[1]

问题： 如何快速获得实验环境，跳过配置中的各种坑

解决方法：https://github.com/chyyuu/ucore_os_lab/blob/master/README-chinese.md 友好地提供了已经预先安装好相关实验环境所需软件的虚拟硬盘 https://pan.baidu.com/s/11zjRK

结论：基于x86的实验环境环境真好用，然而我需要的qemu是 qemu-system-mips哇

[2]

问题: 配置qemu-system-mips后，执行qemu-system-mips显示缺少 mips_bios.bin

解决方法:

https://www.linux-mips.org/wiki/QEMU 中给出了解决方法 如果传递 -kernel参数，则qemu不会调用 mips_bios.bin,所以生成如下空文件：

```
dd if=/dev/zero of=/usr/local/share/qemu/mips_bios.bin bs=1024 count=128
```

结论： qemu-system-mips能执行辽，弹出了一个黑框，然后，然后就没有然后了，只弹出了一个黑框,加上ucore for mips的kernel试一下，还是黑框，fine

[3]

问题： 既然有 bbl-ucore for risc_v 以及相应的分步骤实验，那么mips应该也有吧

解决方法： 根据github的检索经验，搜索 ucore 最为保险，不然可能会错过一些个奇怪的命名， 一共有377个仓库与ucore相关，然而并没有关于 ucore on mips的分步实验，倒是存在关于 risc_V 的一些个分享，都是 risc 指令集，可以先瞅瞅。

结论： 大佬果然很少写博客

[4]

问题： 只需要更改 启动阶段 和 相关的寄存器？？？

结论： 毕竟启动阶段是汇编代码写的，确实需要更改，内联汇编的地方似乎也要修改，寄存器修改没话说，那为啥看 mips的ucore就跟重写了一遍一样，先把x86的搞明白是正解啊

[5]

问题： 进入lab目录，发现有多个 .sh 文件，lab1在CSDN和知乎中找到一些内容，说明shell脚本与makefile的区别

一些个扩展资料：

Makefile与shell脚本的区别

- 一些个格式区别
  - makefile中可以调用shell脚本
  - shell中所有引用 以$开头的变量后面加{},而makefile用()
  - $---makefile变量 $$---shell变量
  - shell中的通配符是 * makefile中的通配符是 %
  - makefile中执行shell命令，一行创建一个进程
    - 所以很多makefile中有很多行的末尾都是 “ ;\ ” ,以此保证代码是一行而不是多行
- 从分工上看，shell是给系统管理员使用，makefile是给软件配置管理员使用
- shell使用sed awk grep等指令，移植性差
- makefile 可以方便地支持多线程

执行 .sh文件的方法：

- ./ hello.sh 【hello.sh必须有x权限】
- sh hello.sh 【hello.sh可以没有x权限】

makefile

make是一个命令工具，是一个解释makefile中指令的命令工具，大多数IDE都有make指令，Delphi的make,Vistual C++的nmake,Linux下的GNU的make

文件依赖性是关键

源文件->编译->中间文件（.obj for win and .o for unix）->链接->执行文件

- 编译需要语法正确，函数与变量的**声明**正确，告诉编译器头文件所在的位置，一个*源文件*对应于一个*中间目标文件*
- 链接函数和全局变量。链接器不管函数所在的源文件，只管中间文件，中间文件太多，打个包吧， .lib for win (Library File)and .a for unix(Archive File)

makefile告诉make需要怎么去编译和链接程序

https://seisman.github.io/how-to-write-makefile/introduction.html

工作方式

- 读入所有makefile

- 读入被include的其他makefile
- 初始化文件中的变量
- 推导隐晦规则，并分析所有规则
- 为所有的目标文件创建**依赖关系链**
- 根据依赖关系，决定哪些文件要重新生成
- 执行命令

规则告诉make两件事,文件的**依赖关系**和如何**生成目标文件**

结论： 改脚本分数的同学是个高手

[6]

- BIOS是固化在主板Flash/CMOS的软件，所以实验环境中它应该在qemu中，实验代码从**bootloader**开始
  - bootasm.S 开启保护模式 全局描述符表的大小与位置入全局描述符表寄存器 进 bootmain
  - bootmain.c 读elf到内存， 安排到相应位置 ，控制权交给ucore
- init.c 里面的kern_init()中的各种调用函数构成了我们在 qemu中看到的**所有内容**，调用过程涵盖了**所有.c文件**
  - 按着kern_init() 中的执行流程，到 clock_init() 调用时钟中断出现问题，这里涉及到产生中断，根据中断描述符表确定中断服务例程入口地址，中间需要通过 vector.S进入all_trap,再建立相应的 **trapframe**,之后进入trap.c，根据trap_dispatch()执行相应功能
  - 初步来看，intr_enable() 和clock_init() 的调用顺序可更改

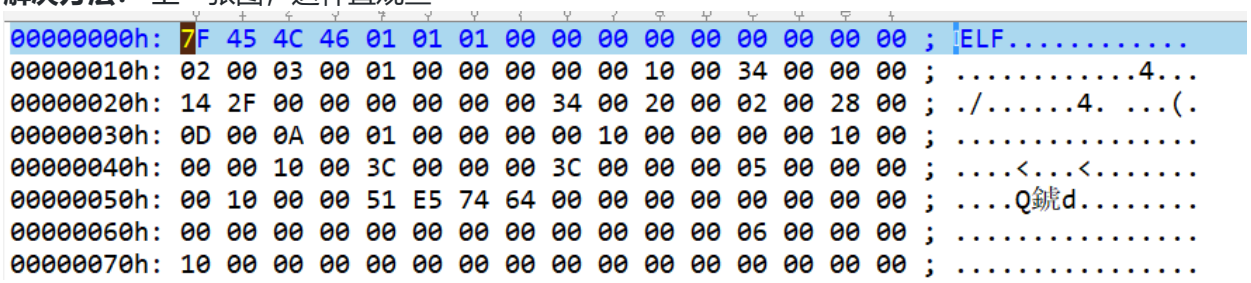
[7]

问题：

```
elf.h
#define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian
```

似乎有道理，为什么尼

解决方法： 上一张图，这样直观些



结论： WinHex大法好

[8]

问题：

```
tf->tf_esp=(uint32_t)tf+sizeof(struct trapframe)-8;
*((uint32_t*)tf-1)=(uint32_t)tf;
```

解决方法： 熟读百遍，其意自现

结论：我们指定了两个中断120，121，中断发生按照上面的流程进入trap_dispatch(),我们希望中断返回后实现特权级的转换，很明显需要人为把 代码段、数据段、标志等寄存器进行转换，问题是 esp 这个东西不听话，需要谨慎对待。

[9]

问题： make grade 之前 **make qemu** 与 make grade 之后**make qemu**两次的执行结果有差异，在grade之后执行可进入命令行

解决方法： 阅读代码，最终发现是 print_ticks()的问题

```
static void print_ticks() {
    cprintf("%d ticks\n",TICK_NUM);
#ifdef DEBUG_GRADE
    cprintf("End of Test.\n");
    panic("EOT: kernel seems ok.");
#endif
}
```

trap.c->panic.c->kmonitor.c kmonitor.c 打开命令行新世界，添加命令全靠*她*-> static struct commmand commonds[]

结论： kern_init()中的调用过程真的涵盖了所有.c文件