找指定 <code>inode</code> , 并在 <code>inode</code> 表中利用前面申请的空闲 <code>inode</code> 节点在 <code>inode</code> 表中建立该节点, 从相应设备上读取 <code>inode</code> 信息, 返回该 <code>inode</code> .	<code>iget</code> 操作	√
(58)	<code>sys_open</code> 设置该文件 <code>inode</code> 节点, 初始化文件结构	连接到 <code>f</code> 上	√
(59)	<code>sys_open</code> 返回文件句柄	打开文件操作结束	√
(60)	程序 <code>filee</code> 读取文件 <code>list.txt</code> , 调用 <code>sys_read()</code>	读取文件开始	√
(61)	<code>sys_read</code> 判断文件类型, 调用 <code>file_read()</code>	执行 <code>file_read()</code>	√
(62)	<code>file_read()</code> 调用 <code>bmap()</code> 确定要读的部分在哪个块上	读取操作 <code>file_read()</code>	√
(63)	<code>file_read()</code> 根据 <code>i</code> 节点和文件表结构信息, 取数据块文件当前读写位置在设备上对应的逻辑块号 <code>nr</code> 。若 <code>nr</code> 不为 0, 则从 <code>i</code> 节点指定的设备上读取该逻辑块, 如果读操作失败则退出循环。若 <code>nr</code> 为 0, 表示指定的数据块不存在, 置缓冲块指针为 <code>NULL</code> 。	读取操作 <code>file_read()</code>	√
(64)	<code>file_read()</code> 计算剩余字节数, 复制数据到指定用户空间	<code>file_read()</code> 读取结束	√

(65)	程序 filee 写入文件 list.txt, 调用 write() 函数	写入文件 a	√
(66)	write() 调用系统接口 sys_write()	写入开始	√
(67)	sys_write 取文件对应的 inode, 调用 file_write() 写入文件	写入文件操作结束	√
(68)	file_write() 判断已写入字节是否小于需要写入字节 count, 如果是则找到需要写入的块号, 计算字节, 拷贝 buf 到缓冲区, 返回写入字节数	file_write() 操作完成	√
(69)	程序 filee 关闭文件 list.txt, 调用 close()	关闭文件 a	√
(70)	close() 调用 sys_close() 函数通过操作文件句柄关闭文件	关闭文件 a	√
(71)	filee 结束, 调用 exit() 退出	程序退出	√
(72)	进程 filee 终止	进程终止	√

1 只完成了读取进程号, 展示了大致过程调用

10.3 界面与操作方法

首先, 整个 UI 界面由一个图像框, 三个文本框, 两个按钮组成。

其中, 左上角是我手绘的展示现在正在进行的大致步骤图像, 右上角的文本框显示的是现在正在发生的事件, 左下角的文本框是事件触发的动作, 右下角的文本框是这一步动作所涉及到的断点的数据, 最右下角的两个按钮是 Next 和 Back, 鼠标点击 Next 按钮可以跳转到下一个动作, Back 则是返回上一个动作。右下角的文本框是可以拖拽的, 然后总体的窗口做了

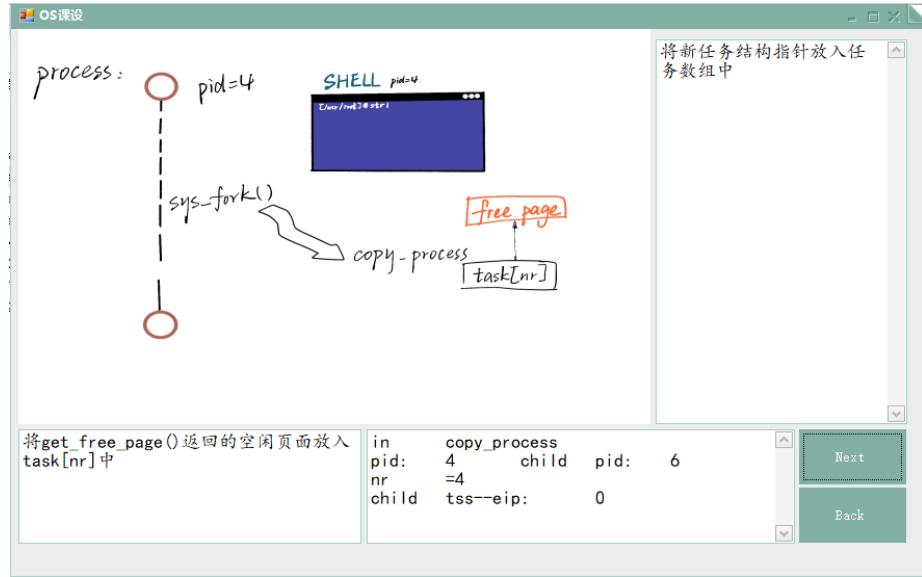


图 10.1: UI 界面一览

全屏功能，可以拖拽窗口调整大小，里面的组件也能够随着窗口的变化而相应变化。

11

可视化方法的描述

11.1 使用的工具

用到的编程语言是 C#, 用到的编译器是 Visual Studio 2019, 使用了库 .NET Framework 4.7.2 以及皮肤库 IrisSkin4¹以及其他正常包含在 Winform 应用程序中的库.

界面中提到的手绘画面使用了概念画板²和 Notability。绘画出来的图片转成 GIF 使用了 ScreenToGif 应用。

11.2 实现功能和原理

使用 VS2019 中的 winform 程序画出大概界面, 其中使用 C# 语言编写, 将自己用概念画板手绘出来的图片通过 ScreenToGif 转成 gif 图片, 用图片插入的方式插入到 winform 程序界面左上角的 Picturebox 中, 右上角和左下角的文本框内的内容是通过 ReadData 这个前置 C# 程序筛选自己预想的每一步的动作和具体事件成 C# 中的字符串, 并且将输出结果复制到 Alpha 这个程序的定义段中, 作为这两个文本框的内容切换。

¹ 参照这一篇 <https://www.cnblogs.com/mq0036/p/6654219.html> 中对于 IrisSkin4 的做法

² 是在 iOS 端的一个软件

右下角的文本框内是数据驱动的断点数据，提取的函数定义在 Alpha 程序中。每按一次 Next 按钮，进度就往下一格，同时展现的画面会变成下一个动作的界面。

11.3 程序架构及详细介绍

整个可视化界面的程序显示的顺序是由一个全局变量 ProcessNum 决定的，即每按一次 Next 按钮，ProcessNum 加一，同时界面刷新，对应这个 ProcessNum 的数据文字和图片显示覆盖上一个 ProcessNum 时显示的数据图片，每按一次 Back 按钮同理。

每一个 ProcessNum 对应的数据除了数据驱动的那部分断点数据没有定死之外，其他的都是之前已经处理好了，直接就可以显示的字符串数组或者是图片组。由于即使是数据驱动的断点数据，在显示的时候也已经提前做好一定的处理，所以在定点显示的时候也是直接按照在原文本里面的顺序进行展示而不是对于内容进行筛选和展示，所以加载的时候需要的计算量较少，运行起来比较顺滑。

界面能够自动放大缩小，其中组件按照比例同时放大缩小，这个功能定义在 Alpha 程序的 AutoSizeForm.cs 文件中，同时对界面还进行了一定的美化，即引进了皮肤库 IrisSkin4。

11.4 最终效果以及相关截图

运行出来效果较好，部分截图如下：（附件中也有相应的运行录屏：展示截屏.mp4）

11.5 编译、配置、运行的具体方法

此程序的编译配置运行等操作都是基于 Windows 10 环境下的，没有尝试过在其他环境下运行。

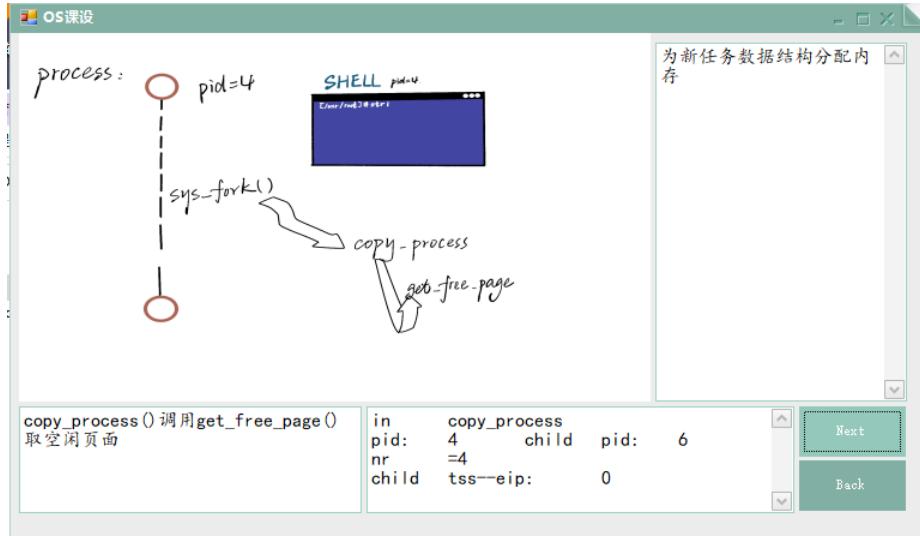


图 11.1: UI 界面

源程序是位于 Alpha 文件夹³下的 Alpha.sln(VS2019 环境下的 C# 项目),如果有 VS2019 即可直接双击打开此项目,就可以直接进入编译运行界面了。在与 Alpha.sln 同等级的文件夹 Alpha 里的 bin/Debug/Alpha.exe 是可以直接运行的程序,同时,皮肤加载文件夹 Skins 和配置的 IrisSkin4.dll 都在这个文件夹⁴下。

数据提取初步处理完成的三个文件 log00.txt,log01.txt,log02.txt 需要放在 Debug 文件夹下。

³是位于和此报告文件同一个文件夹下的 Alpha 文件夹

⁴指 Alpha/Alpha/bin/Debug

12

内核运行数据输出方法

12.1 使用的工具

本课程设计没有使用提供的 Bochs 调试环境。

模拟器: QEMU emulator version 2.8.1(Debian 1:2.8+dfsg-6+deb9u5)

调试器: GNU gdb (Debian 7.12-6) 7.12.0.20161007-git

通过调试器连接到模拟器提供的调试端口进行内核调试，通过调试器输出的信息进行数据提取。

选用 QEMU 而不使用 Bochs 的原因:

1. Bochs 在进行 gdb 调试过程中，触发缺页中断时会产生 Signal 0 信号，使调试器中断执行，妨碍调试过程。
2. QEMU 仍是目前业界一种主流的虚拟化方案，尤其是当其使用 KVM 组件加速后。
3. QEMU 支持的架构体系较 Bochs 更全面。

12.2 提取目标

编号	位置	目的
----	----	----

(1)	Kernel/fork.c:copy_process	显示父子任务 pid, 任务号, tss_eip
(2)	Kernel/fork.c:copy_mem	显示 old_code_base,old_data_base 和 new_data_base
(3)	fs/exec.c:201	Do_execve 取的 inode 的对应 inode 码
(4)	fs/exec.c:224	显示 do_execve 取可执行文件的执行权限, uid 和 gid, 执行文件的第一块数据的任务号
(5)	fs/exec.c:273	do_execve 复制的脚本程序文件名
(6)	fs/exec.c:276	do_execve 复制的解释程序的参数 (可能有, 因为是 optional)
(7)	fs/exec.c:279	do_execve 复制的解释程序文件名
(8)	fs/exec.c:347	显示进程堆结尾字段 brk, 堆栈开始字段, 新的 eip 和 esp
(9)	mm/page.s:page_fault	显示这里引发了缺页异常
(10)	mm/memory.c:do_no_page	错误码和页面地址
(11)	mm/memory.c:384	指定线性地址在进程空间中相对于基址的偏移长度值以及缺页所在的数据块项
(12)	kernel/exit.c:do_exit	当前退出的错误码和相应退出的进程的 pid
(13)	fs/open.c:sys_open	许可的文件模式, 文件的 inode 号, 返回的文件句柄
(14)	fs/namei.c:open_namei	打开的文件路径名, 对应的 inode 号
(15)	fs/read_write.c:sys_write	文件的权限, 需要读取的字节数
(16)	fs/open.c:sys_close	关闭的文件对应的 inode 号, 对应文件结构种的句柄引用数目
(17)	fs/namei.c:dir_namei	文件名和路径
(18)	fs/namei.c:get_dir	得到路径和 inode 节点号
(19)	fs/namei.c:find_entry	寻找的名字和长度
(20)	fs/inode.c:iget	设备的号码和 inode 的序号
(21)	fs/read_write.c:sys_read	标志着开始读文件
(22)	fs/file_dev.c:file_read	读取文件所在的块号和当时修改的时间

(23)	<code>fs/file_dev.c:file_write</code>	需要写入的字节数目以及已写入的数目
------	---------------------------------------	-------------------

12.3 提取数据的具体方法

将提交的附件中的 `qemu.sh` , `wwwfunction` , `filee.c` , `list.txt` , `str1.c` 全部复制到提供的“操作系统课程设计实验环境”虚拟机中的`/home/debian` 这个文件夹下, 然后进入`/home/debian/桌面/functions` 文件夹下, 双击打开 `edit_rootfs_hd`, 将刚刚的文件 `filee.c` , `list.txt` , `str1.c` 全部复制粘贴到打开之后的文件系统的`/home/debian/linux0.11-lab/rootfs/_hda/usr/root` 这个文件夹下¹。完成这一步之后, 关闭 `edit_rootfs_hd`。

以上操作可以参照附件中录屏: 在虚拟机数据提取准备.mp4

双击打开`/home/debian/桌面/functions` 文件夹下的 `normal_boot_hd`, 然后在跳出来的命令行中输入 `gcc -o str1 str1.c` 和 `gcc -o filee filee.c` 这两条指令编译我们之前放入文件夹下面的两个 C 程序, 完成后退出 `normal_boot_hd`。

准备工作完成后, 回到`/home/debian` 这个文件夹下, 先在这个文件夹下打开 `shell` 窗口, 同时双击 `qemu.sh` 使模拟窗口打开, 在`/home/debian` 的 `shell` 窗口中输入 `gdb -x gbdscript > log0.txt`, 然后 `qemu` 模拟窗口开始运行, 在模拟窗口初始化完成²后, 先在刚刚双击打开 `qemu.sh` 的那个文件夹里面找到 `log0.txt`, 将这个 `txt` 文件复制以下再次粘贴到相同目录下, 命名为 `log00.txt`, 然后返回 `QEMU` 模拟窗口开始输入 `str1` 并回车以运行 `str1`, 在命令行打出 6 行输出语句后重新出现 `[/usr/root]#` 的时候, 回到双击打开 `qemu.sh` 的那个文件夹中找到 `log0.txt`, 将这个 `txt` 文件复制以下再次粘贴到相同目录下, 命名为 `log01.txt`, 然后回到 `QEMU` 窗口输入 `filee` 并回车以运行 `filee` 程序, 在命令行重新出现 `[/usr/root]#` 的时候, 回到双击打开 `qemu.sh` 的那个文件夹中找到 `log0.txt`, 将这个 `txt` 文件复制以下再次粘贴到相同目录下, 命名为 `log02.txt`。

现在关闭 `QEMU` 和 `debian` 的命令行窗口, 回到`/home/debian` 这

¹即双击 `edit_rootfs_hd` 后, 对于新弹出的文件系统双击 `usr` 文件夹, 再双击 `usr` 文件夹下的 `root` 文件夹, 然后进去 `root` 文件夹后, 将文件复制粘贴进去

²即最新一行出现 `[/usr/root]#`

个目录下，我们可以看到有 log00.txt,log01.txt,log02.txt 三个 txt 文件，将这三个文件复制粘贴到你的主机所在的 Windows 系统的任何位置，打开这三个文件开始进行最后的处理。

以上的操作可以参照附件中的录屏：在虚拟机中数据提取.mp4

同时打开 log00.txt,log01.txt,log02.txt,首先查看 log01.txt 中的行数记作 n_1 行³并在 log02.txt 中删掉 n_1 行；查看 log00.txt 有多少行⁴记作 n_0 行，在 log01.txt 中删掉 n_0 行，这么操作之后我们得到了三个阶段各自的数据 txt 文件。接下来对于 log01.txt 和 log02.txt 进行进一步的筛选操作：

对于 log01.txt，全文搜索 copy_process，找到从上到下第一个匹配的地方（比如第 n 行），然后将这上面的数据全部删掉⁵，然后在剩下的文章中搜索 do_exit，找到从上到下第一个匹配的地方（比如第 m 行），然后将这一个 do_exit 后（不包含这个 do_exit 的断点数据）的所有行全部删掉⁶，到此，log01.txt 的处理正式完毕。

对于 log02.txt，全文搜索 copy_process，找到从上到下第一个匹配的地方（比如第 n 行），然后将这上面的数据全部删掉⁷，然后在剩下的文章中搜索 do_exit，找到从上到下第一个匹配的地方（比如第 m 行），然后将这一个 do_exit 后（不包含这个 do_exit 的断点数据）的所有行全部删掉⁸，到此，数据的处理正式完毕。

将处理完成后的 log00.txt, log01.txt, log02.txt 三个文件放入附件中的 Alpha/Alpha/bin/Debug 文件夹下，然后就可以运行同目录下的 Alpha.exe 得到可视化效果了。

³本例子中有 12043 行

⁴本例子中有 9646 行

⁵即第一行到第 $n-1$ 行

⁶假设现在剩下的共有 mm 行，则删掉第 $m+10$ 行到第 mm 行

⁷即第一行到第 $n-1$ 行

⁸假设现在剩下的共有 mm 行，则删掉第 $m+10$ 行到第 mm 行

13

系统运行过程实例

13.1 例 1：创建进程

13.1.1 shell 进程调用 sys_fork() 创建 str1 进程

涉及到的断点信息是打在 copy_process 函数中的，详细信息如下面截图所示：

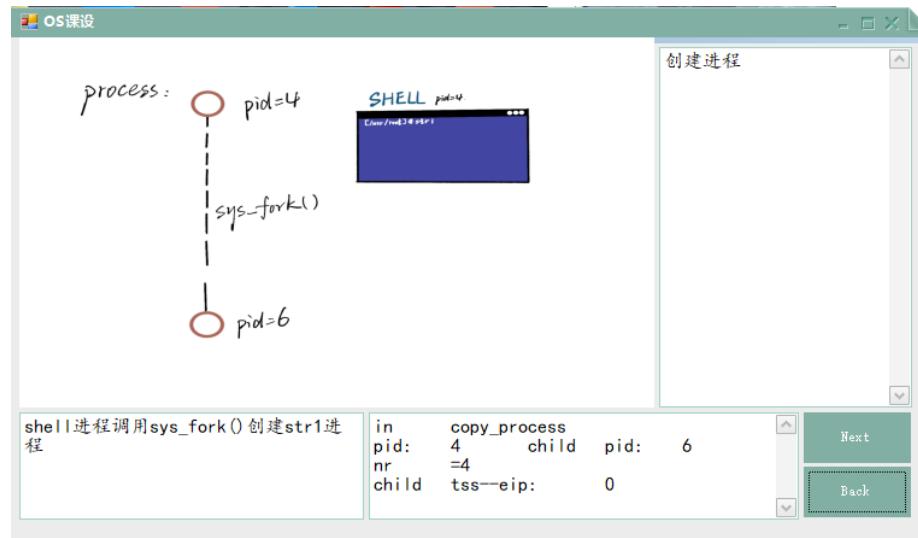


图 13.1：创建进程

13.1.2 调用 copy_mem() 取得父进程段限长和段基址

涉及到断点信息打在 copy_mem() 中，详细信息如下：

```
1: in      copy_mem  
2: old_code_base: 63293  
3: old_data_base: 161528  
4: data_limit is: 139264  
5: code_limit is: 162928  
6: new_data_base is: 139264
```

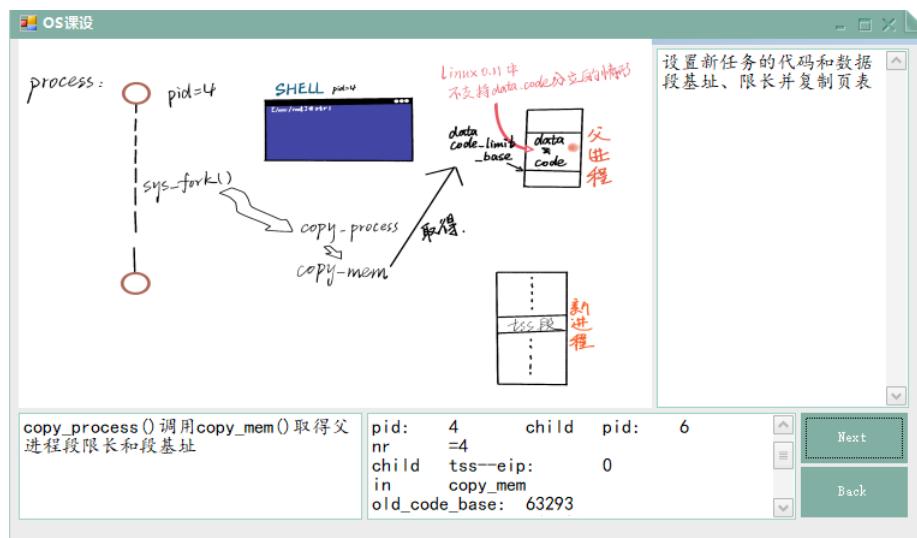


图 13.2: copy_mem()

13.2 例 2: str1 进程的加载运行

13.2.1 调用 do_execve 等函数

涉及到的断点信息是在 do_execve(), do_execve_namei() 等函数中的，截图如下：

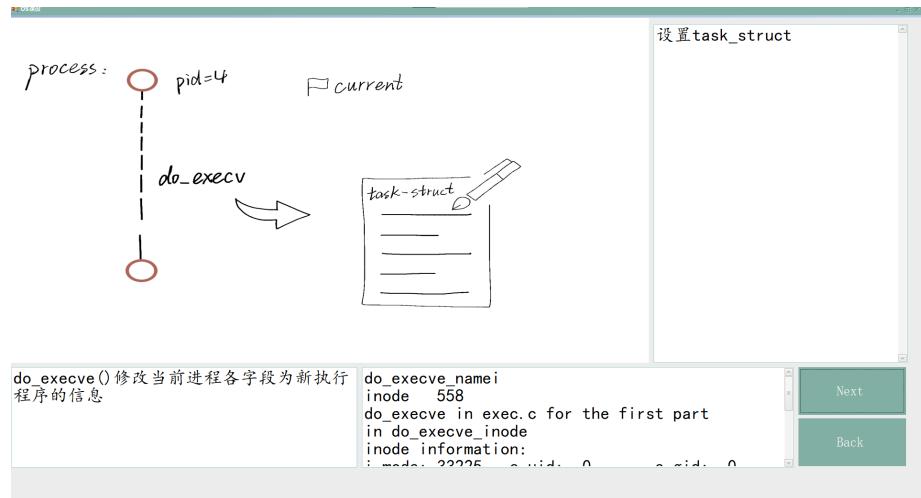


图 13.3：加载进程

13.2.2 str1 运行

首先 str1.c 的代码如下：

```

1:      #include <stdio.h>
2:      int foo(int n)
3:      {
4:          char text[2048];
5:          if(n==0)
6:              return 0;
7:          else{
8:              int i=0;
9:              for(i; i<2048; i++)
10:                  text[i] = '\0';
11:              printf("text_%d=%0x%x, pid=%d\n", n, text, getpid());
12:              sleep(5);
13:              foo(n-1);
14:          }
15:      }
16:      int main(int argc, char **argv)

```

```
17:         {
18:             foo(6);
19:             return 0;
20:         }
```

断点信息主要集中在 `do_no_page` 和相关的信号处理上，部分截图如下：

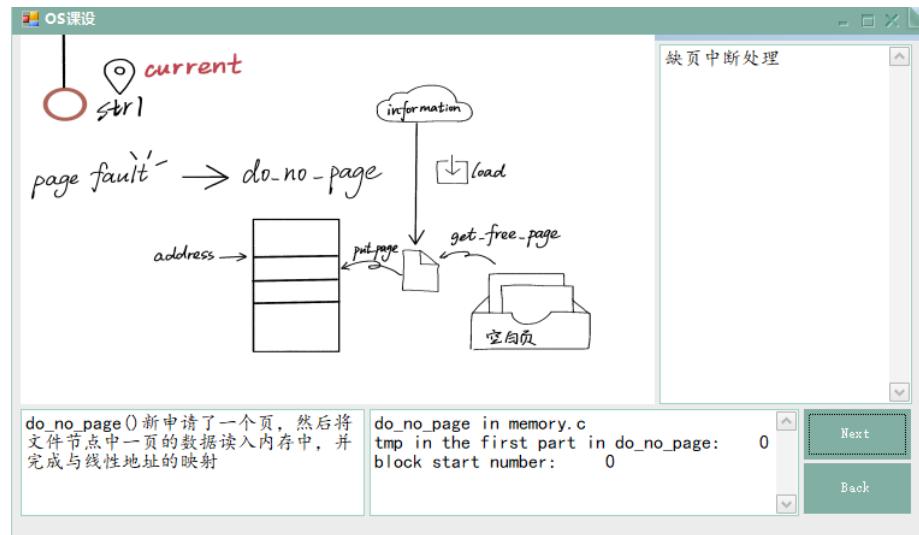


图 13.4：运行程序

13.2.3 进程退出

进程退出主要涉及的断点都在 `do_exit()` 上，截图如下：

13.3 filee 运行以及相关文件操作

13.3.1 filee 源代码

`filee.c` 的源代码如下：

```
1: #include <stdio.h>
```

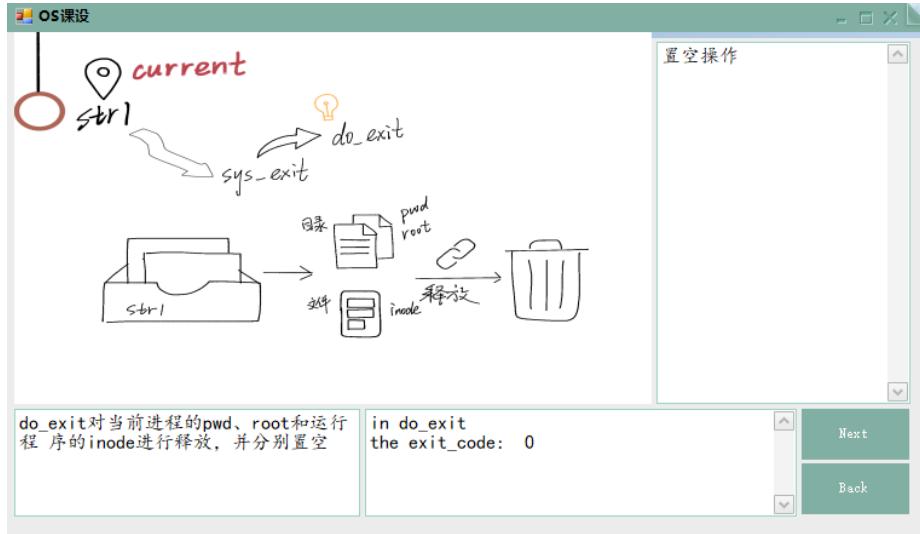


图 13.5: 进程退出

```

2:         #include <stdlib.h>
3:         #include <unistd.h>
4:         #include <sys/types.h>
5:         #include <sys/stat.h>
6:         #include <fcntl.h>
7:         int main(int argc, char *argv[])
8:         {
9:             char buffer[12000];
10:            int fd = open("/usr/root/list.txt", O_RDWR, 0644);
11:            int size = read(fd, buffer, sizeof(buffer));
12:            char s[] = "Linux Programmer!\n";
13:            int sizee = write(fd, s, sizeof(s));
14:            return 0;
15:         }

```

13.3.2 相关文件操作以及截图

具体的文件操作已经在上面解释过了，所以接下来放入几张截图作为展示：

第一张是打开文件操作截图：

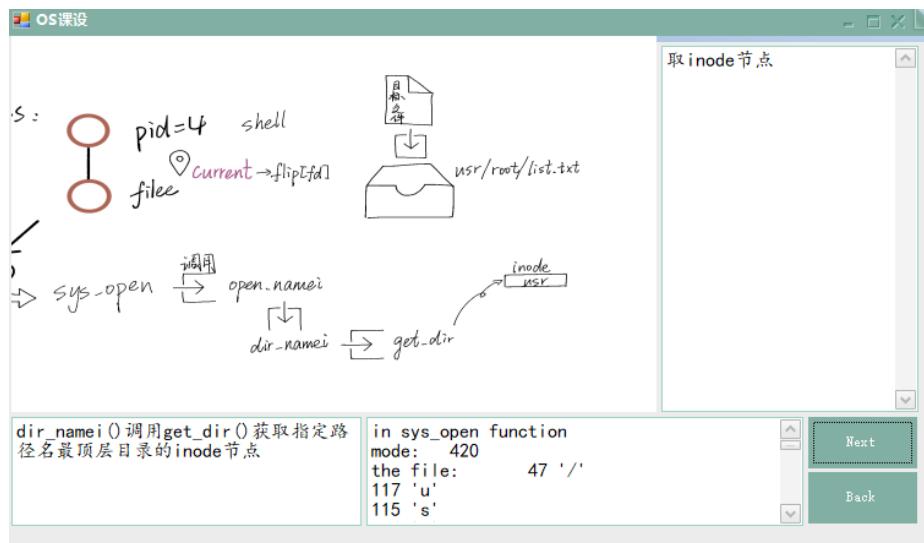


图 13.6： 打开文件

第二张是读文件的截图：

第三章是写文件的截图：

第四张是关闭文件的截图：

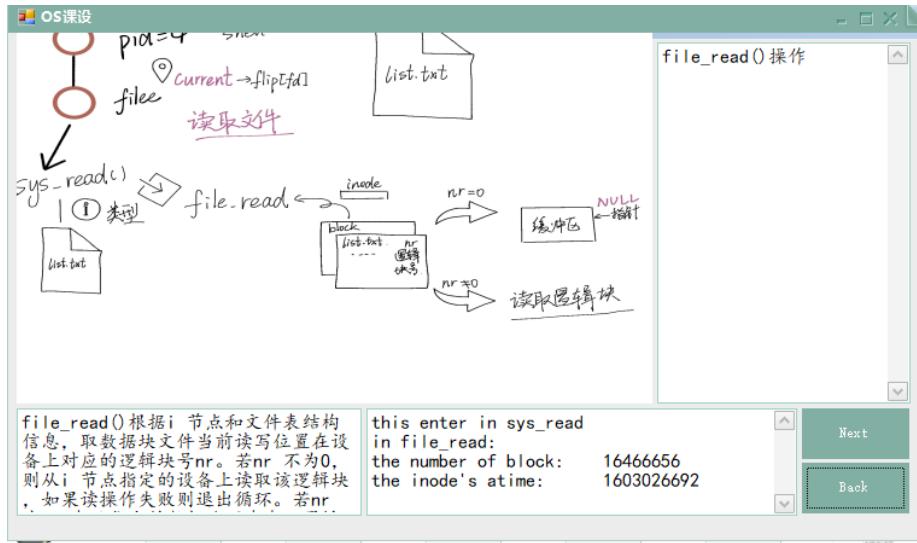


图 13.7：读取文件

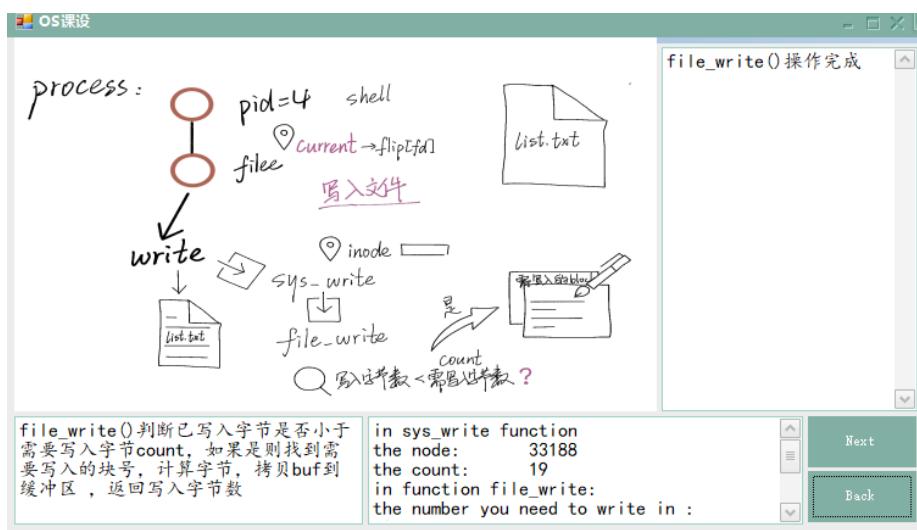


图 13.8：写文件

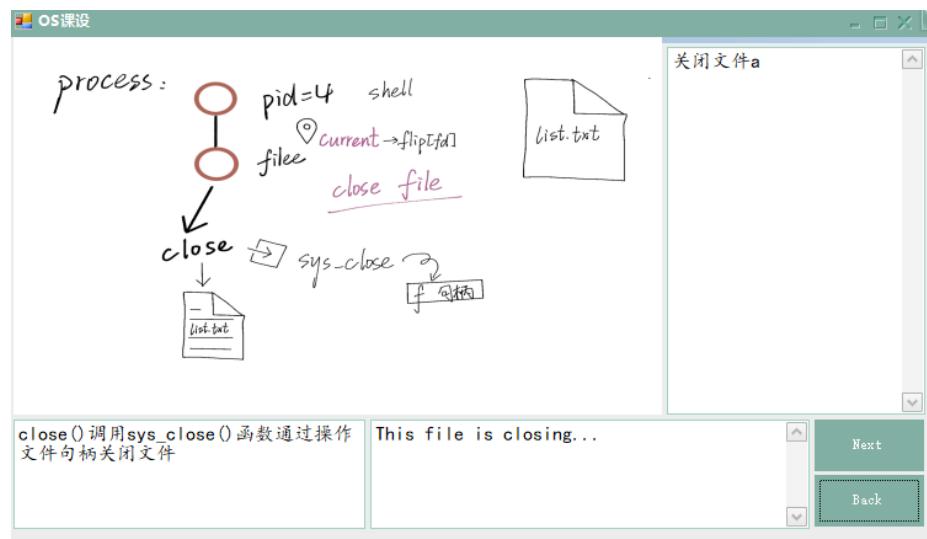


图 13.9：关闭文件

14

Linux 系统新旧版本区别

14.1 Linux0.11 在用户进程上与现在的 Linux 区别

1. 可以从文件 `fork.c` 中的函数 `copy_men()` 中看出, Linux0.11 在创建进程的时候不允许进程的代码段和数据段的段基址不在一起, 而现在的 Linux 明显是可以的; 并且, 此函数中还显示出 Linux0.11 对于段限长的要求比现在的 Linux 对段限长的限制要小很多, 这是因为从前的 Linux0.11 所在的电脑的硬件和现在的硬件条件不一样。
2. 在文件 `exec.c` 的函数 `do_execve()` 中可以看出, Linux0.11 对于代码、数据、堆长度的判定条件是“是否超过 48MB”, 这与现在的 Linux 明显是不同的。

14.2 Linux0.11 在文件操作上与现在的 Linux 区别

文件操作上与现在 Linux 的版本区别主要是在很多文件操作的具体操作上, 在 Linux0.11 中很多文件类型的操作是不支持的, 相应的函数中并没有写入有意义的代码, 现在的 Linux 系统中已经完善了这些文件

操作。