操作系统课程设计报告

陈宇翔 201605301353

2018 年 12 月 23 日

目录

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性趣的模块重点分析。

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

3.1 实验以 Linux 操作系统为背景，涉及

- 操作系统原理 (80%)
- 计算机组成 (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

3.2 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 引导/启动/初始化

9.1.1 引导器——bootsect.s

将全部 linux 内核载入内存，跳转至 setup.s 执行。

9.1.2 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。

9.1.3 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。

9.1.4 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。

9.2 运行

9.2.1 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。

9.2.2 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。

9.2.3 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。

9.2.4 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。

9.2.5 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。

10

系统运行过程的形式化描述方法

10.1 选择可视化模块

10.1.1 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

10.1.2 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.

10.2 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;

11

内核运行数据输出方法

完成这项工作面临这许多的挑战，首先如何让 linux0.11 的代码运行起来就是问题。而且由于 linux0.11 是一个操作系统，从操作系统里面提取数据与普通的程序会有很大的不同，比如我们甚至无法找到一个可以使用的输入输出流来输出信息。

众多的挑战提示我们，需要摒弃传统思路，开发新的方案。下面是我研究方案的过程以及研究出的方案的具体操作方法。

11.1 方案研究的过程

11.1.1 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github 上的 tinyclub/linux-0.11-lab[1]。这是一个可以直接运行的 linux0.11 环境。

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

11.1.2 搭建环境

linux-0.11-lab 项目需要在一定的环境下运行。

根据该项目的文档，创建一个 debian 虚拟机，配置好图形化界面，安装 cscope exuberant-ctags build-essential qemu bochs vgabios bochsbios bochs-doc bochs-x libltdl7 bochs-sdl bochs-term 软件包。

11.1.3 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[2]和调试派[3]，本人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。

11.1.4 协作

不是所有人都会使用 gdb，不是所有人都会配置环境，为了让协作变得简单，我将一个已经配置完好的系统打包共享。这样就可以和同组的人合作进行工作，也能减少因为环境不一致而导致的各种不便。

11.1.5 技术细节

在这里，按条目列举一下这个系统的所有技术细节，以便有一定基础的人更加细致地了解本系统的工作原理。

- 使用 Debian9.6 发行版
- 安装了 LXDE 桌面环境和一些常用软件
- 安装了所有必要的 VMware guest 驱动
- 开启了自动登录，默认用户为 debian，这个用户可以不输入密码使用 sudo

[2]修改程序，直接输出中间变量

[3]使用调试器调试程序，在执行到断点处查看程序的状态

- 默认用户 debian 的密码为 admin
- 安装了必要的编译套件，使得 linux-0.11-lab 的所有功能均可正常运行
- 写了若干脚本，使得许多事情可以依靠双击一个脚本来完成
- linux-0.11-lab 放在默认用户主目录下，内部做了如下修改

  - 使用 linux-0.11-lab 中 examples/syscall/syscall.patch 给内核添加了一个系统调用，作用详见后面部分

  - 在 linux-0.11-lab 目录中添加了 gdb_script 脚本，作为根脚本，用来初始化 gdb 以及提供一些功能

  - 在 linux-0.11-lab 目录中添加了 gdb_script_beforeboot 和 gdb_script_aftercall 脚本，内部加了几个例子，作用详见后面部分
- 写了几个基本的说明文件

11.2 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。
6. 还有很多……

11.3 具体提取方法

预备知识：首先简单介绍一下 gdb 调试脚本基础知识，有基础的可以直接跳过这一部分

代码 11.1 常用语句

```
# 井号表示注释
echo <内容> # 显示一些内容（不会自动换行，但是可以添加\n令其换行）
print <变量名> # 显示变量
print <表达式> # 显示表达式的计算结果
i locals # 显示所有局部变量
i r <寄存器名> # 显示寄存器内容
i r a # 显示所有寄存器
bt [<最大深度>] # 以给定的最大深度显示调用堆栈，不给定则输出全部
set $<变量名>=<内容> # 定义gdb内部变量
$<变量名> # 引用gdb内部变量
```

代码 11.2 条件分支语句

```
if <条件>
    [操作1]
    [操作2]
    [...]
[else]
    [操作1]
    [操作2]
    [...]
end
```

代码 11.3 循环语句

```
while <条件>
    [操作1]
    [操作2]
    [...]
end
```

代码 11.4 调用其他的 gdb 脚本（可以模块化调试脚本，使其更加简洁优美）

```
source <路径>
```

代码 11.5 定义函数/过程

```
define <函数名>
    [操作1]
    [操作2]
    [...]
end
# 函数支持传参数，$arg1是第一个参数的值，$arg2是第二个参数的值...
# 调用时直接把函数名写在程序中即可，
# 不需要加括号，参数直接写在后面，以空格分隔
# 下面有一个完整的例子
```

代码 11.6 一个完整的定义函数例子

```
define myprint
    print $arg1
end
myprint "this is my print"
```

11.3.1 系统需求

- 只支持在 VMware 中运行，版本要求 15.0 及以上。
- 默认需要 512M 内存（可以降低到 128M），2 个 CPU 核心（可以降低到 1 个）。[4]

11.3.2 获取打包的虚拟机

可以从下面的渠道下载：

- **山大云盘**[5]
- **百度网盘**[6] 提取码: m3qw
- **BT 种子磁力链接**[7]

[4]不建议提高该虚拟机的配置，经过测试，该方案的性能瓶颈在于单核 CPU 性能，大于 2 核不比 2 核快，大于 512M 内存也不会带来任何好处，128M 内存可以保证所有功能正常运行但不能运行浏览器。

[5]https://icloud.qd.sdu.edu.cn:7777/#/link/FC8D568388B828307FDFB70D514E47F4?_k=hh71q9

[6]https://pan.baidu.com/s/1c4vpGbMi-UboggE7W0ixTA

[7]magnet:?xt=urn:btih:4977be76532f24650a3b90430283bf9b9c1178ef&dn=%e6%93%8d%e4%bd%9c%e7%b3%bb%e7%bb%9f%e8%af%be%e7%a8%8b%e8%ae%be%e8%ae%a1%e5%ae%9e%e9%aa%8c%e7%8e%af%e5%a2%83.ova

- **GitHub Release**[8]

11.3.3 导入 VMware

如果电脑中只安装了 VMware Workstation，可以直接双击导入 VMware，中间会弹出一次警告，点击重试即可。（在此吐槽一下 VMware 警告自己导出的虚拟机不符合规范）

11.3.4 开机进入

将虚拟机开机，等待桌面出现。

开机后，桌面上应该会有四个文件：

- **回收站**
- **linux-0.11-lab** 文件夹的链接
- **function** 里面装了所有功能
- **说明** 里面写了有关所有功能的简单说明

所有功能的命名遵循一种规范：

功能均是下划线分隔的单词组成，每个单词有一定的含义，详细含义请参考下面的介绍。

在这里列举一下所有功能：（有点多，不用着急，可以先跳过这一部分，把这里当成一个可以查的表就行）

- **clean** 清理中间文件（建议加断点前清理一下，免得把中间文件加上断点导致一些奇怪的错误）
- **edit_gdb_script_*** 编辑 gdb 脚本（打开会出现详细说明）
- **edit_rootfs_floppy** 修改软盘启动模式下的文件
- **edit_rootfs_hd** 修改硬盘启动模式下的文件

[8] https://github.com/sfd158/oldlinux-homework/releases/tag/v1.0

- **normal__boot__floppy** 软盘模式启动
- **normal__boot__hd** 硬盘模式启动
- **run__gdb__script__floppy__console__livedisplay** 软盘模式 debug 启动，将屏幕显示为命令行（影响键盘输入时 Linux 的行为），实时显示 debug 输出
- **run__gdb__script__floppy__console__redirect** 软盘模式 debug 启动，将屏幕显示为命令行（影响键盘输入时 Linux 的行为），重定向 debug 输出到文件
- **run__gdb__script__floppy__livedisplay** 软盘模式 debug 启动，正常显示屏幕（无法复制内容），实时显示 debug 输出
- **run__gdb__script__floppy__redirect** 软盘模式 debug 启动，正常显示屏幕（无法复制内容），重定向 debug 输出到文件
- **run__gdb__script__hd__console__livedisplay** 硬盘模式 debug 启动，将屏幕显示为命令行（影响键盘输入时 Linux 的行为），实时显示 debug 输出
- **run__gdb__script__hd__console__redirect** 硬盘模式 debug 启动，将屏幕显示为命令行（影响键盘输入时 Linux 的行为），重定向 debug 输出到文件
- **run__gdb__script__hd__livedisplay** 硬盘模式 debug 启动，正常显示屏幕（无法复制内容），实时显示 debug 输出
- **run__gdb__script__hd__redirect** 硬盘模式 debug 启动，正常显示屏幕（无法复制内容），重定向 debug 输出到文件
- **start__debug__floppy** 软盘模式 debug 启动，正常显示屏幕（无法复制内容），打开带图形界面的调试
- **start__debug__floppy__console** 软盘模式 debug 启动，将屏幕显示为命令行（影响键盘输入时 Linux 的行为），打开带图形界面的调试
- **start__debug__hd** 硬盘模式 debug 启动，正常显示屏幕（无法复制内容），打开带图形界面的调试

- **start_debug_hd_console** 硬盘模式 debug 启动，将屏幕显示为命令行（影响键盘输入时 Linux 的行为），打开带图形界面的调试
- **start_terminal** 在 linux0.11 lab 目录下启动命令行（如果你对 linux0.11 lab 非常熟悉，这可能会非常有用）
- **umount_rootfs_floppy** 非法关闭 edit_rootfs_floppy 后的补救措施
- **umount_rootfs_hd** 非法关闭 edit_rootfs_hd 后的补救措施

11.3.5 正常运行 linux0.11

打开 function 文件夹里的 normal_boot_hd 和 normal_boot_floppy 可以直接启动 linux0.11。这将有助于你理解 linux0.11 的基本操作方法，预先了解 linux0.11 的系统特点。

11.3.6 图形化调试

打开 function 文件夹里的 start_debug_hd 和 start_debug_floppy 可以启动一个图形化的调试界面，可以在里面进行单步调试。这将有助于你理解 linux0.11 的代码结构、执行过程以及基本的调试方法。

11.3.7 查看说明，准备高级调试

本系统的原理是利用 gdb 调试脚本输出 linux 系统运行过程中的数据，为了减少大家的工作量，我对 gdb 脚本以及 gdb 的输出做了一下简单的封装，提供了更为方便的功能，让大家能够更为专注的进行数据提取。

打开 function 文件夹中的 edit_gdb_script_before_boot 和 edit_gdb_script_after_call_hello 会分别打开两个文件，可以在这两个文件中添加 gdb 调试脚本的内容，可以在这里面定义断点。这两个文件不是空的，可以参考里面的例子添加自己想添加的断点，也可以删除这些例子。最好只定义断点不要做别的事情，除非你知道自己在做什么，否则可能会影响数据导出系统的正常运行。

- **edit_gdb_script_before_boot** 脚本打开的文件中的断点会在系统启动之前添加。

- **edit_gdb_script_after_call_hello**　脚本打开的文件中的断点会在调用 sys_hello() 系统调用之后添加，借助这个文件，可以过滤掉开机过程中的所有无用信息，后面会介绍这个系统调用的具体意义以及使用方法。

添加断点的基本格式如下

代码 11.7 添加断点

```
break <文件名>:<行号> [条件]
    comm
        start_output
        <在此处写上你的操作>
        stop_output
    end
```

其中,start_output 是已经定义好的一个函数,作用是开始输出。stop_output 也是一个已经定义好的函数，作用是停止输出。因为 gdb 输出过多，我在 gdb 的输出后加了一个简单的过滤器来过滤无用输出，所以在每个断点处你都应该在开始和结束加上这两句话。

预置的函数

- **start_output**　开始输出
- **stop_output**　结束输出
- **printscreen**　输出当前屏幕上的内容
- **printscreen_preboot**　输出当前屏幕上的内容,与前者不同的是,这个函数用来输出 console 初始化完成之前屏幕的内容（比如 bootsect.s 时期）。

按照上述格式添加断点，得到的输出格式如下

代码 11.8 输出格式

```
<保留行>
<断点所在行号> <该行的内容>
```

```
[你的输出第1行]
[你的输出第2行]
[……]
```

其中，< 保留行 > 是指：

代码 11.9 保留行

```
Breakpoint <断点号>, <断点所在函数名>
    (<参数1>=<该参数传入值> [若该参数是指针，指针指向的内容]
    [, 其他参数，格式与之前相同]) at <文件名>:<行号>
```

由于保留行 1 可能会很长，导致 gdb 分多行显示保留行 1，为了方便程序处理，保留行只保留最后一行来确保保留行只包含一行。被省略的信息需要自行输出。

此外，调用 sys_hello() 系统调用之后会输出三行固定内容，以提示从这里开始调用了一次 sys_hello() 系统调用，具体内容可以看最后一张截图。

成功添加断点之后

添加完断点，可以打开 function 文件夹中的以 _livedisplay 结尾的文件来预览输出效果，若调试脚本有语法错误会导致整个程序直接退出，发生闪退现象时请注意查看自己的脚本是否有错误。

edit_gdb_script_after_call_hello 的使用方法

在这个文件里添加的断点只会在调用 sys_hello() 系统调用后被添加。

关于这个系统调用： 这个系统调用是我为了实现这个功能而添加到 linux 源码中去的，称为第 73 个系统调用，没有程序会调用这个系统调用，所以这项修改对系统是完全没有影响的。调用这个系统调用，linux0.11 会在内核态向屏幕输出一行字。

为了调用这个系统调用，我们需要自行编写程序进行调用，调用这个系统调用的程序源码就在 linux0.11 硬盘模式正常启动[9]后的 /usr/root/examples/syscall 目录下，在 linux0.11 中，cd 到这个目录，执行 make 即可编译并执行这个程序。经过一次编译之后，你可以把编译好的程

[9]如果你不明白“硬盘模式启动”是什么意思，可以参考前面对 normal_boot_hd 功能的介绍

序从 linux0.11 中复制出来[10]备用，因为软盘启动模式由于软盘容量太小所以没有放入这个文件。

其他注意事项

每次使用 normal_boot_hd 功能时，会将虚拟机中的～/linux-0.11-lab/examples 目录复制到 linux0.11 硬盘模式的/usr/root/examples 文件中，可以利用这一点实现更为方便的将文件从外边复制到 linux0.11 内

文件里有几个断点触发操作的例子,gdb 还支持值修改触发，特定条件下（如 a==b）触发，中断触发等

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

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，建议像第一行一样先定义再输出，定义可以加在任何地方，建议加在 11 行及以后，输出静态字符串还是建议使用 echo

printstrlable 的实际意思是输出一个让过滤器识别的标志，过滤器会从第一次标志出现开始的前 2 行一直输出到第二次标志出现 (标志行不输出)，然后重置过滤器状态等待下一个标志

[10]复制出来的方法参考 edit_rootfs_hd 功能

12

可视化方法的描述

12.1 数据的进一步处理

由于前面提到的 gdb 调试脚本输出的数据的格式不太容易直接用在网页中，所以需要在可视化与数据提取中间再加一个步骤，数据的处理。通过这一步，将 gdb 脚本的输出转换为 json 格式，从而达到目的。

12.1.1 数据格式

由于数据极多, 此处只列举输出数据示例. [1]

代码 12.1 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()"
}
```

代码 12.2 函数类别说明

```
{
```

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

```
    "verify_area":"MemoryManage",
    "write_verify":"MemoryManage",
    "rw_char":"CharDrv",
    "rw_ttyx":"CharDrv",
    "tty_read":"CharDrv",
    ...
}
```

代码 12.3 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
]
```

12.2 数据处理与可视化的总结

对于使用 gdb 脚本调试, 应当注意以下几个方面:

12.2.1 添加断点

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. gdb 单步调试, 会将函数初始化过程一步完成, 从而不会在参数初始化过程停留.

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.

12.2.2 其它问题

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 及时使用 clean 功能, 清除中间文件.

12.2.3 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.

12.3 最终效果

在网页上, 呈现出以下效果: [2]

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

图 12.1: 主页, 包含对每个文件功能的介绍; 模拟内存及 console

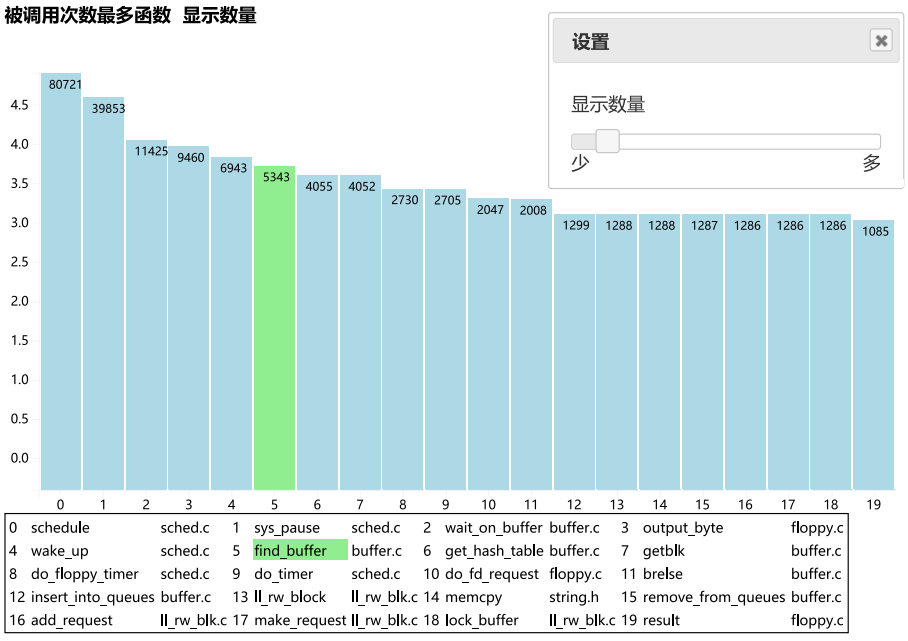

图 12.2: “引导界面调用次数最多”的函数统计图. 表格与条形图可进行简单的交互. 纵坐标为 log 坐标轴

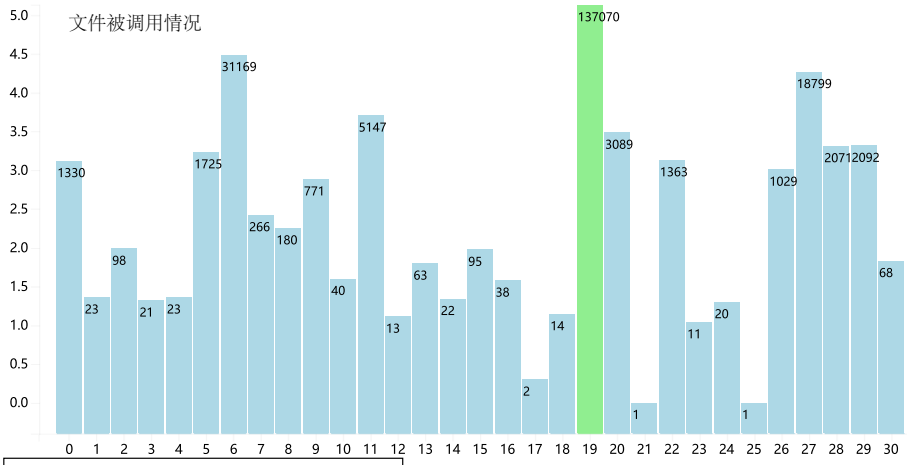

图 12.3: “引导界面调用次数最多”的文件统计图

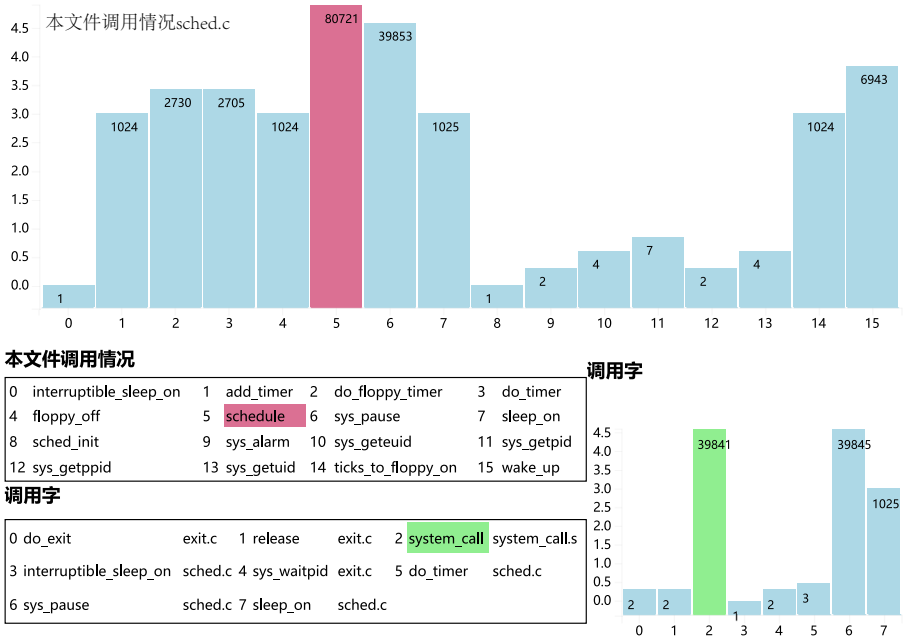

图 12.4: 这是 sched.c 中函数统计调用图. 点击某函数, 可显示对应的调用字.

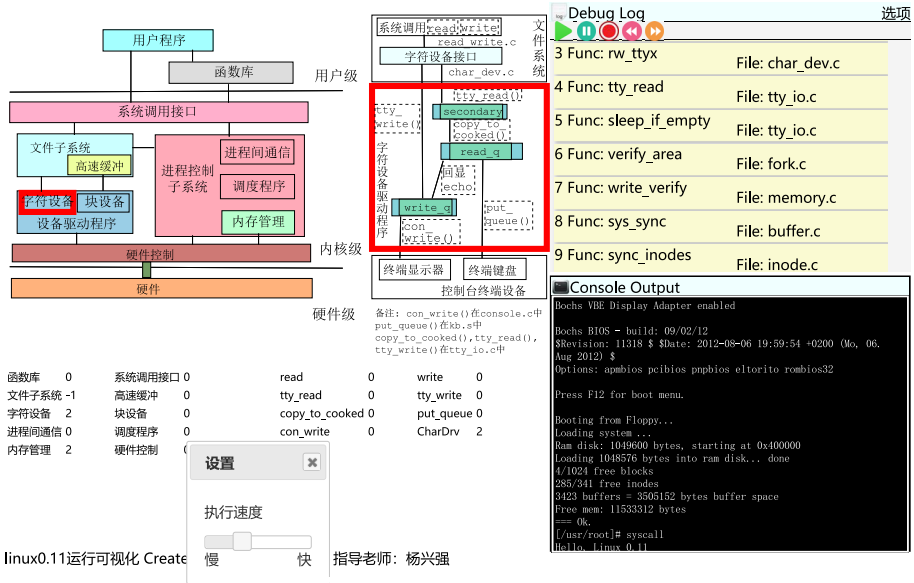

图 12.5: 字符设备动图

13

系统运行过程实例

不多说，直接上图

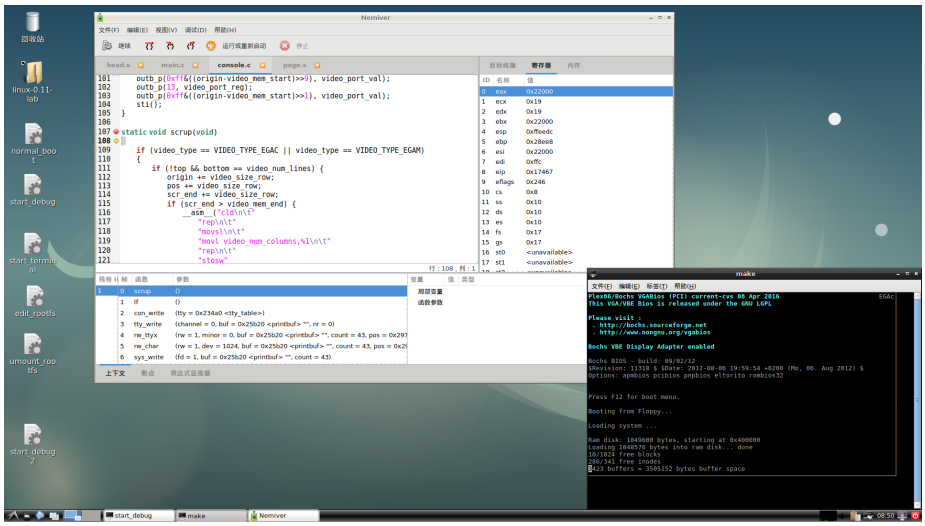

图 13.1: 图形化单步调试

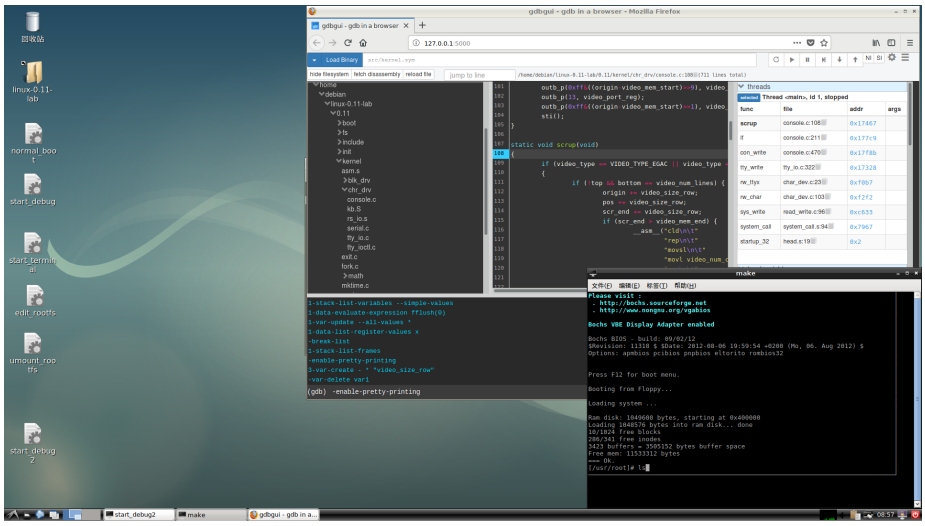

图 13.2: 图形化单步调试 2

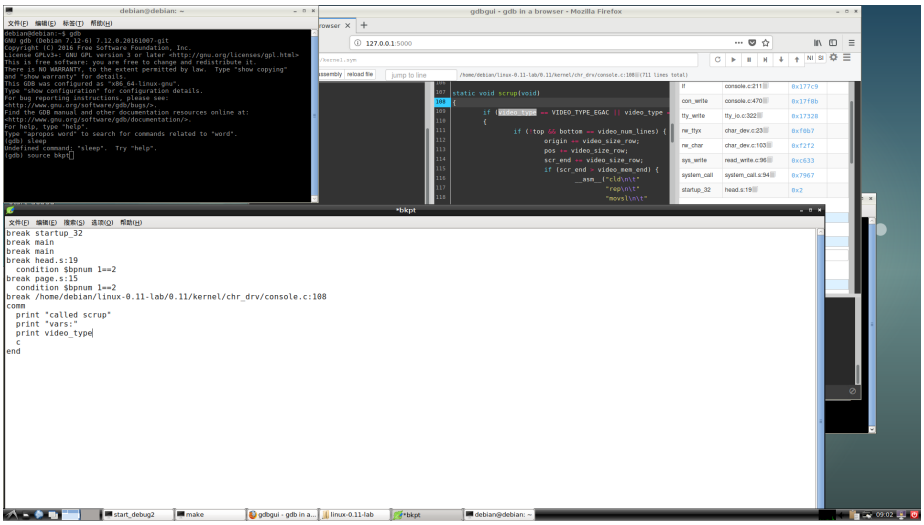

图 13.3: 手动使用 GDB 调试

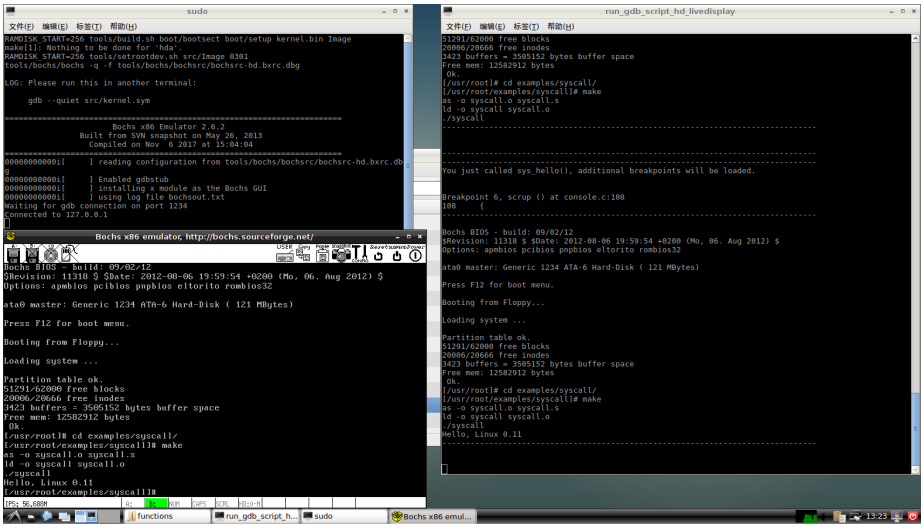

图 13.4: 启动脚本调试并调用 sys_call 系统调用