操作系统课程设计实验报告

宋振华

学号: 201605301357

1 简述

1.1 Linux 简述

Linux 操作系统是 UNIX 操作系统的一种克隆系统. 它诞生于 1991 年, 此后借助于 Internet 网络, 现已成为当今世界上使用最多的一种 UNIX 类操作系统, 并且使用人数还在迅猛增长.

Linux 操作系统的诞生、发展和成长过程依赖于以下 5 个重要支柱: UNIX 操作系统、MINIX 操作系统、GNU 计划、POSIX 标准和 Internet 网络.

通过阅读 Linux 早期内核版本的源代码, 是学习 Linux 系统的一种行之有效的途径, 并且对研究和应用 Linux 嵌入式系统也有很大帮助.

正如 Linux 系统创始人在一篇新闻组投稿上所说的, 要理解一个软件系统的真正运行机制, 一定要阅读其源代码 (RTFSC). 系统本身是一个完整的整体, 具有很多看似不重要的细节存在, 但是若忽略这些细节, 就会对整个系统的理解带来困难.

目前的 Linux 内核源代码量都在几百万行的数量上, 极其庞大, 对这些版本进行完全注释和说明几乎是不可能的, 而 0.11 版内核不超过 2 万行代码量, 麻雀虽小, 五脏俱全.

1.2 实验目的

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.

2 阅读源码

2.1 思维导图

在阅读源码的过程中, 我利用 XMind 8 思维导图工具, 将 Linux 0.11 结构, 绘制成一张静态的思维导图, 以加深对 Linux0.11 内核各部分的理解. [1]

2.2 源码理解

以下是部分源码理解概述, 更详细的源码理解, 请参考我的 Github 网站[2].

2.2.1 引导/启动/初始化

引导器——bootsect.s 将全部 linux 内核载入内存, 跳转至 setup.s 执行.

初始化 CPU 和其他硬件——setup.s 获取系统信息, 初始化 CPU 和其他硬件.

初始化 c 语言运行环境——head.s 进一步设置 CPU 和内存, 使得可以直接运行 C 语言编译好的程序.

初始化内核功能——main.c 初始化内核的所有模块和功能, 创建 idle 进程, 调用 shell 程序.

[1] 思维导图源码见https://github.com/sfd158/oldlinux-homework/blob/master/linux0.11_v1.0.xmind

[2] https://github.com/sfd158/oldlinux-homework/blob/master/html/explains.json

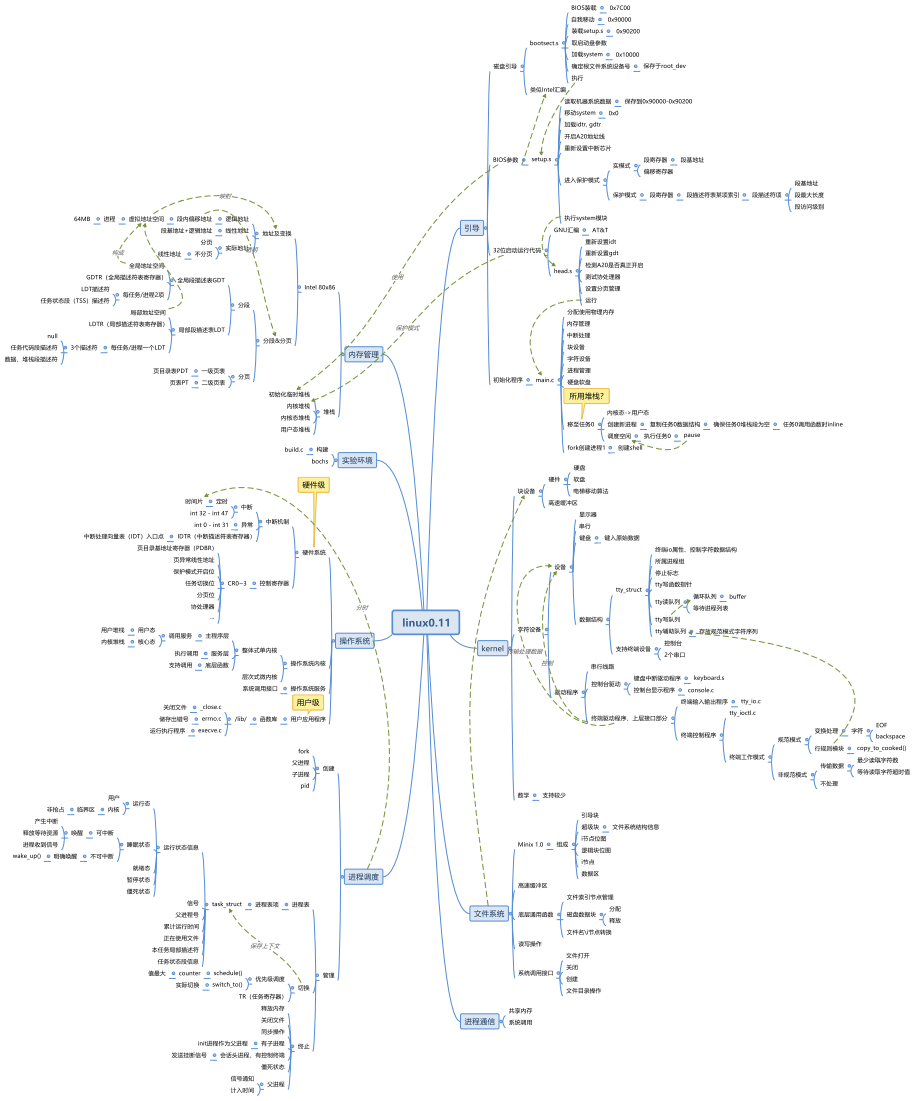

图 1: linux0.11 思维导图

2.2.2 运行

系统调用 向用户程序提供系统调用接口, 是一个操作系统内核所必须具有的功能. 在 include/unistd.h 中列出了所有（72 个）系统调用号, 用户程序通过使用 int 0x80 指令调用这些系统调用. 大部分系统调用是对文件的操作, 只有少部分涉及退出、返回等.

内存管理 内存管理主要靠硬件实现, linux0.11 的内存管理代码是配合着 80x86 的内存管理方案编写的.

它对外提供的功能有内存的申请和释放, 以及对用户程序运行环境的基本保护. 用户程序通过系统调用可以使用其中部分功能.

进程调度 实现了进程的创建、销毁、切换、暂停、唤醒、中断等, 在内核内部使用一个结构体保存了每一个进程的基本信息, 在系统忙时通过时钟中断来实现多任务切换.

进程通信 支持内存共享、信号的进程通信方式, 通过特定的系统调用来实现.

文件系统 在 linux 中, 文件系统不仅仅是用来访问存储的. 根据 unix 的标准, 所有一般硬件都会通过文件系统来抽象成统一的访问接口, 这就使得文件系统的功能非常强大.

在 linux0.11 中, 只有两种设备文件：块设备和字符设备, 使用同一个系统调用, 提供了不同的访问方法.

3 选择可视化模块

3.1 可视化部分

1. 开机启动过程: 统计了开机启动过程中, 所有 C 语言函数被调用的情况.

2. 字符显示过程:

   控制台输入 echo hello 系统的执行情况;

控制台输入 a.out(该程序在控制台输出 hello,world!) 系统的执行情况.

3.2 提取什么数据

1. 对于开机启动过程, 当每个 C 语言函数被执行到时, 输出调用栈, 以观察调用、被调用情况.
2. 对于字符设备, 当每个 C 语言函数被执行到时, 输出调用栈, 以观察调用、被调用情况; 同时当屏幕被修改时, 输出屏幕内容.

4 可视化方案

编程语言 可视化展示界面使用 HTML5+JavaScript+CSS3. 优点: 设计界面较为快捷.

实现如下功能:

1. 介绍每个源文件的用途;
2. 统计启动过程调用信息, 以柱状图形式展现;
3. 实现字符输出过程的动画, 包括

   在终端执行 echo hello 与 a.out;

4.1 最终效果

在网页上, 呈现出以下效果: [3]

5 提取数据细节

5.1 数据格式

由于数据极多, 此处只列举输出数据示例. [4]

[3] 可通过http://123.207.166.164:23333/使用此可视化界面.

[4] 详细数据和源代码可以查看https://github.com/sfd158/oldlinux-homework

图 2: 主页, 包含对每个文件功能的介绍; 模拟内存及 console

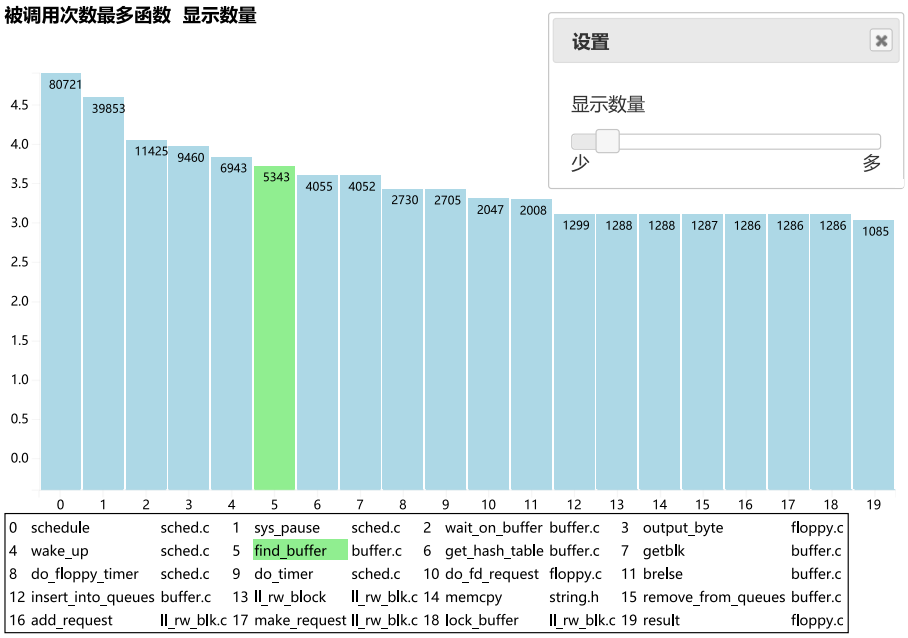

图 3: “引导界面调用次数最多”的函数统计图. 表格与条形图可进行简单的交互. 纵坐标为 log 坐标轴

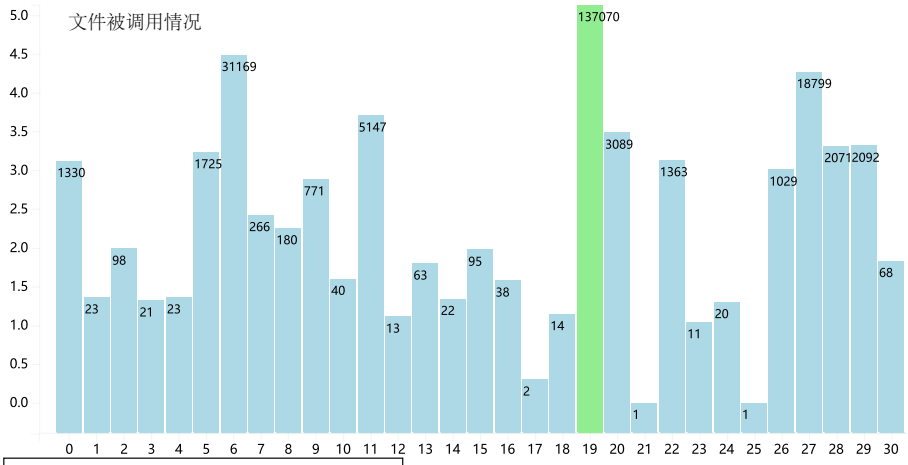

图 4: “引导界面调用次数最多”的文件统计图

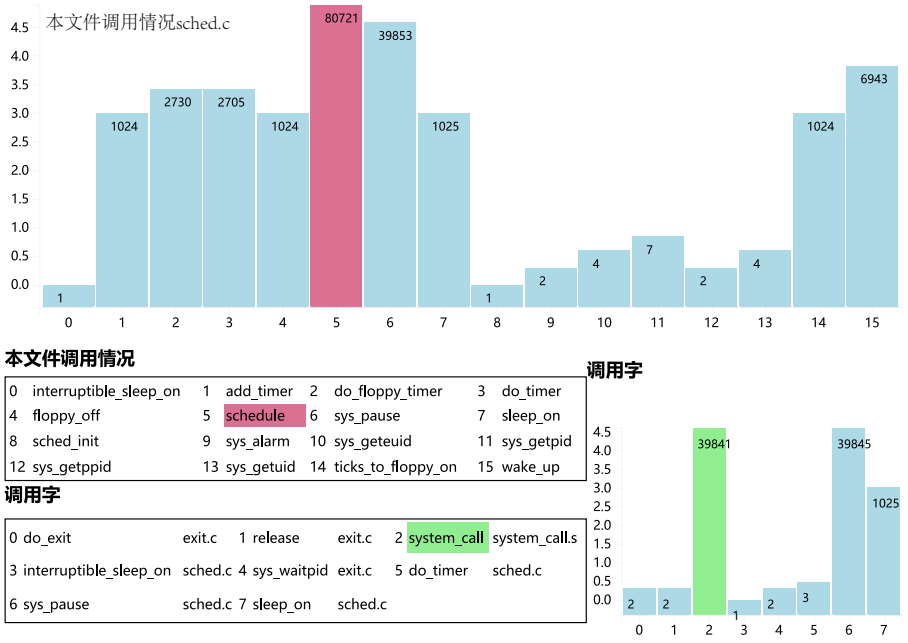

图 5: 这是 sched.c 中函数统计调用图. 点击某函数, 可显示对应的调用字.

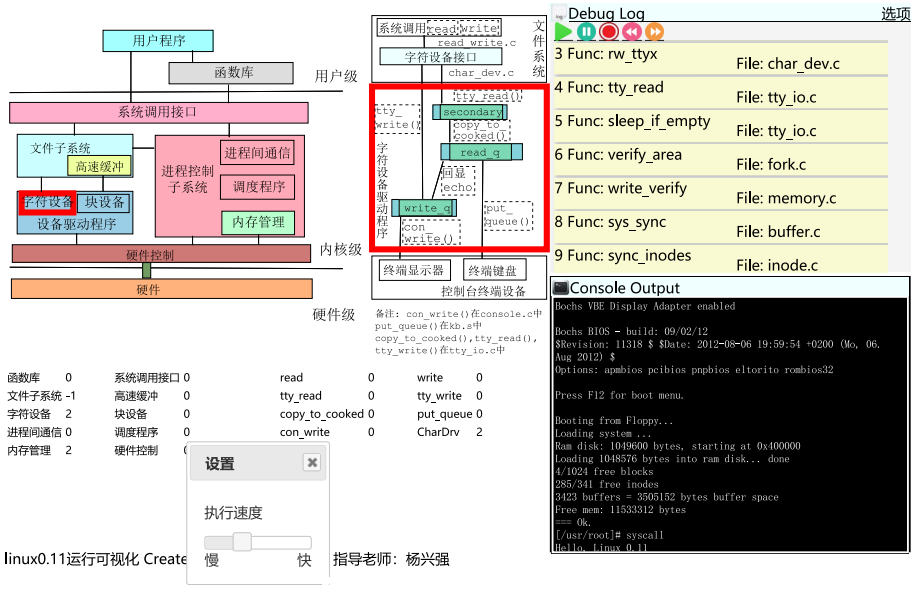

图 6: 字符设备动图

文件功能说明

```
"bitmap_c":
{
"src":"0.11/fs/bitmap.c",
"brief":"文件系统部分",
"detailed":
"包括对i节点位图和逻辑块位图进行释放和占用处理函数.
操作i节点位图的函数是free_inode()和new_inode(),
操作逻辑块位图的函数是free_block()和new_block()"
}
```

函数类别说明

```
{
"verify_area":"MemoryManage",
"write_verify":"MemoryManage",
"rw_char":"CharDrv",
"rw_ttyx":"CharDrv",
"tty_read":"CharDrv",
```

```
...
}
```

执行过程记录

```
[
{"funcName":"verify_area", "fileName":"fork.c"},     //调用栈第0层
{"funcName":"sys_read", "fileName":"read_write.c"} //调用栈第1层
]
```

5.2 注意事项

对于使用 gdb 脚本调试, 应当注意以下几个方面:

5.2.1 添加断点

缺页中断加断点 在执行过程中, 会频繁访问中断向量表, 或发生缺页中断. 此时 gdb 会收到一个信号, 并暂停.

解决方案 在对应位置添加断点, 让 gdb 在此处继续执行. 如下:

```
break head.s:19
comm
c
end
break page.s:15
comm
c
end
```

断点不宜太多 在 gdb 脚本中, 不能一次性添加太多断点, 否则会严重影响执行速度. 原因如下:

1. 频繁执行 gdb 脚本, 耗费大量时间在 gdb 输出, 保存断点等方面;

   举例来说, console.c 中的所有函数, 在系统启动时, 总计被调用了 5000 多次.

如果在诸多函数添加断点, 系统启动过程, 断点可能被执行到几十万次.

解决方案 如果需要大量记录数据, 可以将大量断点分成若干批次, 每次执行只加入其中一部分断点.

优点: 每次调试产生的数据量较小. 由于 gdb 执行时间不影响 bochs 时钟中断产生次数, 因此不会产生时间片轮转次数大幅增加的误差.

缺点: 不能完全保证两次执行操作完全相同, (如时钟中断时机不一定完全相同).

相比而言, 由于 linux 具有很好的模块性, 各模块之间耦合性较低, 分别调试问题不大.

加断点需谨慎 在 linux0.11 中, 有些函数执行极其频繁, 如 sched.c 中 void schedule(void) 函数 (该函数用于时间片轮转). 如需要 gdb 连续地执行代码 (区别与单步调试), 这些函数会频繁被执行, 从而严重影响速度. 原因同上.

有些函数定义后, 从未被使用, 在编译时这些函数会被自动优化掉. 如 include/string.h 中 memmove, memcmp, memchr 函数. 如果在这些函数上添加断点, gdb 会在别的位置暂停, 或是崩溃.

汇编语言添加断点 汇编语言中定义的函数, 类似于这样:

```
keyboard_interrupt:
pushl %eax
pushl %ebx
pushl %ecx
pushl %edx
push %ds
push %es
movl $0x10,%eax
mov %ax,%ds
mov %ax,%es
xor %al,%al             /* %eax is scan code */
inb $0x60,%al
cmpb $0xe0,%al
```

其中 keyboard_interrupt 为函数入口, 随后几条 pushl 等语句, 为函数参数初始化过程. 调试汇编时, 汇编文件中不是所有的行都会执行, 比如 pushl eax；pushl ebx 连接在一起可能在 x86 机器语言中只有一条指令, 这个时候在 pushl ebx 加断点是无效的, 该断点会被跳过.

因此, 在 pushl %eax 这一条语句上添加断点, 是不会被执行到的.

解决方案 使用 gdb 图形化工具, 在汇编语言函数入口处, 多设几个断点, 观察会在哪里暂停. 会暂停的位置, 可以在 gdb 脚本中设为断点.

5.2.2 关于 Makefile

及时清理生成代码 代码生成过程会有许多中间文件, 不要将中间文件错误地当做源代码文件. 如 kernel/chr_drv/kb.S(源代码) 和 keyboard.s(中间代码).

亲测在 keyboard.s 中添加断点, 会输出很多奇怪的东西.

解决方案 及时执行 make distclean 操作, 清除中间文件. (该操作已在陈宇翔脚本中实现, 自动完成)

5.2.3 gdb 脚本技巧

1. gdb 脚本可以添加函数, 以简化代码;
2. gdb 脚本 source 语句可以导入别的 gdb 脚本, 类似于 C 语言的 #include 语句;
3. gdb 脚本中 set 定义的变量, 是全局变量;
4. 在 bochs 环境中, gdb 脚本某一处出现 bug, 可能导致 bochs 退出. 建议对 gdb 脚本做好备份;
5. 直接 print 字符串, gdb 会每次生成一个新字符串, 时间一长内存消耗极大, 建议先定义再输出, 定义可以加在任何地方, 输出静态字符串还是建议使用 echo.

5.3 提取数据环境

使用了陈宇翔配置的 gdb 输出环境.

5.3.1 内核的运行

为了调试方便, 也考虑到可行性, 将编译好的 linux 放在一个 80x86 仿真器（虚拟机）上运行是一个比较好的选择.

但是, 仿真器并不能直接运行代码, 为了让 linux 真正的运行起来, 还有非常多的工作需要做. 比如为 linux0.11 制作虚拟硬盘, 这个步骤又牵涉出了 linux 只是个内核, 我们需要给硬盘里填入需要的用户程序, 也就是一整套的 unix 根目录环境. 遇到的问题真是数不胜数.

为了减少工作量, 为了“站在巨人的肩膀上”, 我们找到了 linux-0.11-lab[5]. 这是一个可以直接运行的 linux0.11 环境.

5.3.2 提取方案的选择

自古以来, 从运行的软件中提取运行状态数据就分为两大派别：输出派[6]和调试派[7], 本组成员一直是比较坚定的调试派, 而且根据本次实验的无处输出特点, 调试派有着明显的优势.

经过研究可以发现, linux-0.11-lab 项目使用了 QEMU 和 Bochs 虚拟机, 他们都支持 GDB Stub, 即可以使用 gdb 调试内部正在运行的操作系统. gdb 则是一个非常强大的调试工具, 可以导出软件运行过程中几乎所有数据. 于是, 提取方案的基本方向确定了：GDB 调试.

5.4 该方案的优点

1. 完全不需要修改 linux0.11 源码, 不破坏原有的代码结构.
2. 完全不因为导出数据浪费的时间而影响原来系统中发生的时钟中断.
3. 能够导出非常详细的信息, 从内存到寄存器再到屏幕的内容, 能想到就能做到.
4. 能够导出系统最开始阶段的信息（bootsect.s、setup.s 等）, 此时屏幕、栈等还没有被内核初始化.
5. gdb 脚本功能非常强大（图灵完备）, 可以做很多复杂的操作, 能想到就做得到.

[5]网址为https://github.com/tinyclub/linux-0.11-lab

[6]修改程序, 直接输出中间变量

[7]使用调试器调试程序, 在执行到断点处查看程序的状态

5.5 该环境运行截图

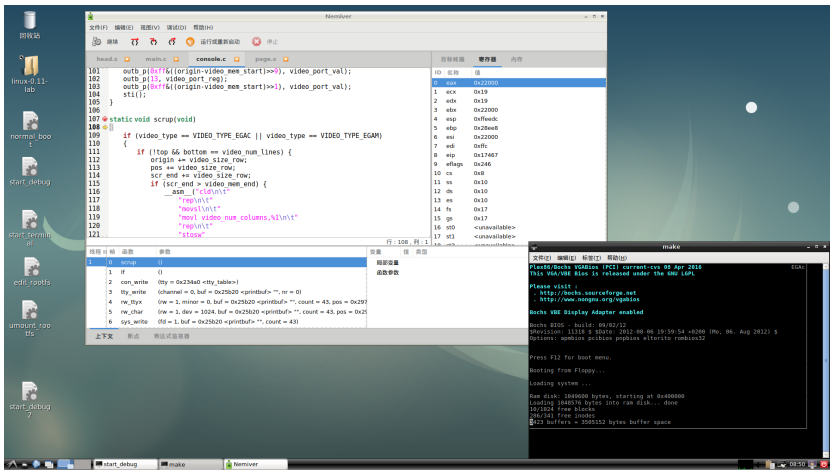

图 7: gdb 图形界面, 用于单步调试 Linux 0.11

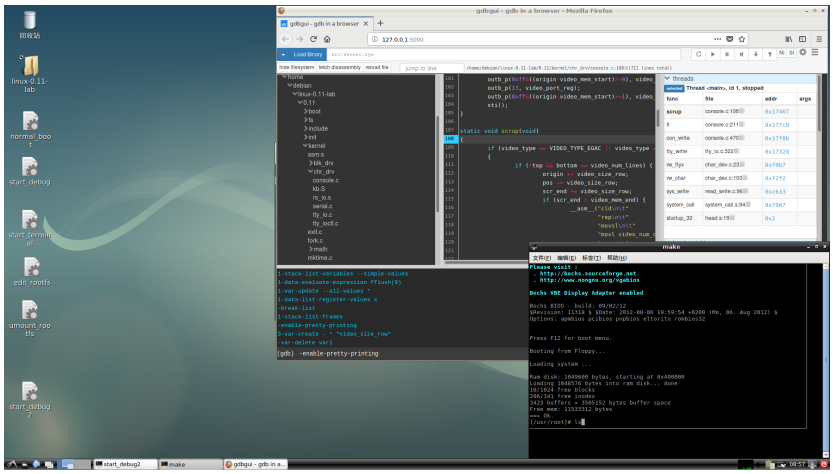

图 8: gdb 网页界面, 用于单步调试 Linux 0.11

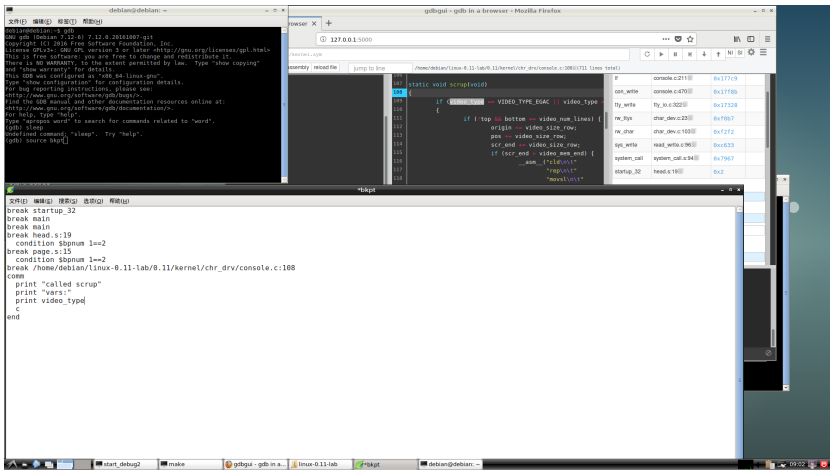

图 9: 使用 gdb 脚本调试

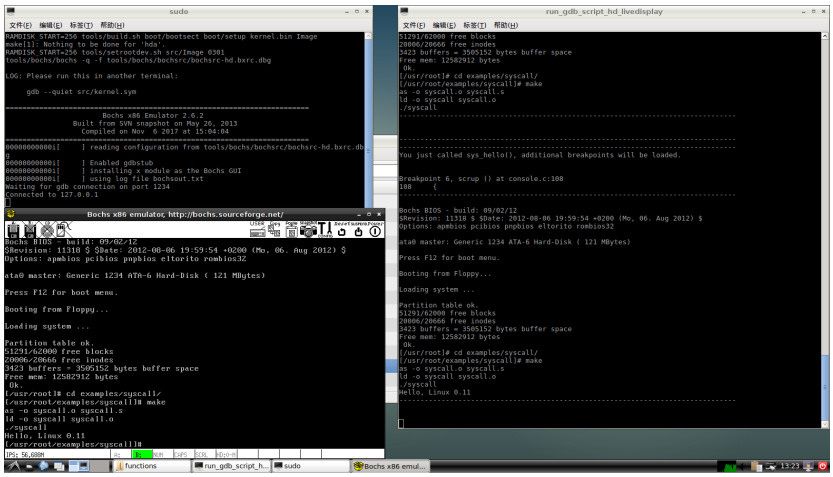

图 10: 执行系统调用 syscall 的过程

6 感想

6.1 不足之处

1. 由于时间原因, 动画效果不是特别精美;
2. 个别时间, 存在同组两人一人忙, 另一人闲的情况;

3. 本实验报告中, 引用格式不是特别规整;
4. 可视化部分代码复用性较差.

6.2 收获

1. 对 Linux 0.11 有了进一步的认识, 加深了对操作系统课程内容的理解;
2. 理解 linux 操作系统的基本工作原理;
3. 对同步、异步有了更深的认识;
4. 更加熟练地使用 gdb 调试脚本;
5. 积累了 JavaScript 制作动图的经验;
6. 积累了处理矢量图的经验, 本实验报告中一部分插图 (包括可视化应用界面), 均为矢量图, 可在 pdf 阅读器中任意放大;
7. 同陈宇翔同学合作的过程中, 加强了我们合作交流的能力.

6.3 对课程的建议

1. 建议同学们加强沟通与交流, 以实现合作互补. 我与陈宇翔同组, 陈宇翔学了汇编语言选修课, 专攻提取数据的方案; 我选了 Web 技术, 专攻可视化方案, 效果很好.
2. 建议下一级同学阅读代码时, 不要太纠结于细节;
3. 建议将提取数据的时间增加一周, 可视化的时间减少一周.