

Book OS

操作系统开发者手册

Operating system developer's manual

系统版本: Ver0.2

修订日期: 2019/2/26

版权声明

BookOS 是一个基于 x86 平台的 32 位操作系统,其版权属于 BookOS 开发者所有。BookOS 使用 C 和汇编开发,我们采用开源的方式发布,其源码遵循 BSD-2 协议。BSD-2 协议声明如下:

BSD 2-Clause License

Copyright (c) 2017-2018, Mediocre Operating System Project Developers.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

目录

第一章	开发环境要求.....	- 5 -
1.1	环境需求.....	- 5 -
1.2	Windows 下的环境搭建.....	- 5 -
1.3	Linux 下的环境搭建.....	- 5 -
第二章	系统的一生.....	- 6 -
2.1	从零到写入磁盘.....	- 6 -
2.2	从引导到进入内核.....	- 6 -
2.3	从初始化到完整运行.....	- 7 -
第三章	系统的要点.....	- 9 -
3.1	loader 部分要点.....	- 9 -
3.2	init 部分要点.....	- 9 -
3.3	console 部分要点.....	- 9 -
3.4	page 部分要点.....	- 9 -
3.5	memory 部分要点.....	- 10 -
3.6	device 部分要点.....	- 10 -
3.7	fs 部分要点.....	- 10 -
3.8	gui 部分要点.....	- 10 -
3.9	syscall 部分要点.....	- 10 -
3.10	lib 部分要点.....	- 11 -
3.11	源码结构部分要点.....	- 11 -
3.12	Makefile 部分要点.....	- 12 -
3.13	内核态与用户态要点.....	- 19 -
第四章	一些问题的解决.....	- 20 -
4.1	如何添加一个中断类 device?	- 20 -
4.2	如何编写自己的图形界面?	- 20 -
4.3	如何编写自己的文件系统?	- 20 -
4.4	如何添加一个系统调用?	- 20 -
4.5	如何添加一个线程/进程?	- 21 -
第五章	应用程序开发.....	- 23 -
5.1	原理介绍.....	- 23 -

5.2 程序的编写、编译与写入磁盘.....	- 23 -
5.3 程序的载入文件系统.....	- 26 -
后记.....	- 29 -

第一章 开发环境要求

1.1 环境需求

你所需要的工具是 gcc, nasm, ld, dd, rm, make, bochs 虚拟机或者 qemu 虚拟机。你可以在 windows 和 linux 下搭建你的环境。你需要做的就是下载好工具，配置好环境变量就可以了。

1.2 Windows 下的环境搭建

Windows 下的环境配置请参考视频（自己制作）：

<https://www.bilibili.com/video/av44483615>

1.3 Linux 下的环境搭建

Linux 下的环境配置请参考如下（Boss 制作）：

How to run BookOS in linux(ubuntu linuxmint...):

1.You need setup some essential packages:

```
sudo apt-get install gcc make build-essential
```

```
sudo apt-get install qemu qemu-user-static
```

```
sudo apt-get install nasm vgabios bochs bochs-x bximage
```

2.Changing your current dictionary to src and make:

```
cd $workdir/src
```

```
make
```

3.Have fun.

到此，你就已经搭建好了开发环境，可以开始你的屌丝人生了。^^

第二章 系统的一生

2.1 从零到写入磁盘

最开始，我们把代码写成源程序，通过 `nasm` 和 `gcc` 分别编译 `.c` 和 `.asm` 文件，生成 `.o` 目标文件，通过 `ld` 连接成我们所需要的可执行文件。由于 `boot` 和 `loader` 分别只有一个文件，所以直接把他们生成可执行文件。内核的话就需要编译链接成可执行文件。于是，我们就有了 `boot/boot.bin`（二进制）、`boot/loader.bin`（二进制）和 `kernel/kernel.elf`（ELF 格式文件，其实也可以做成二进制，但 ELF 优点多，我们就选用 ELF 文件）。

于是，我们的操作系统所需要的文件就制作完成，下一步就是把他们写入磁盘，使他们可以运行。由于软盘是可以直接读取某个扇区的数据，所以我们直接把这些文件写入到一个绝对扇区位置，然后在运行的时候，从软盘对应的地方读取数据就可以了。写入工具是 `dd`，我把 `boot.bin` 写入到 0 扇区（引导必须位于 0 扇区），`loader.bin` 写到 2 扇区（偶数貌似好看点），`loader` 我固定使用 4KB（8 扇区），所以，我把内核放到了 10 扇区，对于内核的扇区数，可以根据内核的大小进行调整，目前写入 400 个扇区（即使内核没有 200KB 也不要紧，就相当于后面写入的是数据为空的扇区）。这些可以再 `src/Makefile` 中查看。

此时，我们的操作系统就成功地住在软盘上了。

2.2 从引导到进入内核

大家都知道，电脑开机后进行一系列操作之后就会检测是否有启动扇区，有的话就把启动扇区加载到 `0x7c00` 这个内存地址。然后跳到那个地方执行我们的系统。在 `BookOS` 中，`boot` 的作用就是加载 `loader`，因为 `boot` 只能存放 512 字节，所以我们把大任交给 `loader`，它先从软盘读取 `loader` 进入内存，然后跳入 `loader` 执行。这下，`loader` 就开始忙碌地工作了（真勤奋啊）。

我们的 `loader` 先从磁盘上读取内核进入内存，这之后，就没有软盘操作了，然后关闭软盘马达，不让他运转（节约用电？`Maybe`）。接下来就检测我们有多少内存，并把这些信息放入到 `ARDS` 结构体中，我们在内核中初始化内存管理的时候需要用到这些信息。下一步，初始化图形模式，根据我们定义的图形模式（在 `boot/const.inc`），切换到指定的图形模式。

“好累呀！不过还是得继续做事情。”`loader` 接下来做的便是进入保护模式，先关闭中断（以免发生中断后出现错误），然后再加 `gdt` 描述符表，再打开 `A20` 总线（让我们可以访问 `1MB` 以上的内存），然后设置 `CR0` 的最低一位为 `1` 就变成保护模式了，下一步是清空处理器流水线，用一个跳转就行。

此后，我们便进入保护模式了，现在访问内存不是段：偏移了，而是段选择子：偏移，所以先初始化数据段，栈段，其他段的的选择子和栈顶。由于数据段和代码的基地址设置成 `0`，所以偏移的地址就是对应的程序内的地址。在前面说过，我们的内核是 `elf` 格式的，不能直接执行，所以需要解析一下，于是就把内核文件的内容解析到另外一个地址，这样的话，我们就可以直接跳转到对应的地址执行内核了。“咚”，进入内核！

2.3 从初始化到完整运行

自从 loader 因为无事可做，抑郁而死之后，kernel 登基了。我们内核的入口函数是 `_start`，所以内核先会在 `_start` 里面执行。在 loader 里面在此初始化段选择子以及栈顶（loader 栈顶和 kernel 栈顶不一样），然后调用 `main` 函数进入 `init/main.c` 中的 `main` 函数初始化整个系统。

首先是初始化系统的分页机制，从此我们访问内存就需要访问页目录表内定义了地址才行。接下来是初始化 GDT/IDT，我们重新定义了 GDT，并且初始化了 IDT，于是就能够正确处理系统产生的异常和外部产生的中断（IRQ 中断和系统调用）。接下来是初始化控制台，这个控制台主要用于文本模式下输出信息，这是没有开启图形模式前用于输出信息的初始化，现在有图形了，可以把这部分去掉，但是不能忘了本，有新欢就把旧爱抛弃了，万一哪天图形出问题了，咱们还可以回到文本模式做测试对吧（应该不会出问题，毕竟我们是最棒的开发者！）

接下来，初始化内存管理模块，内存管理模块是根据分页来进一步进行管理的。分页管理中提供了可以分配/释放整个页（1 PAGE=4096 bytes= 4 kb）的函数。因此内存管理是基于分页管理来的，我们可以分配更小的空间，也可以分配更大的空间。接下来初始化图形模式的基本信息，获取屏幕的宽高，显存地址，像素宽度等。并且显示系统 logo。再然后初始化 cpu，获取 cpu 的相关信息。而后，初始化系统调用表，把系统调用表填写好，产生系统调用后会执行表格内对应的函数。

要迎来新的黎明了，多线程/多进程即将到来。首先是初始化多线程环境（进程也相当于一个线程，只不过拥有一套自己的资源），并把当前执行流当做 `main thread` 主线程，同时初始化一个 `idle` 线程（当没有其它线程和进程运行的时候来运行它）

此时我们的系统还不会发生线程/进程切换，因为我们的任务切换时基于时钟中断的，每个 0.01 秒检测一次是否需要切换任务。那么接下来就是初始化时钟中断，初始化完之后还不能进行任务切换，必须开启中断才会发生任务切换，所以我们 `io_sti` 打开中断，此时，就可以发生任务切换了。

我们的 `main thread` 主线程还在继续工作，它现在初始化硬盘（因为硬盘初始化会发出检测硬盘信息的命令，会产生中断，所以必须在此之前打开中断，硬盘中断程序才能相应）。硬盘可能有多个，我们会将有的 IDE 硬盘都初始化（最多 4 个硬盘）。然后选择 `hda` 第一个硬盘，并且在它上面初始化一个 `Fatxe` 文件系统。如果我们有要往文件系统中写入文件的需要，也就是说 `WRITE_DISK` 为 1 时，我们写入一个名字为 `WRITE_NAME`，大小为 `FILE_SECTORS` 的文件进入文件系统。这个机制是我用来把可执行文件，文本或者图形文件写入我们的文件系统所写的一套方法，你也可以自己实现一种，毕竟，条条大路通罗马。然后 `ls` 一下，看我们的文件系统里面有什么文件。

接下来就是初始化键盘，鼠标，并且开启几个线程，`keyboard` 线程用于处理键盘操作产生的数据，`mouse` 线程用于处理鼠标操作产生的数据，`clock` 线程用于每次发生时钟中断后，把当前的时间增加，这样我们的时间就是变化的（不然就只有一个启动时的时间，你都不知道你启动了多久了）

由于之前初始化图形后就显示了一个 logo，所以屏幕上只有现实的 logo，没有其它内容。在这里，我们初始化图形界面，就可以替代原有的 logo，现实自己的内容了。这样做的原因是，启动后就显示 logo，中间初始化其它内容需要一定时间，等它们都初始化完后，在初始化这里的图形界面，这样，就有一种类似于 Windows 启动的感觉了。

目前都是运行在内核中的，所以我们初始化一个进程，进程运行在用户态，但是呢，

init 进程是运行在内核中的，它拥有进程的所有东西，只是说它的执行入口在内核里面，和一般的进程没有区别，其他的进程我们将采用从磁盘加载的方式。而后，主线程就进入一个循环，先判断是否有被强制结束的进程，有的话就去把该进程从系统中释放掉，然后在休眠 100 毫秒。

就这样主线程的所作所为已经被我们看的清清楚楚，我们已经看清他这个人了。（斜眼笑）

那么 init 做了什么呢？在 `src/kernel/thread.c` 中我们可以找到 `init`，它其实也没做啥，就是从磁盘加载了一个 `boshell` 程序，然后循环等待子进程结束。那么刚好，`boshell` 就是我们进入图形界面后看到的那个 `shell` 程序。`Shell` 程序就等待用户输入命令，并作出相应的操作。

从此，我们的系统就进入就绪状态，等待和用户交互了。

第三章 系统的要点

3.1 loader 部分要点

要点 1:

加载内核的时候，如果内核大小变大了，就需要把在 `LoadKernel` 后面把内核的扇区数和写入磁盘的扇区数满足，只要比内核实际的大小大都可以的。与此同时，他也会加载一个应用程序的数据进入一个内存地址，我们将在后面写入文件系统的时候使用。

要点 2:

在切换图形模式的时候，在 `const.inc` 中修改要切换的模式。如果不想开启图形模式就把切换到指定的模式的那个 `int` 注释掉，不产生中断请求，就不会执行，后面就会进入文本模式，就不能初始化图形（当然这个功能可能不太会用到）

3.2 init 部分要点

要点 1:

在 `src/init/main.c` 中，会执行很多初始化操作，他们之间其实是有关联的，也就是不能随便修改他们的顺序。例如在初始化内存管理中，需要调用到分页管理里面的函数，因此，就不能把内存管理放到分页机制前初始化，不然会出错。

要点 2:

你可以修改 `WRITE_ID` 来往文件系统写入不同的文件，这样就比较方便从外部导入文件到文件系统了。

3.3 console 部分要点

要点 1:

在初始化完 `console` 之后，你就可以调用 `printk` 来打印你想要输出的信息，`printk` 也许还不太完善，你可以自己去完善，让它可以支持更多格式。

3.4 page 部分要点

要点 1:

在初始化完 `page` 之后，你就可以调用 `kernel_free_page` 和 `kernel_free_page` 来分配和释放 `N` 个页。

要点 2:

在分页机制中，我把低端 `2G` 内存给内核使用，用户使用高端 `2G`，这通过页目录表映射之后就可以实现。

3.5 memory 部分要点

要点 1:

在初始化完 memory 之后，你就可以调用 `kmalloc` 和 `kfree` 来分配和释放内存，便于后面的开发，他们是在内核下面的内存分配和释放的函数。

要点 2:

如果你在开发系统的时候需要分配大量内存，如果害怕内存不够，可以在你的虚拟机配置的时候配置更多内存。

3.6 device 部分要点

要点 1:

在 device 中，有中断设备（时钟，鼠标，键盘，硬盘，以及未实现的网络），也有非中断设备（视频，ramdisk），在中断设备中，他们都很类似，如果要实现自己的某个设备，可以仿造他们的方法来实现（例如网络）。

要点 2:

如果是中断设备，要记得到 descriptor 中注册一个中断描述符，以及在 interrupt 中注册一个入口函数。

3.7 fs 部分要点

要点 1:

在文件系统中，删除小文件（几 kb）没有问题，如果文件太大就可能会出现问題，所以大家在使用的时候稍微注意一下，后面更新文件系统后可能会好点。

要点 2:

文件系统是建立在硬盘的读写函数之上的，你可以根据这两个基层函数建立一套自己的文件系统，也可以去移植 ext2 或者 fat32 文件系统，但是要完善一点，不然的话，残缺了就不好了。

3.8 gui 部分要点

要点 1:

图形界面是基于绘制像素点进入显存来实现的，因此在 video 中已经有了绘制和读取像素的函数，我们后面只需要在这基础上进行开发就行了。也可以在视频信息的结构体获取屏幕宽高。

3.9 syscall 部分要点

要点 1:

系统调用是一个数组，里面存放的是函数的地址，一个新的系统调用就需要有

一个新的 NR，并且在你系统调用函数的前面加上 `sys_` 来表示它是一个系统调用函数。

3.10 lib 部分要点

要点 1:

Lib 下面的函数分为 2 种，一种是独立可以被调用的，一种是调用系统调用的接口。系统调用接口使用汇编编写的，因为要调用 `int` 软中断，来实现调用系统内部的函数。你可以把某一类的函数写到一个文件里面，例如文件操作的都写到 `file.asm` 中，也可以把他们独立出来 `fopen.asm, fread.asm, fwrite.asm...` 都可以的。

3.11 源码结构部分要点

要点 1:

打开 BookOS，你会发现有这些目录和文件，`bin, bochs, doc, img, src, LICENSE, README.md`。

`Bin` 目录下面存放的是应用程序的源代码以及其 `Makefile`，在需要的是后可以重新编写代码。

`Bochs` 目录下存放的是 `bochs` 虚拟机的配置文件，如果你想用 `bochs` 调试，你可以使用到这个文件。











`Doc` 目录下存放的是一些文档，一些 `log, bug, note` 之类的东西。可以把内容写在这里面，以辅助开发。

`Img` 下面存放的是磁盘镜像，`a.img` 表示软盘，`c.img` 表示第一个硬盘。还可以添加 `d.img, e.img, f.img` 等等。

`LICENSE` 是源码开放协议，BookOS 遵循 `BSD-2` 协议开放。












`README.md` 是 BookOS 的 `readme` 文件。

`Src` 目录下是存放的系统源代码。

 <code>boot</code>	2019/2/24 22:25	文件夹
 <code>device</code>	2019/2/24 22:25	文件夹
 <code>fs</code>	2019/2/24 22:25	文件夹
 <code>gui</code>	2019/2/24 22:25	文件夹
 <code>include</code>	2019/2/21 1:01	文件夹
 <code>init</code>	2019/2/24 22:25	文件夹
 <code>kernel</code>	2019/2/24 22:25	文件夹
 <code>lib</code>	2019/2/24 22:25	文件夹
 <code>cmd.bat</code>	2019/1/20 19:41	Windows 批处理...
 <code>Makefile</code>	2019/2/25 20:37	文件

在 `boot` 中存放的是引导代码，`device` 中存放的是设备/驱动文件，`fs` 存放的是文件系统的代码，`gui` 存放的是图形界面的代码，`init` 存放的是初始化的代码，`kernel` 存放的是内核代码，`lib` 存放的是库文件代码，`makefile` 是整个系统编译运行的主管，`cmd` 是用于在 `windows` 下面打开控制台后快速进入当前目录。`include` 存放的是头文件代码，在 `include` 中还有个 `sys` 目录，下面存放的是系统相关的代码，例如内存管理，进程管理，文件系统，图形界面的头文件等。

而直接在 `include` 下面的其他文件是一些类似于标准库的代码，这些既可以在内核中使用，也可以在用户程序中使用，用户程序只需要调用这些头文件。

 <code>sys</code>	2019/2/21 1:01	文件夹
 <code>graphic.h</code>	2019/2/19 20:32	H 文件
 <code>math.h</code>	2019/1/25 17:51	H 文件
 <code>stdarg.h</code>	2018/10/26 10:59	H 文件
 <code>stdint.h</code>	2019/1/25 0:03	H 文件
 <code>stdio.h</code>	2019/2/19 17:35	H 文件
 <code>stdlib.h</code>	2019/2/20 21:01	H 文件
 <code>string.h</code>	2019/2/12 23:33	H 文件
 <code>time.h</code>	2019/2/11 0:45	H 文件
 <code>types.h</code>	2019/2/11 14:01	H 文件
 <code>unistd.h</code>	2019/2/10 22:48	H 文件

要点 2:

如果你想添加一个文件，记得在对应的目录中添加，例如要添加一个 `kernel` 内部的文件，就添加到 `src/kernel` 目录文件下，其头文件应当放在 `src/include/sys` 下面。如果是添加一个系统调用的接口，把文件放在 `src/lib` 下面，头文件放到 `src/include` 下的某个头文件和或者自己添加的头文件里面。并且记得添加一个文件后，得到对应目录下面的 `makefile` 中修改添加该文件，并且还要到 `src/makefile` 这个主管里面添加该文件。

3.12 Makefile 部分要点

要点 1:

搞清楚 `Makefile` 后就知道整个系统是如何编译，链接，写入磁盘，虚拟机运行的了。先打开 `src/makefile`，可以发现这里面就是设置了一些工具的变量，以方便在后面的引用。由于在环境搭建的时候设置了环境变量，这个地方可以直接使用这些工具的名字。

```
#tool dir
BOCHS_DIR = ../bochs/

NASM      = nasm
CC        = gcc
LD        = ld
DD        = dd
QEMU      = qemu-system-i386
BOCHS     = bochs
BXRC      = $(BOCHS_DIR)bochsrc.bxrc
```

后面也是定义一些变量,这些事 `img` 镜像的位置,在写入磁盘的时候需要用到。如果要更改写入的镜像,可以在这儿修改。

```
#img dir
IMG_DIR = ../img/

FLAPPY_IMG = $(IMG_DIR)a.img
HDA_IMG    = $(IMG_DIR)c.img
```

再往后就是连接所需要的参数,如需要更改连接的方式,可以在这儿修改。

```
#flags

LDFLAGS = -m elf_i386 -e _start -Ttext 0x100000
```

然后定义了 `loader` 和 `kernel` 写入磁盘的扇区偏移以及扇区数量。

```
#system disk
LOADER_OFF = 2
LOADER_CNTS = 8

KERNEL_OFF = 10
KERNEL_CNTS = 400
```

为了能让主 `makefile` 调用其它目录下面的 `makefile`,我们需要把其它目录写进来。

```
#src dir
BOOT_DIR = ./boot/
KERNEL_DIR = ./kernel/
INIT_DIR = ./init/
LIB_DIR = ./lib/
DEVICE_DIR = ./device/
```

```
GUI_DIR = ./gui/  
FS_DIR = ./fs/
```

然后就是要输出那些文件，输出到哪个地方。

```
BOOT_BIN = $(BOOT_DIR)boot.bin  
LOADER_BIN = $(BOOT_DIR)loader.bin  
KERNEL_ELF = $(KERNEL_DIR)kernel.elf
```

因为链接操作是在这个主管 makefile 里进行的，所以我们需要知道其它目录产生的目标.o 文件。因此，我们用一个 OBJS 变量保存，如下：

```
#objs  
OBJS = $(KERNEL_DIR)_start.o\  
$(KERNEL_DIR)x86.o\  
$(KERNEL_DIR)descriptor.o\  
$(KERNEL_DIR)8259a.o\  
$(KERNEL_DIR)interrupt.o\  
$(KERNEL_DIR)console.o\  
$(KERNEL_DIR)bitmap.o\  
$(KERNEL_DIR)page.o\  
$(KERNEL_DIR)kernel.o\  
$(KERNEL_DIR)memory.o\  
$(KERNEL_DIR)ards.o\  
$(KERNEL_DIR)cpu.o\  
$(KERNEL_DIR)ioqueue.o\  
$(KERNEL_DIR)debug.o\  
$(KERNEL_DIR)cmos.o\  
$(KERNEL_DIR)syscall.o\  
$(KERNEL_DIR)thread.o\  
$(KERNEL_DIR)sync.o\  
$(KERNEL_DIR)tss.o\  
$(KERNEL_DIR)process.o\  
$(KERNEL_DIR)system.o\  
$(FS_DIR)dir.o\  
$(FS_DIR)fatxe.o\  
$(FS_DIR)fat.o\  
$(FS_DIR)file.o\  
$(INIT_DIR)main.o\  
$(GUI_DIR)main.o\  
$(GUI_DIR)graphic.o\  
$(GUI_DIR)font.o
```

\$(GUI_DIR)ft_simsun.o\
\$(GUI_DIR)ft_standard.o\
\$(GUI_DIR)image.o\
\$(DEVICE_DIR)vga.o\
\$(DEVICE_DIR)clock.o\
\$(DEVICE_DIR)keyboard.o\
\$(DEVICE_DIR)harddisk.o\
\$(DEVICE_DIR)video.o\
\$(DEVICE_DIR)ramdisk.o\
\$(DEVICE_DIR)mouse.o\
\$(LIB_DIR)string.o\
\$(LIB_DIR)vsprintf.o\
\$(LIB_DIR)math.o\
\$(LIB_DIR)printf.o\
\$(LIB_DIR)malloc.o\
\$(LIB_DIR)free.o\
\$(LIB_DIR)exit.o\
\$(LIB_DIR)getchar.o\
\$(LIB_DIR)putchar.o\
\$(LIB_DIR)write.o\
\$(LIB_DIR)execv.o\
\$(LIB_DIR)wait.o\
\$(LIB_DIR)fdopen.o\
\$(LIB_DIR)fclose.o\
\$(LIB_DIR)freadd.o\
\$(LIB_DIR)fwrite.o\
\$(LIB_DIR)fstat.o\
\$(LIB_DIR)lseek.o\
\$(LIB_DIR)unlink.o\
\$(LIB_DIR)opendir.o\
\$(LIB_DIR)closedir.o\
\$(LIB_DIR)readdir.o\
\$(LIB_DIR)rewinddir.o\
\$(LIB_DIR)mkdir.o\
\$(LIB_DIR)rmdir.o\
\$(LIB_DIR)rename.o\
\$(LIB_DIR)copy.o\
\$(LIB_DIR)move.o\
\$(LIB_DIR)getcwd.o\
\$(LIB_DIR)chdir.o\
\$(LIB_DIR)clear.o\
\$(LIB_DIR)ps.o\
\$(LIB_DIR)reboot.o\
\$(LIB_DIR)random.o\

```
$(LIB_DIR)access.o\  
$(LIB_DIR)gettime.o\  
$(LIB_DIR)graphic.o\  
$(LIB_DIR)mouse.o\  
$(LIB_DIR)igl.o\  
$(LIB_DIR)video.o\  
$(LIB_DIR)mm.o
```

准备工作就开始了。接下来就是开始真正地做事情了。由于 makefile 的第一个命令开始是文件中的第一个标签，那么就找到了 `all: compile link disk qemu`，默认的话就是执行这里面的操作，可以知道，它的第一步就是 `compile`，于是找到 `compile` 如下：

```
#compile file  
compile:  
  cd $(BOOT_DIR) && $(MAKE)  
  cd $(KERNEL_DIR) && $(MAKE)  
  cd $(INIT_DIR) && $(MAKE)  
  cd $(LIB_DIR) && $(MAKE)  
  cd $(DEVICE_DIR) && $(MAKE)  
  cd $(GUI_DIR) && $(MAKE)  
  cd $(FS_DIR) && $(MAKE)
```

它所做的就是进入某个目录，然后运行 `make`，引用该 `makefile` 执行操作，执行完后会返回这儿，然后执行下一个 `cd`，知道把所有子目录都 `make` 完。因此 `compile` 就完成了，产生了许多 `.o` 文件以及 `boot.bin` 和 `loader.bin`，`.o` 文件是用来链接成 `kernel.elf` 的。

下一步就是链接，看看它的代码：

```
link: $(KERNEL_ELF)  
  
$(KERNEL_ELF): $(OBJS)  
  $(LD) $(LDFLAGS) -o $(KERNEL_ELF) $(OBJS)
```

它要产生 `KERNEL_ELF`，于是就会执行 `$(LD) $(LDFLAGS) -o $(KERNEL_ELF) $(OBJS)` 这句代码，由此，链接操作完成，生成了 `kernel.elf` 文件。

接下来就是把文件写入磁盘了。在这里，默认把 `boot` 写入第 0 个扇区，`loader` 写入第二个扇区，`kernel` 写入第 10 个扇区。

```
#write file into disk  
disk:
```



```
$(DD) if=$(BOOT_BIN) of=$(FLAPPY_IMG) bs=512 count=1 conv=notrunc
$(DD) if=$(LOADER_BIN) of=$(FLAPPY_IMG) bs=512 seek=$(LOADER_OFF)
count=$(LOADER_CNTRS) conv=notrunc
$(DD) if=$(KERNEL_elf) of=$(FLAPPY_IMG) bs=512 seek=$(KERNEL_OFF)
count=$(KERNEL_CNTRS) conv=notrunc
```

这样，我们的系统就住在软盘上了。等运行的时候就可以启动咱们的系统了，好激动。

默认是使用 `qemu` 虚拟机启动，你也可以选择 `bochs`，甚至可以把镜像放到 `virtual box` 或者 `vmware` 虚拟机中运行。

```
#run kernel in vm
bochs:
    $(BOCHS) -q -f $(BXRC)

qemu:
    $(QEMU) -m 64 -fda $(FLAPPY_IMG) -hda $(HDA_IMG) -boot a
```

由此，我们的系统就在虚拟机中飞速运行啦，`qemu` 比 `bochs` 快，你可以自己选择。

我们还有一个 `clean` 标签，可以用它来清除编译过程中产生的临时文件，让我们的源码变得干净。

```
#clean temporary files
clean:
    cd $(BOOT_DIR) && $(MAKE) clean
    cd $(KERNEL_DIR) && $(MAKE) clean
    cd $(INIT_DIR) && $(MAKE) clean
    cd $(LIB_DIR) && $(MAKE) clean
    cd $(DEVICE_DIR) && $(MAKE) clean
    cd $(GUI_DIR) && $(MAKE) clean
    cd $(FS_DIR) && $(MAKE) clean
```

至此，主 `makefile` 就完了，你可以根据自己的需求添加其他的标签，满足自己的需要。

要点 2:

除此之外，再看一个子 `makefile`，就能明白整个流程了。首先也是进行准备操作，定义变量保存工具和头文件路径。

```
#The tools name
NASM      = nasm
CC        = gcc

#The file path
INCLUDE_PATH = ../include/
```

然后定义 `gcc` 和 `nasm` 的编译参数:

```
#flags
ASM_FLAGS = -I $(INCLUDE_PATH) -f elf
C_FLAGS = -I $(INCLUDE_PATH) -c -fno-builtin -Wall -Wunused
```

就像这样, 会在后面编译的时候引用。

用 `OBJS` 把所有的对象文件列出来, 我们根据列出来的文件来选择那些事要编译的。

```
OBJS = _start.o\  
      x86.o\  
      descriptor.o\  
      8259a.o\  
      interrupt.o\  
      console.o\  
      bitmap.o\  
      page.o\  
      kernel.o\  
      memory.o\  
      ards.o\  
      cpu.o\  
      ioqueue.o\  
      debug.o\  
      cmos.o\  
      syscall.o\  
      thread.o\  
      sync.o\  
      tss.o\  
      process.o\  
      system.o
```

然后就是第一个标签是需要操作的标签, 因此如下

```
#First read here
.PHONY: all
```

```
all: compile

#Compile files
compile: $(OBJS)
```

可以看出，默认就是编译文件。于是便会引用到自动编译汇编和 c 语言的命令。

```
%o: %.asm
$(NASM) $(ASM_FLAGS) -o $@ $<

%.o: %.c
$(CC) $(C_FLAGS) -o $@ $<
```

就是像这样，%号就是任意一个文件名。后面就是后缀名，汇编文件是 **NASM** 编译，**gcc** 文件是 **CC** 编译。这样就会逐渐编译每一个文件。

与此同时你也可以在里面输入 **clean** 来清楚产生的临时文件。

```
#Clean temporary files
clean:
-rm *.o
-rm kernel.elf
```

由此，**makefile** 就解析完了，你就知道整个系统是怎么编译，链接，运行的了。

3.13 内核态与用户态要点

要点 1:

我们要注意某个线程/进程对一个函数的调用，如果你是一个内核线程，那么你可以调用内核态的函数，也可以调用用户态的函数（不会出错，但尽量不要调用，毕竟要通过中断才可以调用到内核里面的函数），用户态的函数只能调用库函数，不能直接调用内核态的函数，不然会出错。

要点 2:

用户进入内核调用某个函数，是通过系统调用中断实现的，他要通过发生中断来切换运行的特权级，从 3 特权级到 0 特权级，然后调用 0 特权级的函数，就不会发生错误了，调用完后就会切换特权级，从 0 特权级切换到 3 特权级。返回到用户态继续执行 3 特权级的函数。

第四章 一些问题的解决

4.1 如何添加一个中断类 device?

我们拿 Keyboard 来举例。

第一步，创建一个 keyboard.c。

第二步，写一个 init_keyboard 函数，用于初始化 keyboard

第三步，写一个 keyboard_handler 函数，用于初始化处理键盘中断

第四步，在 init_descriptor 中初始化一个 keyboard 对应的中断描述符

```
set_gate_descriptor(idt + 0x20+KEYBOARD_IRQ, (int *)&IRQ_keyboard, 0x08, DA_386IGate, 0);
```

这里涉及到一个 IRQ_keyboard，他是键盘中断发生后的入口，是汇编函数编写的。

第五步，在 interrupt.asm 中编写 IRQ_keyboard 函数，当发生中断后，会首先执行这个函数，这个函数会调用 keyboard_handler 来进行 c 语言部分的处理。

到这里，就可以接受 keyboard 中断了，如果要进行其它处理，可以自己添加函数处理。

4.2 如何编写自己的图形界面?

由于底层的像素写入和读取函数已经实现了，就可以自己实现一个图形界面了，也可以使用原生的图形界面，在那个基础之上进行图形编程。

```
/*把颜色值写入显存*/
```

```
void vram_write_pixel (int32 x, int32 y,uint32 color);
```

```
/*从显存中读取一个像素保存到 color 里面*/
```

```
void vram_read_pixel (int32 x, int32 y,uint32 *color);
```

4.3 如何编写自己的文件系统?

文件系统是基于扇区的读/写函数来实现的，所以只要有了这两个函数，你就可以实现任何类型的文件系统了。

```
/*读取几个扇区进入 buf*/
```

```
void hd_read_sectors (uint32 lba, void *buf, uint32 counts);
```

```
/*从 buf 中写入几个扇区到磁盘*/
```

```
void hd_write_sectors (uint32 lba, void *buf, uint32 counts);
```

4.4 如何添加一个系统调用?

如果你想要让用户程序调用内核函数，你需要添加一个系统调用，让用户程序陷入内核态执行内核里面的某个函数。

我们拿 `_NR_FOPEN` 来举例：

第一步，编写你的系统里面的函数

```
int32_t sys_open(const char *pathname,uint8_t flags);
```

第二步，需要在 `syscall.h` 中添加一个系统调用函数的编号

```
#define _NR_FOPEN 10
```

如果你要添加的编号超过了 `MAX_SYS_CALLS`，你需要更新 `MAX_SYS_CALLS` 的大小，因为它是一个数组，你要把函数的地址存放在这个数组里面才行。

第三步，你需要在 `src/kernel/syscall.c` 里面把函数地址放到 `sys_call_table` 里面对应的位置上去。

```
sys_call_table[_NR_FOPEN] = sys_open;
```

这样，当系统调用发生的时候，他就可以找到 `sys_open` 这个函数了。

第四步，编写用户的接口。把它保存到 `src/lib` 目录下。我给它起名叫 `fopen.asm`，他是一个汇编文件。

```
[bits 32]
[section .text]

INT_VECTOR_SYS_CALL equ 0x80

_NR_FOPEN EQU 10

global fopen
fopen:
    mov eax, _NR_FOPEN
    mov ebx, [esp + 4]
    mov ecx, [esp + 8]
    int INT_VECTOR_SYS_CALL
    ret
```

在这里先声明成 32 位程序，`.text` 代码段，然后有个系统调用号，这个我们和 linux 一样，都是 `0x80`，我们让 `_NR_FOPEN` 为 `10`，和我们之前定义的一样。下面就是写了 `fopen` 的标签，并且把它变成全局标签，也就是导出去，让其它函数可以调用到。`eax` 的值是系统调用函数对应的 NR 号，然后 `ebx,ecx` 来进行参数的传递，再调用系统调用，完了后就返回，这样，只要调用 `fopen`，就可以到内核里去调用某个函数。（这里的参数传递，我们最多只支持 4 个参数分别是 `ebx,ecx,edx,esi`）

第五步，写一个头文件，`fopen(const char *pathname,uint8_t flags);`，然后放到 `include/stdio.h` 里面，只要某个程序导入了这个头文件就可以调用这个函数了。

第六步，把 `fopen.o` 写入 `src/lib/Makefile` 的 `OBJS`，与此同时，为了让整个系统也知道它，还要把 `fopen.o` 写入 `src/Makefile`，这样内核就能知道它了

第七步，当你需要写某个程序的时候，你也需要把 `fopen.o` 写到你程序的 `Makefile` 里面，例如 `/bin/test/Makefile` 里面的 `OBJS` 后面。

就这样，你就可以添加一个系统调用了。

4.5 如何添加一个线程/进程？

我们的线程是运行在内核中的，所以只能在内核里面创建线程，那这些线程有什么用呢？它可以来处理中断产生的数据，也可以是一段运行在内核里面的代码。那么如何添加一个内核线程呢？其实很简单的。

```
struct thread *thread_start(char *name, int prio, thread_func function, void *arg);
```

Name 参数是线程的名字，prio 是线程的特权级，function 是线程要执行的函数，arg 是传入的参数。返回值是一个 thread 结构体。这样，你就可以生成一个线程，它可以执行系统中的某段代码。

进程的话，有运行在内核中的进程和执行磁盘程序的进程。

```
int process_execute(void *filename, char *name);
```

Filename 是进程的入口地址，name 是进程的名字，返回进程的 pid。这个进程是有自己的资源（斜眼笑）的，有页目录表，一套完整的映射。不过，他是直接运行某个函数，并把那个函数当做一个进程。

```
int sys_execv(char *path, char *argv[]);
```

是只从磁盘加载一个程序运行。Path 是路径，argv 是传入的参数，返回进程 pid。这个函数可以在内核调用，他还有个用户接口 `int execv(char *path, char *argv[])`；这个函数可以在应用程序中调用，以执行另外一个程序。只要在文件系统中某个可执行文件，你就可以把它拿来执行了。

第五章 应用程序开发

5.1 原理介绍

对于应用程序的开发，我们先来搞懂它的原理。

首先，我们得编写我们的程序代码，如果调用需要调用库函数，那么导入对应的头文件就可以了。然后，需要对代码进行编译，编译成.o 目标文件。再和库函数的.o 文件进行链接，生成可执行二进制文件。接下来，把文件写入磁盘，我们这里是软盘。在 loader 中，我们会把磁盘上的文件读入到一个内存里面。在文件系统初始化完之后，我们把那个内存里面的数据通过 `sys_write` 写入到文件系统中，后面就可以直接从文件系统中加载执行了。

5.2 程序的编写、编译与写入磁盘

第一步，先从/bin 目录下面复制一个文件目录，在这里，我们复制一下 `brainfuck` 目录，并把它命名为 `test`，如下：

 <code>boshell</code>	2019/2/23 15:03	文件夹
 <code>brainfuck</code>	2019/2/23 15:03	文件夹
 <code>test</code>	2019/2/26 21:01	文件夹
 <code>tinytext</code>	2019/2/23 15:03	文件夹

第二步，打开 `test`，把 `main.c` 清理一下，保留基本框架。

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    return 0;
}
```

然后把 `example.txt` 删除掉，我们不用它。`Version.txt` 可以保留，他是用来记录该程序的版本的。

第三部，再打开 `makefile`，和其它 `makefile` 很相似，也是先定义工具和路径变量以及其他信息。

```
#The file path
INCLUDE_DIR = ../../src/include/
LIB_DIR = ../../src/lib/
```

这里的头文件目录需要引用到 `src/include`。

我们生成的是一个二进制文件所以有 `BIN = bin`。然后就是文件写入哪个扇区和写入多少个扇区数，写入的扇区数可以根据程序的大小调整，如果程序是 `5.26kb`，那么你可以把扇区的大小调整为 `12` 个扇区，也就是说和文件大小最为接近但是大于文件大小。

再例如文件大小 22.3kb，你可以写入 45 个扇区。

```
BIN_OFF = 500
BIN_CNTS = 20
```

后面也是一些目录的定义和编译的参数传递，和其它 `makefile` 差不多。

```
#flags
ASM_FLAGS = -I $(INCLUDE_DIR) -f elf
C_FLAGS = -I $(INCLUDE_DIR) -c -fno-builtin -Wall -Wunused

LDLFLAGS = -e _start -Ttext 0x80000000 --oformat binary
```

值得注意的是，我们最终把它链接成一个二进制文件。而 `--oformat binary` 就是做这个请求的。

下面就是 `OBJS` 文件，这个是在程序源码中的 `.o` 对象文件。除此之外，还有一个 `LD_OBJS`，这个变量存放的是需要链接的链接库文件，如果你需要调用某个函数，你就要链接对应的链接库才行。

```
OBJS = _start.o\
      main.o

LD_OBJS = $(LIB_DIR)printf.o\
          $(LIB_DIR)malloc.o\
          $(LIB_DIR)free.o\
          $(LIB_DIR)exit.o\
          $(LIB_DIR)getchar.o\
          $(LIB_DIR)putchar.o\
          $(LIB_DIR)write.o\
          $(LIB_DIR)vsprintf.o\
          $(LIB_DIR)string.o\
          $(LIB_DIR)fopen.o\
          $(LIB_DIR)fclose.o\
          $(LIB_DIR)freopen.o\
          $(LIB_DIR)fwrite.o\
          $(LIB_DIR)fstat.o\
          $(LIB_DIR)lseek.o\
          $(LIB_DIR)unlink.o\
```

后面的内容和之前的 `makefile` 一样，在这儿就不在阐述，就是编译，链接，写入磁盘的具体命令操作。你不需要改变它们，你需要做的就是根据你的源文件数量和调用的库函数数量来添加不同的 `OBJS` 文件和 `LD_OBJS` 文件。

我们也把 `Makefile` 的 `OBJS` 文件也整理一下。如下：

```
OBJS = _start.o\
```



```
main.o
```

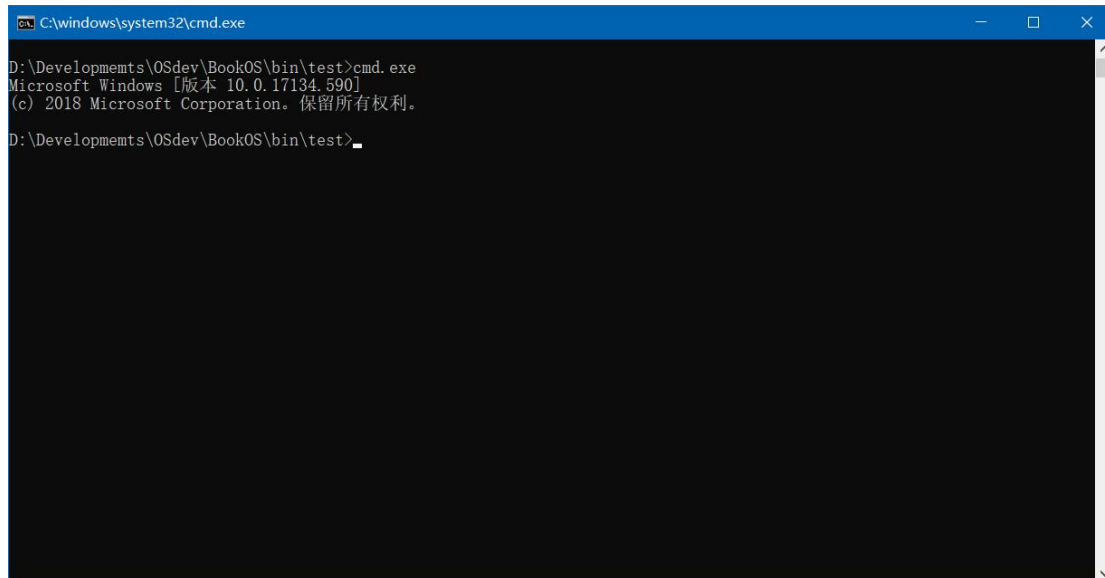
```
LD_OBJS = $(LIB_DIR)printf.o\  
          $(LIB_DIR)exit.o\  
          $(LIB_DIR)write.o\  
          $(LIB_DIR)vsprintf.o\  
          $(LIB_DIR)string.o\  
          $(LIB_DIR)string.o
```

我们的 `test` 程序只需要调用 `printf`，所以只需要以上链接库文件。`write` 是用来往控制台写字符串，`printf` 会调用它。`printf` 是用来解析传入的参数，`vsprintf` 是 `printf` 中调用的一个中间函数，`vsprintf` 要调用 `string` 中的某些函数。那 `exit` 呢？它是程序调用 `exit` 库文件。如果代码中没有，我们也会默认调用它，所以这里要保留它。如果你想添加其它功能，那么就针对不同的功能添加不同的库文件就行了。

第四步，就是编写我们自己程序代码了。我们用 `printf` 输出 `hello, world! 123`

```
#include <stdio.h>  
int main(int argc, char *argv[])  
{  
    printf("hello, world!\n%d", 123);  
    return 0;  
}
```

代码写好后，可以进行编译了。打开 `cmd.bat`，输入 `make` 就可以，如果想删除生成的文件，就 `make clean`。如下：



```
C:\windows\system32\cmd.exe  
D:\Development\OSdev\BookOS\bin\test>cmd.exe  
Microsoft Windows [版本 10.0.17134.590]  
(c) 2018 Microsoft Corporation. 保留所有权利。  
D:\Development\OSdev\BookOS\bin\test>_
```

```

D:\Developments\OSdev\BookOS\bin\test>make
gcc -I ../../src/include/ -c -fno-builtin -Wall -Wunused -o main.o main.c
ld -e _start -Ttext 0x80000000 --oformat binary -o bin_start.o main.o ../../
../src/lib/printf.o ../../src/lib/exit.o ../../src/lib/write.o ../../src/l
ib/vsprintf.o ../../src/lib/string.o
dd if=bin of=../../img/a.img bs=512 seek=500 count=20 conv=notrunc
13+1 records in
13+1 records out
7068 bytes (7.1 kB, 6.9 KiB) copied, 0.010696 s, 661 kB/s

D:\Developments\OSdev\BookOS\bin\test>_

```

如果没有错误提示，就说明编译成功了，并且生成二进制文件，写入了磁盘。

5.3 程序的载入文件系统

现在，我们的程序已经写入磁盘了，接下来，我们就是把它载入内存。这就需要 loader 来做了。

第一步，在 `src/boot/loader.asm` 中找到 `LoadeApp` 这个标签，它后面做的就是加载程序到一个内存地址。

```

LoadeApp:
    ;loade app
    mov ax, APP_SEG
    mov si, APP_OFF
    mov cx, BLOCK_SIZE
    call LoadeBlock

    add ax, 0x1000
    mov cx, BLOCK_SIZE
    call LoadeBlock

```

在 `boot/const.inc` 中找到 `APP_SEG` 和 `APP_OFF` 的定义

```
APP_OFF EQU 500
```

```
APP_SEG EQU 0x6000
```

```
BLOCK_SIZE EQU 128
```

应用程序的偏移和我们之前 `makefile` 中的写入扇区偏移一样。我们把 `app` 加载到段为 `0x6000` 的地址里面（这里用 `ax` 保存参数，会在 `Loadeblock` 中给 `es`），其物理地址是 `0x60000`，`BLOCK_SIZE` 是 `128`（因为扇区寻址原因，这儿最大只能是 `128`），也就是每次加载 `128` 个扇区，然后 `call LoadeBlock`，就把从磁盘上 `500` 偏移出开始的 `128` 个扇区读取到 `0x60000` 这个地址了。然后 `ax` 加上 `0x1000`，也就是说段地址变成了 `0x7000`，那为什么是加 `0x1000` 呢？我们来计算一下：`128 扇区 * 512 字节 = 65536 字节 = 0x10000 字节`。也就是说读取完 `128` 扇区后，应用程序就读取了 `0x10000` 字节，下一次读取就把段移动一下，然后偏移清零，就可以继续在后面读取数据了。

在第二次读取之后，就读取了 `256` 个扇区到内存中，也就是 `0x20000 字节 = 128KB`，也就是说，我们要写入的文件的的大小最大为 `128kb`。

第二步，打开 `src/init/main.c`

你会看到一些宏定义如下：

```
#define WRITE_DISK 1

#define WRITE_ID 1

#if WRITE_ID == 1
    #define WRITE_NAME "/test"
    #define FILE_SECTORS 20
#elif WRITE_ID == 2
    #define WRITE_NAME "/boshell"
    #define FILE_SECTORS 40
#elif WRITE_ID == 3
    #define WRITE_NAME "/tinytext"
    #define FILE_SECTORS 40
#elif WRITE_ID == 4
    #define WRITE_NAME "/brainfuck"
    #define FILE_SECTORS 20
#endif

#define APP_PHY_ADDR 0x60000
```

WRITE_DISK 为 1 的时候就是需要从内存读取数据进入文件系统，为 0 则不。

WRITE_ID 是选择要写入哪个类型的文件。下面会根据不同的 ID 定义不同的 WRITE_NAME 和 FILE_SECTORS。在这里我们把它设置为 1，就对应了/test 这个文件

APP_PHY_ADDR 是 0x60000，这和 loader 中把磁盘上的数据读入内存的地址一样。我们将在这个地址中读取数据来写入文件系统。

在 main 函数中你会看到这个：

```
if(WRITE_DISK){
    write_bin();
}
```

它拿来写入文件操作，write_bin 就是执行操作的核心。

```
void write_bin()
{
    printk("write_bin start...\n");

    char *app_addr = (char *)APP_PHY_ADDR;
    printk("app:%x\n", *app_addr);
    int fd;
    int written;

    fd = sys_open(WRITE_NAME, O_CREAT|O_RDWR);
    written = sys_write(fd, app_addr, FILE_SECTORS*SECTOR_SIZE);
```

```
printf("write bin %s size:%d success!\n",WRITE_NAME, written);
sys_close(fd);
}
```

它先输出一段文字，再定义一个指针变量指向 APP_PHY_ADDR 然后打开 WRITE_NAME 这个文件,如果不存在就创建，如果存在就读写。然后 sys_write 把 app_addr 指向的内存的数据写入 FILE_SECTORS*SECTOR_SIZE 字节到文件。就这样，我们就把这个我们编写的程序写入了文件系统。可以直接在文件系统中加载执行了。

然后启动系统，输入 ls 查看文件系统上的文件，就会看到有一个 test 文件

```
/>ls
. boshell tingtext brainfuck test
/>
```

我们知道他是一个可执行文件，所以直接输入该文件就可以执行了。

```
/>test
hello,world!
123
/>
```

你看，就输出了 hello,world!123，是不是很棒？（必须的，杠杠的）

就这样，一个程序的编写，编译，链接，写入磁盘，载入内存，载入文件系统，运行就这样完成了，是不是很期待根据系统提供的库函数来开发自己的软件呢？那就赶快动手试试吧。

至此，你的屌丝人生就已经到达了巅峰！

后记

由于本人能力有限，文档中可能还存在些许错误，希望大家能够指出，我将会及时修正。可能大家还会在开发中遇到问题，就可以联系我，我会更新解决方法到文档之中。也感谢大家对本系统的关注与支持。

编撰者：胡自成

联系方式

电子邮箱：2323168280@qq.com

QQ:2323168280

QQ 群：913813452

微信：hu2323168280

文档版本发布历史记录

2019/2/26

基于 BookOSv0.2

发布 v0.1 文档