

实验指导书目录

2020年9月22日 15:42

目录

[实验一 熟悉实验环境](#)

[实验二 操作系统的引导](#)

[实验三 系统调用](#)

[实验四 进程运行轨迹的跟踪与统计](#)

[实验五 基于内核栈切换的进程切换](#)

[实验六 信号量的实现与应用](#)

[实验七 地址映射与共享](#)

[实验八 终端设备的控制](#)

[实验九 proc文件系统的实现](#)

1) 熟悉实验环境

2020年9月23日 17:37

实验一 熟悉实验环境

1. 课程说明

本实验是 [操作系统之基础](#) 课程的配套实验，推荐大家进行实验之前先学习相关课程：

- L1 什么是操作系统

Tips: 点击上方文字中的超链接或者输入

入 <https://mooc.study.163.com/course/1000002004#/info> 进入理论课程的学习。如果网易云上的课程无法查看，也可以看 Bilibili 上的 [操作系统哈尔滨工业大学李治军老师](#)。

2. 主要平台和工具简介

本操作系统实验的硬件环境是 IA-32 (x86) 架构的 PC 机（在实验楼的环境中就是右侧的窗口），主要软件环境是 Bochs + gcc + 你最喜欢的编辑器 / IDE + 你最喜欢的操作系统 + Linux 0.11 源代码。

实验的基本流程是根据实验要求编写应用程序、修改 Linux 0.11 的源代码，用 gcc 编译后，在 Bochs 的虚拟环境中运行、调试目标代码。

上述实验环境涉及到的软件都是免费且开源的，具有较强的可移植性，可以在多种计算机的多种操作系统上搭建。为方便实验者，我们在最常见的平台 Ubuntu（最流行的 GNU/Linux 发行版之一）——上制作了 hit-oslab 集成环境，它基本包含了实验所需的所有软件，安装过程非常简单，基本上是直接解压就可以使用。

2.1 x86 模拟器 Bochs

Bochs 是一个免费且开放源代码的 IA-32 (x86) 架构 PC 机模拟器。在它模拟出的环境中可以运行 Linux、DOS 和各种版本的 Windows 等多种操作系统。而 Bochs 本身具有很高的移植性，可以运行在多种软硬件平台之上，这也是我们选择它作为本书的指定模拟器的主要原因。如果您想拥抱自由的 Linux，那么 Bochs 几乎是您的不二选择。如果您想继续把自己绑定在 Windows 平台上，那么除了 Bochs，您还可以选用 VMware 或者 Microsoft Virtual PC。它们是最著名虚拟机软件，而且都可以免费使用。因为 Bochs 的是模拟器，其原理决定了它的运行效率会低于虚拟机。

但对于本书所设计的实验来说，效率上的差别很不明显。而且，Bochs 有虚拟机无可比拟的调试操作系统的能力，所以我们更建议您选用 Bochs。hit-oslab 已经内置了 bochs，本实验后文假定的缺省环境也是 Bochs。

关于 Bochs 的更详细的介绍请访问它的 [主页](#) 及 Bochs 使用手册。

2.2 GCC 编译器

GCC 是和 Linux 一起成长起来的编译器。Linux 最初的版本就是由 GCC 编译的。现在 GCC 也是在自由软件领域应用最广泛的编译器。所以，我们也选择 GCC 作为本书实验的指定编译器。

2.3 GDB 调试器

GDB 调试器是 GCC 编译器的兄弟。做为自由软件领域几乎是唯一的调试器，它秉承了 Unix 类操作系统的一贯风格，采用纯命令行操作，有点儿类似 dos 下的 debug。

关于它的使用方法请看 GDB 使用手册。

另外，可以学习实验楼的 [《GDB 简明教程》](#)，通过动手实验学习 Linux 上 GDB 调试 C 语言程序的基本技巧。

2.4 Ubuntu (GNU/Linux)

Ubuntu 也许不是目前最好用的 Linux 桌面发行版，但它一定是最流行的。主要特点是易用，非常的易用。现在，已经有越来越多的人开始用 Ubuntu 完全代替 Windows，享受更加自由、

安全、守法的感觉。

Ubuntu 的主页是 <http://www.ubuntu.com/>，这里不仅可以免费下载到 iso 文件，甚至能免费申领 Ubuntu 的安装光盘。

我们强烈建议您在 Ubuntu 下做实验。因为有些实验内容涉及到在自己改进的 Linux 0.11 下，运行自己编的应用程序。被改进的功能都是高版本 Linux 内核已经具有的，在其上确认自己编写的应用程序无误后，再用之测试自己改进的 Linux 0.11，可以更有信心些。

3. 实验环境的工作模式

(1) 准备环境

hit-oslab 实验环境简称 oslab，是一个压缩文件 (hit-oslab-linux-20110823.tar.gz)，这个文件已经下载到了 /home/teacher 目录和 /home/shiyanlou/oslab (大家一进入实验环境，就是点击左边的 terminal 打开终端以后，所在的目录就是 /home/shiyanlou，这是大家的主目录) 下，大家可以使用下面的命令解压展开压缩包即可工作。

推荐大家使用如下的命令解压到 /home/shiyanlou/oslab/ 中。

```
# 进入到 oslab 所在的文件夹
$ cd /home/shiyanlou/oslab/
# 解压，并指定解压到 /home/shiyanlou/
# 这样的话，在 /home/shiyanlou/oslab/ 中就能找到解压后的所有文件
$ tar -zxvf hit-oslab-linux-20110823.tar.gz \
  -C /home/shiyanlou/
# 查看是否解压成功
$ ls -al
# 除了压缩包 hit-oslab-linux-20110823.tar.gz 之外，其他的就是压缩包中的内容
```

(2) 文件结构

- Image 文件

oslab 工作在一个宿主操作系统之上，我们使用的 Linux，在宿主操作系统之上完成对 Linux 0.11 的开发、修改和编译之后，在 linux-0.11 目录下会生产一个名为 Image 的文件，它就是编译之后的目标文件。

该文件内已经包含引导和所有内核的二进制代码。如果拿来一张软盘，从它的 0 扇区开始，逐字节写入 Image 文件的内容，就可以用这张软盘启动一台真正的计算机，并进入 Linux 0.11 内核。

oslab 采用 bochs 模拟器加载这个 Image 文件，模拟执行 Linux 0.11，这样省却了重新启动计算机的麻烦。

- bochs 目录

bochs 目录下是与 bochs 相关的执行文件、数据文件和配置文件。

- run 脚本

run 是运行 bochs 的脚本命令。

运行后 bochs 会自动在它的虚拟软驱 A 和虚拟硬盘上各挂载一个镜像文件，软驱上挂载是 linux-0.11/Image，硬盘上挂载的是 hdc-0.11.img。

因为 bochs 配置文件中的设置是从软驱 A 启动，所以 Linux 0.11 会被自动加载。

而 Linux 0.11 会驱动硬盘，并 mount 硬盘上的文件系统，也就是将 hdc-0.11.img 内镜像的文件系统挂载到 0.11 系统内的根目录 —— /。在 0.11 下访问文件系统，访问的就是 hdc-0.11.img 文件内虚拟的文件系统。

- hdc-0.11.img 文件

hdc-0.11.img 文件的格式是 Minix 文件系统的镜像。

Linux 所有版本都支持这种格式的文件系统，所以可以直接在宿主 Linux 上通过 mount 命令访问此文件内的文件，达到宿主系统和 bochs 内运行的 Linux 0.11 之间交换文件的效果。

Windows 下目前没有 (或者是还没发现) 直接访问 Minix 文件系统的办法，所以要借助于 fdb.img，这是一个 1.44M 软盘的镜像文件，内部是 FAT12 文件系统。将它挂载到 bochs 的软驱 B，就可以在 0.11 中访问它。而通过 filedisk 或者 WinImage，可以在 Windows 下访问它内部的文件。

hdc-0.11.img 内包含有：

- Bash shell;

- 一些基本的 Linux 命令、工具，比如 cp、rm、mv、tar；
 - vi 编辑器；
 - gcc 1.4 编译器，可用来编译标准 C 程序；
 - as86 和 ld86；
 - Linux 0.11 的源代码，可在 0.11 下编译，然后覆盖现有的二进制内核。
- 其他文件在后面用到的时候会进行单独讲解。

4. 使用方法

开始使用之前的准备活动：把当前目录切换到 oslab 下，用 pwd 命令确认，用 ls -l 列目录内容。

```
# 切换目录
$ cd /home/shiyanlou/oslab/
# 确认路径
$ pwd
# 查看目录内容
$ ls -l
```

本实验的所有内容都在本目录或其下级目录内完成。

```
shiyanlou@82a7ecabb02f:~$ cd /home/shiyanlou/oslab/
shiyanlou@82a7ecabb02f:~/oslab$ pwd
/home/shiyanlou/oslab
shiyanlou@82a7ecabb02f:~/oslab$ ls -l
总用量 95272
drwxr-xr-x  2 shiyanlou shiyanlou    4096  9月  5  2010 bochs
-rwxr-xr-x  1 shiyanlou shiyanlou     115  8月 30  2008 dbg-asm
-rwxr-xr-x  1 shiyanlou shiyanlou     119  8月 30  2008 dbg-c
-rwxr-xr-x  1 shiyanlou shiyanlou 12423461 8月 28  2008 gdb
-rw-r--r--  1 shiyanlou shiyanlou      75  8月 28  2008 gdb-cmd.txt
drwxr-xr-x  2 shiyanlou shiyanlou    4096  9月  5  2010 hdc
-rw-r--r--  1 shiyanlou shiyanlou 63504384 10月  2  2009 hdc-0.11.img
-rw-rw-r--  1 shiyanlou shiyanlou 21586449  1月 13  2015 hit-oslab-linux-20110823.tar.gz
drwxr-xr-x 10 shiyanlou shiyanlou    4096  8月 23  2011 linux-0.11
-rwxr-xr-x  1 shiyanlou shiyanlou     126 10月  9  2008 mount-hdc
-rwxr-xr-x  1 shiyanlou shiyanlou     254  9月  1  2009 run
-rwxr-xr-x  1 shiyanlou shiyanlou     268  9月  1  2009 run gdb
shiyanlou@82a7ecabb02f:~/oslab$
```

4.1 编译内核

“编译内核”比“编写内核”要简单得多。

首先要进入 linux-0.11 目录，然后执行 make 命令：

```
$ cd ./linux-0.11/
$ make all
```

因为 all 是最常用的参数，所以可以省略，只用 make，效果一样。

在多线程的系统上，可以用 -j 参数进行并行编译，加快速度。例如双 CPU 的系统可以：

```
$ make -j 2
```

make 命令会显示很多很多的信息，你可以尽量去看懂，也可以装作没看见。只要最后几行中没有“error”就说明编译成功。

最后生成的目标文件是一个软盘镜像文件——linux-0.11/Image（下面的图中给出了详细的信息）。如果将此镜像文件写到一张 1.44MB 的软盘上，就可以启动一台真正的计算机。

```

shiyanolou@82a7ecabb02f:~/oslab/linux-0.11$ ls -al
总用量 292
drwxr-xr-x 10 shiyanolou shiyanolou 4096 6月 18 09:57 .
drwxr-xr-x 5 shiyanolou shiyanolou 4096 9月 5 2010 ..
drwxr-xr-x 2 shiyanolou shiyanolou 4096 6月 18 09:56 boot
drwxr-xr-x 2 shiyanolou shiyanolou 4096 6月 18 09:56 fs
-rw-rw-r-- 1 shiyanolou shiyanolou 124068 6月 18 09:57 Image
drwxr-xr-x 5 shiyanolou shiyanolou 4096 9月 5 2010 include
drwxr-xr-x 2 shiyanolou shiyanolou 4096 6月 18 09:56 init
drwxr-xr-x 5 shiyanolou shiyanolou 4096 6月 18 09:56 kernel
drwxr-xr-x 2 shiyanolou shiyanolou 4096 6月 18 09:56 lib
-rw-r--r-- 1 shiyanolou shiyanolou 3442 8月 23 2011 Makefile
drwxr-xr-x 2 shiyanolou shiyanolou 4096 6月 18 09:56 mm
-rw-rw-r-- 1 shiyanolou shiyanolou 10729 6月 18 09:56 System.map
-rw-r--r-- 1 shiyanolou shiyanolou 110724 10月 9 2008 tags
drwxr-xr-x 2 shiyanolou shiyanolou 4096 6月 18 09:57 tools
shiyanolou@82a7ecabb02f:~/oslab/linux-0.11$

```

linux-0.11 目录下是全部的源代码，很多实验内容都是要靠修改这些代码来完成。修改后需要重新编译内核，还是执行命令：make all。

make 命令会自动跳过未被修改的文件，链接时直接使用上次编译生成的目标文件，从而节约编译时间。但如果重新编译后，你的修改貌似没有生效，可以试试先 make clean，再 make all（或者一行命令：make clean && make all。make clean 是删除上一次编译生成的所有中间文件和目标文件，确保是在全新的状态下编译整个工程。

4.2 运行

在 Bochs 中运行最新编译好的内核很简单，在 oslab 目录下执行：

```

# 注意是在上层目录
# 刚刚编译是在 oslab/linux-0.11/ 文件夹下
$ cd ~/oslab/
# 执行 run 脚本
$ ./run

```

如果出现 Bochs 的窗口，里面显示 linux 的引导过程，最后停止在 [/usr/root/]#，表示运行成功，如下图所示。

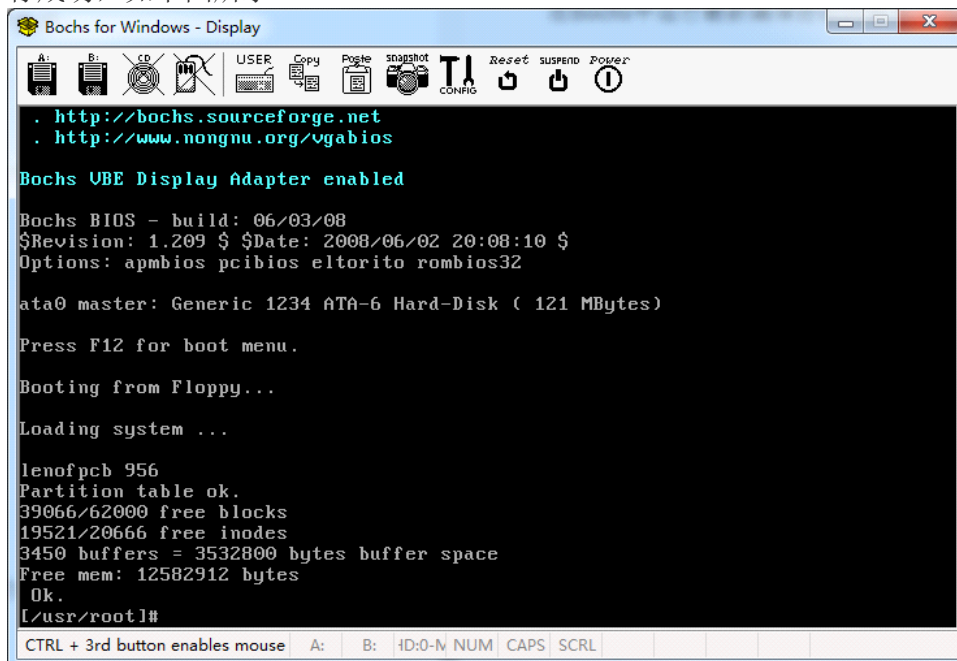


图 1 用 Bochs 启动 Linux 0.11 以后的样子

4.3 调试

内核调试分为两种模式：汇编级调试和 C 语言级调试。

（1）汇编级调试

汇编级调试需要执行命令：

```

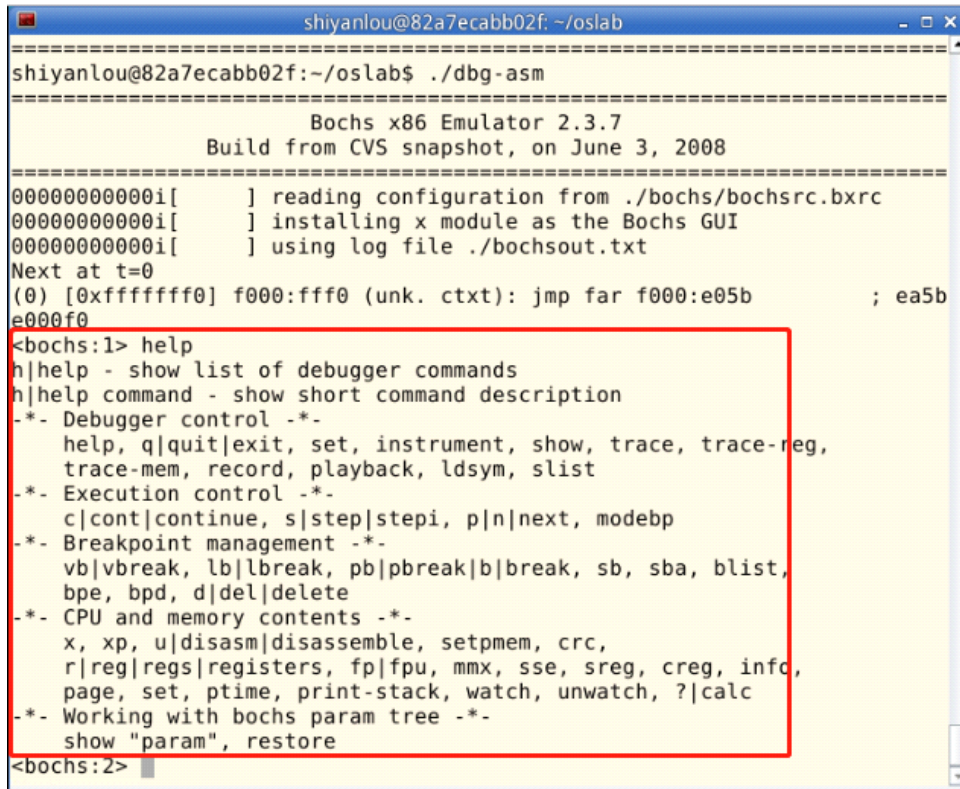
# 确认在 oslab 目录下
$ cd ~/oslab/
# 运行脚本前确定已经关闭刚刚运行的 Bochs

```



```
$ ./dbg-asm
```

汇编级调试的启动之后 Bochs 是黑屏，这是正常的。
可以用命令 `help` 来查看调试系统用的基本命令。更详细的信息请查阅 Bochs 使用手册。



```
shiyanolou@82a7ecabb02f:~/oslab$ ./dbg-asm
=====
Bochs x86 Emulator 2.3.7
Build from CVS snapshot, on June 3, 2008
=====
00000000000i[      ] reading configuration from ./bochs/bochsrc.bxrc
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file ./bochsout.txt
Next at t=0
(0) [0xffffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b      ; ea5b
e000f0
<bochs:1> help
h|help - show list of debugger commands
h|help command - show short command description
-- Debugger control --
  help, q|quit|exit, set, instrument, show, trace, trace-reg,
  trace-mem, record, playback, ldsym, slist
-- Execution control --
  c|cont|continue, s|step|stepi, p|n|next, modebp
-- Breakpoint management --
  vb|vbreak, lb|lbreak, pb|pbreak|b|break, sb, sba, blist,
  bpe, bpd, d|del|delete
-- CPU and memory contents --
  x, xp, u|disasm|disassemble, setpmem, crc,
  r|reg|regs|registers, fp|fpu, mmx, sse, sreg, creg, infc,
  page, set, ptime, print-stack, watch, unwatch, ?|calc
-- Working with bochs param tree --
  show "param", restore
<bochs:2>
```

(2) C 语言级调试

C 语言级调试稍微复杂一些。首先执行如下命令：

```
$ cd ~/oslab
$ ./dbg-c
```

然后再打开一个终端窗口，执行：

```
$ cd ~/oslab
$ ./rungdb
```

注意：启动的顺序不能交换，否则 `gdb` 无法连接。
出现下图所示的提示，才说明连接成功：

```
shiyanolou@82a7ecabb02f: ~/oslab
/home/shiyanolou/oslab/gdb-cmd.txt:2: Error in sourced command file:
localhost:1234: Connection refused.
(gdb) exi
Undefined command: "exi". Try "help".
(gdb) q
shiyanolou@82a7ecabb02f:~/oslab$ ./run
=====
Bochs x86 Emulator 2.3.7
Build from CVS snapshot, on June 3, 2008
=====
00000000000i[      ] reading configuration from ./bochs/bochsrc.bxrc
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file ./bochsout.txt
=====
Bochs is exiting with the following message:
[VGUI ] POWER button turned off.
=====
shiyanolou@82a7ecabb02f:~/oslab$ ./dbg-c
=====
Bochs x86 Emulator 2.3.7
Build from CVS snapshot, on June 3, 2008
=====
00000000000i[      ] reading configuration from ./bochs/bochsrc-gdb.bxrc
00000000000i[      ] Enabled gdbstub
00000000000i[      ] reading configuration from ./bochs/bochsrc.bxrc
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file ./bochsout.txt
Waiting for gdb connection on port 1234
Connected to 127.0.0.1
```

新终端窗口中运行的是 GDB 调试器。关于 gdb 调试器请查阅 GDB 使用手册。

4.4 文件交换

接下来讲解一下 Ubuntu 和 Linux 0.11 之间的文件交换如何启动。

开始设置文件交换之前，务必关闭所有的 Bochs 进程。

oslab 下的 hdc-0.11-new.img 是 0.11 内核启动后的根文件系统镜像文件，相当于在 bochs 虚拟机里装载的硬盘。在 Ubuntu 上访问其内容的方法是：

```
$ cd ~/oslab/
# 启动挂载脚本
$ sudo ./mount-hdc
```

大家使用 sudo 时，password 是 shiyanolou，也有可能不会提示输入密码。

之后，hdc 目录下就是和 0.11 内核一模一样的文件系统了，可以读写任何文件（可能有些文件要用 sudo 才能访问）。

```
# 进入挂载到 Ubuntu 上的目录
$ cd ~/oslab/hdc
# 查看内容
$ ls -al
```

读写完毕，不要忘了卸载这个文件系统：

```
$ cd ~/oslab/
# 卸载
$ sudo umount hdc
```

经过 sudo ./mount-hdc 这样处理以后，我们可以在 Ubuntu 的 hdc 目录下创建一个 xxx.c 文件，然后利用 Ubuntu 上的编辑工具（如 gedit 等）实现对 xxx.c 文件的编辑工作，在编辑保存以后。

执行 sudo umount hdc 后，再进入 Linux 0.11（即 run 启动 bochs 以后）就会看到这个 xxx.c（即如下图所示），这样就避免了在 Linux 0.11 上进行编辑 xxx.c 的麻烦，因为 Linux 0.11 作为一个很小的操作系统，其上的编辑工具只有 vi，使用起来非常不便。

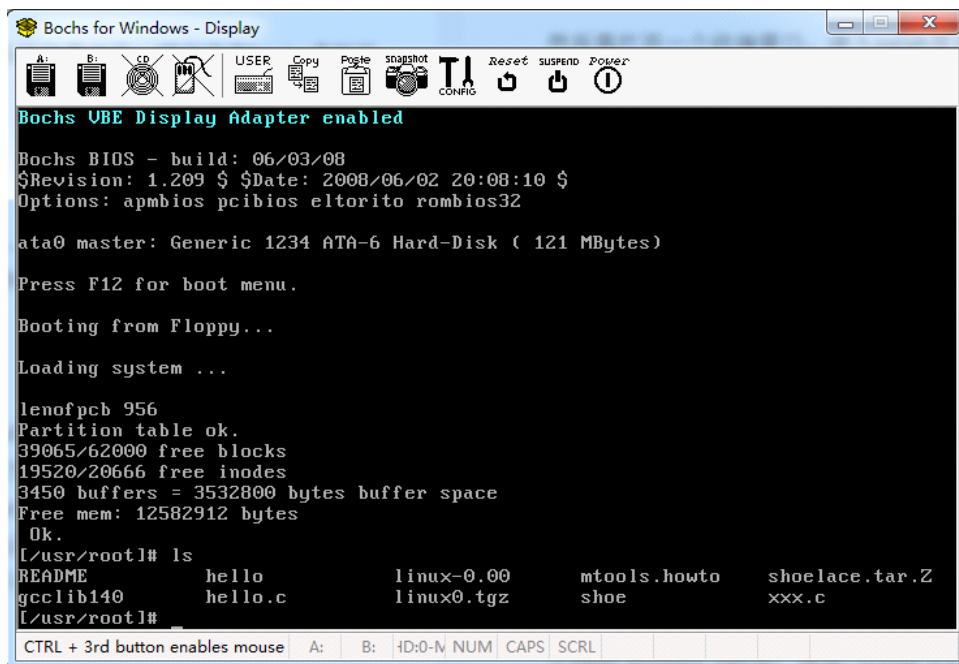


图 2 用 Ubuntu 和 Linux 0.11 完成文件交换以后再启动 Linux 0.11 以后另外在 Linux 0.11 上产生的文件，如后面实验中产生的 process.log 文件，可以按这种方式“拿到”Ubuntu 下用 python 程序进行处理，当然这个 python 程序在 Linux 0.11 上显然是不好使的，因为 Linux 0.11 上搭建不了 python 解释环境。

注意 1：不要在 0.11 内核运行的时候 mount 镜像文件，否则可能会损坏文件系统。同理，也不要再在已经 mount 的时候运行 0.11 内核。

注意 2：在关闭 Bochs 之前，需要先在 0.11 的命令行运行“sync”，确保所有缓存数据都存盘后，再关闭 Bochs。

2) 操作系统引导

2020年9月23日 17:41

实验二 操作系统的引导

1. 课程说明

本实验是 [操作系统之基础 - 网易云课堂](#) 课程的配套实验，推荐大家进行实验之前先学习相关课程：

- L2 开始揭开钢琴的盖子
- L3 操作系统启动

Tips: 点击上方文字中的超链接或者输入

<https://mooc.study.163.com/course/1000002004#/info> 进入理论课程的学习。如果网易云上的课程无法查看，也可以看 Bilibili 上的 [操作系统哈尔滨工业大学李治军老师](#)。

2. 实验目的

- 熟悉 hit-oslab 实验环境；
- 建立对操作系统引导过程的深入认识；
- 掌握操作系统的基本开发过程；
- 能对操作系统代码进行简单的控制，揭开操作系统的神秘面纱。

3. 实验内容

此次实验的基本内容是：

1. 阅读《Linux 内核完全注释》的第 6 章，对计算机和 Linux 0.11 的引导过程进行初步的了解；
2. 按照下面的要求改写 0.11 的引导程序 bootsect.s
3. 有兴趣同学可以做做进入保护模式前的设置程序 setup.s。

改写 bootsect.s 主要完成如下功能：

1. bootsect.s 能在屏幕上打印一段提示信息“XXX is booting...”，其中 XXX 是你给自己的操作系统起的名字，例如 LZJos、Sunix 等（可以上论坛上秀秀谁的 OS 名字最帅，也可以显示一个特色 logo，以表示自己操作系统的与众不同。）

改写 setup.s 主要完成如下功能：

1. bootsect.s 能完成 setup.s 的载入，并跳转到 setup.s 开始地址执行。而 setup.s 向屏幕输出一行“Now we are in SETUP”。
2. setup.s 能获取至少一个基本的硬件参数（如内存参数、显卡参数、硬盘参数等），将其存放在内存的特定地址，并输出到屏幕上。
3. setup.s 不再加载 Linux 内核，保持上述信息显示在屏幕上即可。

4. 实验报告

在实验报告中回答如下问题：

1. 有时，继承传统意味着别手蹩脚。x86 计算机为了向下兼容，导致启动过程比较复杂。请找出 x86 计算机启动过程中，被硬件强制，软件必须遵守的两个“多此一举”的步骤（多找几个也无妨），说说它们为什么多此一举，并设计更简洁的替代方案。

5. 评分标准

- bootsect 显示正确，30%
- bootsect 正确读入 setup，10%
- setup 获取硬件参数正确，20%
- setup 正确显示硬件参数，20%
- 实验报告，20%

6. 实验提示

操作系统的 boot 代码有很多，并且大部分是相似的。本实验仿照 Linux-0.11/boot 目录下的 bootsect.s 和 setup.s，以剪裁它们为主线。当然，如果能完全从头编写，并实现实验所要求的功能，是再好不过了。

同济大学赵炯博士的《Linux 内核 0.11 完全注释（修正版 V3.0）》（以后简称《注释》）的第 6 章是非常有帮助的参考，实验中可能遇到的各种问题，几乎都能找到答案。也是一份很好的参考。

需要注意的是，oslab 中的汇编代码使用 as86 编译。

下面将给出一些更具体的“提示”。这些提示并不是实验的一步一步的指导，而是罗列了一些实验中可能遇到的困难，并给予相关提示。它们肯定不会涵盖所有问题，也不保证其中的每个字都对完成实验有帮助。所以，它们更适合在你遇到问题时查阅，而不是当作指南一样地亦步亦趋。本课程所有实验的提示都是秉承这个思想编写的。

6.1 开始实验前

在正式开始实验之前，你需要先了解下面的内容：

（1）相关代码文件

Linux 0.11 文件夹中的 boot/bootsect.s、boot/setup.s 和 tools/build.c 是本实验会涉及到的源文件。它们的功能详见《注释》的 6.2、6.3 节和 16 章。

（2）引导程序的运行环境

引导程序由 BIOS 加载并运行。它活动时，操作系统还不存在，整台计算机的所有资源都由它掌控，而能利用的功能只有 BIOS 中断调用。

实验中主要使用 BIOS 0x10 和 0x13 中断。

6.2 完成 bootsect.s 的屏幕输出功能

代码中以 ! 开头的行都是注释，实际在写代码时可以忽略。

实验中所有提到的修改，均是指相对于 linux-0.11 中的代码。

首先来看完成屏幕显示的关键代码，如下：

```
! 首先读入光标位置
mov ah, #0x03
xor bh, bh
int 0x10
! 显示字符串 “Hello OS world, my name is LZJ”
! 要显示的字符串长度
mov cx, #36
mov bx, #0x0007
mov bp, #msg1
! es:bp 是显示字符串的地址
! 相比与 linux-0.11 中的代码，需要增加对 es 的处理，因为原代码中在输出之前已经处理了 es
mov ax, #0x07c0
mov es, ax
mov ax, #0x1301
int 0x10
! 设置一个无限循环
inf_loop:
jmp inf_loop
```

这里需要修改的是字符串长度，即用需要输出的字符串长度替换 mov cx, #24 中的 24。要注意：除了我们设置的字符串 msg1 之外，还有三个换行 + 回车，一共是 6 个字符。比如这里 Hello OS world, my name is LZJ 的长度是 30，加上 6 后是 36，所以代码应该修改为 mov cx, #36。

接下来就是修改输出的字符串了：

```
! msg1 处放置字符串
msg1:
! 换行 + 回车
.byte 13, 10
.ascii "Hello OS world, my name is LZJ"
```

```
! 两对换行 + 回车
.byte 13, 10, 13, 10
! boot_flag 必须在最后两个字节
.org 510
! 设置引导扇区标记 0xAA55
! 必须有它, 才能引导
boot_flag:
.word 0xAA55
```

将 .org 508 修改为 .org 510, 是因为这里不需要 root_dev: .word ROOT_DEV, 为了保证 boot_flag 一定在最后两个字节, 所以要修改 .org。

完整的代码如下:

```
entry _start
_start:
    mov ah, #0x03
    xor bh, bh
    int 0x10
    mov cx, #36
    mov bx, #0x0007
    mov bp, #msg1
    mov ax, #0x07c0
    mov es, ax
    mov ax, #0x1301
    int 0x10
inf_loop:
    jmp inf_loop
msg1:
    .byte 13, 10
    .ascii "Hello OS world, my name is LZJ"
    .byte 13, 10, 13, 10
.org 510
boot_flag:
.word 0xAA55
```

接下来, 将完成屏幕显示的代码在开发环境中编译, 并将编译后的目标文件做成 Image 文件。

6.3 编译和运行

Ubuntu 上先从终端进入 ~/oslab/linux-0.11/boot/ 目录。

Windows 上则先双击快捷方式 “MinGW32.bat”, 将打开一个命令行窗口, 当前目录是 oslab, 用 cd 命令进入 linux-0.11\boot。

无论那种系统, 都执行下面两个命令编译和链接 bootsect.s:

```
$ as86 -O -a -o bootsect.o bootsect.s
$ ld86 -O -s -o bootsect bootsect.o
```

其中 -O (注意: 这是数字 0, 不是字母 O) 表示生成 8086 的 16 位目标程序, -a 表示生成与 GNU as 和 ld 部分兼容的代码, -s 告诉链接器 ld86 去除最后生成的可执行文件中的符号信息。

如果这两个命令没有任何输出, 说明编译与链接都通过了。

Ubuntu 下用 ls -l 可列出下面的信息:

```
-rw-x--x  1 root root 544 Jul 25 15:07 bootsect
-rw----- 1 root root 257 Jul 25 15:07 bootsect.o
-rw----- 1 root root 686 Jul 25 14:28 bootsect.s
```

Windows 下用 dir 可列出下面的信息:

```
2008-07-28 20:14          544 bootsect
2008-07-28 20:14          924 bootsect.o
2008-07-26 20:13       5,059 bootsect.s
```

其中 bootsect.o 是中间文件。bootsect 是编译、链接后的目标文件。
 需要留意的是 bootsect 的文件大小是 544 字节，而引导程序必须要正好占用一个磁盘扇区，即 512 个字节。造成多了 32 个字节的原因是 ld86 产生的是 Minix 可执行文件格式，这样的可执行文件处理文本段、数据段等部分以外，还包括一个 Minix 可执行文件头部，它的结构如下：

```
struct exec {
    unsigned char a_magic[2]; //执行文件魔数
    unsigned char a_flags;
    unsigned char a_cpu; //CPU标识号
    unsigned char a_hdrlen; //头部长度，32字节或48字节
    unsigned char a_unused;
    unsigned short a_version;
    long a_text; long a_data; long a_bss; //代码段长度、数据段长度、堆长度
    long a_entry; //执行入口地址
    long a_total; //分配的内存总量
    long a_syms; //符号表大小
};
```

算一算：6 char (6 字节) + 1 short (2 字节) + 6 long (24 字节) = 32，正好是 32 个字节，去掉这 32 个字节后就可以放入引导扇区了（这是 tools/build.c 的用途之一）。
 对于上面的 Minix 可执行文件，其 a_magic[0]=0x01, a_magic[1]=0x03, a_flags=0x10（可执行文件），a_cpu=0x04（表示 Intel i8086/8088，如果是 0x17 则表示 Sun 公司的 SPARC），所以 bootsect 文件的头几个字节应该是 01 03 10 04。为了验证一下，Ubuntu 下用命令“hexdump -C bootsect”可以看到：

```
00000000 01 03 10 04 20 00 00 00 00 02 00 00 00 00 00 00 |.....|
00000010 00 00 00 00 00 00 00 00 00 82 00 00 00 00 00 00 |.....|
00000020 b8 c0 07 8e d8 8e c0 b4 03 30 ff cd 10 b9 17 00 |.....0....|
00000030 bb 07 00 bd 3f 00 b8 01 13 cd 10 b8 00 90 8e c0 |....?.....|
00000040 ba 00 00 b9 02 00 bb 00 02 b8 04 02 cd 13 73 0a |.....s....|
00000050 ba 00 00 b8 00 00 cd 13 eb e1 ea 00 00 20 90 0d |..... ..|
00000060 0a 53 75 6e 69 78 20 69 73 20 72 75 6e 6e 69 6e |.Sunix is runn|
00000070 67 21 0d 0a 0d 0a 00 00 00 00 00 00 00 00 00 |g!.....|
00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000210 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa |.....U....|
00000220
```

Windows 下用 UltraEdit 把该文件打开，果然如此。

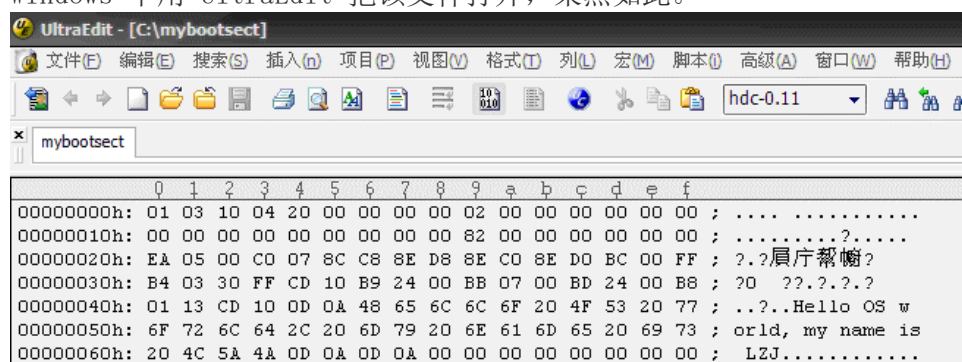


图 1 用 UltraEdit 打开文件 bootsect

接下来干什么呢？是的，要去掉这 32 个字节的文件头部（tools/build.c 的功能之一就是这这个）！随手编个小的文件读写程序都可以去掉它。不过，懒且聪明的人会在 Ubuntu 下用命令：

```
$ dd bs=1 if=bootsect of=Image skip=32
```

生成的 Image 就是去掉文件头的 bootsect。

Windows 下可以用 UltraEdit 直接删除（选中这 32 个字节，然后按 Ctrl+X）。
去掉这 32 个字节后，将生成的文件拷贝到 linux-0.11 目录下，并一定要命名为 “Image”
（注意大小写）。然后就 “run” 吧！

```
# 当前的工作路径为 /home/shiyanlou/oslab/linux-0.11/boot/  
# 将刚刚生成的 Image 复制到 linux-0.11 目录下  
$ cp ./Image ../Image  
# 执行 oslab 目录中的 run 脚本  
$ ../../run
```

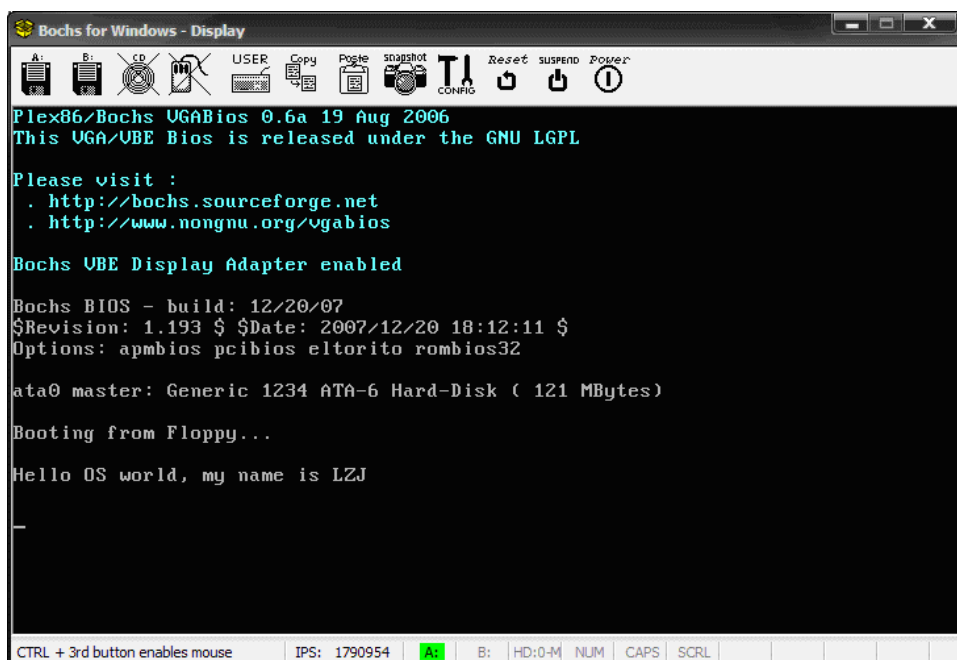
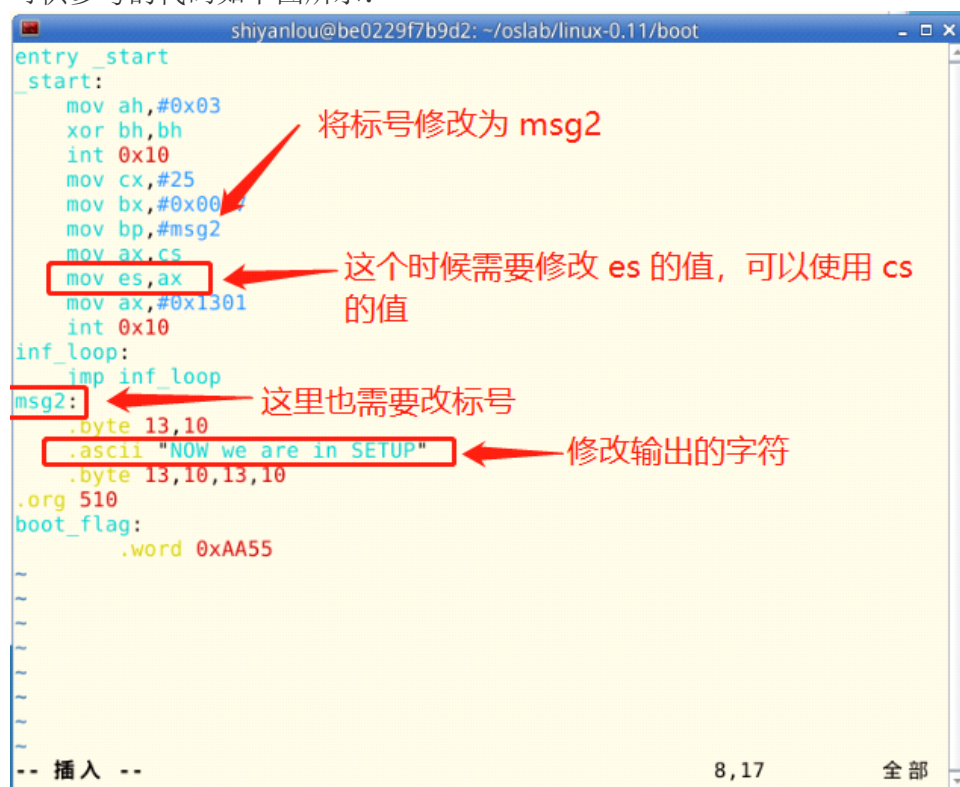


图 2 bootsect 引导后的系统启动情况

6.4 bootsect.s 读入 setup.s

首先编写一个 setup.s，该 setup.s 可以直接拷贝前面的 bootsect.s（还需要简单的调整），然后将其中的显示的信息改为：“Now we are in SETUP”。

可供参考的代码如下图所示：



接下来需要编写 bootsect.s 中载入 setup.s 的关键代码。原版 bootsect.s 中下面的代码

就是做这个的。

```
load_setup:
! 设置驱动器和磁头(drive 0, head 0): 软盘 0 磁头
    mov dx,#0x0000
! 设置扇区号和磁道(sector 2, track 0): 0 磁头、0 磁道、2 扇区
    mov cx,#0x0002
! 设置读入的内存地址: BOOTSEG+address = 512, 偏移512字节
    mov bx,#0x0200
! 设置读入的扇区个数(service 2, nr of sectors),
! SETUPLEN是读入的扇区个数, Linux 0.11 设置的是 4,
! 我们不需要那么多, 我们设置为 2 (因此还需要添加变量 SETUPLEN=2)
    mov ax,#0x0200+SETUPLEN
! 应用 0x13 号 BIOS 中断读入 2 个 setup.s扇区
    int 0x13
! 读入成功, 跳转到 ok_load_setup: ok - continue
    jnc ok_load_setup
! 软驱、软盘有问题才会执行到这里。我们的镜像文件比它们可靠多了
    mov dx,#0x0000
! 否则复位软驱 reset the diskette
    mov ax,#0x0000
    int 0x13
! 重新循环, 再次尝试读取
    jmp load_setup
ok_load_setup:
! 接下来要干什么? 当然是跳到 setup 执行。
! 要注意: 我们没有将 bootsect 移到 0x9000, 因此跳转后的段地址应该是 0x7ce0
! 即我们要设置 SETUPSEG=0x07e0
```

所有需要的功能在原版 bootsect.s 中都是存在的, 我们要做的仅仅是将这些代码添加到新的 bootsect.s 中去。

除了新增代码, 我们还需要去掉 5.2 小节中我们在 bootsect.s 添加的无限循环。

编写完成后大致如下:

```
SETUPLEN=2
SETUPSEG=0x07e0
entry _start
_start:
    mov ah,#0x03
    xor bh,bh
    int 0x10
    mov cx,#36
    mov bx,#0x0007
    mov bp,#msg1
    mov ax,#0x07c0
    mov es,ax
    mov ax,#0x1301
    int 0x10
load_setup:
    mov dx,#0x0000
    mov cx,#0x0002
    mov bx,#0x0200
    mov ax,#0x0200+SETUPLEN
    int 0x13
    jnc ok_load_setup
    mov dx,#0x0000
    mov ax,#0x0000
    int 0x13
    jmp load_setup
ok_load_setup:
```

```

        jmp     0, SETUPSEG
msg1:
        .byte   13, 10
        .ascii  "Hello OS world, my name is LZJ"
        .byte   13, 10, 13, 10
.org 510
boot_flag:
        .word   0xAA55

```

6.5 再次编译

现在有两个文件都要编译、链接。一个个手工编译，效率低下，所以借助 Makefile 是最佳方式。

在 Ubuntu 下，进入 linux-0.11 目录后，使用下面命令（注意大小写）：

```
$ make BootImage
```

Windows 下，在命令行方式，进入 Linux-0.11 目录后，使用同样的命令（不需注意大小写）：

```
makeBootImage
```

无论哪种系统，都会看到：

```

Unable to open 'system'
make: *** [BootImage] Error 1

```

有 Error! 这是因为 make 根据 Makefile 的指引执行了 tools/build.c，它是为生成整个内核的镜像文件而设计的，没考虑我们只需要 bootsect.s 和 setup.s 的情况。它在向我们要“系统”的核心代码。为完成实验，接下来给它打个小补丁。

6.6 修改 build.c

build.c 从命令行参数得到 bootsect、setup 和 system 内核的文件名，将三者做简单的整理后一起写入 Image。其中 system 是第三个参数 (argv[3])。当 “make all” 或者 “makeall” 的时候，这个参数传过来的是正确的文件名，build.c 会打开它，将内容写入 Image。而 “make BootImage” 时，传过来的是字符串 “none”。所以，改造 build.c 的思路就是当 argv[3] 是 “none” 的时候，只写 bootsect 和 setup，忽略所有与 system 有关的工作，或者在该写 system 的位置都写上 “0”。

修改工作主要集中在 build.c 的尾部，可以参考下面的方式，将圈起来的部分注释掉。

```
shiyanolou@be0229f7b9d2: ~/oslab/linux-0.11
die("Setup exceeds " STRINGIFY(SETUP_SECTS)
    " sectors - rewrite build/boot/setup");
fprintf(stderr,"Setup is %d bytes.\n",i);
for (c=0 ; c<sizeof(buf) ; c++)
    buf[c] = '\0';
while (i<SETUP_SECTS*512) {
    c = SETUP_SECTS*512-i;
    if (c > sizeof(buf))
        c = sizeof(buf);
    if (write(1,buf,c) != c)
        die("Write call failed");
    i += c;
}

// if ((id=open(argv[3],0_RDONLY,0))<0)
//     die("Unable to open 'system'");
// if (read(id,buf,GCC_HEADER) != GCC_HEADER)
//     die("Unable to read header of 'system'");
// if (((long *) buf)[5] != 0)
//     die("Non-GCC header of 'system'");
// for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
//     if (write(1,buf,c)!=c)
//         die("Write call failed");
// close(id);
// fprintf(stderr,"System is %d bytes.\n",i);
// if (i > SYS_SIZE*16)
//     die("System is too big");
return(0);
}

192,1  底端
```

注释掉即可

当按照前一节所讲的编译方法编译成功后再 run，就得到了如图 3 所示的运行结果，和我们想得到的结果完全一样。

```
$ cd ~/oslab/linux-0.11
$ make BootImage
$ ../run
```

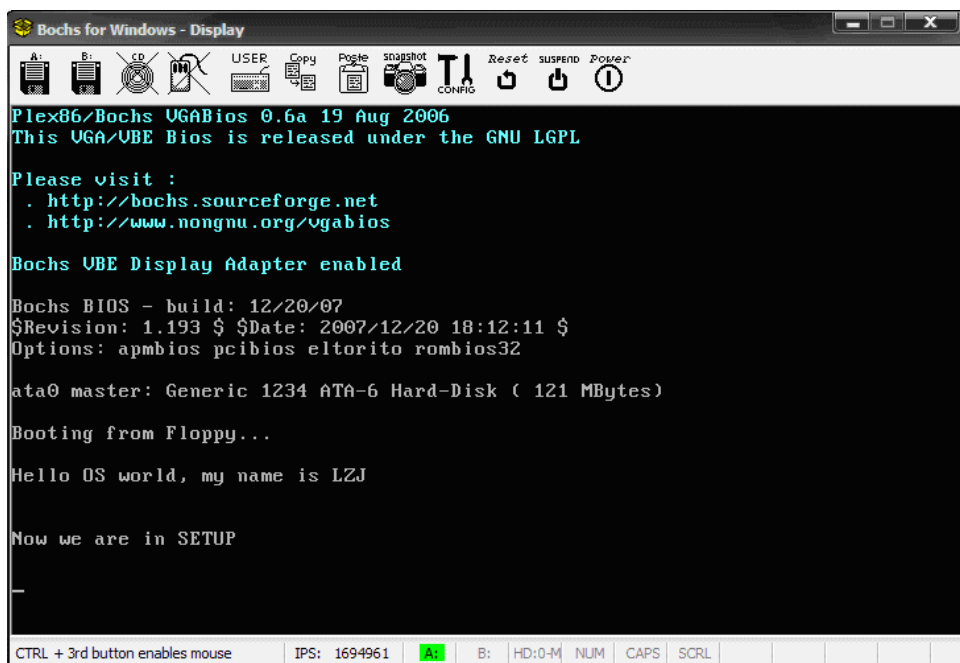


图 3 用修改后的 bootsect.s 和 setup.s 进行引导的结果

6.7 setup.s 获取基本硬件参数

setup.s 将获得硬件参数放在内存的 0x90000 处。原版 setup.s 中已经完成了光标位置、内存大小、显存大小、显卡参数、第一和第二硬盘参数的保存。

用 ah=#0x03 调用 0x10 中断可以读出光标的位置，用 ah=#0x88 调用 0x15 中断可以读出内存的大小。有些硬件参数的获取要稍微复杂一些，如磁盘参数表。在 PC 机中 BIOS 设定的中断向量表中 int 0x41 的中断向量位置 (4*0x41 = 0x0000:0x0104) 存放的并不是中断程序的地址，而是第一个硬盘的基本参数表。第二个硬盘的基本参数表入口地址存于 int 0x46 中断向

量位置处。每个硬盘参数表有 16 个字节大小。下表给出了硬盘基本参数表的内容：

表 1 磁盘基本参数表

位移	大小	说明
0x00	字	柱面数
0x02	字节	磁头数
...
0x0E	字节	每磁道扇区数
0x0F	字节	保留

所以获得磁盘参数的方法就是复制数据。

下面是将硬件参数取出来放在内存 0x90000 的关键代码。

```
mov     ax, #INITSEG
! 设置 ds = 0x9000
mov     ds, ax
mov     ah, #0x03
! 读入光标位置
xor     bh, bh
! 调用 0x10 中断
int     0x10
! 将光标位置写入 0x90000.
mov     [0], dx
! 读入内存大小位置
mov     ah, #0x88
int     0x15
mov     [2], ax
! 从 0x41 处拷贝 16 个字节（磁盘参数表）
mov     ax, #0x0000
mov     ds, ax
lds     si, [4*0x41]
mov     ax, #INITSEG
mov     es, ax
mov     di, #0x0004
mov     cx, #0x10
! 重复16次
rep
movsb
```

6.8 显示获得的参数

现在已经将硬件参数（只包括光标位置、内存大小和硬盘参数，其他硬件参数取出的方法基本相同，此处略去）取出来放在了 0x90000 处，接下来的工作是把这些参数显示在屏幕上。这些参数都是一些无符号整数，所以需要做的的主要工作是用汇编程序在屏幕上将这些整数显示出来。

以十六进制方式显示比较简单。这是因为十六进制与二进制有很好的对应关系（每 4 位二进制数和 1 位十六进制数存在一一对应关系），显示时只需将原二进制数每 4 位划成一组，按组求对应的 ASCII 码送显示器即可。ASCII 码与十六进制数字的对应关系为：0x30 ~ 0x39 对应数字 0 ~ 9，0x41 ~ 0x46 对应数字 a ~ f。从数字 9 到 a，其 ASCII 码间隔了 7h，这一点在转换时要特别注意。为使一个十六进制数能按高位到低位依次显示，实际编程中，需对 bx 中的数每次循环左移一组（4 位二进制），然后屏蔽掉当前高 12 位，对当前余下的 4 位（即 1 位十六进制数）求其 ASCII 码，要判断它是 0 ~ 9 还是 a ~ f，是前者则加 0x30 得对应的 ASCII 码，后者则要加 0x37 才行，最后送显示器输出。以上步骤重复 4 次，就可以完成 bx 中数以 4 位十六进制的形式显示出来。

下面是完成显示 16 进制数的汇编语言程序的关键代码，其中用到的 BIOS 中断为 INT

0x10, 功能号 0x0E (显示一个字符), 即 AH=0x0E, AL=要显示字符的 ASCII 码。

! 以 16 进制方式打印栈顶的16位数

print_hex:

! 4 个十六进制数字

mov cx, #4

! 将 (bp) 所指的值放入 dx 中, 如果 bp 是指向栈顶的话

mov dx, (bp)

print_digit:

! 循环以使低 4 比特用上 !! 取 dx 的高 4 比特移到低 4 比特处。

rol dx, #4

! ah = 请求的功能值, al = 半字节(4 个比特)掩码。

mov ax, #0xe0f

! 取 dl 的低 4 比特值。

and al, dl

! 给 al 数字加上十六进制 0x30

add al, #0x30

cmp al, #0x3a

! 是一个不大于十的数字

jl outp

! 是a~f, 要多加 7

add al, #0x07

outp:

int 0x10

loop print_digit

ret

! 这里用到了一个 loop 指令;

! 每次执行 loop 指令, cx 减 1, 然后判断 cx 是否等于 0。

! 如果不为 0 则转移到 loop 指令后的标号处, 实现循环;

! 如果为0顺序执行。

!

! 另外还有一个非常相似的指令: rep 指令,

! 每次执行 rep 指令, cx 减 1, 然后判断 cx 是否等于 0。

! 如果不为 0 则继续执行 rep 指令后的串操作指令, 直到 cx 为 0, 实现重复。

! 打印回车换行

print_nl:

! CR

mov ax, #0xe0d

int 0x10

! LF

mov al, #0xa

int 0x10

ret

只要在适当的位置调用 print_bx 和 print_nl (注意, 一定要设置好栈, 才能进行函数调用) 就能将获得硬件参数打印到屏幕上, 完成此次实验的任务。但事情往往并不总是顺利的, 前面的两个实验大多数实验者可能一次就编译调试通过了 (这里要提醒大家: 编写操作系统的代码一定要认真, 因为要调试操作系统并不是一件很方便的事)。但在这个实验中会出现运行结果不对的情况 (为什么呢? 因为我们给的代码并不是 100% 好用的)。所以接下来要复习一下汇编, 并阅读《Bochs 使用手册》, 学学在 Bochs 中如何调试操作系统代码。我想经过漫长而痛苦的调试后, 大家一定能兴奋地得到下面的运行结果:


```

Booting from Floppy...

Hello OS world, my name is LZJ

Now we are in SETUP

Cursor POS:1600
Memory SIZE:3C00KB
Cyls:019A
Heads:0010
Sectors:0026

```

图 4 用可以打印硬件参数的 `setup.s` 进行引导的结果

Memory Size 是 0x3C00KB，算一算刚好是 15MB（扩展内存），加上 1MB 正好是 16MB，看看 Bochs 配置文件 `bochs/bochsrc.bxrc`：

```

!.....
megs: 16
!.....
ata0-master: type=disk, mode=flat, cylinders=410, heads=16, spt=38
!.....

```

这些都和上面打出的参数吻合，表示此次实验是成功的。

实验楼的环境中参数可能跟上面给出的不一致。大家需要根据自己环境

中 `bochs/bochsrc.bxrc` 文件中的内容才能确定具体的输出信息。

下面是提供的参考代码，大家可以根据这个来进行编写代码：

```

INITSEG = 0x9000
entry _start
_start:
! Print "NOW we are in SETUP"
    mov ah, #0x03
    xor bh, bh
    int 0x10
    mov cx, #25
    mov bx, #0x0007
    mov bp, #msg2
    mov ax, cs
    mov es, ax
    mov ax, #0x1301
    int 0x10
mov ax, cs
    mov es, ax
! init ss:sp
    mov ax, #INITSEG
    mov ss, ax
    mov sp, #0xFF00
! Get Params
    mov ax, #INITSEG
    mov ds, ax
    mov ah, #0x03
    xor bh, bh
    int 0x10
    mov [0], dx
    mov ah, #0x88
    int 0x15
    mov [2], ax
    mov ax, #0x0000
    mov ds, ax
    lds si, [4*0x41]

```

```

    mov ax, #INITSEG
    mov es, ax
    mov di, #0x0004
    mov cx, #0x10
    rep
    movsb
! Be Ready to Print
    mov ax, cs
    mov es, ax
    mov ax, #INITSEG
    mov ds, ax
! Cursor Position
    mov ah, #0x03
    xor bh, bh
    int 0x10
    mov cx, #18
    mov bx, #0x0007
    mov bp, #msg_cursor
    mov ax, #0x1301
    int 0x10
    mov dx, [0]
    call print_hex
! Memory Size
    mov ah, #0x03
    xor bh, bh
    int 0x10
    mov cx, #14
    mov bx, #0x0007
    mov bp, #msg_memory
    mov ax, #0x1301
    int 0x10
    mov dx, [2]
    call print_hex
! Add KB
    mov ah, #0x03
    xor bh, bh
    int 0x10
    mov cx, #2
    mov bx, #0x0007
    mov bp, #msg_kb
    mov ax, #0x1301
    int 0x10
! Cyles
    mov ah, #0x03
    xor bh, bh
    int 0x10
    mov cx, #7
    mov bx, #0x0007
    mov bp, #msg_cyles
    mov ax, #0x1301
    int 0x10
    mov dx, [4]
    call print_hex
! Heads
    mov ah, #0x03
    xor bh, bh
    int 0x10
    mov cx, #8
    mov bx, #0x0007

```

```

    mov bp, #msg_heads
    mov ax, #0x1301
    int 0x10
    mov dx, [6]
    call print_hex
! Secotrs
    mov ah, #0x03
    xor bh, bh
    int 0x10
    mov cx, #10
    mov bx, #0x0007
    mov bp, #msg_sectors
    mov ax, #0x1301
    int 0x10
    mov dx, [12]
    call print_hex
inf_loop:
    jmp inf_loop
print_hex:
    mov cx, #4
print_digit:
    rol dx, #4
    mov ax, #0xe0f
    and al, dl
    add al, #0x30
    cmp al, #0x3a
    jl outp
    add al, #0x07
outp:
    int 0x10
    loop print_digit
    ret
print_nl:
    mov ax, #0xe0d ! CR
    int 0x10
    mov al, #0xa ! LF
    int 0x10
    ret
msg2:
    .byte 13,10
    .ascii "NOW we are in SETUP"
    .byte 13,10,13,10
msg_cursor:
    .byte 13,10
    .ascii "Cursor position:"
msg_memory:
    .byte 13,10
    .ascii "Memory Size:"
msg_cyles:
    .byte 13,10
    .ascii "Cyls:"
msg_heads:
    .byte 13,10
    .ascii "Heads:"
msg_sectors:
    .byte 13,10
    .ascii "Sectors:"
msg_kb:
    .ascii "KB"

```

```
.org 510  
boot_flag:  
    .word 0xAA55
```

3) 系统调用

2020年9月23日 17:42

实验三 系统调用

系统调用

系统调用

1. 课程说明

本实验是 [操作系统之基础 - 网易云课堂](#) 的配套实验，推荐大家进行实验之前先学习相关课程：

- L4 操作系统接口
- L5 系统调用的实现

Tips: 点击上方文字中的超链接或者输

入 <https://mooc.study.163.com/course/1000002004#/info> 进入理论课程的学习。如果网易云上的课程无法查看，也可以看 Bilibili 上的 [操作系统哈尔滨工业大学李治军老师](#)。

2. 实验目的

- 建立对系统调用接口的深入认识；
- 掌握系统调用的基本过程；
- 能完成系统调用的全面控制；
- 为后续实验做准备。

3. 实验内容

此次实验的基本内容是：在 Linux 0.11 上添加两个系统调用，并编写两个简单的应用程序测试它们。

(1) iam()

第一个系统调用是 iam()，其原型为：

```
int iam(const char * name);
```

完成的功能是将字符串参数 name 的内容拷贝到内核中保存下来。要求 name 的长度不能超过 23 个字符。返回值是拷贝的字符数。如果 name 的字符个数超过了 23，则返回“-1”，并置 errno 为 EINVAL。

在 kernal/who.c 中实现此系统调用。

(2) whoami()

第二个系统调用是 whoami()，其原型为：

```
int whoami(char* name, unsigned int size);
```

它将内核中由 iam() 保存的名字拷贝到 name 指向的用户地址空间中，同时确保不会对 name 越界访存

(name 的大小由 size 说明)。返回值是拷贝的字符数。如果 size 小于需要的空间，则返回“-1”，并置 errno 为 EINVAL。

也是在 kernal/who.c 中实现。

(3) 测试程序

运行添加过新系统调用的 Linux 0.11，在其环境下编写两个测试程序 iam.c 和 whoami.c。最终的运行结果是：

```
$ ./iam lizhijun
$ ./whoami
lizhijun
```

4. 实验报告

在实验报告中回答如下问题：

- 从 Linux 0.11 现在的机制看，它的系统调用最多能传递几个参数？你能想出办法来扩大这个限制吗？
- 用文字简要描述向 Linux 0.11 添加一个系统调用 foo() 的步骤。

5. 评分标准

- 将 testlab2.c（在 /home/teacher 目录下）在修改过的 Linux 0.11 上编译运行，显示的结果即内核程序的得分。满分 50%
- 只要至少一个新增的系统调用被成功调用，并且能和用户空间交换参数，可得满分
- 将脚本 testlab2.sh（在 /home/teacher 目录下）在修改过的 Linux 0.11 上运行，显示的结果即应用程序的得分。满分 30%
- 实验报告，20%

6. 实验提示

首先，请将 Linux 0.11 的源代码恢复到原始状态。

```
# 删除原来的文件
$ cd ~/oslab
$ sudo rm -rf .//*
# 重新拷贝
$ cp -r /home/teacher/oslab/* ./
```

操作系统实现系统调用的基本过程（在 MOOC 课程中已经给出了详细的讲解）是：

- 应用程序调用库函数（API）；
- API 将系统调用号存入 EAX，然后通过中断调用使系统进入内核态；

- 内核中的中断处理函数根据系统调用号，调用对应的内核函数（系统调用）；
- 系统调用完成相应功能，将返回值存入 EAX，返回到中断处理函数；
- 中断处理函数返回到 API 中；
- API 将 EAX 返回给应用程序。

6.1 应用程序如何调用系统调用

在通常情况下，调用系统调用和调用一个普通的自定义函数在代码上并没有什么区别，但调用后发生的事情有很大不同。

调用自定义函数是通过 `call` 指令直接跳转到该函数的地址，继续运行。

而调用系统调用，是调用系统库中为该系统调用编写的一个接口函数，叫 API（Application Programming Interface）。API 并不能完成系统调用的真正功能，它要做的是去调用真正的系统调用，过程是：

- 把系统调用的编号存入 EAX；
- 把函数参数存入其它通用寄存器；
- 触发 0x80 号中断（int 0x80）。

linux-0.11 的 `lib` 目录下有一些已经实现的 API。Linus 编写它们的原因是在内核加载完毕后，会切换到用户模式下，做一些初始化工作，然后启动 shell。而用户模式下的很多工作需要依赖一些系统调用才能完成，因此在内核中实现了这些系统调用的 API。

后面的目录如果没有特殊说明，都是指在 `/home/shiyanlou/oslab/linux-0.11` 中。比如下面的 `lib/close.c`，是指 `/home/shiyanlou/oslab/linux-0.11/lib/close.c`。

我们不妨看看 `lib/close.c`，研究一下 `close()` 的 API：

```
#define __LIBRARY__
#include <unistd.h>
_syscall1(int, close, int, fd)
```

其中 `_syscall1` 是一个宏，在 `include/unistd.h` 中定义。

```
#define _syscall1(type, name, atype, a) \
type name(atype a) \
{ \
long __res; \
```

```

__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name), "b" ((long)(a))); \
if (__res >= 0) \
    return (type) __res; \
errno = -__res; \
return -1; \
}

```

将 `_syscall1(int, close, int, fd)` 进行宏展开，可以得到：

```

int close(int fd)
{
    long __res;
    __asm__ volatile ("int $0x80"
        : "=a" (__res)
        : "0" (__NR_close), "b" ((long)(fd)));
    if (__res >= 0)
        return (int) __res;
    errno = -__res;
    return -1;
}

```

这就是 API 的定义。它先将宏 `__NR_close` 存入 EAX，将参数 `fd` 存入 EBX，然后进行 0x80 中断调用。调用返回后，从 EAX 取出返回值，存入 `__res`，再通过对 `__res` 的判断决定传给 API 的调用者什么样的返回值。

其中 `__NR_close` 就是系统调用的编号，在 `include/unistd.h` 中定义：

```

#define __NR_close 6
/*
所以添加系统调用时需要修改include/unistd.h文件，
使其包含__NR_whoami和__NR_iam。
*/

```

```

/*
而在应用程序中，要有：
*/
/* 有它，_syscall1 等才有效。详见unistd.h */
#define __LIBRARY__

```

```

/* 有它，编译器才能获知自定义的系统调用的编号
*/
#include "unistd.h"
/* iam() 在用户空间的接口函数 */
_syscall1(int, iam, const char*, name);
/* whoami() 在用户空间的接口函数 */
_syscall2(int, whoami, char*, name, unsigned
int, size);

```

在 0.11 环境下编译 C 程序，包含的头文件都在 /usr/include 目录下。

该目录下的 unistd.h 是标准头文件（它和 0.11 源码树中的 unistd.h 并不是同一个文件，虽然内容可能相同），没有 __NR_whoami 和 __NR_iam 两个宏，需要手工加上它们，也可以直接从修改过的 0.11 源码树中拷贝新的 unistd.h 过来。

6.2 从“int 0x80”进入内核函数

int 0x80 触发后，接下来就是内核的中断处理了。先了解一下 0.11 处理 0x80 号中断的过程。

在内核初始化时，主函数（在 init/main.c 中，Linux 实验环境下是 main()，Windows 下因编译器兼容性问题被换名为 start()）调用

了 sched_init() 初始化函数：

```

void main(void)
{
    // .....
    time_init();
    sched_init();
    buffer_init(buffer_memory_end);
    // .....
}

```

sched_init() 在 kernel/sched.c 中定义为：

```

void sched_init(void)
{
    // .....
    set_system_gate(0x80, &system_call);
}

```

set_system_gate 是个宏，

在 include/asm/system.h 中定义为：

```

#define set_system_gate(n, addr) \

```

```
_set_gate(&idt[n], 15, 3, addr)
```

`_set_gate` 的定义是:

```
#define _set_gate(gate_addr, type, dpl, addr) \
__asm__ ("movw %%dx, %%ax\n\t" \
        "movw %0, %%dx\n\t" \
        "movl %%eax, %1\n\t" \
        "movl %%edx, %2" \
        : \
        : "i" ((short) (0x8000 \
+ (dpl<<13) + (type<<8))), \
        "o" (*((char *) (gate_addr))), \
        "o" (*(4+(char *) (gate_addr))), \
        "d" ((char *) (addr)), "a" (0x00080000))
```

虽然看起来挺麻烦,但实际上很简单,就是填写 IDT (中断描述符表),将 `system_call` 函数地址写到 0x80 对应的中断描述符中,也就是在中断 0x80 发生后,自动调用函数 `system_call`。具体细节请参考《注释》的第 4 章。

接下来看 `system_call`。该函数纯汇编打造,定义在 `kernel/system_call.s` 中:

```
!.....
! # 这是系统调用总数。如果增删了系统调用,必须
! 做相应修改
nr_system_calls = 72
!.....
.globl system_call
.align 2
system_call:
! # 检查系统调用编号是否在合法范围内
    cmpl \ $nr_system_calls-1, %eax
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx
! # push %ebx, %ecx, %edx, 是传递给系统调用的参数
    pushl %ebx
! # 让ds, es指向GDT, 内核地址空间
```



```

    movl $0x10,%edx
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx
! # 让fs指向LDT, 用户地址空间
    mov %dx,%fs
    call sys_call_table(,%eax,4)
    pushl %eax
    movl current,%eax
    cmpl $0,state(%eax)
    jne reschedule
    cmpl $0,counter(%eax)
    je reschedule

```

system_call 用 .globl 修饰为其他函数可见。

Windows 实验环境下会看到它有一个下划线前缀，这是不同版本编译器的特质决定的，没有实质区别。

call sys_call_table(,%eax,4) 之前是一些压栈保护，修改段选择子为内核段，call

sys_call_table(,%eax,4) 之后是看看是否需要重新调度，这些都与本实验没有直接关系，此处只关心 call sys_call_table(,%eax,4) 这一句。

根据汇编寻址方法它实际上是：call sys_call_table + 4 * %eax，其中 eax 中放的是系统调用号，即 __NR_XXXXXX。

显然，sys_call_table 一定是一个函数指针数组的起始地址，它定义在 include/linux/sys.h 中：

```

fn_ptr sys_call_table[] = { sys_setup,
sys_exit, sys_fork, sys_read,...

```

增加实验要求的系统调用，需要在这个函数表中增加两个函数引用——sys_iam 和 sys_whoami。当然该函数在 sys_call_table 数组中的位置必须和 __NR_XXXXXX 的值对应上。

同时还要仿照此文件中前面各个系统调用的写法，加上：

```

extern int sys_whoami();
extern int sys_iam();

```

不然，编译会出错的。

6.3 实现 sys_iam() 和 sys_whoami()

添加系统调用的最后一步，是在内核中实现函数 sys_iam() 和 sys_whoami()。

每个系统调用都有一个 `sys_xxxxxx()` 与之对应，它们都是我们学习和模仿的好对象。

比如在 `fs/open.c` 中的 `sys_close(int fd)`:

```
int sys_close(unsigned int fd)
{
    // .....
    return (0);
}
```

它没有什么特别的，都是实实在在地做 `close()` 该做的事情。

所以只要自己创建一个文件：`kernel/who.c`，然后实现两个函数就万事大吉了。

如果完全没有实现的思路，不必担心，本实验的“6.7 在用户态和核心态之间传递数据”还会有提示。

6.4 修改 Makefile

要想让我们添加的 `kernel/who.c` 可以和其它 Linux 代码编译链接到一起，必须要修改 `Makefile` 文件。`Makefile` 里记录的是所有源程序文件的编译、链接规则，《注释》3.6 节有简略介绍。我们之所以简单地运行 `make` 就可以编译整个代码树，是因为 `make` 完全按照 `Makefile` 里的指示工作。

如果想要深入学习 `Makefile`，可以选择实验楼的课程：[《Makefile 基础教程》](#)、[《跟我一起来玩转 Makefile》](#)。

`Makefile` 在代码树中有很多，分别负责不同模块的编译工作。我们要修改的是 `kernel/Makefile`。需要修改两处。

(1) 第一处

```
OBJS = sched.o system_call.o traps.o asm.o
fork.o \
    panic.o printk.o vsprintf.o sys.o
exit.o \
    signal.o mktime.o
```

改为:

```
OBJS = sched.o system_call.o traps.o asm.o
fork.o \
    panic.o printk.o vsprintf.o sys.o
exit.o \
    signal.o mktime.o who.o
```

添加了 who.o。

(2) 第二处

```
### Dependencies:
exit.s exit.o:
exit.c ../include/errno.h ../include/signal.h \
../include/sys/types.h ../include/sys/wait.h
../include/linux/sched.h \
../include/linux/head.h ../include/linux/fs.h
../include/linux/mm.h \
../include/linux/kernel.h ../include/linux/tty.h \
../include/termios.h \
../include/asm/segment.h
```

改为:

```
### Dependencies:
who.s who.o:
who.c ../include/linux/kernel.h ../include/unis
td.h
exit.s exit.o:
exit.c ../include/errno.h ../include/signal.h \
../include/sys/types.h ../include/sys/wait.h
../include/linux/sched.h \
../include/linux/head.h ../include/linux/fs.h
../include/linux/mm.h \
../include/linux/kernel.h ../include/linux/tty.h \
../include/termios.h \
../include/asm/segment.h
```

添加了 who.s who.o:

```
who.c ../include/linux/kernel.h ../include/unis
td.h。
```

Makefile 修改后, 和往常一样 make all 就能自动把 who.c 加入到内核中了。

如果编译时提示 who.c 有错误, 就说明修改生效了。所以, 有意或无意地制造一两个错误也不完全是坏事, 至少能证明 Makefile 是对的。

6.5 用 printk() 调试内核

oslab 实验环境提供了基于 C 语言和汇编语言的两种调试手段。除此之外, 适当地向屏幕输出一些程序运行状态的信息, 也是一种很高效、便捷的调试方法, 有时甚至是唯一的方法, 被称为“printf 法”。

要知道到，`printf()` 是一个只能在用户模式下执行的函数，而系统调用是在内核模式中运行，所以 `printf()` 不可用，要用 `printk()`。
`printk()` 和 `printf()` 的接口和功能基本相同，只是代码上有一点点不同。`printk()` 需要特别处理一下 `fs` 寄存器，它是专用于用户模式的段寄存器。看一看 `printk` 的代码（在 `kernel/printk.c` 中）就知道了：

```
int printk(const char *fmt, ...)
{
    // .....
    __asm__ ("push %%fs\n\t"
            "push %%ds\n\t"
            "pop %%fs\n\t"
            "pushl %0\n\t"
            "pushl $buf\n\t"
            "pushl $0\n\t"
            "call tty_write\n\t"
            "addl $8,%%esp\n\t"
            "popl %0\n\t"
            "pop %%fs"
            :: "r" (i) : "ax", "cx", "dx");
    // .....
}
```

显然，`printk()` 首先 `push %fs` 保存这个指向用户段的寄存器，在最后 `pop %fs` 将其恢复，`printk()` 的核心仍然是调用 `tty_write()`。查看 `printf()` 可以看到，它最终也要落实到这个函数上。

6.6 编写测试程序

激动地运行一下由你亲手修改过的 “Linux 0.11 pro++”！然后编写一个简单的应用程序进行测试。比如在 `sys_iam()` 中向终端 `printk()` 一些信息，让应用程序调用 `iam()`，从结果可以看出系统调用是否被真的调用到了。

可以直接在 Linux 0.11 环境下用 `vi` 编写（别忘了经常执行 “`sync`” 以确保内存缓冲区的数据写入磁盘），也可以在 Ubuntu 或 Windows 下编完后再传到 Linux 0.11 下。无论如何，最终都必须在 Linux 0.11 下编译。编译命令是：

```
$ gcc -o iam iam.c -Wall
```

gcc 的 “-Wall” 参数是给出所有的编译警告信息，“-o” 参数指定生成的执行文件名是 iam，用下面命令运行它：

```
$ ./iam
```

如果如愿输出了你的信息，就说明你添加的系统调用生效了。否则，就还要继续调试，祝你好运！

6.7 在用户态和核心态之间传递数据

指针参数传递的是应用程序所在地址空间的逻辑地址，在内核中如果直接访问这个地址，访问到的是内核空间中的数据，不会为用户空间的。所以这里还需要一点儿特殊工作，才能在内核中从用户空间得到数据。

要实现的两个系统调用参数中都有字符串指针，非常像 `open(char *filename, ...)`，所以我们看一下 `open()` 系统调用是如何处理的。

```
int open(const char * filename, int flag, ...)
{
    // .....
    __asm__ ("int $0x80"
            : "=a" (res)
            : "0" (__NR_open), "b" (filename), "c"
            (flag), "d" (va_arg(arg, int)));
    // .....
}
```

可以看出，系统调用是用 `eax`、`ebx`、`ecx`、`edx` 寄存器来传递参数的。

- 其中 `eax` 传递了系统调用号，而 `ebx`、`ecx`、`edx` 是用来传递函数的参数的
- `ebx` 对应第一个参数，`ecx` 对应第二个参数，依此类推。

如 `open` 所传递的文件名指针是由 `ebx` 传递的，也即进入内核后，通过 `ebx` 取出文件名字符串。`open` 的 `ebx` 指向的数据在用户空间，而当前执行的是内核空间的代码，如何在用户态和核心态之间传递数据？

接下来我们继续看看 `open` 的处理：

```
system_call: //所有的系统调用都从system_call开始
! .....
    pushl %edx
```

```

    pushl %ecx
    pushl %ebx                # push %ebx, %
ecx, %edx, 这是传递给系统调用的参数
    movl $0x10, %edx         # 让ds, es指向
GDT, 指向核心地址空间
    mov %dx, %ds
    mov %dx, %es
    movl $0x17, %edx        # 让fs指向的是
LDT, 指向用户地址空间
    mov %dx, %fs
    call sys_call_table(, %eax, 4)    # 即call
sys_open

```

由上面的代码可以看出，获取用户地址空间（用户数据段）中的数据依靠的就是段寄存器 fs，下面该转到 sys_open 执行了，在 fs/open.c 文件中：

```

int sys_open(const char * filename, int flag, int
mode) //filename这些参数从哪里来?
/*是否记得上面的pushl %edx,    pushl %ecx,
pushl %ebx?
    实际上一个C语言函数调用另一个C语言函数时，编
译时就是将要
    传递的参数压入栈中（第一个参数最后压，...），
然后call ...，
    所以汇编程序调用C函数时，需要自己编写这些参数
压栈的代码...*/
{
    .....
    if ((i=open_namei(filename, flag, mode,
&inode))<0) {
        .....
    }
    .....
}

```

它将参数传给了 open_namei()。
 再沿着 open_namei() 继续查找，文件名先后又被传给 dir_namei()、get_dir()。
 在 get_dir() 中可以看到：

```

static struct m_inode * get_dir(const char *
pathname)
{

```

```

.....
    if ((c=get_fs_byte(pathname))== '/') {
        .....
    }
    .....
}

```

处理方法就很显然了：用 `get_fs_byte()` 获得一个字节的用户空间中的数据。

所以，在实现 `iam()` 时，调用 `get_fs_byte()` 即可。

但如何实现 `whoami()` 呢？即如何实现从核心态拷贝数据到用户态内存空间中呢？

猜一猜，是否有 `put_fs_byte()`？有！看一看 `include/asm/segment.h`：

```

extern inline unsigned char get_fs_byte(const
char * addr)
{
    unsigned register char _v;
    __asm__ ("movb %%fs:%1,%0"::"r" (_v):"m"
(*addr));
    return _v;
}

```

```

extern inline void put_fs_byte(char val, char
*addr)
{
    __asm__ ("movb %0,%%fs:%1"::"r" (val),"m"
(*addr));
}

```

他俩以及所有 `put_fs_xxx()` 和 `get_fs_xxx()` 都是用户空间和内核空间之间的桥梁，在后面的实验中还要经常用到。

6.8 运行脚本程序

Linux 的一大特色是可以编写功能强大的 shell 脚本，提高工作效率。本实验的部分评分工作由脚本 `testlab2.sh` 完成。它的功能是测试

`iam.c` 和 `whoami.c`。

首先将 `iam.c` 和 `whoami.c` 分别编译

成 `iam` 和 `whoami`，然后

将 `testlab2.sh`（在 `/home/teacher` 目录下）拷贝

到同一目录下。

用下面命令为此脚本增加执行权限：

```
$ chmod +x testlab2.sh
```

然后运行之：

```
$ ./testlab2.sh
```

根据输出，可知 iam.c 和 whoami.c 的得分。

errno

errno 是一种传统的错误代码返回机制。

当一个函数调用出错时，通常会返回 -1 给调用者。

但 -1 只能说明出错，不能说明错是什么。为解决此问题，全局变量 errno 登场了。错误值被存放到 errno 中，于是调用者就可以通过判断 errno 来决定如何应对错误了。

各种系统对 errno 的值的含义都有标准定义。Linux 下用 “man errno” 可以看到这些定义

4) 进程运行轨迹的跟踪与统计

2020年9月27日 14:07

进程运行轨迹的跟踪与统计

进程运行轨迹的跟踪与统计

1. 课程说明

难度系数：★★★☆☆

本实验是 [操作系统之进程与线程 - 网易云课堂](#) 的配套实验，推荐大家进行实验之前先学习相关课程：

- L8 CPU 管理的直观想法
- L9 多进程图像

Tips: 点击上方文字中的超链接或者输

入 <https://mooc.study.163.com/course/1000002008#/info> 进入理论课程的学习。如果网易云上的课程无法查看，也可以看 Bilibili 上的 [操作系统哈尔滨工业大学李治军老师](#)。

2. 实验目的

- 掌握 Linux 下的多进程编程技术；
- 通过对进程运行轨迹的跟踪来形象化进程的概念；
- 在进程运行轨迹跟踪的基础上进行相应的数据统计，从而能对进程调度算法进行实际的量化评价，更进一步加深对调度和调度算法的理解，获得能在实际操作系统上对调度算法进行实验数据对比的直接经验。

3. 实验内容

进程从创建（Linux 下调用 `fork()`）到结束的整个过程就是进程的生命期，进程在其生命期中的运行轨迹实际上就表现为进程状态的多次切换，如进程创建以后会成为就绪态；当该进程被调度以后会切换到运行态；在运行的过程中如果启动了一个文件读写操作，操作系统会将该进程切换到阻塞态（等待态）从而让出 CPU；当文件读写完毕以后，操作系统会在将其切换成就绪态，等待进程调度算法来调度该进程执行……

本次实验包括如下内容：

- 基于模板 `process.c` 编写多进程的样本程序，实现如下功能：+ 所有子进程都并行运行，每个子进程的实际运行时间一般不超过 30 秒；+ 父进程向标准输出打印所有子进程的 `id`，并在所有子进程都退出后才退出；
- 在 Linux0.11 上实现进程运行轨迹的跟踪。+ 基本任务是在内核中维护一个日志文件 `/var/process.log`，把从操作系统启动到系统关机过程中所有进程的运行轨迹都记录在这一 `log` 文件中。
- 在修改过的 0.11 上运行样本程序，通过分析 `log` 文件，统计该程序建立的所有进程的等待时间、完成时间（周转时间）和运行时间，然后计算平均等待时间，平均完成时间和吞吐量。可以自己编写统计程序，也可以使用 python 脚本程序——`stat_log.py`（在 `/home/teacher/` 目录下）——进行统计。
- 修改 0.11 进程调度的时间片，然后再运行同样的样本程序，统计同样的时间数据，和原有的情况对比，体会不同时间片带来的差异。

`/var/process.log` 文件的格式必须为：

```
pid    X    time
```

其中：

- pid 是进程的 ID；
- X 可以是 N、J、R、W 和 E 中的任意一个，分别表示进程新建(N)、进入就绪态(J)、进入运行态(R)、进入阻塞态(W) 和退出(E)；
- time 表示 X 发生的时间。这个时间不是物理时间，而是系统的滴答时间(tick)；

三个字段之间用制表符分隔。例如：

```
12  N   1056
12  J   1057
4   W   1057
12  R   1057
13  N   1058
13  J   1059
14  N   1059
14  J   1060
15  N   1060
15  J   1061
12  W   1061
15  R   1061
15  J   1076
14  R   1076
14  E   1076
.....
```

4. 实验报告

完成实验后，在实验报告中回答如下问题：

- 结合自己的体会，谈谈从程序设计者的角度看，单进程编程和多进程编程最大的区别是什么？
- 你是如何修改时间片的？仅针对样本程序建立的进程，在修改时间片前后，log 文件的统计结果（不包括 Graphic）都是什么样？结合你的修改分析一下为什么会这样变化，或者为什么没变化？

5. 评分标准

- process.c, 50%
- 日志文件建立成功, 5%
- 能向日志文件输出信息, 5%
- 5 种状态都能输出, 10%（每种 2 %）
- 调度算法修改, 10%
- 实验报告, 20%

6. 实验提示

process.c 的编写涉及到 fork() 和 wait() 系统调用，请自行查阅相关文献。

0.11 内核修改涉及到 init/main.c、kernel/fork.c 和 kernel/sched.c，开始实验前如果能详细阅读《注释》一书的相关部分，会大有裨益。

6.1 编写样本程序

所谓样本程序，就是一个生成各种进程的程序。我们的对 0.11 的修改把系统对它们的调度情况都记录到 log 文件中。在修改调度算法或调度参数后再运行完全一样的样本程序，可以检验调度算法的优劣。

理论上，此程序可以在任何 Unix/Linux 上运行，所以建议在 Ubuntu 上调试通过后，再拷贝到 0.11 下运行。

process.c 是样本程序的模板（在 /home/teacher/ 目录下）。

它主要实现了一个函数：

```
/*
 * 此函数按照参数占用CPU和I/O时间
 * last: 函数实际占用CPU和I/O的总时间，不含在就绪队列中的时间，>=0是必须的
```

```

* cpu_time: 一次连续占用CPU的时间, >=0是必须的
* io_time: 一次I/O消耗的时间, >=0是必须的
* 如果last > cpu_time + io_time, 则往复多次占用CPU和I/O, 直到总运行时间超过last为止
* 所有时间的单位为秒
*/
cpuio_bound(int last, int cpu_time, int io_time);

```

下面是 4 个使用的例子:

```

// 比如一个进程如果要占用10秒的CPU时间, 它可以调用:
cpuio_bound(10, 1, 0);
// 只要cpu_time>0, io_time=0, 效果相同

```

```

// 以I/O为主要任务:
cpuio_bound(10, 0, 1);
// 只要cpu_time=0, io_time>0, 效果相同

```

```

// CPU和I/O各1秒钟轮回:
cpuio_bound(10, 1, 1);

```

```

// 较多的I/O, 较少的CPU:
// I/O时间是CPU时间的9倍
cpuio_bound(10, 1, 9);

```

修改此模板, 用 `fork()` 建立若干个同时运行的子进程, 父进程等待所有子进程退出后才退出, 每个子进程按照你的意愿做不同或相同的 `cpuio_bound()`, 从而完成一个个性化的样本程序。

它可以用来检验有关 `log` 文件的修改是否正确, 同时还是数据统计工作的基础。

`wait()` 系统调用可以让父进程等待子进程的退出。

小技巧:

在 Ubuntu 下, `top` 命令可以监视即时的进程状态。在 `top` 中, 按 `u`, 再输入你的用户名, 可以限定只显示以你的身份运行的进程, 更方便观察。按 `h` 可得到帮助。

在 Ubuntu 下, `ps` 命令可以显示当时各个进程的状态。`ps aux` 会显示所有进程; `ps aux | grep xxxx` 将只显示名为 `xxxx` 的进程。更详细的用法请问 `man`。

在 Linux 0.11 下, 按 `F1` 可以即时显示当前所有进程的状态。

6.2 log 文件

操作系统启动后先要打开 `/var/process.log`, 然后在每个进程发生状态切换的时候向 `log` 文件内写入一条记录, 其过程和用户态的应用程序没什么两样。然而, 因为内核状态的存在, 使过程中的很多细节变得完全不一样。

打开 `log` 文件

为了能尽早开始记录, 应当在内核启动时就打开 `log` 文件。内核的入口是 `init/main.c` 中的 `main()` (Windows 环境下是 `start()`), 其中一段代码是:

```

//.....
move_to_user_mode();
if (!fork()) { /* we count on this going ok */
    init();
}
//.....

```

这段代码在进程 0 中运行, 先切换到用户模式, 然后全系统第一次调用 `fork()` 建立进程 1。进程 1 调用 `init()`。

在 `init()` 中:

```
// .....  
//加载文件系统  
setup((void *) &drive_info);  
// 打开/dev/tty0, 建立文件描述符0和/dev/tty0的关联  
(void) open("/dev/tty0", O_RDWR, 0);  
// 让文件描述符1也和/dev/tty0关联  
(void) dup(0);  
// 让文件描述符2也和/dev/tty0关联  
(void) dup(0);  
// .....
```

这段代码建立了文件描述符 0、1 和 2, 它们分别就是 `stdin`、`stdout` 和 `stderr`。这三者的值是系统标准 (Windows 也是如此), 不可改变。

可以把 `log` 文件的描述符关联到 3。文件系统初始化, 描述符 0、1 和 2 关联之后, 才能打开 `log` 文件, 开始记录进程的运行轨迹。

为了能尽早访问 `log` 文件, 我们要让上述工作在进程 0 中就完成。所以把这一段代码从 `init()` 移动到 `main()` 中, 放在 `move_to_user_mode()` 之后 (不能再靠前了), 同时加上打开 `log` 文件的代码。

修改后的 `main()` 如下:

```
//.....  
move_to_user_mode();  
/*****添加开始*****/  
setup((void *) &drive_info);  
// 建立文件描述符0和/dev/tty0的关联  
(void) open("/dev/tty0", O_RDWR, 0);  
//文件描述符1也和/dev/tty0关联  
(void) dup(0);  
// 文件描述符2也和/dev/tty0关联  
(void) dup(0);  
(void) open("/var/process.log", O_CREAT|O_TRUNC|O_WRONLY, 0666);  
/*****添加结束*****/  
if (!fork()) { /* we count on this going ok */  
    init();  
}  
//.....
```

打开 `log` 文件的参数的含义是建立只写文件, 如果文件已存在则清空已有内容。文件的权限是所有人可读可写。

这样, 文件描述符 0、1、2 和 3 就在进程 0 中建立了。根据 `fork()` 的原理, 进程 1 会继承这些文件描述符, 所以 `init()` 中就不必再 `open()` 它们。此后所有新建的进程都是进程 1 的子孙, 也会继承它们。但实际上, `init()` 的后续代码和 `/bin/sh` 都会重新初始化它们。所以只有进程 0 和进程 1 的文件描述符肯定关联着 `log` 文件, 这一点在接下来的写 `log` 中很重要。

6.3 写 `log` 文件

`log` 文件将被用来记录进程的状态转移轨迹。所有的状态转移都是在内核进行的。

在内核状态下, `write()` 功能失效, 其原理等同于《系统调用》实验中不能在内核状态调用 `printf()`, 只能调用 `printk()`。编写可在内核调用的 `write()` 的难度较大, 所以这里直接给出源码。它主要参考了 `printk()` 和 `sys_write()` 而写成的:

```
#include "linux/sched.h"  
#include "sys/stat.h"  
static char logbuf[1024];  
int fprintk(int fd, const char *fmt, ...)  
{
```

```

    va_list args;
    int count;
    struct file * file;
    struct m_inode * inode;
va_start(args, fmt);
    count=vsprintf(logbuf, fmt, args);
    va_end(args);
/* 如果输出到stdout或stderr, 直接调用sys_write即可 */
    if (fd < 3)
    {
        __asm__ ("push %%fs\n\t"
                "push %%ds\n\t"
                "pop %%fs\n\t"
                "pushl %0\n\t"
                /* 注意对于Windows环境来说, 是_logbuf, 下同 */
                "pushl $logbuf\n\t"
                "pushl %1\n\t"
                /* 注意对于Windows环境来说, 是_sys_write, 下同 */
                "call sys_write\n\t"
                "addl $8,%%esp\n\t"
                "popl %0\n\t"
                "pop %%fs"
                :: "r" (count), "r" (fd): "ax", "cx", "dx");
    }
    else
/* 假定>=3的描述符都与文件关联。事实上, 还存在很多其它情况, 这里并没有考
虑。*/
    {
        /* 从进程0的文件描述符表中得到文件句柄 */
        if (!(file=task[0]->filp[fd]))
            return 0;
        inode=file->f_inode;
asm ("push %%fs\n\t"
    "push %%ds\n\t"
    "pop %%fs\n\t"
    "pushl %0\n\t"
    "pushl $logbuf\n\t"
    "pushl %1\n\t"
    "pushl %2\n\t"
    "call file_write\n\t"
    "addl $12,%%esp\n\t"
    "popl %0\n\t"
    "pop %%fs"
    :: "r" (count), "r" (file), "r" (inode): "ax", "cx", "dx");
    }
    return count;
}

```

因为和 printk 的功能近似, 建议将此函数放入到 kernel/printk.c 中。fprintk() 的使用方式类同与 C 标准库函数 fprintf(), 唯一的区别是第一个参数是文件描述符, 而不是文件指针。例如:

```

// 向stdout打印正在运行的进程的ID
fprintk(1, "The ID of running process is %ld", current->pid);
// 向log文件输出跟踪进程运行轨迹
fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'R', jiffies);

```

6.4 jiffies, 滴答

jiffies 在 kernel/sched.c 文件中定义为一个全局变量:

```
long volatile jiffies=0;
```

它记录了从开机到当前时间的时钟中断发生次数。在 kernel/sched.c 文件中的 sched_init() 函数中, 时钟中断处理函数被设置为:

```
set_intr_gate(0x20,&timer_interrupt);
```

而在 kernel/system_call.s 文件中将 timer_interrupt 定义为:

```
timer_interrupt:
! .....
! 增加jiffies计数值
    incl jiffies
! .....
```

这说明 jiffies 表示从开机时到现在发生的时钟中断次数, 这个数也被称为“滴答数”。

另外, 在 kernel/sched.c 中的 sched_init() 中有下面的代码:

```
// 设置8253模式
outb_p(0x36, 0x43);
outb_p(LATCH&0xff, 0x40);
outb_p(LATCH>>8, 0x40);
```

这三条语句用来设置每次时钟中断的间隔, 即为 LATCH, 而 LATCH 是定义在文件 kernel/sched.c 中的一个宏:

```
// 在 kernel/sched.c 中
#define LATCH (1193180/HZ)
// 在 include/linux/sched.h 中
#define HZ 100
```

再加上 PC 机 8253 定时芯片的输入时钟频率为 1.193180MHz, 即 1193180/每秒, LATCH=1193180/100, 时钟每跳 11931.8 下产生一次时钟中断, 即每 1/100 秒 (10ms) 产生一次时钟中断, 所以 jiffies 实际上记录了从开机以来共经过了多少个 10ms。

6.5 寻找状态切换点

必须找到所有发生进程状态切换的代码点, 并在这些点添加适当的代码, 来输出进程状态变化的情况到 log 文件中。

此处要面对的情况比较复杂, 需要对 kernel 下的 fork.c、sched.c 有通盘的了解, 而 exit.c 也会涉及到。

我们给出两个例子描述这个工作该如何做, 其他情况实验者可仿照完成。

(1) 例子 1: 记录一个进程生命期的开始

第一个例子是看看如何记录一个进程生命期的开始, 当然这个事件就是进程的创建函数 fork(), 由《系统调用》实验可知, fork() 功能在内核中实现为 sys_fork(), 该“函数”在文件 kernel/system_call.s 中实现为:

```
sys_fork:
    call find_empty_process
! .....
! 传递一些参数
    push %gs
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
! 调用 copy_process 实现进程创建
    call copy_process
    addl $20,%esp
```

所以真正实现进程创建的函数是 `copy_process()`，它在 `kernel/fork.c` 中定义为：

```
int copy_process(int nr,.....)
{
    struct task_struct *p;
    // .....
    // 获得一个 task_struct 结构体空间
    p = (struct task_struct *) get_free_page();
    // .....
    p->pid = last_pid;
    // .....
    // 设置 start_time 为 jiffies
    p->start_time = jiffies;
    // .....
    /* 设置进程状态为就绪。所有就绪进程的状态都是
    TASK_RUNNING(0)，被全局变量 current 指向的
    是正在运行的进程。*/
    p->state = TASK_RUNNING;
    return last_pid;
}
```

因此要完成进程运行轨迹的记录就要在 `copy_process()` 中添加输出语句。这里要输出两种状态，分别是“N（新建）”和“J（就绪）”。

（2）例子 2：记录进入睡眠态的时间

第二个例子是记录进入睡眠态的时间。`sleep_on()` 和 `interruptible_sleep_on()` 让当前进程进入睡眠状态，这两个函数在 `kernel/sched.c` 文件中定义如下：

```
void sleep_on(struct task_struct **p)
{
    struct task_struct *tmp;
    // .....
    tmp = *p;
    // 仔细阅读，实际上是将 current 插入“等待队列”头部，tmp 是原来的头部
    *p = current;
    // 切换到睡眠态
    current->state = TASK_UNINTERRUPTIBLE;
    // 让出 CPU
    schedule();
    // 唤醒队列中的上一个（tmp）睡眠进程。0 换作 TASK_RUNNING 更好
    // 在记录进程被唤醒时一定要考虑到这种情况，实验者一定要注意!!!
    if (tmp)
        tmp->state=0;
}
```

```
/* TASK_UNINTERRUPTIBLE和TASK_INTERRUPTIBLE的区别在于不可中断的睡眠
 * 只能由wake_up()显式唤醒，再由上面的 schedule()语句后的
 *
 * if (tmp) tmp->state=0;
 *
 * 依次唤醒，所以不可中断的睡眠进程一定是按严格从“队列”（一个依靠
 * 放在进程内核栈中的指针变量tmp维护的队列）的首部进行唤醒。而对于可
 * 中断的进程，除了用wake_up唤醒以外，也可以用信号（给进程发送一个信
 * 号，实际上就是将进程PCB中维护的一个向量的某一位置位，进程需要在合
 * 适的时候处理这一位。感兴趣的实验者可以阅读有关代码）来唤醒，如在
 * schedule()中：
 *
 * for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
 *     if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
```

```

*      (*p)->state==TASK_INTERRUPTIBLE)
*      (*p)->state=TASK_RUNNING;//唤醒
*
* 就是当进程是可中断睡眠时，如果遇到一些信号就将其唤醒。这样的唤醒会
* 出现一个问题，那就是可能会唤醒等待队列中间的某个进程，此时这个链就
* 需要进行适当调整。interruptible_sleep_on和sleep_on函数的主要区别就
* 在这里。
*/
void interruptible_sleep_on(struct task_struct **p)
{
    struct task_struct *tmp;
    ...
    tmp=*p;
    *p=current;
repeat:    current->state = TASK_INTERRUPTIBLE;
    schedule();
// 如果队列头进程和刚唤醒的进程 current 不是一个，
// 说明从队列中间唤醒了一个进程，需要处理
    if (*p && *p != current) {
// 将队列头唤醒，并通过 goto repeat 让自己再去睡眠
        (*p).state=0;
        goto repeat;
    }
    *p=NULL;
//作用和 sleep_on 函数中的一样
    if (tmp)
        tmp->state=0;
}

```

相信实验者已经找到合适的地方插入记录进程从运行到睡眠的语句了。

总的来说，Linux 0.11 支持四种进程状态的转移：就绪到运行、运行到就绪、运行到睡眠和睡眠到就绪，此外还有新建和退出两种情况。其中就绪与运行间的状态转移是通过 `schedule()`（它亦是调度算法所在）完成的；运行到睡眠依靠的是 `sleep_on()` 和 `interruptible_sleep_on()`，还有进程主动睡觉的系统调用 `sys_pause()` 和 `sys_waitpid()`；睡眠到就绪的转移依靠的是 `wake_up()`。所以只要在这些函数的适当位置插入适当的处理语句就能完成进程运行轨迹的全面跟踪了。

- 为了让生成的 log 文件更精准，以下几点请注意：
- 进程退出的最后一步是通知父进程自己的退出，目的是唤醒正在等待此事件的父进程。从时序上来说，应该是子进程先退出，父进程才醒来。
- `schedule()` 找到的 next 进程是接下来要运行的进程（注意，一定要分析清楚 next 是什么）。如果 next 恰好是当前正处于运行态的进程，`swtch_to(next)` 也会被调用。这种情况下相当于当前进程的状态没变。
- 系统无事可做的时候，进程 0 会不停地调用 `sys_pause()`，以激活调度算法。此时它的状态可以是等待态，等待有其它可运行的进程；也可以叫运行态，因为它是唯一一个在 CPU 上运行的进程，只不过运行的效果是等待。

6.6 管理 log 文件

日志文件的管理与代码编写无关，有几个要点要注意：

- 每次关闭 bochs 前都要执行一下 `sync` 命令，它会刷新 cache，确保文件确实写入了磁盘。
- 在 0.11 下，可以用 `ls -l /var` 或 `ll /var` 查看 `process.log` 是否建立，及它的属性和长度。
- 一定要实践《实验环境的搭建与使用》一章中关于文件交换的部分。最终肯定要把 `process.log` 文件拷贝到主机环境下处理。
- 在 0.11 下，可以用 `vi /var/process.log` 或 `more /var/process.log` 查

看整个 log 文件。不过，还是拷贝到 Ubuntu 下看，会更舒服。

- 在 0.11 下，可以用 `tail -n NUM /var/process.log` 查看 log 文件的最后 NUM 行。

一种可能的情况下，得到的 process.log 文件的前几行是：

```
1  N   48   //进程1新建（init()）。此前是进程0建立和运行，但为什么没出
现在log文件里？
1  J   49   //新建后进入就绪队列
0  J   49   //进程0从运行->就绪，让出CPU
1  R   49   //进程1运行
2  N   49   //进程1建立进程2。2会运行/etc/rc脚本，然后退出
2  J   49
1  W   49   //进程1开始等待（等待进程2退出）
2  R   49   //进程2运行
3  N   64   //进程2建立进程3。3是/bin/sh建立的运行脚本的子进程
3  J   64
2  E   68   //进程2不等进程3退出，就先走一步了
1  J   68   //进程1此前在等待进程2退出，被阻塞。进程2退出后，重新进入就
绪队列
1  R   68
4  N   69   //进程1建立进程4，即shell
4  J   69
1  W   69   //进程1等待shell退出（除非执行exit命令，否则shell不会退出）
3  R   69   //进程3开始运行
3  W   75
4  R   75
5  N   107  //进程5是shell建立的不知道做什么的进程
5  J   108
4  W   108
5  R   108
4  J   110
5  E   111  //进程5很快退出
4  R   111
4  W   116  //shell等待用户输入命令。
0  R   116  //因为无事可做，所以进程0重出江湖
4  J   239  //用户输入命令了，唤醒了shell
4  R   239
4  W   240
0  R   240
.....
```

6.7 数据统计

为展示实验结果，需要编写一个数据统计程序，它从 log 文件读入原始数据，然后计算平均周转时间、平均等待时间和吞吐率。

任何语言都可以编写这样的程序，实验者可自行设计。我们用 python 语言编写了一个——stat_log.py（这是 python 源程序，可以用任意文本编辑器打开）。

python 是一种跨平台的脚本语言，号称“可执行的伪代码”，非常强大，非常好用，也非常有用，建议闲着的时候学习一下。

其解释器免费且开源，Ubuntu 下这样安装：

```
# 在实验楼的环境中已经安装了 python，可以不必进行此操作
$ sudo apt-get install python
```

然后只要给 stat_log.py 加上执行权限（使用的命令为 `chmod +x stat_log.py`）就可以直接运行它。

此程序必须在命令行下加参数执行，直接运行会打印使用说明。

```
Usage:
./stat_log.py /path/to/process.log [PID1] [PID2] ... [-x PID1
```

```
[PID2] ... ] [-m] [-g]
Example:
# Include process 6, 7, 8 and 9 in statistics only. (Unit: tick)
./stat_log.py /path/to/process.log 6 7 8 9
# Exclude process 0 and 1 from statistics. (Unit: tick)
./stat_log.py /path/to/process.log -x 0 1
# Include process 6 and 7 only. (Unit: millisecond)
./stat_log.py /path/to/process.log 6 7 -m
# Include all processes and print a COOL "graphic"! (Unit: tick)
./stat_log.py /path/to/process.log -g
```

运行 `./stat_log.py process.log 0 1 2 3 4 5 -g` (只统计 PID 为 0、1、2、3、4 和 5 的进程) 的输出示例:

```
(Unit: tick)
Process   Turnaround   Waiting   CPU Burst   I/O Burst
  0         75         67         8           0
  1       2518         0         1       2517
  2         25         4        21           0
  3       3003         0         4       2999
  4       5317         6        51       5260
  5          3         0         3           0
Average:   1823.50    12.83
Throughout: 0.11/s
-----< COOL GRAPHIC OF SCHEDULER >-----
[Symbol]   [Meaning]
~~~~~
      number  PID or tick
      "-"     New or Exit
      "#"     Running
      "|"     Ready
      ":"     Waiting
      "/"     / Running with
      "+-|"   Ready
              \and/or Waiting
-----< !!!!!!!!!!!!!!!!!!!!!!!!!!!!! >-----
40 -0
41 #0
42 #
43 #
44 #
45 #
46 #
47 #
48 |0 -1
49 | :1 -2
50 | : #2
51 | : #
52 | : #
53 | : #
54 | : #
55 | : #
56 | : #
57 | : #
58 | : #
59 | : #
60 | : #
61 | : #
62 | : #
63 | : #
```

```

64 | : | 2 -3
65 | : | #3
66 | : | #
67 | : | #
.....

```

小技巧：如果命令程序输出过多，可以用 `command arguments | more`（`command arguments` 需要替换为脚本执行的命令）的方式运行，结果会一屏一屏地显示。

“more” 在 Linux 和 Windows 下都有。Linux 下还有一个 “less”，和 “more” 类似，但功能更强，可以上下翻页、搜索。

6.8 修改时间片

下面是 0.11 的调度函数 `schedule`，在文件 `kernel/sched.c` 中定义为：

```

while (1) {
    c = -1; next = 0; i = NR_TASKS; p = &task[NR_TASKS];
    // 找到 counter 值最大的就绪态进程
    while (--i) {
        if (!*--p) continue;
        if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
            c = (*p)->counter, next = i;
    }
    // 如果有 counter 值大于 0 的就绪态进程，则退出
    if (c) break;
    // 如果没有：
    // 所有进程的 counter 值除以 2 衰减后再和 priority 值相加，
    // 产生新的时间片
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
        if (*p) (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
    // 切换到 next 进程
    switch_to(next);
}

```

分析代码可知，0.11 的调度算法是选取 `counter` 值最大的就绪进程进行调度。

其中运行态进程（即 `current`）的 `counter` 数值会随着时钟中断而不断减 1（时钟中断 10ms 一次），所以是一种比较典型的时间片轮转调度算法。

另外，由上面的程序可以看出，当没有 `counter` 值大于 0 的就绪进程时，要对所有的进程做 `(*p)->counter = ((*p)->counter >> 1) + (*p)->priority`。其效果是对所有的进程（包括阻塞态进程）都进行 `counter` 的衰减，并再累加 `priority` 值。这样，对正被阻塞的进程来说，一个进程在阻塞队列中停留的时间越长，其优先级越大，被分配的时间片也就会越大。

所以总的来说，Linux 0.11 的进程调度是一种综合考虑进程优先级并能动态反馈调整时间片的轮转调度算法。

此处要求实验者对现有的调度算法进行时间片大小的修改，并进行实验验证。为完成此工作，我们需要知道两件事情：

- 进程 `counter` 是如何初始化的
- 当进程的时间片用完时，被重新赋成何值？

首先回答第一个问题，显然这个值是在 `fork()` 中设定的。Linux 0.11 的 `fork()` 会调用 `copy_process()` 来完成从父进程信息拷贝（所以才称其为 `fork`），看看 `copy_process()` 的实现（也在 `kernel/fork.c` 文件中），会发现其中有下面两条语句：

```

// 用来复制父进程的PCB数据信息，包括 priority 和 counter
*p = *current;
// 初始化 counter
p->counter = p->priority;
// 因为父进程的counter数值已发生变化，而 priority 不会，所以上面的第二句代

```

```
码将p->counter 设置成 p->priority。  
// 每个进程的 priority 都是继承自父亲进程的，除非它自己改变优先级。
```

```
// 查找所有的代码，只有一个地方修改过 priority，那就是 nice 系统调用。  
int sys_nice(long increment)  
{  
    if (current->priority-increment>0)  
        current->priority -= increment;  
    return 0;  
}
```

本实验假定没有人调用过 nice 系统调用，时间片的初值就是进程 0 的 priority，即宏 INIT_TASK 中定义的：

```
#define INIT_TASK \  
    { 0, 15, 15,   
// 上述三个值分别对应 state、counter 和 priority;
```

接下来回答第二个问题，当就绪进程的 counter 为 0 时，不会被调度（schedule 要选取 counter 最大的，大于 0 的进程），而当所有的就绪态进程的 counter 都变成 0 时，会执行下面的语句：

```
(*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
```

显然算出的新的 counter 值也等于 priority，即初始时间片的大小。提示就到这里。如何修改时间片，自己思考、尝试吧

5) 基于内核栈切换的进程切换

2020年9月27日 14:09

基于内核栈切换的进程切换

基于内核栈切换的进程切换

1. 课程说明

难度系数：★★★★☆

本实验是 [操作系统之进程与线程 - 网易云课堂](#) 的配套实验，推荐大家进行实验之前先学习相关课程：

- L10 用户级线程
- L11 内核级线程
- L12 核心级线程实现实例
- L13 操作系统的那棵树

Tips: 点击上方文字中的超链接或者输

入 <https://mooc.study.163.com/course/1000002008#/info> 进入理论课程的学习。如果网易云上的课程无法查看，也可以看 Bilibili 上的 [操作系统哈尔滨工业大学李治军老师](#)。

2. 实验目的

- 深入理解进程和进程切换的概念；
- 综合应用进程、CPU 管理、PCB、LDT、内核栈、内核态等知识解决实际问题；
- 开始建立系统认识。

3. 实验内容

现在的 Linux 0.11 采用 TSS（后面会有详细论述）和一条指令就能完成任务切换，虽然简单，但这指令的执行时间却很长，在实现任务切换时大概需要 200 多个时钟周期。

而通过堆栈实现任务切换可能要更快，而且采用堆栈的切换还可以使用指令流水的并行优化技术，同时又使得 CPU 的设计变得简单。所以无论是 Linux 还是 Windows，进程/线程的切换都没有使用 Intel 提供的这种 TSS 切换手段，而都是通过堆栈实现的。

本次实践项目就是将 Linux 0.11 中采用的 TSS 切换部分去掉，取而代之的是基于堆栈的切换程序。具体的说，就是将 Linux 0.11 中的 switch_to 实现去掉，写成一段基于堆栈切换的代码。

本次实验包括如下内容：

- 编写汇编程序 switch_to；
- 完成主体框架；
- 在主体框架下依次完成 PCB 切换、内核栈切换、LDT 切换等；
- 修改 fork()，由于是基于内核栈的切换，所以进程需要创建出能完成内核栈切换的样子。
- 修改 PCB，即 task_struct 结构，增加相应的内容域，同时处理由于修改了 task_struct 所造成的影响。
- 用修改后的 Linux 0.11 仍然可以启动、可以正常使用。
- （选做）分析实验 3 的日志体会修改前后系统运行的差别。

4. 实验报告

回答下面三个题：

问题 1

针对下面的代码片段：

```
movl tss,%ecx
addl $4096,%ebx
movl %ebx,ESP0(%ecx)
```

回答问题：

- （1）为什么要加 4096；
- （2）为什么没有设置 tss 中的 ss0。

问题 2

针对代码片段：

```
*(--krnstack) = ebp;
*(--krnstack) = ecx;
*(--krnstack) = ebx;
*(--krnstack) = 0;
```

回答问题：

- （1）子进程第一次执行时，eax=? 为什么要等于这个数？哪里的工作让 eax 等于这样一个数？
- （2）这段代码中的 ebx 和 ecx 来自哪里，是什么含义，为什么要通过这些代码将其写到子进程的内核栈中？
- （3）这段代码中的 ebp 来自哪里，是什么含义，为什么要做这样的设置？可以不设置吗？为什么？

问题 3

为什么要在切换完 LDT 之后要重新设置 fs=0x17？而且为什么重设操作要出现在切换完 LDT 之后，出现在 LDT 之前又会怎么样？

5. 评分标准

- switch_to(kernal/system_call.s)，40%
- fork.c，30%
- sched.h 和 sched.c，10%
- 实验报告，20%

6. 实验提示

本次实验将 Linux 0.11 中采用的 TSS 切换部分去掉，取而代之的是基于堆栈的切换程序。具体的说，就是将 Linux 0.11 中

的 `switch_to`（在 `kernal/system_call.s` 中）实现去掉，写成一段基于堆栈切换的代码。

6.1 TSS 切换

在现在的 Linux 0.11 中，真正完成进程切换是依靠任务状态段（Task State Segment，简称 TSS）的切换来完成的。

具体的说，在设计“Intel 架构”（即 x86 系统结构）时，每个任务（进程或线程）都对应一个独立的 TSS，TSS 就是内存中的一个结构体，里面包含了几乎所有的 CPU 寄存器的映像。有一个任务寄存器（Task Register，简称 TR）指向当前进程对应的 TSS 结构体，所谓的 TSS 切换就将 CPU 中几乎所有的寄存器都复制到 TR 指向的那个 TSS 结构体中保存起来，同时找到一个目标 TSS，即要切换到的下一个进程对应的 TSS，将其中存放的寄存器映像“扣在” CPU 上，就完成了执行现场的切换，如下图所示。

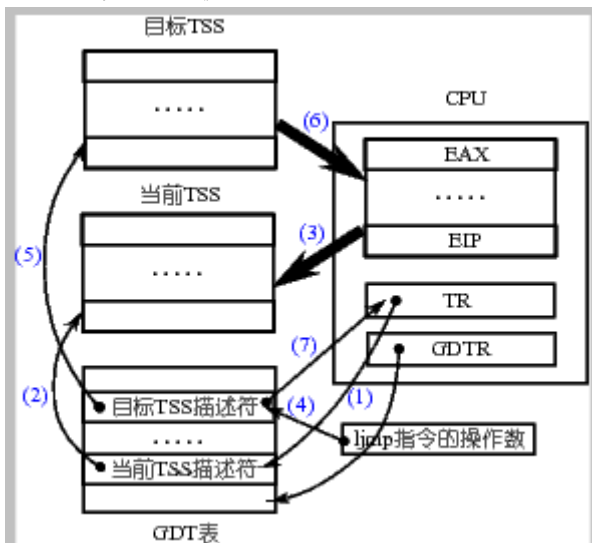


图 1 基于 TSS 的进程切换

Intel 架构不仅提供了 TSS 来实现任务切换，而且只要一条指令就能完成这样的切换，即图中的 `ljmp` 指令。

具体的工作过程是：

- （1）首先用 TR 中存取的段选择符在 GDT 表中找到当前 TSS 的内存位置，由于 TSS 是一个段，所以需要段表中的一个描述符来表示这个段，和在系统启动时论述的内核代码段是一样的，那个段用 GDT 中的某个表项来描述，还记得是哪项吗？是 8 对应的第 1 项。此处的 TSS 也是用 GDT 中的某个表项描述，而 TR 寄存器是用来表示这个段用 GDT 表中的哪一项来描述，所以 TR 和 CS、DS 等寄存器的功能是完全类似的。
- （2）找到了当前的 TSS 段（就是一段内存区域）以后，将 CPU 中的寄存器映像存放到这段内存区域中，即拍了一个快照。
- （3）存放了当前进程的执行现场以后，接下来要找到目标进程的现场，并将其扣在 CPU 上，找目标 TSS 段的方法也是一样的，因为找段都要从一个描述符表中找，描述 TSS 的描述符放在 GDT 表中，所以找目标 TSS 段也要靠 GDT 表，当然只要给出目标 TSS 段对应的描述符在 GDT 表中存放的位置——段选择子就可以了，仔细想想系统启动时那条著名的 `jmp 0, 8` 指令，这个段选择子就放在 `ljmp` 的参数中，实际上就 `jmp 0, 8` 中的 8。
- （4）一旦将目标 TSS 中的全部寄存器映像扣在 CPU 上，就相当于切换到了目标进程的执行现场了，因为那里有目标进程停下

时的 CS:EIP，所以此时就开始从目标进程停下时的那个 CS:EIP 处开始执行，现在目标进程就变成了当前进程，所以 TR 需要修改为目标 TSS 段在 GDT 表中的段描述符所在的位置，因为 TR 总是指向当前 TSS 段的段描述符所在的位置。上面给出的这些工作都是一句长跳转指令 `ljmp` 段选择子:段内偏移，在段选择子指向的段描述符是 TSS 段时 CPU 解释执行的结果，所以基于 TSS 进行进程/线程切换的 `switch_to` 实际上就是一句 `ljmp` 指令：

```
#define switch_to(n) {
    struct{long a,b;} tmp;
    __asm__(
        "movw %%dx,%1"
        "ljmp %0" ::"m"(*&tmp.a), "m"(*&tmp.b), "d"(TSS(n)
    )
}
#define FIRST_TSS_ENTRY 4
#define TSS(n) (((unsigned long) n) << 4) + (FIRST_TSS_ENTRY
<< 3))
```

GDT 表的结构如下图所示，所以第一个 TSS 表项，即 0 号进程的 TSS 表项在第 4 个位置上， $4 \ll 3$ ，即 $4 * 8$ ，相当于 TSS 在 GDT 表中开始的位置，`TSS(n)` 找到的是进程 n 的 TSS 位置，所以还要再加上 $n \ll 4$ ，即 $n * 16$ ，因为每个进程对应有一个 TSS 和一个 LDT，每个描述符的长度都是 8 个字节，所以是乘以 16，其中 LDT 的作用就是上面论述的那个映射表，关于这个表的详细论述要等到内存管理一章。 $TSS(n) = n * 16 + 4 * 8$ ，得到就是进程 n（切换到的目标进程）的 TSS 选择子，将这个值放到 dx 寄存器中，并且又放置到结构体 tmp 中 32 位长整数 b 的前 16 位，现在 64 位 tmp 中的内容是前 32 位为空，这个 32 位数字是段内偏移，就是 `jmp 0, 8` 中的 0；接下来的 16 位是 $n * 16 + 4 * 8$ ，这个数字是段选择子，就是 `jmp 0, 8` 中的 8，再接下来的 16 位也为空。所以 `switch_to` 的核心实际上就是 `ljmp 空, $n * 16 + 4 * 8$` ，现在和前面给出的基于 TSS 的进程切换联系在一起了。

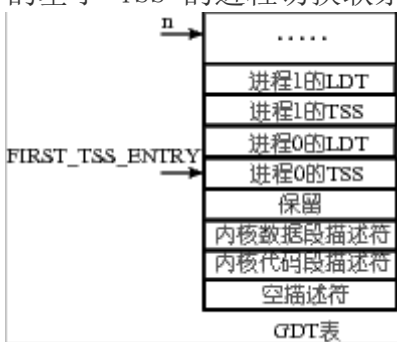


图 2 GDT 表中的内容

6.2 本次实验的内容

虽然用一条指令就能完成任务切换，但这指令的执行时间却很长，这条 `ljmp` 指令在实现任务切换时大概需要 200 多个时钟周期。而通过堆栈实现任务切换可能要更快，而且采用堆栈的切换还可以使用指令流水的并行优化技术，同时又使得 CPU 的设计变得简单。所以无论是 Linux 还是 Windows，进程/线程的切换都没有使用 Intel 提供的这种 TSS 切换手段，而都是通过堆栈实现的。

本次实践项目就是将 Linux 0.11 中采用的 TSS 切换部分去掉，取而代之的是基于堆栈的切换程序。具体的说，就是将 Linux 0.11 中的 switch_to 实现去掉，写成一段基于堆栈切换的代码。

在现在的 Linux 0.11 中，真正完成进程切换是依靠任务状态段（Task State Segment，简称 TSS）的切换来完成的。具体的说，在设计“Intel 架构”（即 x86 系统结构）时，每个任务（进程或线程）都对应一个独立的 TSS，TSS 就是内存中的一个结构体，里面包含了几乎所有的 CPU 寄存器的映像。有一个任务寄存器（Task Register，简称 TR）指向当前进程对应的 TSS 结构体，所谓的 TSS 切换就将 CPU 中几乎所有的寄存器都复制到 TR 指向的那个 TSS 结构体中保存起来，同时找到一个目标 TSS，即要切换到的下一个进程对应的 TSS，将其中存放的寄存器映像“扣在”CPU 上，就完成了执行现场的切换。

要实现基于内核栈的任务切换，主要完成如下三件工作：

- （1）重写 switch_to；
- （2）将重写的 switch_to 和 schedule() 函数接在一起；
- （3）修改现在的 fork()。

6.3 schedule 与 switch_to

目前 Linux 0.11 中工作的 schedule() 函数是首先找到下一个进程的数组位置 next，而这个 next 就是 GDT 中的 n，所以这个 next 是用来找到切换后目标 TSS 段的段描述符的，一旦获得了这个 next 值，直接调用上面剖析的那个宏展开 switch_to(next)；就能完成如图 TSS 切换所示的切换了。

现在，我们不用 TSS 进行切换，而是采用切换内核栈的方式来完成进程切换，所以在新的 switch_to 中将用到当前进程的 PCB、目标进程的 PCB、当前进程的内核栈、目标进程的内核栈等信息。一页内存上（一块 4KB 大小的内存），其中 PCB 位于这页内存的低地址，栈位于这页内存的高地址；另外，由于当前进程的 PCB 是用一个全局变量 current 指向的，所以只要告诉新 switch_to() 函数一个指向目标进程 PCB 的指针就可以了。同时还要将 next 也传递进去，虽然 TSS(next) 不再需要了，但是 LDT(next) 仍然是需要的，也就是说，现在每个进程不用有自己的 TSS 了，因为已经不采用 TSS 进程切换了，但是每个进程需要有自己的 LDT，地址分离地址还是必须要有的，而进程切换必然要涉及到 LDT 的切换。

综上所述，需要将目前的 schedule() 函数

（在 kernel/sched.c 中）做稍许修改，即将下面的代码：

```
if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
    c = (*p)->counter, next = i;
//.....
switch_to(next);
```

修改为：

```
if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
    c = (*p)->counter, next = i, pnext = *p;
//.....
switch_to(pnext, LDT(next));
```

6.4 实现 switch_to

实现 switch_to 是本次实践项目中最重要的一部分。

由于要对内核栈进行精细的操作，所以需要用汇编代码来完成函数 `switch_to` 的编写。

这个函数依次主要完成如下功能：由于是 C 语言调用汇编，所以需要首先在汇编中处理栈帧，即处理 `ebp` 寄存器；接下来要取出表示下一个进程 PCB 的参数，并和 `current` 做一个比较，如果等于 `current`，则什么也不用做；如果不等于 `current`，就开始进程切换，依次完成 PCB 的切换、TSS 中的内核栈指针的重写、内核栈的切换、LDT 的切换以及 PC 指针（即 `CS:EIP`）的切换。

```
switch_to:
    pushl %ebp
    movl %esp,%ebp
    pushl %ecx
    pushl %ebx
    pushl %eax
    movl 8(%ebp),%ebx
    cmpl %ebx,current
    je 1f
! 切换PCB
! ...
! TSS中的内核栈指针的重写
! ...
! 切换内核栈
! ...
! 切换LDT
! ...
    movl $0x17,%ecx
    mov %cx,%fs
! 和后面的 clts 配合来处理协处理器，由于和主题关系不大，此处
! 不做论述
    cmpl %eax,last_task_used_math
    jne 1f
    clts
1:    popl %eax
    popl %ebx
    popl %ecx
    popl %ebp
    ret
```

虽然看起来完成了挺多的切换，但实际上每个部分都只有很简单的几条指令。完成 **PCB 的切换** 可以采用下面两条指令，其中 `ebx` 是从参数中取出来的下一个进程的 PCB 指针，

```
movl %ebx,%eax
xchgl %eax,current
```

经过这两条指令以后，`eax` 指向现在的当前进程，`ebx` 指向下一个进程，全局变量 `current` 也指向下一个进程。

TSS 中的内核栈指针的重写可以用下面三条指令完成，其中宏 `ESP0 = 4`，`struct tss_struct *tss = &(init_task.task.tss)`；也是定义了一个全局变量，和 `current` 类似，用来指向那一段 0 号进程的

TSS 内存。

前面已经详细论述过，在中断的时候，要找到内核栈位置，并将用户态下的 SS:ESP, CS:EIP 以及 EFLAGS 这五个寄存器压到内核栈中，这是沟通用户栈（用户态）和内核栈（内核态）的关键桥梁，而找到内核栈位置就依靠 TR 指向的当前 TSS。

现在虽然不使用 TSS 进行任务切换了，但是 Intel 的这态中断处理机制还要保持，所以仍然需要有一个当前 TSS，这个 TSS 就是我们定义的那个全局变量 tss，即 0 号进程的 tss，所有进程都共用这个 tss，任务切换时不再发生变化。

```
movl tss,%ecx
addl $4096,%ebx
movl %ebx,ESPO(%ecx)
```

定义 ESPO = 4 是因为 TSS 中内核栈指针 esp0 就放在偏移为 4 的地方，看一看 tss 的结构体定义就明白了。

完成内核栈的切换也非常简单，和我们前面给出的论述完全一致，将寄存器 esp（内核栈使用到当前情况时的栈顶位置）的值保存到当前 PCB 中，再从下一个 PCB 中的对应位置上取出保存的内核栈栈顶放入 esp 寄存器，这样处理完以后，再使用内核栈时使用的就是下一个进程的内核栈了。

由于现在的 Linux 0.11 的 PCB 定义中没有保存内核栈指针这个域（kernelstack），所以需要加上，而宏 KERNEL_STACK 就是你加的那个位置，当然将 kernelstack 域加在 task_struct 中的哪个位置都可以，但是在某些汇编文件中（主要是在 kernal/system_call.s 中）有些关于操作这个结构一些汇编硬编码，所以一旦增加了 kernelstack，这些硬编码需要跟着修改，由于第一个位置，即 long state 出现的汇编硬编码很多，所以 kernelstack 千万不要放置在 task_struct 中的第一个位置，当放在其他位置时，修改 kernal/system_call.s 中的那些硬编码就可以了。

```
KERNEL_STACK = 12
movl %esp,KERNEL_STACK(%eax)
! 再取一下 ebx，因为前面修改过 ebx 的值
movl 8(%ebp),%ebx
movl KERNEL_STACK(%ebx),%esp
```

task_struct 的定义：

```
// 在 include/linux/sched.h 中
struct task_struct {
    long state;
    long counter;
    long priority;
    long kernelstack;
    //.....
```

由于这里将 PCB 结构体的定义改变了，所以在产生 0 号进程的 PCB 初始化时也要跟着一起变化，需要将原来的 #define INIT_TASK { 0,15,15, 0,{},{},0,... 修改为 #define INIT_TASK { 0,15,15,PAGE_SIZE+(long)&init_task, 0,{},{},0,...，即在

PCB 的第四项中增加关于内核栈指针的初始化。

再下一个切换就是 LDT 的切换了，指令 `movl 12(%ebp),%ecx` 负责取出对应 LDT(next)的那个参数，指令 `lldt %cx` 负责修改 LDTR 寄存器，一旦完成了修改，下一个进程在执行用户态程序时使用的映射表就是自己的 LDT 表了，地址空间实现了分离。

最后一个切换是关于 PC 的切换，和前面论述的一致，依靠的就是 `switch_to` 的最后一句指令 `ret`，虽然简单，但背后发生的事却很多：

- `schedule()` 函数的最后调用了这个 `switch_to` 函数
- 所以这句指令 `ret` 就返回到下一个进程（目标进程）的 `schedule()` 函数的末尾
- 遇到的是}
- 继续 `ret` 回到调用的 `schedule()` 地方
- 是在中断处理中调用的
- 所以回到了中断处理中
- 就到了中断返回的地址

再调用 `iret` 就到了目标进程的用户态程序去执行和书中论述的内核态线程切换的五段论是完全一致的。

这里还有一个地方需要格外注意，那就是 `switch_to` 代码中在切换完 LDT 后的两句，即：

```
! 切换 LDT 之后
movl $0x17,%ecx
mov %cx,%fs
```

这两句代码的含义是重新取一下段寄存器 `fs` 的值，这两句话必须要加、也必须要出现在切换完 LDT 之后，这是因为在实践项目 2 中曾经看到过 `fs` 的作用——通过 `fs` 访问进程的用户态内存，LDT 切换完成就意味着切换了分配给进程的用户态内存地址空间，所以前一个 `fs` 指向的是上一个进程的用户态内存，而现在需要执行下一个进程的用户态内存，所以就需要用这两条指令来重取 `fs`。

不过，细心的读者可能会发现：`fs` 是一个选择子，即 `fs` 是一个指向描述符表项的指针，这个描述符才是指向实际的用户态内存的指针，所以上一个进程和下一个进程的 `fs` 实际上都是 0x17，真正找到不同的用户态内存是答这个问题就需要对段寄存器有更深刻的认识，实际上段寄存器包含两个部分：显式部分和隐式部分，如下图给出实例所示，就是那个著名的 `jmp 0, 8`，虽然我们的指令是让 `cs=8`，但在执行这条指令时，会在段表（GDT）中找到 8 对应的那个描述符表项，取出基地址和段限长，除了完成和 `eip` 的累加算出 PC 以外，还会将取出的基地址和段限长放在 `cs` 的隐藏部分，即图中的基地址 0 和段限长 7FF。为什么要这样做？下次执行 `jmp 100` 时，由于 `cs` 没有改过，仍然是 8，所以可以不再去查 GDT 表，而是直接用其隐藏部分中的基地址 0 和 100 累加直接得到 PC，增加了执行指令的效率。现在想必明白了为什么重新设置 `fs=0x17` 了吧？而且为什么要出现在切换完 LDT 之后？

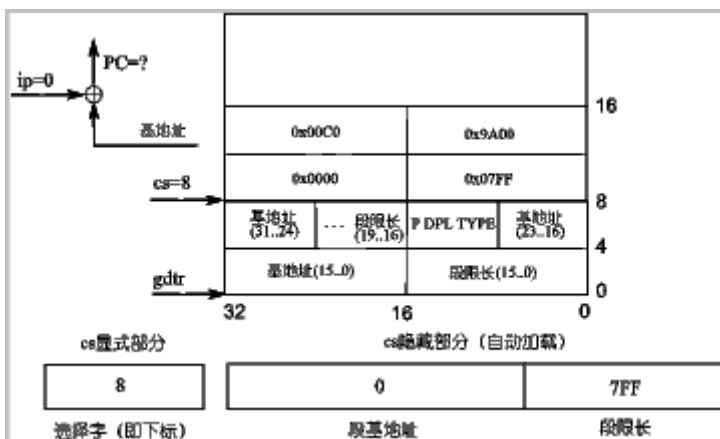


图 3 段寄存器中的两个部分

6.5 修改 fork

开始修改 `fork()` 了，和书中论述的原理一致，就是要把进程的用户栈、用户程序和其内核栈通过压在内核栈中的 `SS:ESP`，`CS:IP` 关联在一起。

另外，由于 `fork()` 这个叉子的含义就是要让父子进程共用同一个代码、数据和堆栈，现在虽然是使用内核栈完成任务切换，但 `fork()` 的基本含义不会发生变化。

将上面两段描述联立在一起，修改 `fork()` 的核心工作就是要形成如下图所示的子进程内核栈结构。

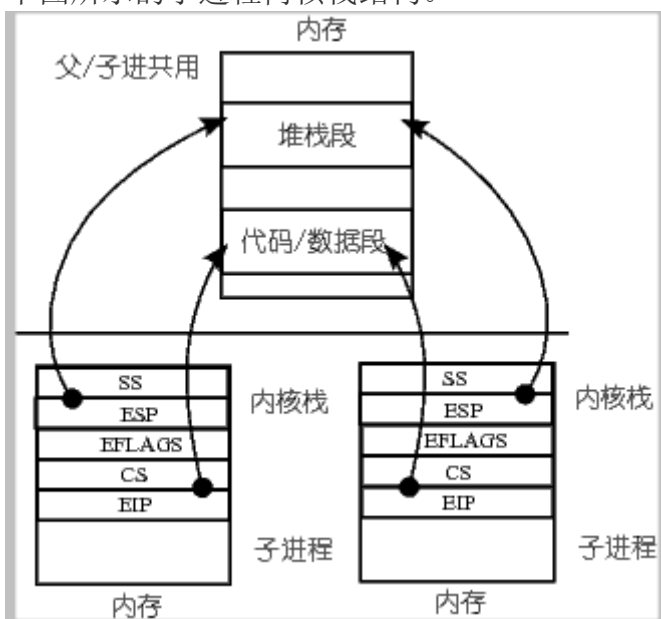


图 4 `fork` 进程的父子进程结构

不难想象，对 `fork()` 的修改就是对子进程的内核栈的初始化，在 `fork()` 的核心实现 `copy_process` 中，`p = (struct task_struct *) get_free_page()`；用来完成申请一页内存作为子进程的 PCB，而 `p` 指针加上页面大小就是子进程的内核栈位置，所以语句 `krnstack = (long *) (PAGE_SIZE + (long) p)`；就可以找到子进程的内核栈位置，接下来就是初始化 `krnstack` 中的内容了。

```
*(--krnstack) = ss & 0xffff;
*(--krnstack) = esp;
*(--krnstack) = eflags;
*(--krnstack) = cs & 0xffff;
*(--krnstack) = eip;
```

这五条语句就完成了上图所示的那个重要的关联，因为其中 `ss, esp`

等内容都是 `copy_proces()` 函数的参数，这些参数来自调用 `copy_proces()` 的进程的内核栈中，就是父进程的内核栈中，所以上面给出的指令不就是将父进程内核栈中的前五个内容拷贝到子进程的内核栈中，图中所示的关联不也就是一个拷贝吗？

接下来的工作就需要和 `switch_to` 接在一起考虑了，故事从哪里开始呢？回顾一下前面给出来的 `switch_to`，应该从“切换内核栈”完事的那个地方开始，现在到子进程的内核栈开始工作了，接下来做的四次弹栈以及 `ret` 处理使用的都是子进程内核栈中的东西，

```
l: popl %eax
    popl %ebx
    popl %ecx
    popl %ebp
ret
```

为了能够顺利完成这些弹栈工作，子进程的内核栈中应该有这些内容，所以需要对 `krnstack` 进行初始化：

```
*(--krnstack) = ebp;
*(--krnstack) = ecx;
*(--krnstack) = ebx;
// 这里的 0 最有意思。
*(--krnstack) = 0;
```

现在到了 `ret` 指令了，这条指令要从内核栈中弹出一个 32 位数作为 EIP 跳去执行，所以需要弄一个函数地址（仍然是一段汇编程序，所以这个地址是这段汇编程序开始处的标号）并将其初始化到栈中。我们弄的一个名为 `first_return_from_kernel` 的汇编标号，然后可以用语句 `*(--krnstack) = (long)`

`first_return_from_kernel`；将这个地址初始化到子进程的内核栈中，现在执行 `ret` 以后就会跳转到 `first_return_from_kernel` 去执行了。

想一想 `first_return_from_kernel` 要完成什么工作？PCB 切换完成、内核栈切换完成、LDT 切换完成，接下来应该那个“内核级线程切换五段论”中的最后一段切换了，即完成用户栈和用户代码的切换，依靠的核心指令就是 `iret`，当然在切换之前应该回复一下执行现场，主要就是 `eax, ebx, ecx, edx, esi, edi, gs, fs, es, ds` 等寄存器的恢复。

下面给出了 `first_return_from_kernel` 的核心代码，当然 `edx` 等寄存器的值也应该先初始化到子进程内核栈，即 `krnstack` 中。

```
popl %edx
popl %edi
popl %esi
pop %gs
pop %fs
pop %es
pop %ds
iret
```

最后别忘了将存放在 PCB 中的内核栈指针修改到初始化完成时内核栈的栈顶，即：

```
p->kernelstack = stack;
```

来自 <<https://www.lanqiao.cn/mobile/courses/115/learning?id=571>>

6) 信号量的实现和应用

2020年9月27日 14:10

信号量的实现和应用

1. 课程说明

难度系数：★★★★☆

本实验是 [操作系统之进程与线程 - 网易云课堂](#) 的配套实验，推荐大家进行实验之前先学习相关课程：

- L16 进程同步与信号量
- L17 对信号量的临界区保护
- L18 信号量的代码实现
- L19 死锁处理

Tips: 点击上方文字中的超链接或者输

入 <https://mooc.study.163.com/course/1000002008#/info> 进入理论课程的学习。如果网易云上的课程无法查看，也可以看 Bilibili 上的 [操作系统哈尔滨工业大学李治军老师](#)。

2. 实验目的

- 加深对进程同步与互斥概念的认识；
- 掌握信号量的使用，并应用它解决生产者——消费者问题；
- 掌握信号量的实现原理。

3. 实验内容

本次实验的基本内容是：

- 在 Ubuntu 下编写程序，用信号量解决生产者——消费者问题；
- 在 0.11 中实现信号量，用生产者—消费者程序检验之。

3.1 用信号量解决生产者—消费者问题

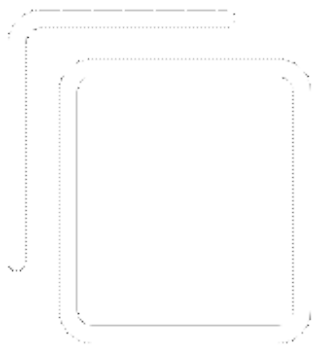
在 Ubuntu 上编写应用程序“pc.c”，解决经典的生产者—消费者问题，完成下面的功能：

- 建立一个生产者进程，N 个消费者进程（ $N>1$ ）；
- 用文件建立一个共享缓冲区；
- 生产者进程依次向缓冲区写入整数 $0, 1, 2, \dots, M$ ， $M \geq 500$ ；
- 消费者进程从缓冲区读数，每次读一个，并将读出的数字从缓冲区删除，然后将本进程 ID 和 + 数字输出到标准输出；
- 缓冲区同时最多只能保存 10 个数。

一种可能的输出效果是：

```
10: 0
10: 1
10: 2
10: 3
10: 4
11: 5
11: 6
```

```
12: 7
10: 8
12: 9
12: 10
12: 11
12: 12
.....
11: 498
11: 499
```



其中 ID 的顺序会有较大变化，但冒号后的数字一定是从 0 开始递增加一的。
pc.c 中将会用到 `sem_open()`、`sem_close()`、`sem_wait()` 和 `sem_post()` 等信号量相关的系统调用，请查阅相关文档。

《UNIX 环境高级编程》是一本关于 Unix/Linux 系统级编程的相当经典的教程。如果你对 POSIX 编程感兴趣，建议买一本常备手边。

哈尔滨工业大学校园网用户可以

在 <ftp://run.hit.edu.cn/study/Computer Science/Linux Unix/> 下载，后续实验也用得到。

3.2 实现信号量

Linux 在 0.11 版还没有实现信号量，Linus 把这件富有挑战的工作留给了你。如果能实现一套山寨版的完全符合 POSIX 规范的信号量，无疑是很有成就感的。但时间暂时不允许我们这么做，所以先弄一套缩水版的类 POSIX 信号量，它的函数原型和标准并不完全相同，而且只包含如下系统调用：

```
sem_t *sem_open(const char *name, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_unlink(const char *name);
```

- `sem_open()` 的功能是创建一个信号量，或打开一个已经存在的信号量。
 - `sem_t` 是信号量类型，根据实现的需要自定义。
 - `name` 是信号量的名字。不同的进程可以通过提供同样的 `name` 而共享同一个信号量。如果该信号量不存在，就创建新的名为 `name` 的信号量；如果存在，就打开已经存在的名为 `name` 的信号量。
 - `value` 是信号量的初值，仅当新建信号量时，此参数才有效，其余情况下它被忽略。当成功时，返回值是该信号量的唯一标识（比如，在内核的地址、ID

等)，由另两个系统调用使用。如失败，返回值是 NULL。

- `sem_wait()` 就是信号量的 P 原子操作。如果继续运行的条件不满足，则令调用进程等待在信号量 `sem` 上。返回 0 表示成功，返回 -1 表示失败。
- `sem_post()` 就是信号量的 V 原子操作。如果有等待 `sem` 的进程，它会唤醒其中的一个。返回 0 表示成功，返回 -1 表示失败。
- `sem_unlink()` 的功能是删除名为 `name` 的信号量。返回 0 表示成功，返回 -1 表示失败。

在 `kernel` 目录下新建 `sem.c` 文件实现如上功能。然后将 `pc.c` 从 Ubuntu 移植到 0.11 下，测试自己实现的信号量。

4. 实验报告

完成实验后，在实验报告中回答如下问题：

在 `pc.c` 中去掉所有与信号量有关的代码，再运行程序，执行效果有变化吗？为什么会这样？实验的设计者在第一次编写生产者——消费者程序的时候，是这么做的：

```
Producer()
{
    // 生产一个产品 item;
// 空闲缓存资源
    P(Empty);
// 互斥信号量
    P(Mutex);
// 将item放到空闲缓存中;
    V(Mutex);
// 产品资源
    V(Full);
}

Consumer()
{
    P(Full);
    P(Mutex);
//从缓存区取出一个赋值给item;
    V(Mutex);
// 消费产品item;
    V(Empty);
}
```

这样可行吗？如果可行，那么它和标准解法在执行效果上会有什么不同？如果不可行，那么它有什么问题使它不可行？

5. 评分标准

- `pc.c`, 40%
- `sem_open()`, 10%
- `sem_post()`, 10%
- `sem_wait()`, 10%
- `sem_unlink()`, 10%

- 实验报告，20%

6. 实验提示

本实验需要完成两个任务：（1）在 Ubuntu 下编写程序，用信号量解决生产者——消费者问题；（2）在 linux-0.11 中实现信号量，用生产者——消费者程序检验之。

6.1 信号量

信号量，英文为 semaphore，最早由荷兰科学家、图灵奖获得者 E. W. Dijkstra 设计，任何操作系统教科书的“进程同步”部分都会有详细叙述。

Linux 的信号量秉承 POSIX 规范，用 `man sem_overview` 可以查看相关信息。

本次实验涉及到的信号量系统调用包括：

`sem_open()`、`sem_wait()`、`sem_post()` 和 `sem_unlink()`。

生产者——消费者问题

生产者——消费者问题的解法几乎在所有操作系统教科书上都有，其基本结构为：

```

Producer()
{
    // 生产一个产品 item;
// 空闲缓存资源
    P(Empty);
// 互斥信号量
    P(Mutex);
// 将item放到空闲缓存中;
    V(Mutex);
// 产品资源
    V(Full);
}

Consumer()
{
    P(Full);
    P(Mutex);
//从缓存区取出一个赋值给item;
    V(Mutex);
// 消费产品item;
    V(Empty);
}

```

显然在演示这一过程时需要创建两类进程，一类执行函数 `Producer()`，另一类执行函数 `Consumer()`。

6.2 多进程共享文件

在 Linux 下使用 C 语言，可以通过三种方法进行文件的读写：

- 使用标准 C 的 `fopen()`、`fread()`、`fwrite()`、`fseek()` 和 `fclose()` 等；
- 使用系统调用 `open()`、`read()`、`write()`、`lseek()` 和 `close()` 等；
- 通过内存镜像文件，使用 `mmap()` 系统调用。
- 在 Linux 0.11 上只能使用前两种方法。

`fork()` 调用成功后，子进程会继承父进程拥有的大多数资源，包括父进程打开的文

件。所以子进程可以直接使用父进程创建的文件指针/描述符/句柄，访问的是与父进程相同的文件。

使用标准 C 的文件操作函数要注意，它们使用的是进程空间内的文件缓冲区，父进程和子进程之间不共享这个缓冲区。因此，任何一个进程做完写操作后，必须 `fflush()` 一下，将数据强制更新到磁盘，其它进程才能读到所需数据。建议直接使用系统调用进行文件操作。

6.3 终端也是临界资源

用 `printf()` 向终端输出信息是很自然的事情，但当多个进程同时输出时，终端也成为了一个临界资源，需要做好互斥保护，否则输出的信息可能错乱。

另外，`printf()` 之后，信息只是保存在输出缓冲区内，还没有真正送到终端上，这也可能造成输出信息时序不一致。用 `fflush(stdout)` 可以确保数据送到终端。

6.4 原子操作、睡眠和唤醒

Linux 0.11 是一个支持并发的现代操作系统，虽然它还没有面向应用实现任何锁或者信号量，但它内部一定使用了锁机制，即在多个进程访问共享的内核数据时一定需要通过锁来实现互斥和同步。

锁必然是一种原子操作。通过模仿 0.11 的锁，就可以实现信号量。

多个进程对磁盘的并发访问是一个需要锁的地方。Linux 0.11 访问磁盘的基本处理方法是在内存中划出一段磁盘缓存，用来加快对磁盘的访问。进程提出的磁盘访问请求首先要到磁盘缓存中去找，如果找到直接返回；如果没有找到则申请一段空闲的磁盘缓存，以这段磁盘缓存为参数发起磁盘读写请求。请求发出后，进程要睡眠等待（因为磁盘读写很慢，应该让出 CPU 让其他进程执行）。这种方法是许多操作系统（包括现代 Linux、UNIX 等）采用的较通用的方法。这里涉及到多个进程共同操作磁盘缓存，而进程在操作过程可能会被调度而失去 CPU。因此操作磁盘缓存时需要考虑互斥问题，所以其中必定用到了锁。而且也一定用到了让进程睡眠和唤醒。

下面是从 `kernel/blk_drv/ll_rw_blk.c` 文件中取出的两个函数：

```
static inline void lock_buffer(struct buffer_head * bh)
{
    // 关中断
    cli();
    // 将当前进程睡眠在 bh->b_wait
    while (bh->b_lock)
        sleep_on(&bh->b_wait);
    bh->b_lock=1;
    // 开中断
    sti();
}

static inline void unlock_buffer(struct buffer_head * bh)
{
    if (!bh->b_lock)
        printk("ll_rw_block.c: buffer not locked\n\r");
    bh->b_lock = 0;
    // 唤醒睡眠在 bh->b_wait 上的进程
    wake_up(&bh->b_wait);
}
```

}

分析 `lock_buffer()` 可以看出，访问锁变量时用开、关中断来实现原子操作，阻止进程切换的发生。当然这种方法有缺点，且不适合用于多处理器环境中，但对于 Linux 0.11，它是一种简单、直接而有效的机制。

另外，上面的函数表明 Linux 0.11 提供了这样的接口：用 `sleep_on()` 实现进程的睡眠，用 `wake_up()` 实现进程的唤醒。它们的参数都是一个结构体指针—— `struct task_struct *`，即进程都睡眠或唤醒在该参数指向的一个进程 PCB 结构链表上。

因此，我们可以用开关中断的方式实现原子操作，而调

用 `sleep_on()` 和 `wake_up()` 进行进程的睡眠和唤醒。

`sleep_on()` 的功能是将当前进程睡眠在参数指定的链表上（注意，这个链表是一个隐式链表，详见《注释》一书）。`wake_up()` 的功能是唤醒链表上睡眠的所有进程。这些进程都会被调度运行，所以它们被唤醒后，还要重新判断一下是否可以继续运行。可参考 `lock_buffer()` 中的那个 `while` 循环。

6.5 应对混乱的 bochs 虚拟屏幕

不知是 Linux 0.11 还是 bochs 的 bug，如果向终端输出的信息较多，bochs 的虚拟屏幕会产生混乱。此时按 `ctrl+L` 可以重新初始化一下屏幕，但输出信息一多，还是会混乱。建议把输出信息重定向到一个文件，然后用 `vi`、`more` 等工具按屏查看这个文件，可以基本解决此问题。

6.6 关于 `string.h` 的提示

下面描述的问题未必具有普遍意义，仅做为提醒，请实验者注意。

`include/string.h` 实现了全套的 C 语言字符串操作，而且都是采用汇编 + `inline` 方式优化。

但在使用中，某些情况下可能会遇到一些奇怪的问题。比如某人就遇到 `strcmp()` 会破坏参数内容的问题。如果调试中遇到有些“诡异”的情况，可以试试不包含头文件，一般都能解决。不包含 `string.h`，就不会用 `inline` 方式调用这些函数，它们工作起来就趋于正常了

来自 <<https://www.lanqiao.cn/mobile/courses/115/learning?id=572>>

7) 地址映射与共享

2020年9月27日 14:11

地址映射与共享

1. 课程说明

难度系数：★★★★☆

本实验是 [操作系统之内存管理 - 网易云课堂](#) 的配套实验，推荐大家进行实验之前先学习相关课程：

- L20 内存使用与分段
- L21 内存分区与分页
- L22 段页结合的实际内存管理
- L23 请求调页内存换入
- L24 内存换出

Tips: 点击上方文字中的超链接或者输入

<https://mooc.study.163.com/course/1000003007#/info> 进入理论课程的学习。如果网易云上的课程无法查看，也可以看 Bilibili 上的 [操作系统哈尔滨工业大学李治军老师](#)。

2. 实验目的

- 深入理解操作系统的段、页式内存管理，深入理解段表、页表、逻辑地址、线性地址、物理地址等概念；
- 实践段、页式内存管理的地址映射过程；
- 编程实现段、页式内存管理上的内存共享，从而深入理解操作系统的内存管理。

3. 实验内容

本次实验的基本内容是：

- 用 Bochs 调试工具跟踪 Linux 0.11 的地址翻译（地址映射）过程，了解 IA-32 和 Linux 0.11 的内存管理机制；
- 在 Ubuntu 上编写多进程的生产者—消费者程序，用共享内存做缓冲区；
- 在信号量实验的基础上，为 Linux 0.11 增加共享内存功能，并将生产者—消费者程序移植到 Linux 0.11。

3.1 跟踪地址翻译过程

首先以汇编级调试的方式启动 Bochs，引导 Linux 0.11，在 0.11 下编译和运行 test.c。它是一个无限循环的程序，永远不会主动退出。然后在调试器中通过查看各项系统参数，从逻辑地址、LDT 表、GDT 表、线性地址到页表，计算出变量 i 的物理地址。最后通过直接修改物理内存的方式让 test.c 退出运行。

test.c 的代码如下：

```
#include <stdio.h>
int i = 0x12345678;
```

```
int main(void)
{
    printf("The logical/virtual address of i is 0x%08x", &i);
    fflush(stdout);
    while (i)
        ;
    return 0;
}
```

3.2 基于共享内存的生产者—消费者程序

本项实验在 Ubuntu 下完成，与信号量实验中的 pc.c 的功能要求基本一致，仅有两点不同：

- 不用文件做缓冲区，而是使用共享内存；
- 生产者和消费者分别是不同的程序。生产者是 producer.c，消费者是 consumer.c。两个程序都是单进程的，通过信号量和缓冲区进行通信。

Linux 下，可以通过 shmget() 和 shmat() 两个系统调用使用共享内存。

3.3 共享内存的实现

进程之间可以通过页共享进行通信，被共享的页叫做共享内存，结构如下图所示：

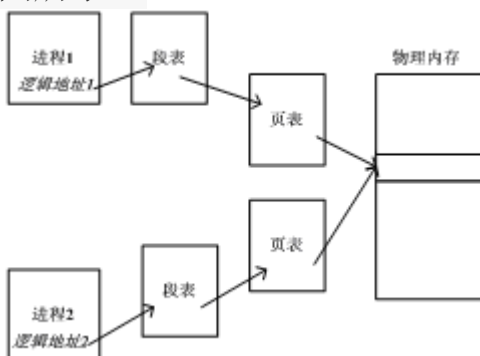


图 1 进程间共享内存的结构

本部分实验内容是在 Linux 0.11 上实现上述页面共享，并将上一部分实现的 producer.c 和 consumer.c 移植过来，验证页面共享的有效性。具体要求在 mm/shm.c 中实现 shmget() 和 shmat() 两个系统调用。它们能支持 producer.c 和 consumer.c 的运行即可，不需要完整地实现 POSIX 所规定的功能。

- shmget()

```
int shmget(key_t key, size_t size, int shmflg);
```

shmget() 会新建/打开一页内存，并返回该页共享内存的 shmid（该块共享内存存在操作系统内部的 id）。

所有使用同一块共享内存的进程都要使用相同的 key 参数。

如果 key 所对应的共享内存已经建立，则直接返回 shmid。如果 size 超过一页内存的大小，返回 -1，并置 errno 为 EINVAL。如果系统无空

闲内存，返回 -1，并置 `errno` 为 `ENOMEM`。

`shmflg` 参数可忽略。

- `shmat()`

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

`shmat()` 会将 `shmid` 指定的共享页面映射到当前进程的虚拟地址空间中，并将其首地址返回。

如果 `shmid` 非法，返回 -1，并置 `errno` 为 `EINVAL`。

`shmaddr` 和 `shmflg` 参数可忽略。

4. 实验报告

完成实验后，在实验报告中回答如下问题：

- 对于地址映射实验部分，列出你认为最重要的那几步（不超过 4 步），并给出你获得的实验数据。
- `test.c` 退出后，如果马上再运行一次，并再进行地址跟踪，你发现有哪些异同？为什么？

5. 评分标准

- 跟踪地址映射的过程，20%
- `shmget()`，10%
- `shmat()`，10%
- `producer.c`，15%
- `consumer.c`，15%
- 实验报告，30%

6. 实验提示

本次需要完成的内容：

- （1）用 Bochs 调试工具跟踪 Linux 0.11 的地址翻译（地址映射）过程，了解 IA-32 和 Linux 0.11 的内存管理机制；
- （2）在 Ubuntu 上编写多进程的生产者—消费者程序，用共享内存做缓冲区；
- （3）在信号量实验的基础上，为 Linux 0.11 增加共享内存功能，并将生产者—消费者程序移植到 Linux 0.11。

6.1 IA-32 的地址翻译过程

Linux 0.11 完全遵循 IA-32 (Intel Architecture 32-bit) 架构进行地址翻译，Windows、后续版本的 Linux 以及一切在 IA-32 保护模式下运行的操作系统都遵循此架构。因为只有这样才能充分发挥 CPU 的 MMU（内存管理单元）的功能。

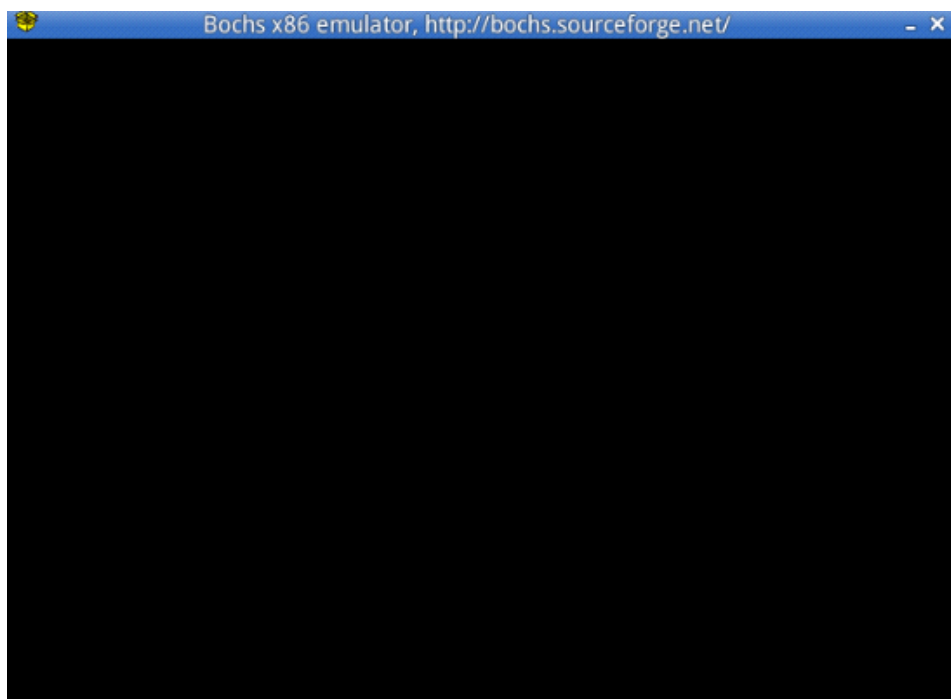
关于此地址翻译过程的细节，请参考《注释》一书中的 5.3.1–5.3.4 节。

6.2 用 Bochs 汇编级调试功能进行人工地址翻译

此过程比较机械，基本不消耗脑细胞，做一下有很多好处。

（1）准备

编译好 Linux 0.11 后，首先通过运行 `./dbg-asm` 启动调试器，此时 Bochs 的窗口处于黑屏状态



而命令行窗口显示：

```
shiyancelou@b3fe3ed157c2: ~/oslab
shiyancelou@b3fe3ed157c2:~/oslab$ ./dbg-asm
=====
Bochs x86 Emulator 2.3.7
Build from CVS snapshot, on June 3, 2008
=====
000000000000i[      ] reading configuration from ./bochs/bochsrc.bxrc
000000000000i[      ] installing x module as the Bochs GUI
000000000000i[      ] using log file ./bochsout.txt
Next at t=0
(0) [0xffffffff] f000:fff0 (unk. ctxt): jmp far f000:e05b      ; ea5b
e000f0
<bochs:1>
```

```
=====
Bochs x86 Emulator 2.3.7
Build from CVS snapshot, on June 3, 2008
=====
000000000000i[      ] reading configuration
from ./bochs/bochsrc.bxrc
000000000000i[      ] installing x module as the Bochs GUI
000000000000i[      ] using log file ./bochsout.txt
Next at t=0
(0) [0xffffffff] f000:fff0 (unk. ctxt): jmp far
f000:e05b      ; ea5be000f0
<bochs:1>_
```

Next at t=0 表示下面的指令是 Bochs 启动后要执行的第一条软件指令。

单步跟踪进去就能看到 BIOS 的代码。不过这不是本实验需要的。直接输入命令 c, continue 程序的运行, Bochs 一如既往地启动了 Linux 0.11。

在 Linux 0.11 下输入（或拷入）test.c（代码在本实验的第 3 小节中），编译为 test，运行之，打印如下信息：

```
The logical/virtual address of i is 0x00003004
```

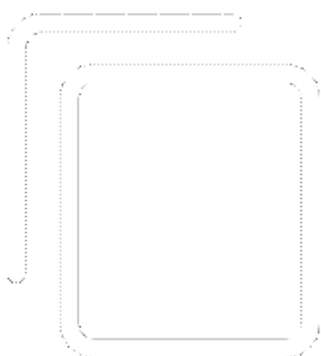
只要 test 不变，0x00003004 这个值在任何人的机器上都是一样的。即使是在同一个机器上多次运行 test，也是一样的。

test 是一个死循环，只会不停占用 CPU，不会退出。

（2）暂停

当 test 运行的时候，在命令行窗口按 Ctrl+c，Bochs 会暂停运行，进入调试状态。绝大多数情况下都会停在 test 内，显示类似如下信息：

```
(0) [0x00fc8031] 000f:00000031 (unk. ctxt): cmp dword ptr ds:0x3004, 0x00000000 ; 833d04300000000
```



其中的 000f 如果是 0008，则说明中断在了内核里。那么就要 c，然后再 ctrl+c，直到变为 000f 为止。

如果显示的下一条指令不是 cmp ...（这里指语句以 cmp 开头），就用 n 命令单步运行几步，直到停在 cmp ...。

使用命令 u /8，显示从当前位置开始 8 条指令的反汇编代码，结构如下：

```
shiyanolou@5ff91ed70775: ~/oslab
shiyanolou@5ff91ed70775:~/oslab$ ./dbg-asm

=====
Bochs x86 Emulator 2.3.7
Build from CVS snapshot, on June 3, 2008
=====

00000000000i[      ] reading configuration from ./bochs/bochsrc.bxrc
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file ./bochsout.txt
Next at t=0
(0) [0xfffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b          ; ea5b
e000f0
<bochs:1> c
^CNext at t=743199410
(0) [0x00fa706c] 000f:0000006c (unk. ctxt): jmp .+0xffffffff5 (0x10000063) ; ebf5
) ; ebf5
<bochs:2> n ← 使用 n 进行单步，直到出现
Next at t=743199411
(0) [0x00fa7063] 000f:00000063 (unk. ctxt): cmp dword ptr ds:0x3004, 0x0
0000000 ; 833d04300000000
<bochs:3> u /8
10000063: (          ) : cmp dword ptr ds:0x3004, 0x00000000 ;
833d0430000000
1000006a: (          ) : jz .+0x00000004          ; 7404
1000006c: (          ) : jmp .+0xffffffff5       ; ebf5
1000006e: (          ) : add byte ptr ds:[eax], al ; 0000
10000070: (          ) : xor eax, eax            ; 31c0
10000072: (          ) : jmp .+0x00000000        ; eb00
10000074: (          ) : leave                   ; c9
10000075: (          ) : ret                     ; c3
<bochs:4>
```

```

<bochs:3> u /8
10000063: ( ): cmp dword ptr ds:0x3004,
0x00000000 ; 833d0430000000
1000006a: ( ): jz .+0x00000004 ;
7404
1000006c: ( ): jmp .+0xffffffff5 ;
ebf5
1000006e: ( ): add byte ptr ds:[eax], al ;
0000
10000070: ( ): xor eax, eax ;
31c0
10000072: ( ): jmp .+0x00000000 ;
eb00
10000074: ( ): leave ; c9
10000075: ( ): ret ; c3

```

这就是 test.c 中从 while 开始一直到 return 的汇编代码。变量 i 保存在 ds:0x3004 这个地址，并不停地和 0 进行比较，直到它为 0，才会跳出循环。

现在，开始寻找 ds:0x3004 对应的物理地址。

6.3 段表

ds:0x3004 是虚拟地址，ds 表明这个地址属于 ds 段。首先要找到段表，然后通过 ds 的值在段表中找到 ds 段的具体信息，才能继续进行地址翻译。

每个在 IA-32 上运行的应用程序都有一个段表，叫 LDT，段的信息叫段描述符。

LDT 在哪里呢？ldtr 寄存器是线索的起点，通过它可以在 GDT（全局描述符表）中找到 LDT 的物理地址。

用 sreg 命令（是在调试窗口输入）：

```

<bochs:4> sreg
cs:s=0x000f, dl=0x00000002, dh=0x10c0fa00, valid=1
ds:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=3
ss:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
es:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
fs:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
gs:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
ldtr:s=0x0068, dl=0xa2d00068, dh=0x000082fa, valid=1
tr:s=0x0060, dl=0xa2e80068, dh=0x00008bfa, valid=1
gdtr:base=0x00005cb8, limit=0x7ff
idtr:base=0x000054b8, limit=0x7ff

```

可以看到 ldtr 的值是 0x0068=0000000001101000（二进制），表示 LDT

表存放在 GDT 表的 1101（二进制）=13（十进制）号位置（每位数据的意义参考后文叙述的段选择子）。

而 GDT 的位置已经由 gdttr 明确给出，在物理地址的 0x00005cb8。

用 `xp /32w 0x00005cb8` 查看从该地址开始，32 个字的内容，及 GDT 表的前 16 项，如下：

```
<bochs:5> xp /32w 0x00005cb8
```

```
[bochs]:
```

```
0x00005cb8 <bogus+ 0>: 0x00000000 0x00000000
```

```
0x00000fff 0x00c09a00
```

```
0x00005cc8 <bogus+ 16>: 0x00000fff 0x00c09300
```

```
0x00000000 0x00000000
```

```
0x00005cd8 <bogus+ 32>: 0xa4280068 0x00008901
```

```
0xa4100068 0x00008201
```

```
0x00005ce8 <bogus+ 48>: 0xf2e80068 0x000089ff
```

```
0xf2d00068 0x000082ff
```

```
0x00005cf8 <bogus+ 64>: 0xd2e80068 0x000089ff
```

```
0xd2d00068 0x000082ff
```

```
0x00005d08 <bogus+ 80>: 0x12e80068 0x000089fc
```

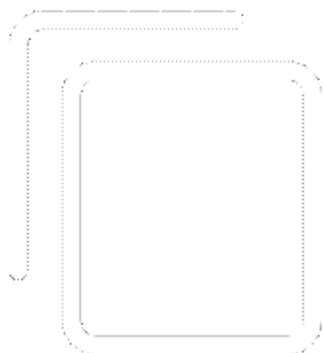
```
0x12d00068 0x000082fc
```

```
0x00005d18 <bogus+ 96>: 0xa2e80068 0x00008bfa
```

```
0xa2d00068 0x000082fa
```

```
0x00005d28 <bogus+ 112>: 0xc2e80068 0x000089f8
```

```
0xc2d00068 0x000082f8
```



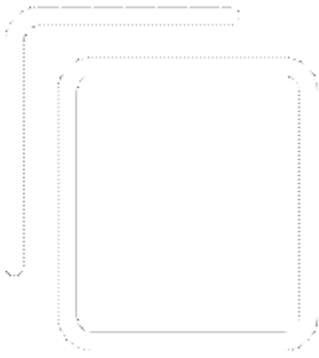
GDT 表中的每一项占 64 位（8 个字节），所以我们要查找的项的地址是 $0x00005cb8 + 13 \times 8$ 。

输入 `xp /2w 0x00005cb8+13*8`，得到：

```
<bochs:6> xp /2w 0x00005cb8+13*8
```

```
[bochs]:
```

```
0x00005d20 <bogus+ 0>: 0xa2d00068 0x000082fa
```



上两步看到的数值可能和这里给出的示例不一致，这是很正常的。如果想确认是否准确，就看 sreg 输出中，ldtr 所在行里，dl 和 dh 的值，它们是 Bochs 的调试器自动计算出的，你寻找到的必须和它们一致。

“0xa2d00068 0x000082fa” 将其中的加粗数字组合为“0x00faa2d0”，这就是 LDT 表的物理地址（为什么这么组合，参考后文介绍的段描述符）。

xp /8w 0x00faa2d0，得到：

```
<bochs:7> xp /8w 0x00faa2d0
```

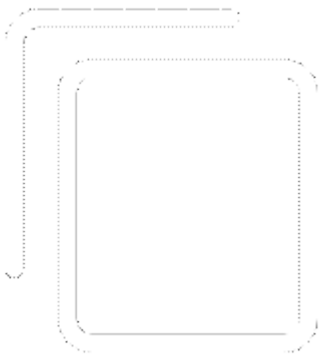
```
[bochs]:
```

```
0x00faa2d0 <bogus+      0>:    0x00000000    0x00000000
```

```
0x00000002    0x10c0fa00
```

```
0x00faa2e0 <bogus+    16>:    0x00003fff    0x10c0f300
```

```
0x00000000    0x00fab000
```



这就是 LDT 表的前 4 项内容了。

6.4 段描述符

在保护模式下，段寄存器有另一个名字，叫段选择子，因为它保存的信息主要是该段在段表里索引值，用这个索引值可以从段表中“选择”出相应的段描述符。

先看看 ds 选择子的内容，还是用 sreg 命令：

```
<bochs:8> sreg
```

```
cs:s=0x000f, dl=0x00000002, dh=0x10c0fa00, valid=1
```

```
ds:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=3
```

```
ss:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
```

```
es:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
```

```
fs:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
```

```
gs:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
ldtr:s=0x0068, dl=0xa2d00068, dh=0x000082fa, valid=1
tr:s=0x0060, dl=0xa2e80068, dh=0x00008bfa, valid=1
gdtr:base=0x00005cb8, limit=0x7ff
idtr:base=0x000054b8, limit=0x7ff
```

可以看到，ds 的值是 0x0017。段选择子是一个 16 位寄存器，它各位的含义如下图：

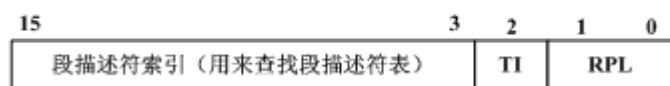


图 2 段选择子的结构

其中 RPL 是请求特权级，当访问一个段时，处理器要检查 RPL 和 CPL（放在 cs 的位 0 和位 1 中，用来表示当前代码的特权级），即使程序有足够的特权级（CPL）来访问一个段，但如果 RPL（如放在 ds 中，表示请求数据段）的特权级不足，则仍然不能访问，即如果 RPL 的数值大于 CPL（数值越大，权限越小），则用 RPL 的值覆盖 CPL 的值。而段选择子中的 TI 是表指示标记，如果 TI=0，则表示段描述符（段的详细信息）在 GDT（全局描述符表）中，即去 GDT 中去查；而 TI=1，则去 LDT（局部描述符表）中去查。

看看上面的 ds，0x0017=00000000000010111（二进制），所以 RPL=11，可见是在最低的特权级（因为在应用程序中执行），TI=1，表示查找 LDT 表，索引值为 10（二进制）= 2（十进制），表示找 LDT 表中的第 3 个段描述符（从 0 开始编号）。

LDT 和 GDT 的结构一样，每项占 8 个字节。所以第 3 项 0x00003fff 0x10c0f300（上一步骤的最后一个输出结果中）就是搜寻好久的 ds 的段描述符了。

用 sreg 输出中 ds 所在行的 dl 和 dh 值可以验证找到的描述符是否正确。

接下来看看段描述符里面放置的是什么内容：

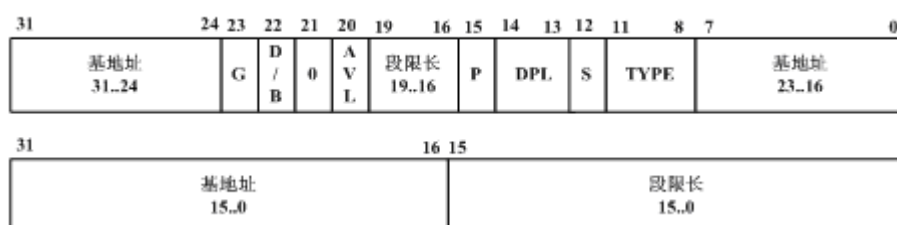


图 3 段描述符的结构

可以看到，段描述符是一个 64 位二进制的数，存放了段基址和段限长等重要数据。其中位 P（Present）是段是否存在的标记；位 S 用来表示是系统段描述符（S=0）还是代码或数据段描述符（S=1）；四位 TYPE 用来表示段的类型，如数据段、代码段、可读、可写等；DPL 是段的权限，和 CPL、RPL 对应使用；位 G 是粒度，G=0 表示段限长以位为单位，G=1 表示段限长以 4KB 为单位；其他内容就不详细解释了。

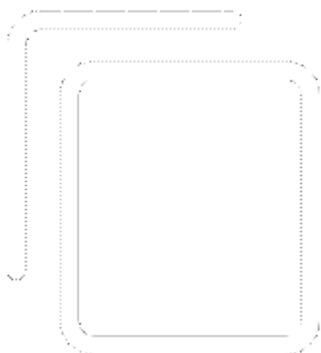
6.5 段基址和线性地址

费了很大的劲，实际上我们需要的只有段基址一项数据，即段描述符

“0x00003fff 0x10c0f300” 中加粗部分组合成的 “0x10000000”。这就是 ds 段在线性地址空间中的起始地址。用同样的方法也可以算算其它段的基址，都是这个数。

段基址+段内偏移，就是线性地址了。所以 ds:0x3004 的线性地址就是：

$$0x10000000 + 0x3004 = 0x10003004$$



用 `calc ds:0x3004` 命令可以验证这个结果。

6.6 页表

从线性地址计算物理地址，需要查找页表。线性地址变成物理地址的过程如下：

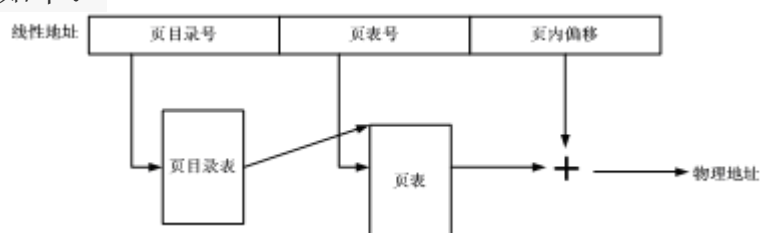


图 4 页表工作原理

线性地址变成物理地址

首先需要算出线性地址中的页目录号、页表号和页内偏移，它们分别对应了 32 位线性地址的 10 位 + 10 位 + 12 位，所以 0x10003004 的页目录号是 64，页号 3，页内偏移是 4。

IA-32 下，页目录表的位置由 CR3 寄存器指引。“`creg`”命令可以看到：

```
CR0=0x8000001b: PG cd nw ac wp ne ET TS em MP PE
```

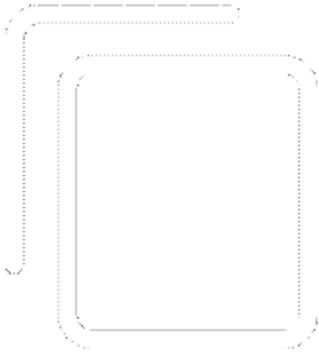
```
CR2=page fault laddr=0x10002f68
```

```
CR3=0x00000000
```

```
PCD=page-level cache disable=0
```

```
PWT=page-level writes transparent=0
```

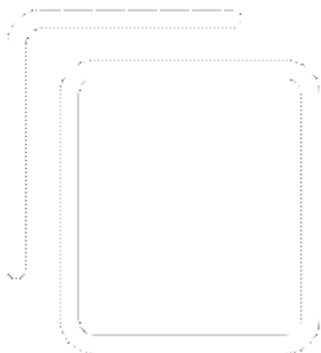
```
CR4=0x00000000: osxmmexcpt osfxsr pce pge mce pae pse de tsd pvi vme
```



说明页目录表的基址为 0。看看其内容，“xp /68w 0”：

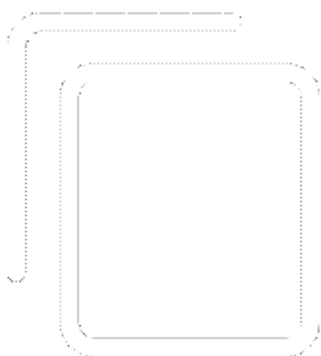
0x00000000	:	0x00001027	0x00002007	0x00003007
0x00004027				
0x00000010	:	0x00000000	0x00024764	0x00000000
0x00000000				
0x00000020	:	0x00000000	0x00000000	0x00000000
0x00000000				
0x00000030	:	0x00000000	0x00000000	0x00000000
0x00000000				
0x00000040	:	0x00ffe027	0x00000000	0x00000000
0x00000000				
0x00000050	:	0x00000000	0x00000000	0x00000000
0x00000000				
0x00000060	:	0x00000000	0x00000000	0x00000000
0x00000000				
0x00000070	:	0x00000000	0x00000000	0x00000000
0x00000000				
0x00000080	:	0x00ff3027	0x00000000	0x00000000
0x00000000				
0x00000090	:	0x00000000	0x00000000	0x00000000
0x00000000				
0x000000a0	:	0x00000000	0x00000000	0x00000000
0x00000000				
0x000000b0	:	0x00000000	0x00000000	0x00000000
0x00ffb027				
0x000000c0	:	0x00ff6027	0x00000000	0x00000000
0x00000000				
0x000000d0	:	0x00000000	0x00000000	0x00000000
0x00000000				
0x000000e0	:	0x00000000	0x00000000	0x00000000
0x00000000				
0x000000f0	:	0x00000000	0x00000000	0x00000000
0x00ffa027				
0x00000100	:	0x00faa027	0x00000000	0x00000000

0x00000000



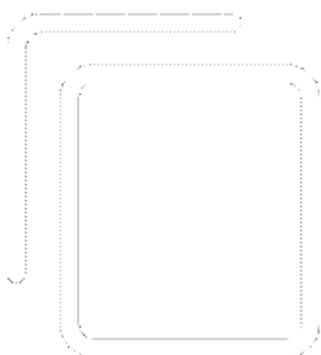
页目录表和页表中的内容很简单，是 1024 个 32 位（正好是 4K）数。这 32 位中前 20 位是物理页框号，后面是一些属性信息（其中最重要的是最后一位 P）。其中第 65 个页目录项就是我们要找的内容，用“xp /w 0+64*4”查看：

0x00000100 : 0x00faa027



其中的 027 是属性，显然 P=1，其他属性实验者自己分析吧。页表所在物理页框号为 0x00faa，即页表在物理内存的 0x00faa000 位置。从该位置开始查找 3 号页表项，得到（xp /w 0x00faa000+3*4）：

0x00faa00c : 0x00fa7067



其中 067 是属性，显然 P=1，应该是这样。

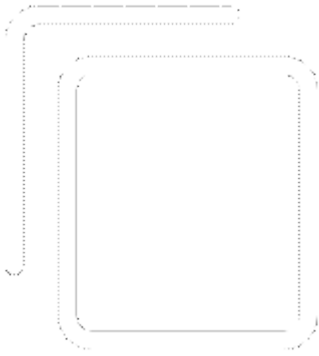
6.7 物理地址

最终结果马上就要出现了！

线性地址 0x10003004 对应的物理页框号为 0x00fa7，和页内偏移 0x004 接到一起，得到 0x00fa7004，这就是变量 i 的物理地址。可以通过两种方法验证。

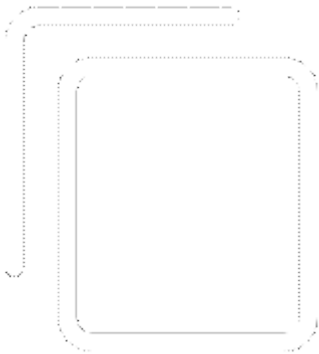
第一种方法是用命令 page 0x10003004，可以得到信息：

linear page 0x10003000 maps to physical page 0x00fa7000



第二种方法是用命令 `xp /w 0x00fa7004`，可以看到：

`0x00fa7004 : 0x12345678`



这个数值确实是 `test.c` 中 `i` 的初值。

现在，通过直接修改内存来改变 `i` 的值为 0，命令是：`setpmem 0x00fa7004 4 0`，表示从 `0x00fa7004` 地址开始的 4 个字节都设为 0。然后再用“c”命令继续 Bochs 的运行，可以看到 `test` 退出了，说明 `i` 的修改成功了，此项实验结束。

6.8 在 Linux 0.11 中实现共享内存

(1) Linux 中的共享内存

Linux 支持两种方式的共享内存。一种方式

是 `shm_open()`、`mmap()` 和 `shm_unlink()` 的组合；另一种方式是 `shmget()`、`shmat()` 和 `shmdt()` 的组合。本实验建议使用后一种方式。

这些系统调用的详情，请查阅 `man` 及相关资料。

特别提醒：没有父子关系的进程之间进行共享内存，`shmget()` 的第一个参数 `key` 不要用 `IPC_PRIVATE`，否则无法共享。用什么数字可视心情而定。

(2) 获得空闲物理页面

实验者需要考虑如何实现页面共享。首先看一下 Linux 0.11 如何操作页面，如何管理进程地址空间。

在 `kernel/fork.c` 文件中有：

```
int copy_process(...)  
{  
    struct task_struct *p;
```

```

    p = (struct task_struct *) get_free_page();
    if (!p)
        return -EAGAIN;
//     .....
}

```

函数 `get_free_page()` 用来获得一个空闲物理页面，在 `mm/memory.c` 文件中：

```

unsigned long get_free_page(void)
{
    register unsigned long __res asm("ax");
    __asm__ ("std ; repne ; scasb\n\t"
            "jne 1f\n\t"
            "movb $1,1(%%edi)\n\t"
            // 页面数*4KB=相对页面起始地址
            "sall $12,%%ecx\n\t"
            // 在加上低端的内存地址，得到的是物理起始地址
            "addl %2,%%ecx\n\t"
            "movl %%ecx,%%edx\n\t"
            "movl $1024,%%ecx\n\t"
            "leal 4092(%%edx),%%edi\n\t"
            "rep ; stosl\n\t"
            //edx赋给eax, eax返回了物理起始地址
            "movl %%edx,%%eax\n\t"
            "1: : "=a" (__res) : "0" (0), "i" (LOW_MEM), "c"
(PAGING_PAGES),
            "D" (mem_map+PAGING_PAGES-1): "di", "cx", "dx");
    return __res;
}

static unsigned char mem_map [ PAGING_PAGES ] = {0,};

```

显然 `get_free_page` 函数就是在 `mem_map` 位图中寻找值为 0 的项（空闲页面），该函数返回的是该页面的起始物理地址。

（3）地址映射

有了空闲的物理页面，接下来需要完成线性地址和物理页面的映射，Linux 0.11 中也有这样的代码，看看 `mm/memory.c` 中的 `do_no_page(unsigned long address)`，该函数用来处理线性地址 `address` 对应的物理页面无效的情况（即缺页中断），`do_no_page` 函数中调用一个重要的函数 `get_empty_page(address)`，其中有：

```

// 函数 get_empty_page(address)
unsigned long tmp=get_free_page();
// 建立线性地址和物理地址的映射

```

```
put_page(tmp, address);
```

显然这两条语句就用来获得空闲物理页面，然后填写线性地址 address 对应的页目录和页表。

(4) 寻找空闲的虚拟地址空间

有了空闲物理页面，也有了建立线性地址和物理页面的映射，但要完成本实验还需要能获得一段空闲的虚拟地址空间。

要从数据段中划出一段空间，首先需要了解进程数据段空间的分布，而这个分布显然是由 exec 系统调用决定的，所以要详细看一看 exec 的核心代码，do_execve（在文件 fs/exec.c 中）。

在函数 do_execve() 中，修改数据段（当然是修改 LDT）的地方是 change_ldt，函数 change_ldt 实现如下：

```
static unsigned long change_ldt(unsigned long text_size, unsigned
long * page)
{
    /*其中text_size是代码段长度，从可执行文件的头部取出，page为
    参数和环境页*/
    unsigned long code_limit, data_limit, code_base, data_base;
    int i;
    code_limit = text_size + PAGE_SIZE - 1;
    code_limit &= 0xFFFFF000;
    //code_limit为代码段限长=text_size对应的页数（向上取整）
    data_limit = 0x4000000; //数据段限长64MB
    code_base = get_base(current->ldt[1]);
    data_base = code_base;
    // 数据段基址 = 代码段基址
    set_base(current->ldt[1], code_base);
    set_limit(current->ldt[1], code_limit);
    set_base(current->ldt[2], data_base);
    set_limit(current->ldt[2], data_limit);
    __asm__("pushl $0x17\n\tpop %%fs"::);
    // 从数据段的末尾开始
    data_base += data_limit;
    // 向前处理
    for (i = MAX_ARG_PAGES - 1; i >= 0; i--) {
        // 一次处理一页
        data_base -= PAGE_SIZE;
        // 建立线性地址到物理页的映射
        if (page[i]) put_page(page[i], data_base);
    }
    // 返回段界限
    return data_limit;
}
```

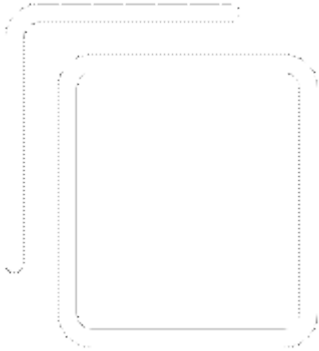
}

仔细分析过函数 `change_ldt`，想必实验者已经知道该如何从数据段中找到一页空闲的线性地址。《注释》中的图 13-6 也能给你很大帮助。

6.9 在同一终端中同时运行两个程序

Linux 的 shell 有后台运行程序的功能。只要在命令的最后输入一个 `&`，命令就会进入后台运行，前台马上回到提示符，进而能运行下一个命令，例如：

```
$ sudo ./producer &  
$ sudo ./consumer
```



当运行 `./consumer` 的时候，`producer` 正在后台运行

来自 <<https://www.lanqiao.cn/mobile/courses/115/learning?id=573>>

8) 终端设备的控制

2020年9月27日 14:11

字符显示的控制

1. 课程说明

难度系数：★★☆☆☆

本实验是 [操作系统之外设与文件系统 - 网易云课堂](#) 的配套实验，推荐大家进行实验之前先学习相关课程：

- L26 I/O 与显示器
- L27 键盘

Tips: 点击上方文字中的超链接或者输

入 <https://mooc.study.163.com/course/1000002009#/info> 进入理论课程的学习。如果网易云上的课程无法查看，也可以看 Bilibili 上的 [操作系统哈尔滨工业大学李治军老师](#)。

2. 实验目的

- 加深对操作系统设备管理基本原理的认识，实践键盘中断、扫描码等概念；
- 通过实践掌握 Linux 0.11 对键盘终端和显示器终端的处理过程。

3. 实验内容

本实验的基本内容是修改 Linux 0.11 的终端设备处理代码，对键盘输入和字符显示进行非常规的控制。

在初始状态，一切如常。用户按一次 F12 后，把应用程序向终端输出所有字母都替换为“*”。用户再按一次 F12，又恢复正常。第三次按 F12，再进行输出替换。依此类推。

以 ls 命令为例：

正常情况：

```
# ls
hello.c hello.o hello
```

第一次按 F12，然后输入 ls：

```
# **
*****. * *****. * *****
```

第二次按 F12，然后输入 ls：

```
# ls
hello.c hello.o hello
```

第三次按 F12，然后输入 ls：

```
# **
*****. * *****. * *****
```

4. 实验报告

完成实验后，在实验报告中回答如下问题：

- 在原始代码中，按下 F12，中断响应后，中断服务程序会调用 func？它实现的是什么功能？
- 在你的实现中，是否把向文件输出的字符也过滤了？如果是，那么怎么能只过滤向终端输出的字符？如果不是，那么怎么能把向文件输出的字符也一并进行过滤？

5. 评分标准

- F12 切换，40%
- 输出字符隐藏，40%
- 实验报告，20%

6. 实验提示

本实验需要修改 Linux 0.11 的终端设备处理代码

（kernel/chr_drv/console.c 文件），对键盘输入和字符显示进行非常规的控制。

6.1 键盘输入处理过程

键盘 I/O 是典型的中断驱动，在 kernel/chr_drv/console.c 文件中：

```
void con_init(void) //控制台的初始化
{
    // 键盘中断响应函数设为 keyboard_interrupt
    set_trap_gate(0x21, &keyboard_interrupt);
}
```

所以每次按键有动作，keyboard_interrupt 函数就会被调用，它在文件 kernel/chr_drv/keyboard.S（注意，扩展名是大写的 S）中实现。所有与键盘输入相关的功能都是在此文件中实现的，所以本实验的部分功能也可以在此文件中实现。

简单说，keyboard_interrupt 被调用后，会将键盘扫描码做为下标，调用数组 key_table 保存的与该按键对应的响应函数。

6.2 输出字符的控制

printf() 等输出函数最终都是调用 write() 系统调用，所以控制好 write()，就能控制好输出字符。

来自 <<https://www.lanqiao.cn/mobile/courses/115/learning?id=574>>

9) proc文件系统的实现

2020年9月27日 14:12

proc 文件系统的实现

1. 课程说明

难度系数：★★★☆☆

本实验是 [操作系统之外设与文件系统 - 网易云课堂](#) 的配套实验，推荐大家进行实验之前先学习相关课程：

- L28 生磁盘的使用
- L29 用文件使用磁盘
- L30 文件使用磁盘的实现
- L31 目录与文件系统
- L32 目录解析代码实现

Tips: 点击上方文字中的超链接或者输入 <https://mooc.study.163.com/course/1000002009#/info> 进入理论课程的学习。如果网易云上的课程无法查看，也可以看 Bilibili 上的 [操作系统哈尔滨工业大学李治军老师](#)。

2. 实验目的

- 掌握虚拟文件系统的实现原理；
- 实践文件、目录、文件系统等概念。

3. 实验内容

在 Linux 0.11 上实现 procfs (proc 文件系统) 内的 psinfo 结点。当读取此结点的内容时，可得到系统当前所有进程的状态信息。例如，用 cat 命令显示 /proc/psinfo 的内容，可得到：

```
$ cat /proc/psinfo
pid    state  father counter  start_time
0      1     -1      0         0
1      1      0     28         1
4      1      1      1         73
3      1      1     27         63
6      0      4     12        817
```

```
$ cat /proc/hdinfo
total_blocks: 62000;
free_blocks: 39037;
used_blocks: 22963;
...
```

procfs 及其结点要在内核启动时自动创建。
相关功能实现在 fs/proc.c 文件内。

4. 实验报告

完成实验后，在实验报告中回答如下问题：

- 如果要求你在 psinfo 之外再实现另一个结点，具体内容自选，那么你会实现一个给出什么信息的结点？为什么？
- 一次 read() 未必能读出所有的数据，需要继续 read()，直到把数据读空为止。而数次 read() 之间，进程的状态可能会发生变化。你认为后几次 read() 传给用户的数据，应该是变化后的，还是变化前的？ + 如果是变化后的，那么用户得到的数据衔接部分是否会有混乱？如何防止混乱？ + 如果是变化前的，那么该在什么样的情况下更新 psinfo 的内容？

5. 评分标准

- 自动创建 /proc、/proc/psinfo、/proc/hdinfo、/proc/inodeinfo，20%
- psinfo 内容可读，内容符合题目要求，40%

- hdinfo 内容可读，符合题目要求，30%
- 实验报告，10%

6. 实验提示

本实验文档在 Linux 0.11 上实现 procfs (proc 文件系统) 内的 psinfo 结点。当读取 psinfo 结点的内容时，可得到系统当前所有进程的状态信息。

最后还给出来 hdinfo 结点实现的提示。

6.1 procfs 简介

正式的 Linux 内核实现了 procfs，它是一个虚拟文件系统，通常被 mount (挂载) 到 /proc 目录上，通过虚拟文件和虚拟目录的方式提供访问系统参数的机会，所以有人称它为 “了解系统信息的一个窗口”。

这些虚拟的文件和目录并没有真实地存在在磁盘上，而是内核中各种数据的一种直观表示。虽然是虚拟的，但它们都可以通过标准的系统调用 (open()、read() 等) 访问。

例如，/proc/meminfo 中包含内存使用的信息，可以用 cat 命令显示其内容：

```
$ cat /proc/meminfo
MemTotal:      384780 kB
MemFree:       13636 kB
Buffers:       13928 kB
Cached:        101680 kB
SwapCached:    132 kB
Active:        207764 kB
Inactive:      45720 kB
SwapTotal:     329324 kB
SwapFree:      329192 kB
Dirty:         0 kB
Writeback:     0 kB
.....
```

其实，Linux 的很多系统命令就是通过读取 /proc 实现的。例如 uname -a 的部分信息就来自 /proc/version，而 uptime 的部分信息来自 /proc/uptime 和 /proc/loadavg。

关于 procfs 更多的信息请访问：<http://en.wikipedia.org/wiki/Procfs>

6.2 基本思路

Linux 是通过文件系统接口实现 procfs，并在启动时自动将其 mount 到 /proc 目录上。

此目录下的所有内容都是随着系统的运行自动建立、删除和更新的，而且它们完全存在于内存中，不占用任何外存空间。

Linux 0.11 还没有实现虚拟文件系统，也就是，还没有提供增加新文件系统支持的接口。所以本实验只能在现有文件系统的基础上，通过打补丁的方式模拟一个 procfs。

Linux 0.11 使用的是 Minix 的文件系统，这是一个典型的基于 inode 的文件系统，《注释》一书对它有详细描述。它的每个文件都要对应至少一个 inode，而 inode 中记录着文件的各种属性，包括文件类型。文件类型有普通文件、目录、字符设备文件和 9b 块设备文件等。在内核中，每种类型的文件都有不同的处理函数与之对应。我们可以增加一种新的文件类型——proc 文件，并在相应的处理函数内实现 procfs 要实现的功能。

6.3 增加新文件类型

在 include/sys/stat.h 文件中定义了几种文件类型和相应的测试宏：

```
#define S_IFMT 00170000
// 普通文件
#define S_IFREG 0100000
// 块设备
#define S_IFBLK 0060000
// 目录
#define S_IFDIR 0040000
// 字符设备
```

```

#define S_IFCHR 0020000
#define S_IFIFO 0010000
//.....
// 测试 m 是否是普通文件
#define S_ISREG(m) ((m) & S_IFMT) == S_IFREG)
// 测试 m 是否是目录
#define S_ISDIR(m) ((m) & S_IFMT) == S_IFDIR)
// 测试 m 是否是字符设备
#define S_ISCHR(m) ((m) & S_IFMT) == S_IFCHR)
// 测试 m 是否是块设备
#define S_ISBLK(m) ((m) & S_IFMT) == S_IFBLK)
#define S_ISFIFO(m) ((m) & S_IFMT) == S_IFIFO)

```

增加新的类型的方法分两步：

- （1）定义一个类型宏 `S_IFPROC`，其值应在 0010000 到 0100000 之间，但后四位八进制数必须是 0（这是 `S_IFMT` 的限制，分析测试宏可知原因），而且不能和已有的任意一个 `S_IFXXX` 相同；
- （2）定义一个测试宏 `S_ISPROC(m)`，形式仿照其它的 `S_ISXXX(m)`

注意，C 语言中以 “0” 直接接数字的常数是八进制数。

6.4 让 `mknod()` 支持新的文件类型

`psinfo` 结点要通过 `mknod()` 系统调用建立，所以要让它支持新的文件类型。

直接修改 `fs/namei.c` 文件中的 `sys_mknod()` 函数中的一行代码，如下：

```

if (S_ISBLK(mode) || S_ISCHR(mode) || S_ISPROC(mode))
    inode->i_zone[0] = dev;
// 文件系统初始化

```

内核初始化的全部工作是在 `main()` 中完成，而 `main()` 在最后从内核态切换到用户态，并调用 `init()`。

`init()` 做的第一件事情就是挂载根文件系统：

```

void init(void)
{
    // .....
    setup((void *) &drive_info);
    // .....
}

```

`procfs` 的初始化工作应该在根文件系统挂载之后开始。它包括两个步骤：

- （1）建立 `/proc` 目录；建立 `/proc` 目录下的各个结点。本实验只建立 `/proc/psinfo`。
- （2）建立目录和结点分别需要调用 `mkdir()` 和 `mknod()` 系统调用。因为初始化时已经在用户态，所以不能直接调用 `sys_mkdir()` 和 `sys_mknod()`。必须在初始化代码所在文件中实现这两个系统调用的用户态接口，即 API：

```

#ifndef __LIBRARY__
#define __LIBRARY__
#endif
_syscall2(int, mkdir, const char*, name, mode_t, mode)
_syscall3(int, mknod, const char*, filename, mode_t, mode, dev_t, dev)

```

`mkdir()` 时 `mode` 参数的值可以是 “0755”（对应 `rw-r--r--`），表示只允许 `root` 用户改写此目录，其它人只能进入和读取此目录。

`procfs` 是一个只读文件系统，所以用 `mknod()` 建立 `psinfo` 结点时，必须通过 `mode` 参数将其设为只读。建议使用 `S_IFPROC|0444` 做为 `mode` 值，表示这是一个 `proc` 文件，权限为 0444 (`r--r--r--`)，对所有用户只读。

`mknod()` 的第三个参数 `dev` 用来说明结点所代表的设备编号。对于 `procfs` 来说，此编号可以完全自定

义。proc 文件的处理函数将通过这个编号决定对应文件包含的信息是什么。例如，可以把 0 对应 psinfo, 1 对应 meminfo, 2 对应 cpuinfo。

如此项工作完成得没有问题，那么编译、运行 0.11 内核后，用 ll /proc 可以看到：

```
# ll /proc
total 0
?r--r--r-- 1 root    root          0 ??? ?  ??? psinfo
```

此时可以试着读一下此文件：

```
# cat /proc/psinfo
(Read)inode->i_mode=XXX444
cat: /proc/psinfo: EINVAL
```

inode->i_mode 就是通过 mknod() 设置的 mode。信息中的 XXX 和你设置的 S_IFPROC 有关。通过此值可以了解 mknod() 工作是否正常。这些信息说明内核在对 psinfo 进行读操作时不能正确处理，向 cat 返回了 EINVAL 错误。因为还没有实现处理函数，所以这是很正常的。

这些信息至少说明，psinfo 被正确 open() 了。所以我们不需要对 sys_open() 动任何手脚，唯一要打补丁的，是 sys_read()。

6.5 让 proc 文件可读

open() 没有变化，那么需要修改的就是 sys_read() 了。

首先分析 sys_read (在文件 fs/read_write.c 中)：

```
int sys_read(unsigned int fd, char * buf, int count)
{
    struct file * file;
    struct m_inode * inode;
    // .....
    inode = file->f_inode;
    if (inode->i_pipe)
        return (file->f_mode&1)?read_pipe(inode, buf, count):-EIO;
    if (S_ISCHR(inode->i_mode))
        return rw_char(READ, inode->i_zone[0], buf, count, &file->f_pos);
    if (S_ISBLK(inode->i_mode))
        return block_read(inode->i_zone[0], &file->f_pos, buf, count);
    if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
        if (count+file->f_pos > inode->i_size)
            count = inode->i_size - file->f_pos;
        if (count<=0)
            return 0;
        return file_read(inode, file, buf, count);
    }
    printk("(Read) inode->i_mode=%06o\n\r", inode->i_mode); //这条信息很面善吧?
    return -EINVAL;
}
```

显然，要在这里一群 if 的排比中，加上 S_IFPROC() 的分支，进入对 proc 文件的处理函数。需要传给处理函数的参数包括：

- inode->i_zone[0]，这就是 mknod() 时指定的 dev ——设备编号
- buf，指向用户空间，就是 read() 的第二个参数，用来接收数据
- count，就是 read() 的第三个参数，说明 buf 指向的缓冲区大小
- &file->f_pos, f_pos 是上一次读文件结束时“文件位置指针”的指向。这里必须传指针，因为处理函数需要根据传给 buf 的数据量修改 f_pos 的值。

6.6 proc 文件的处理函数

proc 文件的处理函数的功能是根据设备编号，把不同的内容写入到用户空间的 buf。写入的数据要

从 `f_pos` 指向的位置开始，每次最多写 `count` 个字节，并根据实际写入的字节数调整 `f_pos` 的值，最后返回实际写入的字节数。当设备编号表明要读的是 `psinfo` 的内容时，就要按照 `psinfo` 的形式组织数据。

实现此函数可能要用到如下几个函数：

- `malloc()` 函数
- `free()` 函数

包含 `linux/kernel.h` 头文件后，就可以使用 `malloc()` 和 `free()` 函数。它们是可以被核心态代码调用的，唯一的限制是一次申请的内存大小不能超过一个页面。

6.7 实现 `sprintf()` 函数

Linux 0.11 没有 `sprintf()`，可以参考 `printf()` 自己实现一个。

可以借鉴如下代码：

```
#include <stdarg.h>
//.....
int sprintf(char *buf, const char *fmt, ...)
{
    va_list args; int i;
    va_start(args, fmt);
    i=vsprintf(buf, fmt, args);
    va_end(args);
    return i;
}
```

6.8 `cat` 命令的实现

`cat` 是 Linux 下的一个常用命令，功能是将文件的内容打印到标准输出。

它核心实现大体如下：

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char* argv[])
{
    char buf[513] = {'\0'};
    int nread;
    int fd = open(argv[1], O_RDONLY, 0);
    while(nread = read(fd, buf, 512))
    {
        buf[nread] = '\0';
        puts(buf);
    }
    return 0;
}
```

6.9 `psinfo` 的内容

进程的信息就来源于内核全局结构数组 `struct task_struct * task[NR_TASKS]` 中，具体读取细节可参照 `sched.c` 中的函数 `schedule()`。

可以借鉴一下代码：

```
for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
if (*p)
    (*p)->counter = ((*p)->counter >> 1)+...;
```

6.10 `hdinfo` 的内容

硬盘总共有多少块，多少块空闲，有多少 `inode` 等信息都放在 `super` 块中，`super` 块可以通过 `get_super()` 函数获得。

其中的信息可以借鉴如下代码：

```
struct super_block * sb;
sb = get_super(inode->i_dev);
struct buffer_head * bh;
total_blocks = sb->s_nzones;
for(i=0; is_zmap_blocks; i++)
{
    bh = sb->s_zmap[i];
```

```
    p=(char *)bh->b_data;
}
```

来自 <<https://www.lanqiao.cn/mobile/courses/115/learning?id=575>>

2) 操作系统的引导

2020年9月22日 15:17

实验内容：

此次实验的基本内容是：

1. 阅读《Linux 内核完全注释》的第 6 章，对计算机和 Linux 0.11 的引导过程进行初步的了解；
2. 按照下面的要求改写 0.11 的引导程序 bootsect.s
3. 有兴趣同学可以做做进入保护模式前的设置程序 setup.s。

改写 bootsect.s 主要完成如下功能：

1. bootsect.s 能在屏幕上打印一段提示信息“XXX is booting...”，其中 XXX 是你给自己的操作系统起的名字，例如 LZJos、Sunix 等（可以上论坛上秀秀谁的 OS 名字最帅，也可以显示一个特色 logo，以表示自己操作系统的与众不同。）

改写 setup.s 主要完成如下功能：

1. bootsect.s 能完成 setup.s 的载入，并跳转到 setup.s 开始地址执行。而 setup.s 向屏幕输出一行“Now we are in SETUP”。
2. setup.s 能获取至少一个基本的硬件参数（如内存参数、显卡参数、硬盘参数等），将其存放在内存的特定地址，并输出到屏幕上。
3. setup.s 不再加载 Linux 内核，保持上述信息显示在屏幕上即可。

在实验报告中回答如下问题：

1. 有时，继承传统意味着别手蹩脚。x86 计算机为了向下兼容，导致启动过程比较复杂。请找出 x86 计算机启动过程中，被硬件强制，软件必须遵守的两个“多此一举”的步骤（多找几个也无妨），说说它们为什么多此一举，并设计更简洁的替代方案

实验完成情况：

bootsect.s 能在屏幕上打印一段提示信息“XXX is booting...”

以下没有完成()

bootsect.s 能完成 setup.s 的载入，并跳转到 setup.s 开始地址执行。而 setup.s 向屏幕输出一行“Now we are in SETUP”。

setup.s 能获取至少一个基本的硬件参数（如内存参数、显卡参数、硬盘参数等），将其存放在内存的特定地址，并输出到屏幕上。

setup.s 不再加载 Linux 内核，保持上述信息显示在屏幕上即可。

3) 系统调用

2020年9月22日 15:19

实验内容

此次实验的基本内容是：在 Linux 0.11 上添加两个系统调用，并编写两个简单的应用程序测试它们。

(1) iam()

第一个系统调用是 iam(), 其原型为：

```
int iam(const char * name);
```

完成的功能是将字符串参数 name 的内容拷贝到内核中保存下来。要求 name 的长度不能超过 23 个字符。返回值是拷贝的字符数。如果 name 的字符个数超过了 23，则返回“-1”，并置 errno 为 EINVAL。

在 kernal/who.c 中实现此系统调用。

(2) whoami()

第二个系统调用是 whoami(), 其原型为：

```
int whoami(char* name, unsigned int size);
```

它将内核中由 iam() 保存的名字拷贝到 name 指向的用户地址空间中，同时确保不会对 name 越界访存（name 的大小由 size 说明）。返回值是拷贝的字符数。如果 size 小于需要的空间，则返回“-1”，并置 errno 为 EINVAL。

也是在 kernal/who.c 中实现。

(3) 测试程序

运行添加过新系统调用的 Linux 0.11，在其环境下编写两个测试程序 iam.c 和 whoami.c。最终的运行结果是：

```
$ ./iam lizhijun
$ ./whoami
Lizhijun
```

实验报告

在实验报告中回答如下问题：

- 从 Linux 0.11 现在的机制看，它的系统调用最多能传递几个参数？你能想出办法来扩大这个限制吗？

5.5.2 系统调用处理过程

当应用程序经过库函数向内核发出一个中断调用 int 0x80 时，就开始执行一个系统调用。其中寄存器 eax 中存放着系统调用号，而携带的参数可依次存放在寄存器 ebx、ecx 和 edx 中。因此 Linux 0.12 内核中用户程序能够向内核最多直接传递三个参数，当然也可以不带参数。处理系统调用中断 int 0x80 的过程是程序 kernel/system_call.s 中的 system_call。

为了方便执行系统调用，内核源代码在 include/unistd.h 文件(150—200 行)中定义了宏函数 syscalln()，其中 n 代表携带的参数个数，可以分别 0 至 3。因此最多可以直接传递 3 个参数。若需要传递大块数据给内核，则可以传递这块数据的指针值，例如对于 read() 系统调用，其定义是：

```
int read(int fd, char *buf, int n);
```

- 用文字简要描述向 Linux 0.11 添加一个系统调用 foo() 的步骤。

1. 添加系统调用号，在模拟器 bochs 中 include/unistd.h 中定义：

2. 也就是在中断 0x80 发生后，自动调用函数 `system_call`
3. `sys_call_table` 一定是一个函数指针数组的起始地址，它定义在 `include/linux/sys.h`
4. 增加实验要求的系统调用，需要在这个函数表中增加两个函数引用 —— `sys_iam` 和 `sys_whoami`。当然该函数在 `sys_call_table` 数组中的位置必须和 `__NR_XXXXXX` 的值对应上。
5. 同时还要仿照此文件中前面各个系统调用的写法，加上：

```
extern int sys_whoami();
extern int sys_iam();
```
6. 在内核中实现 `foo ()`

实验完成情况：

完整完成

4) 进程运行轨迹的跟踪与统计

2020年9月22日 15:19

实验内容：

- 基于模板 process.c 编写多进程的样本程序，实现如下功能： + 所有子进程都并行运行，每个子进程的实际运行时间一般不超过 30 秒； + 父进程向标准输出打印所有子进程的 id，并在所有子进程都退出后才退出；
完成（wait要调用三次）
- 在 Linux0.11 上实现进程运行轨迹的跟踪。 + 基本任务是在内核中维护一个日志文件 /var/process.log，把从操作系统启动到系统关机过程中所有进程的运行轨迹都记录在这一 log 文件中。
完成不彻底，会有报错
- 在修改过的 0.11 上运行样本程序，通过分析 log 文件，统计该程序建立的所有进程的等待时间、完成时间（周转时间）和运行时间，然后计算平均等待时间，平均完成时间和吞吐量。可以自己编写统计程序，也可以使用 python 脚本程序——stat_log.py（在 /home/teacher/ 目录下）——进行统计。
没有分析
- 修改 0.11 进程调度的时间片，然后再运行同样的样本程序，统计同样的时间数据，和原有的情况对比，体会不同时间片带来的差异。
没有做

完成实验后，在实验报告中回答如下问题：

- 结合自己的体会，谈谈从程序设计者的角度看，单进程编程和多进程编程最大的区别是什么？
Cpu利用效率提高
- 你是如何修改时间片的？仅针对样本程序建立的进程，在修改时间片前后，log 文件的统计结果（不包括 Graphic）都是什么样？结合你的修改分析一下为什么会这样变化，或者为什么没变化？

5) 基于内核栈切换的进程切换

2020年9月22日 15:19

本次实践项目就是将 Linux 0.11 中采用的 TSS 切换部分去掉，取而代之的是基于堆栈的切换程序。具体的说，就是将 Linux 0.11 中的 `switch_to` 实现去掉，写成一段基于堆栈切换的代码

本次实验包括如下内容：

- 编写汇编程序 `switch_to`；
- 完成主体框架；
- 在主体框架下依次完成 PCB 切换、内核栈切换、LDT 切换等；
- 修改 `fork()`，由于是基于内核栈的切换，所以进程需要创建出能完成内核栈切换的样子。
- 修改 PCB，即 `task_struct` 结构，增加相应的内容域，同时处理由于修改了 `task_struct` 所造成的影响。
- 用修改后的 Linux 0.11 仍然可以启动、可以正常使用。
- （选做）分析实验 3 的日志体会修改前后系统运行的差别。

下一步

7) 地址映射

2020年10月20日 星期二 17:26

本次需要完成的内容：

- (1) 用 Bochs 调试工具跟踪 Linux 0.11 的地址翻译（地址映射）过程，了解 IA-32 和 Linux 0.11 的内存管理机制；
- (2) 在 Ubuntu 上编写多进程的生产者—消费者程序，用共享内存做缓冲区；
- (3) 在信号量实验的基础上，为 Linux 0.11 增加共享内存功能，并将生产者—消费者程序移植到 Linux 0.11。

bochs 调试

i 变量虚拟地址：ds:0x3004

- ds:0x3004 是虚拟地址，ds 表明这个地址属于 ds 段。首先要找到段表，然后通过 ds 的值在段表中找到 ds 段的具体信息，才能继续进行地址翻译
- 通过 LDTR 找到 LDT 表在 GDT 表中的位置，GDT 的位置已经由 gdtr 明确给出

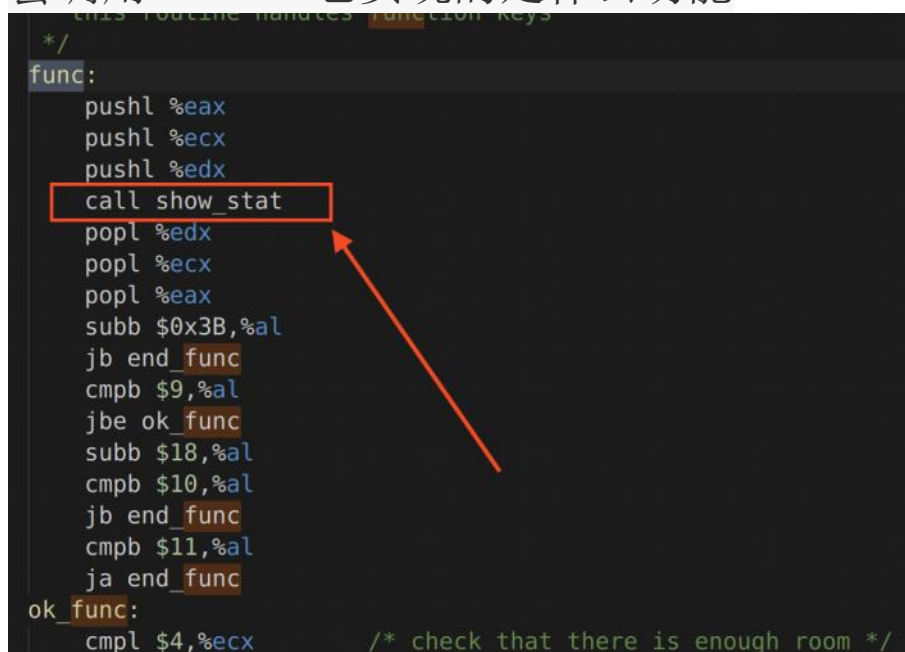
8) 终端设备

2020年11月4日 星期三 15:36

4. 实验报告

完成实验后，在实验报告中回答如下问题：

- 在原始代码中，按下 F12，中断响应后，中断服务程序会调用 func？它实现的是什么功能



```
/*  
this routine handles function keys  
*/  
func:  
    pushl %eax  
    pushl %ecx  
    pushl %edx  
    call show_stat  
    popl %edx  
    popl %ecx  
    popl %eax  
    subb $0x3B,%al  
    jb end_func  
    cmpb $9,%al  
    jbe ok_func  
    subb $18,%al  
    cmpb $10,%al  
    jb end_func  
    cmpb $11,%al  
    ja end_func  
ok_func:  
    cmpl $4,%ecx    /* check that there is enough room */
```

- 在你的实现中，是否把向文件输出的字符也过滤了？如果是，那么怎么能只过滤向终端输出的字符？如果不是，那么怎么能把向文件输出的字符也一并进行过滤？
- 没有把文件输出的字符过滤，因为sys_write那里有判断语句，可以判断是写字符设备还是文件

```

int sys_write(unsigned int fd, char * buf, int count)
{
    struct file * file;
    struct m_inode * inode;

    if (fd >= NR_OPEN || count < 0 || !(file = current->filp[fd]))
        return -EINVAL;
    if (!count)
        return 0;
    inode = file->f_inode;
    if (inode->i_pipe)
        return (file->f_mode & 2) ? write_pipe(inode, buf, count) : -EIO;
    if (S_ISCHR(inode->i_mode))
        return rw_char(WRITE, inode->i_zone[0], buf, count, &file->f_pos);
    if (S_ISBLK(inode->i_mode))
        return block_write(inode->i_zone[0], &file->f_pos, buf, count);
    if (S_ISREG(inode->i_mode))
        return file_write(inode, file, buf, count);
    printk("(Write)inode->i_mode=%06o\n\r", inode->i_mode);
    return -EINVAL;
}

```


9) proc文件系统的实现

2021年3月3日 星期三 10:10

在 Linux 0.11 上实现 procfs (proc 文件系统) 内的 psinfo 结点和 hdinfo 结点

procfs 及其结点要在内核启动时自动创建。
相关功能实现在 fs/proc.c 文件内

- 在系统中增加新的文件类型 proc 文件
- 让 mknod () 支持新的文件类型
 - 实现系统调用
 - mkdir
 - mknod
 - 使用系统调用建立目录和结点
-