

# Lab1.初探xv6

## 框架代码初探

框架代码的内容比较多，随着实验进度的推进，我们会逐步解释代码的内容，所以在阅读代码的时候，只需要关心当前进度相关的模块即可，不需要过度纠缠于与当前进度无关的代码。

未经过任何编译过程的xv6-riscv目录下的代码源文件组织如下：

```
xv6-riscv
├── kernel
│   ├── bio.c           # 文件系统的磁盘块缓存
│   ├── buf.h
│   ├── console.c       # 连接到用户的键盘和屏幕
│   ├── defs.h
│   ├── elf.h
│   ├── entry.S         # 首次启动指令
│   ├── exec.c          # exec() 系统调用
│   ├── fcntl.h
│   ├── file.c          # 文件描述符支持
│   ├── file.h
│   ├── fs.c            # 文件系统
│   ├── fs.h
│   ├── kalloc.c        # 物理页面分配器
│   ├── kernel.ld
│   ├── kernelvec.S     # 处理来自内核的陷入指令以及计时器中断
│   ├── log.c           # 文件系统日志记录以及崩溃修复
│   ├── main.c          # 在启动过程中控制其他模块初始化
│   ├── memlayout.h
│   ├── param.h
│   ├── pipe.c          # 管道
│   ├── plic.c          # RISC-V中断控制器
│   ├── printf.c        # 格式化输出到控制台
│   ├── proc.c          # 进程和调度
│   ├── proc.h
│   ├── ramdisk.c
│   ├── riscv.h
│   ├── sleeplock.c     # 会让出CPU的锁
│   ├── sleeplock.h
│   ├── spinlock.c      # 不让出CPU的锁
│   └── spinlock.h
```

```

|   └─ start.c           # machine mode启动代码
|   └─ stat.h
|   └─ string.c          # 字符串和字节数组库
|   └─ swtch.S           # 线程切换
|   └─ syscall.c         # 系统调用相关
|   └─ syscall.h
|   └─ sysfile.c         # 文件相关的系统调用
|   └─ sysproc.c         # 进程相关的系统调用
|   └─ trampoline.S      # 用于在用户和内核之间切换的汇编代码
|   └─ trap.c            # 对陷入指令和中断进行处理并返回
|   └─ types.h
|   └─ uart.c            # 串口控制台设备驱动程序
|   └─ virtio.h
|   └─ virtio_disk.c     # 磁盘设备驱动程序
|   └─ vm.c              # 管理页表和地址空间
└─ mkfs
    └─ mkfs.c
└─ user
    └─ cat.c
    └─ echo.c
    └─ forktest.c
    └─ grep.c
    └─ grind.c
    └─ init.c
    └─ initcode.S
    └─ kill.c
    └─ ln.c
    └─ ls.c
    └─ mkdir.c
    └─ printf.c
    └─ rm.c
    └─ sh.c
    └─ stressfs.c
    └─ ulib.c
    └─ umalloc.c
    └─ user.h
    └─ user.ld
    └─ usertests.c
    └─ usys.pl
    └─ wc.c
    └─ zombie.c

```

我们主要关心 `kernel/` 中的内容，因为这是 xv6 的内核代码，用户层的代码也是基于内核层实现功能的。

除此之外，[xv6 book](#)中第一章还列出了xv6所有的系统调用，在这里不——列出，有兴趣的同学可以自行查看。

大致了解了上述的目录树之后，就可以开始阅读代码了。

至于从哪里开始阅读代码，你们从程序设计课中学到的应该是从`main.c`开始，但是冷静下来想想，代码的起点真的是这里吗？

## xv6的启动过程

### 观察make的执行命令

在此之前，我们为了测试工具链安装是否成功而构建和编译了整个项目，所以我们需要让项目代码回到最初的样子，从头开始观察。

输入：

```
make clean
```

对于观察整个项目启动过程的动态行为，这里介绍一个非常有用的命令：`make -n`，可以只打印命令但是不执行。

所以可以通过下面的命令观察make执行的命令：

```
make -nB
```

其中，`-B`可以强制make构建所有目标，即使它们已经是最新的。

#### 扩展阅读

想想为什么我们在启动xv6时使用`make qemu`，而这里不使用`make -n qemu`呢？好吧，其实这里的目的是为了让你们看命令的时候更简单。那`make qemu`和`make`有什么区别呢？想知道的同学请去学习一下makefile吧，在本实验中并不要求完全读懂xv6的makefile，欢迎有兴趣的同学自行去学习。

为了更加方便处理命令，观察核心行为，还可以将结果重定向到文件中。

```
make -nB > make.txt
```

接下来就可以使用编辑器打开`make.txt`文件，处理make执行的命令了。当然，这里并不一定要将结果重定向到.txt文件中，只是变成.txt文件使用VSCode处理起来会比较方便，而且也不至于与其他文件混淆。

#### make执行了什么命令？

使用上述的命令，观察make执行了什么命令，在报告中简述该执行过程，观察`main.c`什么时候被编译。

# 逐步调试启动过程

除了通过观察make执行的命令来研究xv6的启动过程，使用gdb也是一种非常好的办法。在后续实验中，使用gdb进行调试或许会给你带来极大的便利。

完成上述的观察make执行了什么命令的任务后，相信你已经可以通过阅读命令进一步开始阅读源码，并且找到内核从哪一个地址开始运行第一条指令了。

## 用gdb调试xv6的启动过程

给第一条指令所在的地址打一个断点，从此处逐步调试观察寄存器的变化，再根据源代码理解启动xv6的每一步行为，并在报告中写出系统什么时候打印基本信息。

在xv6中启动gdb，首先要在一个窗口输入：

```
make qemu-gdb
```

然后你会看到一句话：

```
** Now run 'gdb' in another window.
```

这句话提醒你在另一个窗口打开gdb，所以打开另一个窗口，输入：

```
gdb-multiarch
```

就可以在xv6中启动gdb了。

这时候你可能会发现gdb给你报了一些警告，你无法正常启动gdb来调试代码，无需担心，你只需要退出gdb，将警告中的 `add-auto-load-safe-path YOUR_PATH/.gdbinit` 加入到对应的 `.gdbinit` 文件中，然后再重新打开 `gdb-multiarch` 就可以了。

至于如何使用gdb？导读中有gdb的手册，如果看不懂，可以询问搜索引擎来获得你想要找到的功能的命令，当然，更快速的方法是询问ChatGPT。

## 扩展阅读

如何使用工具往往不是最重要的，最重要的是在正确的时候能够反应出正确的问题，然后通过正确的问题找到答案。当ChatGPT这种AI走入我们的生活，正确的问出问题或许会成为人与人之间的差别。

## 调试main函数

尝试给main函数处打断点，从main函数处出发调试，观察系统什么时候打印基本信息，和在第一条指令处开始调试有何不同？经过这两次调试过程，你就已经能完全理解xv6的启动过程了。

# 定制化xv6的界面

在源代码中找到xv6的shell是如何设置的，可以通过阅读代码过程，也可以通过编辑器的搜索功能找到，还有在Linux中其实一行命令就能搞定查找，无论用哪种方法都可以，去找找看吧。

## 定制属于自己的xv6界面

定制属于自己的xv6界面，在启动后加入欢迎语，可以是一句话，也可以是字符组成的标志，把xv6最原始的命令提示符光标“\$”定制为自己的专属，可以加入学号和昵称之类的，在报告中展示你的定制化界面。

## 继续定制shell

xv6的shell只是一个最小的shell，你可以对它进行改进，增加真实的shell中的许多特性。例如：

- 将shell修改为支持用 `;` 分隔的命令列表
- 通过实现左括号以及右括号来修改shell以支持子shell
- 将shell修改为支持 `tab` 键补全

你可以实现任何你想实现的功能，定制一个专属你自己的shell。

## 关于实验报告

请将Lab0中的实验结果放在本实验的报告中。