OS第 2 次实验

内核与系统调用

151220030 → 高子腾 : GZT@outlook.com

2017 年 4 月 9 日

谨以此实验报告纪录惨淡的清明和之后的周末

https://github.com/sebgao/sebopesyslab

第一部分 实验环境

编译虚拟机环境 Debian (32bit)

GCC 编译版本 gcc version 4.9.2 (Debian 4.9.2-10)

第二部分 实验结果

完成了所有要求，没有实现分页

1. 实现一个简单的内核，将内核和游戏分开
2. 实现用户态格式化输出函数 printf
3. 根据游戏要求实现系统调用
4. 将系统调用封装为库函数并将游戏重构

第三部分 详细实验结果

1 内核与游戏的分离

内核在于 kernel 目录下，游戏在于 game 目录下，在 Makefile 中可以找到它们俩的编译选项，可以看出来确实是分开编译的。

```
$(KERNEL): $(KERNEL_O) $(LIB_O)
  $(LD) -e _start -Ttext=0x100000 -m elf_i386 -T $(LD_SCRIPT)
     -nostdlib -o $@ $^ $(shell $(CC) $(KERNEL_CFLAGS)
     -print-libgcc-file-name)
  ./fill.sh $@

$(GAME): $(GAME_O) $(LIB_O)
  $(LD) -e main -Ttext=0x200000 -m elf_i386 -nostdlib -o $@ $^
     $(shell $(CC) $(GAME_CFLAGS) -print-libgcc-file-name)
```

然后通过以下代码拼成 disk.bin 文件。

```
$(IMAGE): $(BOOT) $(KERNEL) $(GAME)
  @$(DD) if=/dev/zero of=$(IMAGE) count=10000  > /dev/null #
     准备磁盘文件
  @$(DD) if=$(BOOT) of=$(IMAGE) conv=notrunc  > /dev/null # 填充 boot
     loader
  @$(DD) if=$(KERNEL) of=$(IMAGE) seek=1 conv=notrunc > /dev/null #
     填充 kernel, 跨过 mbr
  @$(DD) if=$(GAME) of=$(IMAGE) seek=201 conv=notrunc > /dev/null #
     填充 game
```

那么问题就来了，kernel 怎么加载镜像中的游戏并执行呢？其实也类似于 boot 中的操作，读取磁盘加载 elf 等等，然后得到 elf->entry 这个游戏入口地址。既然是有这个入口的，我们直接 call 或者 jmp 就好了。不过，这太低级了，不是真正的内核与程序分离的思想实现。我们在内核中要做的事除了加载 elf、初始化串口、时钟、中断描述符表开启中断外，还有很重要的一步就是开启分段机制。这里借鉴了框架代码，但没有实现分页机制，而且分段也是扁平的，只用来作为特权检查用。开启分段机制包括填充描述符表，初始化一个 tss 段描述符，设置 tr 段选择符指向它。在一切就绪后，kernel 初始化 pcb 池，创建进程，设置进程的用户栈，内核栈和段描述符，设置进程的入口为 elf->entry。由于这里没有页表的概念，我写的程序用户栈是固定的，所以进程理论上只能运行一个。在所有就绪之后，就可以 iret 到这个进程的用户态。

一个坑是最初没注意到特权级发生了变化，没有压入 ss 和 esp，造成 iret 失败，枉费我四个小时的青春 debug。

另外一个小的坑是加载 elf 时没注意到在映像中的偏移量。

2 实现用户态格式化输出函数 printf

我实现的这个 printf 其实是在用户态格式化后一个个用字符系统调用输出的，这还是比较 naïve 的做法。

还有一点就是在 kernel 中是保留 printk 的。因为在 kernel 的 debug 过程中，尤其异常和中断处理程序中，由于没实现嵌套中断，所以还是由 printk 当道的。

3 游戏的系统调用

这个游戏的系统调用比较多，可以在 lib/syscall.h 中查到它们的定义。另外，我本来是想加上一个添加时钟回调函数的系统调用，可是后来想了一想，在时钟中断时程序处在内核态要再调用用户态的程序，就必须要做特权级、栈的切换，还是比较耗时的，而且不适合内核精简的架构，所以这一系列过程可以在代码中找到，可是没有被启用。

4 将系统调用封装为库函数并将游戏重构

关于屏幕的系统调用都在 game/include/stage.h 下进行封装。实验一的两层屏幕架构 video.h，stage.h 就自然地被分到了 kernel 和 game 中去。

第四部分 必答问题

(a) 在这里，建议你搭上顺风车，让内核就跑在从 0 开始的物理地址中，至于为什么，你可以思考如果不这么干应该怎么做

我觉得不这么干也是可以的，只是占用了一定的内存空间，比如我的实验就是从 0x100000 开始的

(b) 系统调用过程中如果涉及到指针的传送，由于段式存储的特点，我们需要知道指针对应的段到底是哪一个，这样我们才能获取对应的基地址。到了这里，请你仔细思考为什么扁平模式下我们不需要考虑这个问题。

因为扁平模式下所有段描述符的基址和界限都是一样的，基址都是 0，所以在用户态和内核态的寻址过程中，既然所有段描述符的寻址效果都是一样的，段的切换相当于透明，所以就不用管段是哪一个了。

(c) 经过讨论，我们发现还是将代码段，数据段，栈都放到一个段里比较好，虽然这样会比较危险（用户栈向下增长时会将我们的代码和数据冲掉），但这是目前最实际的实现方法了。在这里请思考段页式存储是如何解决这个问题的。

段页式储存因为有页的存在，页是可以被动态分配的，所以寻址空间能较之段寻址方式大得多。所以段页式储存可以将代码段，数据段，栈分得很开，并使用扁平模式。

(d) eip，中断返回现场，请你自行思考应该设置为哪个值

eip 设置成 elf->entry，即游戏程序的入口，这样 iret 时才能以用户态进入游戏程序。

（e）也许你该仔细地思考为什么每个进程都需要自己的内核栈

如果多个进程共用一个内核栈，那么在多线程调度时如果刚好也同时系统调用，那么就 push %ebp 和 pop %ebp 来说，它们的正确性就很难保证。

第五部分　实验心得

这次的实验虽然看起来挺简单的，而且只有两周，但是真正一开始做起来却感觉难以下手，为此我还专门去复习了一下 ics 那本书中关于段机制和中断的内容，现在回看那本书，才真正知道它在讲什么，果然基础是要打扎实的，不然书越读越新。其实在两天前就写好了 Makefile 分开编译的内容，还觉得 game 和 kernel 分开原来是这么简单。但真正要进入 game 的用户态确是很难的，用户和内核栈要指定等等，总是意外来临，措手不及。而且理论知识的匮乏也使我举步维艰，最终捧起上学期的计算机系统基础。花了两天才写好进入用户态，这也是一种体会吧。另外我觉得现在我写的实验里的内存管理很笨啊，不能同时运行多个进程，下次的分页、进程调度又是一次挑战了。