

MOS 系统设计

湖南大学 计算机科学与技术系 刘年

关于MOS

MOS是一个基于MIT6.828 JOS lab的小型操作系统。其设计继承了实验中的微内核思维，提供了低耦合度的图形界面以及内核中的新特性。详细特性列表请见第一章。

该系统前前后后写了大概一个月，其中20多天用于完成MIT的lab，之后修修补补添加新的特性仅有10天左右，时间比较匆忙，设计与编码中会遗留很多或抽象或细节的问题，如发现或有不同看法，欢迎邮件与我联系。

文档构成

开发过程中，堆积了大量的文档。由于网上关于MIT 6.828的资料非常丰富，关于lab部分的文档就不放入这份资料中了。实验相关的文档可以在[MOS Github Repo](#)中的documents文件夹下找到。

本文档适用于在完成MIT JOS lab或者THU ucore OS lab实验的基础上，希望进一步扩展系统的功能，添加图形化界面的同学进行参考。

- 第一章：Micro homemade OS 系统概览

本章主要介绍了MOS支持的特性以及开发环境。

- 第二章：在macOS上搭建MOS编译运行环境

- 第三章：被动读写锁的理解与实现（RWLOCK、LAPIC与IPI）

本章在MOS上实现原子操作，读写锁，核间中断，与被动读写锁。

- 第四章：命令行工具补全计划

本章实现了一些基本的命令行工具，使得MOS的shell:MSH可以自由的浏览文件系统并进行操作。

- 第五章：初探图形化编程

本章利用VBE开启了MOS的图像模式，并在内核中进行了MMIO显存映射，实现了内核的图像显示。

- 第六章：内存管理进阶

本章实现了连续内存分配，提供了kmalloc/kfree工具。

- 第七章：低耦合度图形界面与API设计

本章实现了一个单页面的，低耦合度，易移植的图形化界面。

相关资料

Github Repo: <https://github.com/He11oLiu/MOS>

CSDN Blog : http://blog.csdn.net/he11o_liu

CNBlogs : <http://www.cnblogs.com/he11o-liu>

Email : he11oliu830@gmail.com

第一章：Micro homemade OS

系统总览

MOS是一个基于MIT 6.828 JOS的自制操作系统。在完成JOS实验的基础上添加了很多新特性，并提供了基础的图形界面与应用程序接口。

CGA 显示界面



```
QEMU
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2 4
FS is running
FS can do I/O
Device 1 presence: 1
block cache is good
superblock is good
bitmap is good

# msh in / [10:47:12]
$ sysinfo

-----
| He11o_Liu's JOS version 0.1          |
| Github : https://github.com/He11oLiu/JOS |
| Blog   : http://blog.csdn.net/he11o_liu |
|-----|
System time : [10:47:22]
# msh in / [10:47:22]
$
```

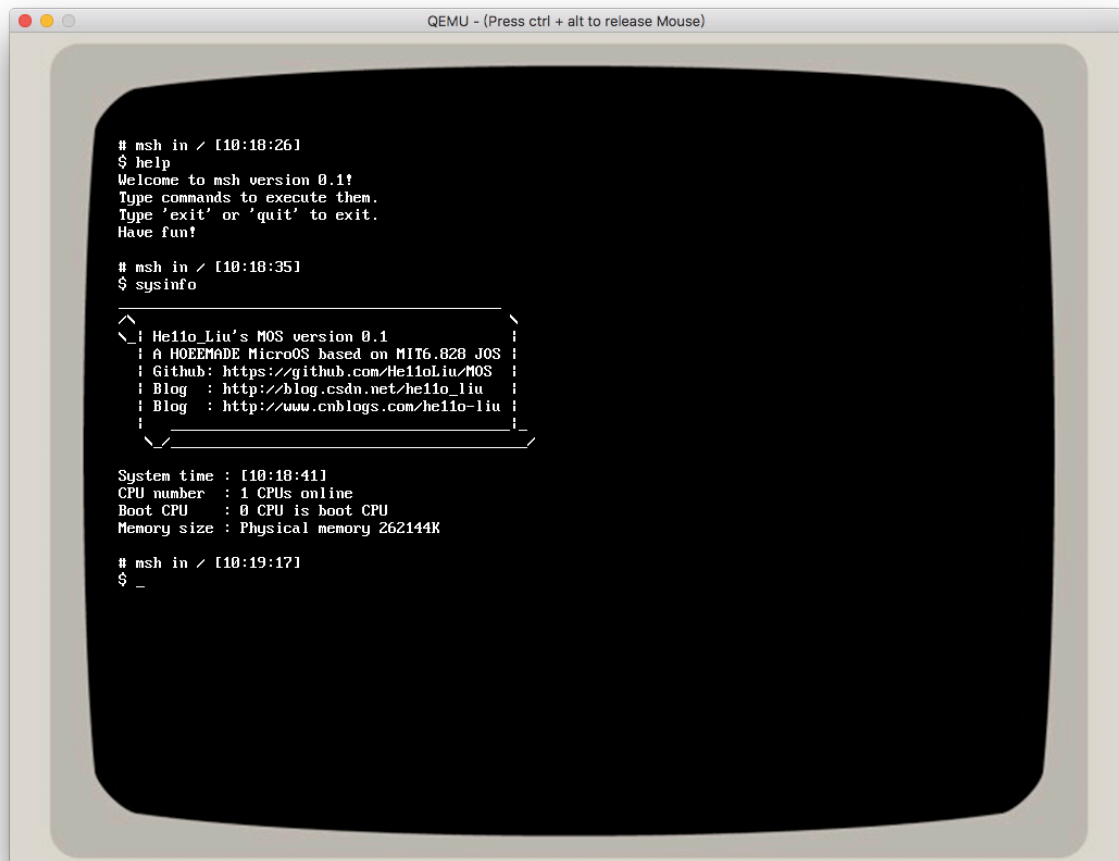
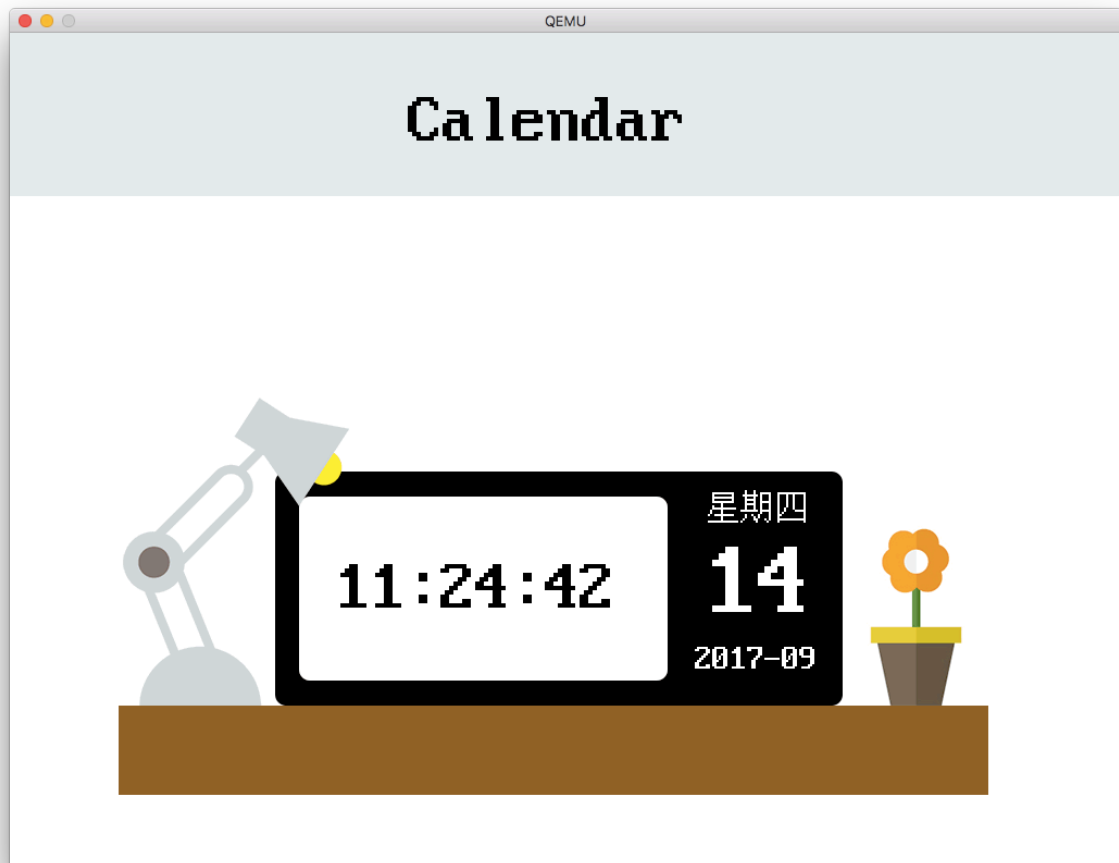
图像显示界面

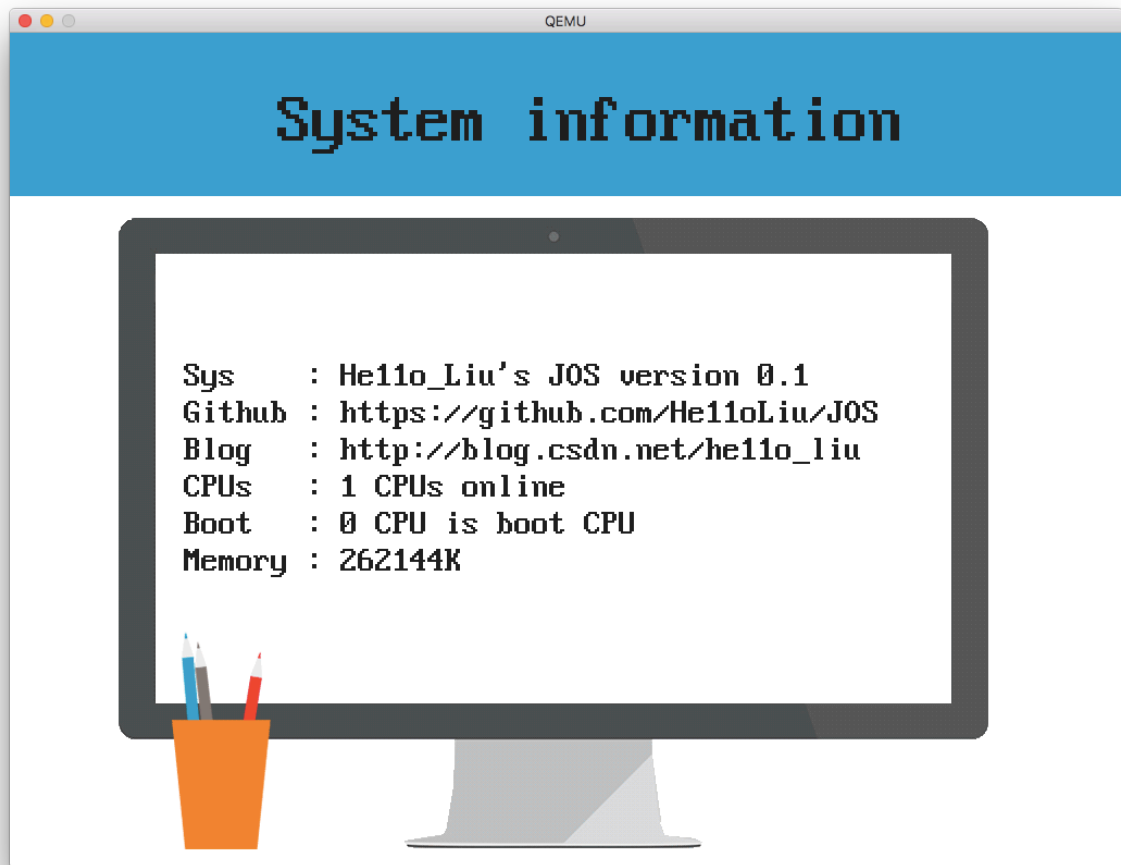
在启动后msh中输入\$ applauncher启动图形化界面应用启动器

```
====Graph mode on====
  scrnx  = 1024
  scrny  = 768
MMIO VRAM = 0xef800000
=====
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2 4
FS is running
FS can do I/O
Device 1 presence: 1
block cache is good
superblock is good
bitmap is good

# msh in / [10:02:07]
$ applauncher
```







环境

需要配置/conf/env.mk下的qemu所在路径

```
$ i386-elf-gcc -v
使用内建 specs。
COLLECT_GCC=i386-elf-gcc
COLLECT_LTO_WRAPPER=/opt/local/libexec/gcc/i386-elf/4.7.2/lto-wrapper
目标: i386-elf
配置
为: /opt/local/var/macports/build/_opt_local_var_macports_sources_rsyn
c.macports.org_macports_release_tarballs_ports_cross_i386-elf-
gcc/i386-elf-gcc/work/gcc-4.7.2/configure --prefix=/opt/local --
target=i386-elf --infodir=/opt/local/share/info --
mandir=/opt/local/share/man --datarootdir=/opt/local/share/i386-elf-
gcc --with-system-zlib --with-gmp=/opt/local --with-mpfr=/opt/local -
-with-mpc=/opt/local --enable-stage1-checking --enable-multilib --
with-newlib --enable-languages=c,c++
线程模型: single
gcc 版本 4.7.2 (GCC)
$ /usr/local/Cellar/qemu/2.10.0/bin/qemu-system-i386 -version
QEMU emulator version 2.10.0
Copyright (c) 2003-2017 Fabrice Bellard and the QEMU Project
developers
```

MOS特性

- 段页式内存管理
- 支持进程(Environments)
 - 进程切换
 - 进程间通讯，通过syscall实现
 - 进程单独地址空间
 - spawn创建进程，fork使用Read Copy Update策略
 - 支持抢断式任务调度
- 支持多核CPU
 - 支持IPI，提供IPI接口
 - 支持大内核锁（基于自旋锁）
- 系统服务syscall
 - 打印字符
 - 获取字符
 - 获取进程编号
 - 回收进程
 - 主动调度
 - fork
 - 设置进程状态

- 申请页，映射页，取消映射
- 用户空间页错误处理入口设置
- IPC进程间通讯
- 用户空间异常处理栈设置
- 支持页错误用户空间处理
- 支持简易文件系统（CS结构）
- 支持文件描述符
- 支持pipe
- 用户空间工具
 - sh简易shell
- 支持原子操作
- 支持读写锁
- 支持针对单一核心IPI
- 支持PRWLock
- 支持基本图形显示
- 支持中英文显示（中英文点阵字库）
- 支持进程工作目录 提供getcwd与chdir
- 新的syscall
 - SYS_env_set_workpath 修改工作路径
- 新的用户程序
 - ls 功能完善
 - pwd 输出当前工作目录
 - cat 接入工作目录
 - touch 由于文件属性没啥可改的，用于创建文件
 - mkdir 创建目录文件
 - msh 更高级的shell 还未完全完工 支持cd 支持默认二进制路径为 bin
- 调整目标磁盘生成工具
- 支持从RTC读取时间
- 支持kmalloc/kfree，支持分配连续空间
- 提供RW/RW用户内核共享framebuffer/palette
- 提供用户GUI接口
- 提供调色板预设
- 提供Applauncher
 - 日历应用
 - 系统信息应用
 - 模拟CGA显示模式的终端程序（基于pipe）

第二章：在macOS上搭建MOS编译运行环境

Tools we need

在搭建环境之前，首先macOS上需要有以下两个工具：

- Homebrew [Homebrew — The missing package manager for macOS](#)
- MacPorts [The MacPorts Project -- Home](#)

运行MOS

- QEMU

有了Homebrew，直接利用brew安装即可安装（自动安装依赖库）

```
$brew install qemu
```

- 将kernel.img与fs.img放在目标目录下（也可以在其他位置，为了下面的Makefile好写）

```
.
├─ Makefile
├─ fs.img
└─ kernel.img
```

- 书写Makefile

```
QEMU=/usr/local/Cellar/qemu/2.10.0/bin/qemu-system-i386 # path to
qemu
run:
    $(QEMU) -drive file=./kernel.img,index=0,media=disk,format=raw
    -serial mon:stdio -vga std -smp 1 -drive
    file=./fs.img,index=1,media=disk,format=raw
```

编译MOS

- i386-elf-gcc

利用Macports来安装i386-elf-gcc

```
$ sudo port -v selfupdate
$ sudo port install i386-elf-gcc
```

Macports会帮你下载源码，编译（非常漫长）

- 修改Makefile中的一些内容

```
diff --git a/GNUMakefile b/GNUMakefile
index adc693e..60fe010 100644
--- a/GNUMakefile
+++ b/GNUMakefile
@@ -33,15 +33,15 @@ TOP = .

# try to infer the correct GCCPREFIX
ifndef GCCPREFIX
-GCCPREFIX := $(shell if i386-MOS-elf-objdump -i 2>&1 | grep
+'^elf32-i386$$' >/dev/null 2>&1; \
-      then echo 'i386-MOS-elf-'; \
+GCCPREFIX := $(shell if i386-elf-objdump -i 2>&1 | grep '^elf32-
i386$$' >/dev/null 2>&1; \
+      then echo 'i386-elf-'; \
        elif objdump -i 2>&1 | grep 'elf32-i386' >/dev/null 2>&1; \
        then echo ''; \
        else echo "***" 1>&2; \
        echo "*** Error: Couldn't find an i386-*-elf version of
GCC/binutils." 1>&2; \
-      echo "*** Is the directory with i386-MOS-elf-gcc in your
PATH?" 1>&2; \
+      echo "*** Is the directory with i386-elf-gcc in your PATH?"
1>&2; \
        echo "*** If your i386-*-elf toolchain is installed with a
command" 1>&2; \
-      echo "*** prefix other than 'i386-MOS-elf-', set your
GCCPREFIX" 1>&2; \
+      echo "*** prefix other than 'i386-elf-', set your
GCCPREFIX" 1>&2; \
        echo "*** environment variable to that prefix and run
'make' again." 1>&2; \
        echo "*** To turn off this error, run 'gmake GCCPREFIX=
...'" 1>&2; \
        echo "***" 1>&2; exit 1; fi)
```

- 修改`.deps`中一些内容

删除`fsformat`的依赖检查

```
obj/fs/: fs/fsformat.c
```

- 修改配置文件中的`qemu`参数

```
QEMU=/usr/local/Cellar/qemu/2.10.0/bin/qemu-system-i386
```

第三章：被动读写锁的理解与实现（RWLOCK、LAPIC与IPI）

本文主要为读论文[Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks](#)的记录，并且尝试将其在MOS上实现。

论文阅读记录

研究背景

- 单核性能提升遇到瓶颈 转向多核提升性能
- 单核主要为计算密集型模型，多核主要为并行化模型
- 并行化模型面临的问题：多个处理器共享数据结构，需要同步原语来保证其一致性
- 更底层：内存一致性 然而会导致可扩展性问题
 - Strict Consistency Memory Barrier
 - Sequential Consistency TSO

可扩展性 Scalable

提到可扩展性，不得不提Amdahl's law

$$S(\text{latency})(s) = 1 / ((1-p) + p/s)$$

其中1-p极为不可以并行的部分，而对于一个处理器，造成(1-P)部分有以下几种原因：

- 内存屏障时等待之前的指令执行完
- MESI模型中等待获取最新值

- 等待其他处理器释放锁
- 多核之间的通讯带宽受限，阻塞

关于读写锁

- 读读不阻塞，读写阻塞
- 适合读数据频率远大于写数据频率的应用
- 单核上的实现思路：读者锁不冲突，当需要加写者锁的时候需要等所有的读者锁释放。利用一个读者计数器来实现。
- 多核上最直观的实现思路：每个核上保存自己的读者锁，写者需要等到所有的读者锁全部释放了才能开始获取锁。

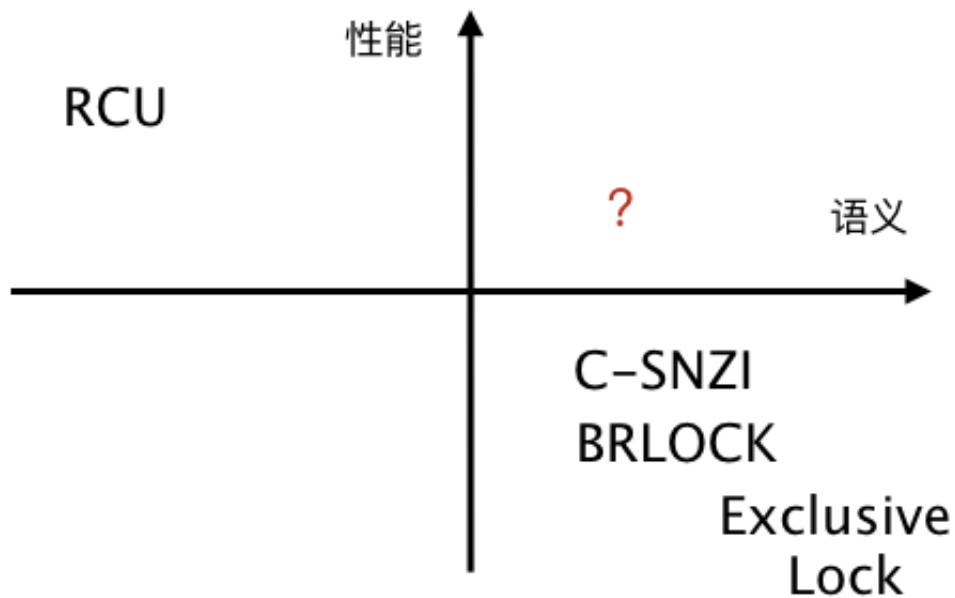
现有的RWLock所做的尝试

BRLOCK C-SNZI

- 读者申请自己核上的锁
- 当只有读者时，由于只是访问自己的核上的锁，所以有良好的扩展性
- 写者需要获取所有核上的互斥锁，恒定延迟，与处理器数量有关。
- SNZI利用树进行了一定优化

RCU

- 弱化语义，可能会读到脏的数据（逻辑矛盾）
- 读者无锁，写着读者可以同时进行
- 先写值再改指针
- 写者开销大，要处理旧的数据
- 垃圾回收(无抢占调度一圈)



Bounded staleness 短内存可见性

所谓短内存可见性，也就是在很短的时间周期内，由于每个核上面的单独cache非常的小，很大几率会被替换掉，从而能看到最新的数据。下面是具体的图表

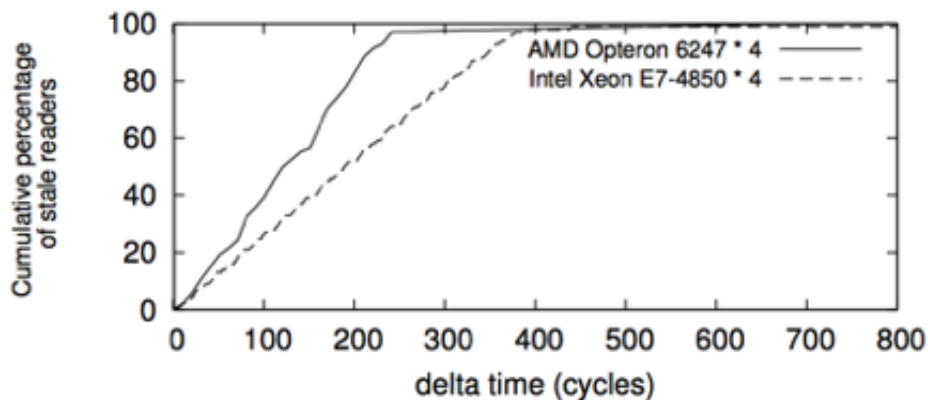


Figure 1: Cumulative percentage of stale readers

*PRWLock*的设计思路

- 在短时间内各个处理器均可以看到最新的版本号
- 利用TSO架构的优势，版本控制隐式表示退出CS区域
- 并不完全依赖于短时间可见，对于特殊情况，保证一致性，利用IPI要求进行Report IPI消息

传递开销较小，且可以相互掩盖。

- 两种模式 支持调度（睡眠与抢占）

PRWLock 的两种模式

Passive Mode

- 用于处理没有经过调度的读者
- 共享内存陈旧化
- 弱控制，通过版本控制隐式反馈+少数情况IPI

Active Mode (用于支持睡眠与调度类似BRLock)

- 用于处理被调度过的进程（睡眠 / 抢占）
- 通过维护active检测数量
- 强控制，主动监听
- 主动等待

PRWLock流程视频：

[优酷视频](#)

PRWLock的正确性

- 写者发现读者获取了最新的版本变量时，由于TSO的特性，也一定看到了写者上的写锁，确信其不会再次进入临界区。
- 对于需要较长时间才能看到最新的版本号或没有读者期望获取读者锁提供了IPI来降低等待时间，避免无限等待。

PRWLock 内核中减少IPIs

- 锁域(Lock Domain)用于表示一个锁对应的处理器范围
- 若上下文切换到了其他的进程，就没必要管这个核的锁了
- 锁域的上下线可以避免一些没有必要的一致性检测
- 注意利用内存屏障来保证一致性

PRWLock 用户态实现

由于在用户态有以下两个特点

- 用户态不能随时关闭抢占(Preemption)
- 用户态不能发送核间中断(IPI)

所以PRWLock在用户态实现的思路如下：

- 利用抢断标记位避免特殊窗口时被抢断
- 写者必须陷入内核态发送IPI

PRWLock 性能分析

读者

- 读者之间无内存屏障（无关联）
- 锁域上下线本来就是极少的操作，用来改善性能的，所以其中的内存屏障影响不大
- 对于长CS区的读者，与传统一样

写者

- IPI只要几百个cycle 本身也要等待
- 多个写者可以直接把锁传递

总结

- 利用了短内存写全局可见时间
- 利用了TSO的特性设计的版本控制来隐式维护语义
- 利用IPI来保证特殊情况

- 利用两种模式支持调度
- 读者之间无关联（内存屏障），提升读者性能
- PWAKE 分布式唤醒，提高了唤醒并行性

移植PRWLock到MOS

MOS的核间中断实现

关于Local APIC

在一个基于APIC的系统中，每一个核心都有一个Local APIC，Local APIC负责处理CPU中特定的中断配置。还有其他事情，它包含了Local Vector Table(LVT)负责配置事件中断。

此外，还有一个CPU外面的IO APIC(例如Intel82093AA)的芯片组，并且提供基于多处理器的中断管理，在多个处理器之间，实现静态或动态的中断触发路由。

Inter-Processor Interrupts (IPIs)是一种由Local APIC触发的中断，一般可以用于多CPU间调度之类的使用

想要开启Local APIC接收中断，则需要设置Spurious Interrupt Vector Register的第8位即可。

使用APIC Timer的最大好处是每个cpu内核都有一个定时器。相反PIT(Programmable Interval Timer)就不这样，PIT是共用的一个。

- 周期触发模式

周期触发模式中，程序设置一个“初始计数”寄存器（Initial Count），同时Local APIC会将这个数复制到“当前计数”寄存器（Current Count）。Local APIC会将这个数（当前计数）递减，直到减到0为止，这时候将触发一个IRQ（可以理解为触发一次中断），与此同时将当前计数恢复到初始计数寄存器的值，然后周而复始的执行上述逻辑。可以使用这种方法通过Local APIC实现定时按照一定时间间隔触发中断的功能。

- 一次性触发模式

同之前一样，但是不会恢复到初始计数。

- TSC-Deadline Modie

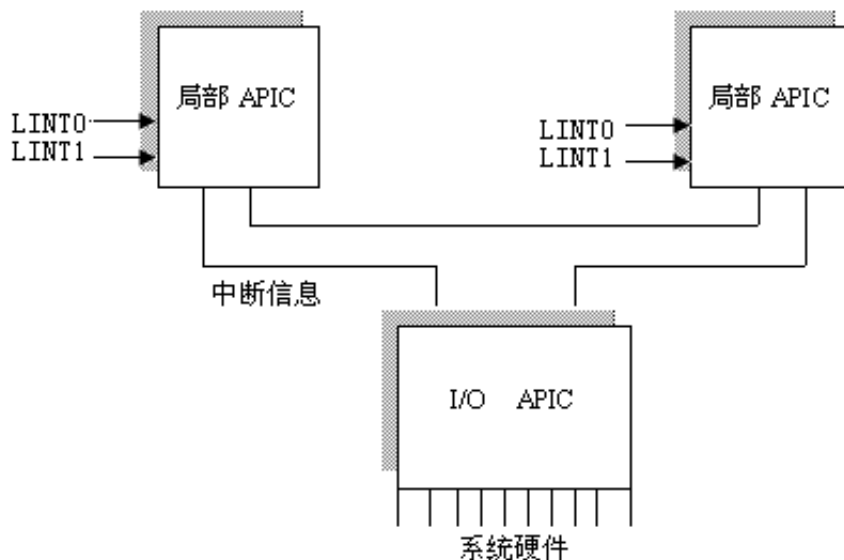
cpu的时间戳到达deadline的时候会触发IRQ

[来源Blog](#)

每个本地APIC都有 32 位的寄存器，一个内部时钟，一个本地定时设备以及为本地中断保留的两条额外的 IRQ 线 LINT0 和 LINT1。所有本地 APIC 都连接到 I/O APIC，形成一个多级 APIC 系统。

Intel x86架构提供LINT0和LINT1两个中断引脚，他们通常与Local APIC相连，用于接收Local APIC传递的中断信号，另外，当Local APIC被禁用的时候，LINT0和LINT1即被配置为INTR和NMI管脚，即外部IO中断管脚和非屏蔽中断管脚。

[来源博客](#)



The local APIC registers are memory mapped to an address that can be found in the MP/MADT tables. Make sure you map these to virtual memory if you are using paging. Each register is 32 bits long, and expects to be written and read as a 32 bit integer. Although each register is 4 bytes, they are all aligned on a 16 byte boundary.

再一次详细查看MOS中的核间中断的实现方式，

由于这段映射，设置了nocache和直写的特性，便于对于IO的操作。

```
void lapic_init(void)
{
    if (!lapicaddr)
        return;
    // lapicaddr is the physical address of the LAPIC's 4K MMIO
    // region. Map it in to virtual memory so we can access it.
    lapic = mmio_map_region(lapicaddr, 4096);
    // Enable local APIC; set spurious interrupt vector.
    lapicw(SVR, ENABLE | (IRQ_OFFSET + IRQ_SPURIOUS));

    // The timer repeatedly counts down at bus frequency
    // from lapic[TICR] and then issues an interrupt.
    // If we cared more about precise timekeeping,
    // TICR would be calibrated using an external time source.
```

```

lapicw(TDCR, X1);
lapicw(TIMER, PERIODIC | (IRQ_OFFSET + IRQ_TIMER));
lapicw(TICR, 10000000);

// Leave LINT0 of the BSP enabled so that it can get
// interrupts from the 8259A chip.
//
// According to Intel MP Specification, the BIOS should
initialize
// BSP's local APIC in Virtual Wire Mode, in which 8259A's
// INTR is virtually connected to BSP's LINTIN0. In this mode,
// we do not need to program the IOAPIC.
if (thiscpu != bootcpu)
    lapicw(LINT0, MASKED);

// Disable NMI (LINT1) on all CPUs
lapicw(LINT1, MASKED);

// Disable performance counter overflow interrupts
// on machines that provide that interrupt entry.
if (((lapic[VER] >> 16) & 0xFF) >= 4)
    lapicw(PCINT, MASKED);

// Map error interrupt to IRQ_ERROR.
lapicw(ERROR, IRQ_OFFSET + IRQ_ERROR);

// Clear error status register (requires back-to-back writes).
lapicw(ESR, 0);
lapicw(ESR, 0);

// Ack any outstanding interrupts.
lapicw(EOI, 0);

// Send an Init Level De-Assert to synchronize arbitration ID's.
lapicw(ICRHI, 0);
lapicw(ICRLO, BCAST | INIT | LEVEL);
while (lapic[ICRLO] & DELIVS)
    ;

// Enable interrupts on the APIC (but not on the processor).
lapicw(TPR, 0);
}

```

lapic_init将LAPIC映射到lapicaddr地址上，并且初始化LAPIC各种中断参数。

```

// Local APIC registers, divided by 4 for use as uint32_t[] indices.
#define ID (0x0020 / 4) // ID

```

这里的宏定义为/4是因为MMIO映射到MMIOaddr，保存在volatile uint32_t *lapic;中。这个单位是uint32_t，故所有的地址均/4

下面来看一下主要的APIC Registers

▪ EOI Register

Write to the register with offset 0xB0 using the value 0 to signal an end of interrupt. A non-zero values causes a general protection fault.

```
#define EOI (0x00B0 / 4)    // EOI
// Acknowledge interrupt.
void lapic_eoi(void)
{
    if (lapic)
        lapicw(EOI, 0);
}
```

▪ Local Vector Table Registers

There are some special interrupts that the processor and LAPIC can generate themselves. While external interrupts are configured in the I/O APIC, these interrupts must be configured using registers in the LAPIC. The most interesting registers are:

0x320 = lapic timer

0x350 = lint0

0x360 = lint1

See the Intel SDM vol 3 for more info.

Bits 0-7	The vector number
Bit 8-11 (reserved for timer)	100b if NMI
Bit 12	Set if interrupt pending.
Bit 13 (reserved for timer)	Polarity, set is low triggered
Bit 14 (reserved for timer)	Remote IRR
Bit 15 (reserved for timer)	trigger mode, set is level triggered
Bit 16	Set to mask
Bits 17-31	Reserved

MOS在这里只保留了BSP的LINT0用于接受8259A的中断，其他的LINT0与LINT1非屏蔽中断，均设置为MASKED

```

// Leave LINT0 of the BSP enabled so that it can get
// interrupts from the 8259A chip.
//
// According to Intel MP Specification, the BIOS should
initialize
// BSP's local APIC in Virtual Wire Mode, in which 8259A's
// INTR is virtually connected to BSP's LINTIN0. In this mode,
// we do not need to program the IOAPIC.
if (thiscpu != bootcpu)
    lapicw(LINT0, MASKED);

// Disable NMI (LINT1) on all CPUs
lapicw(LINT1, MASKED);

```

▪ Spurious Interrupt Vector Register

The offset is 0xF0. The low byte contains the number of the spurious interrupt. As noted above, you should probably set this to 0xFF. To enable the APIC, set bit 8 (or 0x100) of this register. If bit 12 is set then EOI messages will not be broadcast. All the other bits are currently reserved.

```

// Enable local APIC; set spurious interrupt vector.
lapicw(SVR, ENABLE | (IRQ_OFFSET + IRQ_SPURIOUS));

```

▪ Interrupt Command Register

The interrupt command register is made of two 32-bit registers; one at 0x300 and the other at 0x310.

```

#define ICRHI (0x0310 / 4) // Interrupt Command [63:32]
#define ICRL0 (0x0300 / 4) // Interrupt Command [31:0]

```

It is used for sending interrupts to different processors.

The interrupt is issued when 0x300 is written to, but not when 0x310 is written to. Thus, to send an interrupt command one should first write to 0x310, then to 0x300.

需要先写ICRHI，然后在写ICRL0的时候就会产生中断。

At 0x310 there is one field at bits 24-27, which is local APIC ID of the target processor (for a physical destination mode).

```

lapicw(ICRHI, apicid << 24);

```

给ICRHI中断目标核心的local APIC ID。这里的apicid是在MP Floating Pointer Structure读的时候顺序给的cpu_id。

Here is how 0x300 is structured:

Bits

0-7 The vector number, or starting page number for SIPIs

Bits 8-10 The destination mode. 0 is normal, 1 is lowest priority, 2 is SMI, 4 is NMI, 5 can be INIT or INIT level de-assert, 6 is a SIPI.

Bit 11 The destination mode. Clear for a physical destination, or set for a logical destination. If the bit is clear, then the destination field in 0x310 is treated normally.

Bit 12 Delivery status. Cleared when the interrupt has been accepted by the target. You should usually wait until this bit clears after sending an interrupt.

Bit 13 Reserved

Bit 14 Clear for INIT level de-assert, otherwise set.

Bit 15 Set for INIT level de-assert, otherwise clear.

Bits 18-19 Destination type. If this is > 0 then the destination field in 0x310 is ignored. 1 will always send the interrupt to the itself, 2 will send it to all processors, and 3 will send it to all processors aside from the current one. It is best to avoid using modes 1, 2 and 3, and stick with 0.

Bits 20-31 Reserved

ICRLO的分布比较重要

- 其中目标模式有(8-10)

```
#define INIT 0x00000500 // INIT/RESET
#define STARTUP 0x00000600 // Startup IPI
```

- 其中发送模式有(18~19)

```
#define SELF 0x00040000 // Send to self
#define BCAST 0x00080000 // Send to all APICs, including self.
#define OTHERS 0x000C0000 // Send to all APICs, excluding self.
```

不设置的话则为发送给0x310 ICRHI制定的核心。

综上，打包了一个IPI发送的接口，

```
void lapic_ipi(int vector)
{
    lapicw(ICRLO, OTHERS | FIXED | vector);
    while (lapic[ICRLO] & DELIVS)
        ;
}
```

用于发送IPI与IPI ACK均是利用MMIO直接对相应地址书写，比较简单。

这里测试一下，先设置trap中的IPI中断

```
#define T_PRWIPI    20        // IPI report for PRWLock
void prw_ipi_report(struct Trapframe *tf)
{
    cprintf("%d in ipi report\n", cpunum());
}
```

在trap_dispatch中加入对这个中断的分发

```
case PRWIPI:
    prw_ipi_report(tf);
    break;
```

最后在init的时候用bsp发送IPI给所有其他核心

```
lapic_ipi(PRWIPI);
```

设置QEMU模拟4个核心来测试IPI是否正确

```
1 in ipi report
3 in ipi report
2 in ipi report
```

非BSP可以正确的接受IPI并进入中断处理历程。

针对向具体核发送IPI，抽象函数lapic_ipi_dest

```
void lapic_ipi_dest(int dest, int vector)
{
    lapicw(ICRHI, dest << 24);
    lapicw(ICRLO, FIXED | vector);
    while (lapic[ICRLO] & DELIVS)
        ;
}
```

MOS实现传统内核态读写锁

```
typedef struct dumb_rwlock {
    struct spinlock lock;
    atomic_t readers;
} dumb_rwlock;

void rw_initlock(dumb_rwlock *rwlk)
{
    spin_initlock(&rwlk->lock);
    rwlk->readers.counter = 0;
}

void dumb_wrlck(dumb_rwlock *rwlk)
{
    spin_lock(&rwlk->lock);
    while (rwlk->readers.counter > 0)
        asm volatile("pause");
}

void dumb_wrunlock(dumb_rwlock *rwlk)
{
    spin_unlock(&rwlk->lock);
}

void dumb_rdlock(dumb_rwlock *rwlk)
{
    while (1)
    {
        atomic_inc(&rwlk->readers);
        if (!rwlk->lock.locked)
            return;
        atomic_dec(&rwlk->readers);
        while (rwlk->lock.locked)
            asm volatile("pause");
    }
}

void dumb_rdonlock(dumb_rwlock *rwlk)
{
    atomic_dec(&rwlk->readers);
}
```

然后发现一个比较大的问题，MOS没有实现原子操作，先实现原子操作再进行下面的尝试。

MOS 实现原子操作

仿造linux 2.6内核，实现原子操作

```
#ifndef MOS_INC_ATOMIC_H_
#define MOS_INC_ATOMIC_H_

/*
 * Atomic operations that C can't guarantee us. Useful for
 * resource counting etc..
 */

#include <inc/types.h>

#define LOCK "lock ; "

/*
 * Make sure gcc doesn't try to be clever and move things around
 * on us. We need to use _exactly_ the address the user gave us,
 * not some alias that contains the same information.
 */
typedef struct
{
    volatile int counter;
} atomic_t;

#define ATOMIC_INIT(i) \
    { \
        (i) \
    }

/**
 * atomic_read - read atomic variable
 * @v: pointer of type atomic_t
 *
 * Atomically reads the value of @v.
 */
#define atomic_read(v) ((v)->counter)

/**
 * atomic_set - set atomic variable
 * @v: pointer of type atomic_t
 * @i: required value
 *
 * Atomically sets the value of @v to @i.
 */
#define atomic_set(v, i) (((v)->counter) = (i))
```

```

/**
 * atomic_add - add integer to atomic variable
 * @i: integer value to add
 * @v: pointer of type atomic_t
 *
 * Atomically adds @i to @v.
 */
static __inline__ void atomic_add(int i, atomic_t *v)
{
    __asm__ __volatile__(
        LOCK "addl %1,%0"
        : "=m"(v->counter)
        : "ir"(i), "m"(v->counter));
}

/**
 * atomic_sub - subtract the atomic variable
 * @i: integer value to subtract
 * @v: pointer of type atomic_t
 *
 * Atomically subtracts @i from @v.
 */
static __inline__ void atomic_sub(int i, atomic_t *v)
{
    __asm__ __volatile__(
        LOCK "subl %1,%0"
        : "=m"(v->counter)
        : "ir"(i), "m"(v->counter));
}

/**
 * atomic_sub_and_test - subtract value from variable and test result
 * @i: integer value to subtract
 * @v: pointer of type atomic_t
 *
 * Atomically subtracts @i from @v and returns
 * true if the result is zero, or false for all
 * other cases.
 */
static __inline__ int atomic_sub_and_test(int i, atomic_t *v)
{
    unsigned char c;

    __asm__ __volatile__(
        LOCK "subl %2,%0; sete %1"
        : "=m"(v->counter), "=qm"(c)
        : "ir"(i), "m"(v->counter)
        : "memory");
    return c;
}

```

```

}

/**
 * atomic_inc - increment atomic variable
 * @v: pointer of type atomic_t
 *
 * Atomically increments @v by 1.
 */
static __inline__ void atomic_inc(atomic_t *v)
{
    __asm__ __volatile__(
        LOCK "incl %0"
        : "=m"(v->counter)
        : "m"(v->counter));
}

/**
 * atomic_dec - decrement atomic variable
 * @v: pointer of type atomic_t
 *
 * Atomically decrements @v by 1.
 */
static __inline__ void atomic_dec(atomic_t *v)
{
    __asm__ __volatile__(
        LOCK "decl %0"
        : "=m"(v->counter)
        : "m"(v->counter));
}

/**
 * atomic_dec_and_test - decrement and test
 * @v: pointer of type atomic_t
 *
 * Atomically decrements @v by 1 and
 * returns true if the result is 0, or false for all other
 * cases.
 */
static __inline__ int atomic_dec_and_test(atomic_t *v)
{
    unsigned char c;

    __asm__ __volatile__(
        LOCK "decl %0; sete %1"
        : "=m"(v->counter), "=qm"(c)
        : "m"(v->counter)
        : "memory");
    return c != 0;
}

```

```

/**
 * atomic_inc_and_test - increment and test
 * @v: pointer of type atomic_t
 *
 * Atomically increments @v by 1
 * and returns true if the result is zero, or false for all
 * other cases.
 */
static __inline__ int atomic_inc_and_test(atomic_t *v)
{
    unsigned char c;

    __asm__ __volatile__(
        LOCK "incl %0; sete %1"
        : "=m"(v->counter), "=qm"(c)
        : "m"(v->counter)
        : "memory");
    return c != 0;
}

/**
 * atomic_add_negative - add and test if negative
 * @v: pointer of type atomic_t
 * @i: integer value to add
 *
 * Atomically adds @i to @v and returns true
 * if the result is negative, or false when
 * result is greater than or equal to zero.
 */
static __inline__ int atomic_add_negative(int i, atomic_t *v)
{
    unsigned char c;

    __asm__ __volatile__(
        LOCK "addl %2,%0; sets %1"
        : "=m"(v->counter), "=qm"(c)
        : "ir"(i), "m"(v->counter)
        : "memory");
    return c;
}

/**
 * atomic_add_return - add and return
 * @v: pointer of type atomic_t
 * @i: integer value to add
 *
 * Atomically adds @i to @v and returns @i + @v
 */

```

```

static __inline__ int atomic_add_return(int i, atomic_t *v)
{
    int __i;
    /* Modern 486+ processor */
    __i = i;
    __asm__ __volatile__(
        LOCK "xaddl %0, %1;"
        : "=r"(i)
        : "m"(v->counter), "0"(i));
    return i + __i;
}

static __inline__ int atomic_sub_return(int i, atomic_t *v)
{
    return atomic_add_return(-i, v);
}

#define atomic_inc_return(v) (atomic_add_return(1, v))
#define atomic_dec_return(v) (atomic_sub_return(1, v))

/* These are x86-specific, used by some header files */
#define atomic_clear_mask(mask, addr) \
    __asm__ __volatile__(LOCK "andl %0,%1" \
        : \
        : "r"(~(mask)), "m"(*addr) \
        : "memory")

#define atomic_set_mask(mask, addr) \
    __asm__ __volatile__(LOCK "orl %0,%1" \
        : \
        : "r"(mask), "m"(*addr) \
        : "memory")

#endif

```

然后在内核中对读写锁的功能进行测试。

遇到两个问题

- 一个是asm volatile("pause");容易死在那个循环里面，不会重新换到这个CPU中，在DEBUG的时候发现在前后加上cprintf其就会顺利换回来。

```

while (rwlk->lock.locked)
{
    cprintf("");
    asm volatile("pause");
}

```

- 另一个是设计内核中的测试
 - 多核上的输出可能会并行化，要减短输出内容。
 - 在用户空间的锁分享目前不好做，linux是基于文件的。
 - 故设计了两个锁来进行测试

一个是CPU 0的writer锁，一个是reader锁。

```
// test reader-writer lock
rw_initlock(&lock1);
rw_initlock(&lock2);

dumb_wrlock(&lock1);
cprintf("[rw] CPU %d gain writer lock1\n", cpunum());
dumb_rdlock(&lock2);
cprintf("[rw] CPU %d gain reader lock2\n", cpunum());

// Starting non-boot CPUs
boot_aps();

cprintf("[rw] CPU %d going to release writer lock1\n",
cpunum());
dumb_wrunlock(&lock1);
cprintf("[rw] CPU %d going to release reader lock2\n",
cpunum());
dumb_rdunlock(&lock2);
```

对于每个核上，分别获取 lock1 的读着锁与 lock2 的写者锁。添加 `asm volatile("pause");`是想让其他核模拟上线来检测各种情况。

```
dumb_rdlock(&lock1);
cprintf("[rw] %d l1\n", cpunum());
asm volatile("pause");
dumb_rdunlock(&lock1);
cprintf("[rw] %d unl1\n", cpunum());

dumb_wrlock(&lock2);
cprintf("[rw] %d l2\n", cpunum());
asm volatile("pause");
cprintf("[rw] %d unl2\n", cpunum());
dumb_wrunlock(&lock2);
```

在给QEMU四核参数CPUS=4的时候下的运行情况如下：

```
[rw] CPU 0 gain writer lock1
[rw] CPU 0 gain reader lock2
[MP] CPU 1 starting
[MP] CPU 2 starting
[MP] CPU 3 starting
[rw] CPU 0 going to release writer lock1
[rw] CPU 0 going to release reader lock2
[rw] 1 l1
[rw] 2 l1
[rw] 3 l1
[rw] 2 unl1
[rw] 2 l2
[rw] 3 unl1
[rw] 1 unl1
[rw] 2 unl2
[MP] CPU 2 sched
[rw] 3 l2
[rw] 3 unl2
[rw] 1 l2
[MP] CPU 3 sched
[rw] 1 unl2
[MP] CPU 1 sched
```

可以观察到一旦CPU0释放了lock1的写者锁，所有的核均可以获得lock1的读者锁。而后CPU2获得了lock2的写者锁后，其他核上线，CPU3与CPU1只是释放了lock1，无法获得lock2，只有等CPU2释放了lock2才能获取。

这与期望的读写锁的功能是一致的。至此普通读写锁的实现完成。

MOS实现PRWLock

首先有几个重点：

- PRWLock数据结构设计
- 锁的具体实现
- 调度时调用内容
- PRWLock的测试

PRWLock的数据结构

```

enum lock_status
{
    FREE = 0,
    LOCKED,
    PASS,
    PASSIVE
};

struct percpu_prwlock
{
    enum lock_status reader;
    atomic_t version;
};

typedef struct prwlock
{
    enum lock_status writer;
    struct percpu_prwlock lockinfo[NCPU];
    atomic_t active;
    atomic_t version;
} prwlock;

```

对于一个prwlock，除了其主要的版本以及ACTIVE的读者数量，还需要保存每个核心持有该锁的版本号，以及每个核上该锁的读者状态。这里直接通过lockinfo数组索引每个核对应的该锁信息。

而全局内核所拥有的读写锁通过locklist进行索引，在init的时候加入到这个list中去。

```

extern unsigned int prwlocknum;
extern prwlock *locklist[MAXPRWLock];

```

锁的具体操作

初始化操作的时候需要设置各种初值，并将其添加到list中


```

void prw_initlock(prwlock *rwlk)
{
    int i = 0;
    rwlk->writer = FREE;
    for (i = 0; i < NCPU; i++)
    {
        rwlk->lockinfo[i].reader = FREE;
        atomic_set(&rwlk->lockinfo[i].version, 0);
    }
    atomic_set(&rwlk->active, 0);
    atomic_set(&rwlk->version, 0);
    locklist[prwlocknum++] = rwlk;
}

```

剩下的与论文中伪代码的实现思路相同，只是具体调用的函数有一些差别。

读者锁中包括向核心发送ipi。这里只是示意，就没有写PASS的具体部分，可以通过添加一个等待标志变量来实现。

```

void prw_wrlock(prwlock *rwlk)
{
    int newVersion;
    int id = 0;
    unsigned int corewait = 0;
    if (rwlk->writer == PASS)
        return;
    rwlk->writer = LOCKED;
    newVersion = atomic_inc_return(&rwlk->version);
    for (id = 0; id < ncpu; id++)
    {
#ifdef TESTPRW
        cprintf("CPU %d Ver %d\n", id, atomic_read(&rwlk->lockinfo[id].version));
#endif
        if (id != cpunum() && atomic_read(&rwlk->lockinfo[id].version) != newVersion)
        {
            lapic_ipi_dest(id, PRWIPI);
            corewait |= binlist[id];
#ifdef TESTPRW
            cprintf("send ipi %d\n", id);
#endif
        }
    }
    for (id = 0; id < ncpu; id++)
    {
        if (corewait & binlist[id])
        {

```

```

        while (atomic_read(&rwlk->lockinfo[id].version) !=
newVersion)
            asm volatile("pause");
    }
}
while (atomic_read(&rwlk->active) != 0)
{
    lock_kernel();
    sched_yield();
}
}

void prw_wrunlock(prwlock *rwlk)
{
    // if someone waiting to gain write lock rwlk->writer should be
PASS
    rwlk->writer = FREE;
}

void prw_rdlock(prwlock *rwlk)
{
    struct percpu_prwlock *st;
    int lockversion;
    st = &rwlk->lockinfo[cpunum()];
    st->reader = PASSIVE;
    while (rwlk->writer != FREE)
    {
        st->reader = FREE;
        lockversion = atomic_read(&rwlk->version);
        atomic_set(&st->version, lockversion);
        while (rwlk->writer != FREE)
            asm volatile("pause");
        st = &rwlk->lockinfo[cpunum()];
        st->reader = PASSIVE;
    }
}

void prw_rdunlock(prwlock *rwlk)
{
    struct percpu_prwlock *st;
    int lockversion;
    st = &rwlk->lockinfo[cpunum()];
    if (st->reader == PASSIVE)
        st->reader = FREE;
    else
        atomic_dec(&rwlk->active);
    lockversion = atomic_read(&rwlk->version);
    atomic_set(&st->version, lockversion);
}

```

每个核心接到PRWIPI的处理函数

```
void prw_ipi_report(struct Trapframe *tf)
{
    int lockversion, i;
    struct percpu_prwlock *st;
    cprintf("In IPI_report CPU %d\n", cpunum());
    for (i = 0; i < prwlocknum; i++)
    {
        st = &locklist[i]->lockinfo[cpunum()];
        if (st->reader != PASSIVE)
        {
            lockversion = atomic_read(&locklist[i]->version);
            atomic_set(&st->version, lockversion);
        }
    }
}
```

调度时调用内容

调度时需要将所有的锁均进行处理，所以要遍历locklist

```
// Implement PRWLock
if (prwlocknum != 0)
    for (j = 0; j < prwlocknum; j++)
        prw_sched(locklist[j]);
```

具体的prw_sched如下：

```
void prw_sched(prwlock *rwlk)
{
    struct percpu_prwlock *st;
    int lockversion;
    st = &rwlk->lockinfo[cpunum()];
    if (st->reader == PASSIVE)
    {
        atomic_inc(&rwlk->active);
        st->reader = FREE;
    }
    lockversion = atomic_read(&rwlk->version);
    atomic_set(&st->version, lockversion);
}
```

PRWLock的测试

测试PRWLock也比较复杂，由于我们使用的是big kernel lock，所以内核态里面不好测试，直接在初始化开始RR之前测试。这里引入一个新的IPI进行测试。

```
void prw_debug(struct Trapframe *tf)
{
    int needlock = 0;
    cprintf("====CPU %d in prw debug====\n", cpunum());
    if(kernel_lock.cpu == thiscpu && kernel_lock.locked == 1)
    {
        unlock_kernel();
        needlock = 1;
    }
    prw_wrlock(&lock1);
    cprintf("====%d gain lock1====\n", cpunum());
    prw_wrunlock(&lock1);
    cprintf("====%d release lock1====\n", cpunum());
    if(needlock)
        lock_kernel();
}
```

给一个核心发送DEBUGPRW中断，即让其获取lock1的写者锁。

```
#ifdef TESTPRW
    unlock_kernel();
    prw_initlock(&lock1);
    prw_wrlock(&lock1);
    prw_wrunlock(&lock1);
    prw_rdlock(&lock1);
    cprintf("====%d Gain Reader Lock====\n", cpunum());
    lapic_ipi_dest(3, DEBUGPRW);
    for (int i = 0; i < 10000; i++)
        asm volatile("pause");
    prw_rdunlock(&lock1);
    cprintf("====%d release Reader Lock====\n", cpunum());
    lock_kernel();
#endif
```

这里先用unlock_kernel，避免其他核心无法接收中断，最后再lock_kernel，才能开始sched。

测试选择6个核心

```
SMP: CPU 0 found 6 CPU(s)
enabled interrupts: 1 2 4
[MP] CPU 1 starting
[MP] CPU 2 starting
[MP] CPU 3 starting
[MP] CPU 4 starting
```

```
[MP] CPU 5 starting
[MP] CPU 1 sched
[MP] CPU 2 sched
[MP] CPU 3 sched
[MP] CPU 4 sched
[MP] CPU 5 sched
CPU 0 Ver 0
CPU 1 Ver 0
send ipi 1
CPU 2 Ver 0
send ipi 2
CPU 3 Ver 0
send ipi 3
CPU 4 Ver 0
send ipi 4
CPU 5 Ver 0
====0 Gain Reader Lock====
In IPI_report CPU 1
In IPI_report CPU 2
In IPI_report CPU 4
FS is running
====CPU 3 in prw debug====
FS can do I/O
CPU 0 Ver 0
Dsend ipi 0
evice 1 presence: 1
CPU 1 Ver 1
send ipi 1
CPU 2 Ver 2
CPU 3 Ver 1
CPU 4 Ver 1
send ipi 4
CPU 5 Ver 1
send ipi 5
In IPI_report CPU 5
$ block cache is good
superblock is good
bitmap is good
alloc_block is good
file_open is good
file_get_block is good
file_flush is good
file_truncate is good
file rewrite is good
====0 release Reader Lock====
Init finish! Sched start...
====3 gain lock1====
====3 release lock1====
```

当CPU0释放了读着锁之后，CPU3才能够获取lock1，测试正确

第四章：命令行工具补全计划

本文将主要关注用户程序，并补全内核功能。本文主要包括以下用户应用程序：

```
ls      list directory contents
pwd      return working directory name
mkdir    make directories
touch    change file access and modification times(we only support
create file)
cat      concatenate and print files
shell
```

list directory contents

读文件

由于写到这里第一次在用户空间读取文件，简要记录一下读取文件的过程。

首先是文件结构，在lab5中设计文件系统的时候设计的，保存在struct File中，用户可以根据此结构体偏移来找具体的信息。

再是fsformat中提供的与文件系统相关的接口。这里用到了readn。其只是对于read的一层包装。

功能实现

回到ls本身的逻辑上。ls 主要是读取path文件，并将其下所有的文件名全部打印出来。

return working directory name

由于之前写的MOS中每个进程没有写工作目录。这里再加上工作目录。

在struct env中加入工作目录，添加后env如下：

```
struct Env {
    struct Trapframe env_tf;    // Saved registers
    struct Env *env_link;      // Next free Env
    envid_t env_id;            // Unique environment identifier
    envid_t env_parent_id;      // env_id of this env's parent
    enum EnvType env_type;      // Indicates special system
environments
    unsigned env_status;        // Status of the environment
    uint32_t env_runs;          // Number of times environment has run
    int env_cpunum;             // The CPU that the env is running on

    // Address space
    pde_t *env_pgdir;           // Kernel virtual address of page dir

    // Exception handling
    void *env_pgfault_upcall;   // Page fault upcall entry point

    // IPC
    bool env_ipc_recving;       // Env is blocked receiving
    void *env_ipc_dstva;        // VA at which to map received page
    uint32_t env_ipc_value;     // Data value sent to us
    envid_t env_ipc_from;       // envid of the sender
    int env_ipc_perm;           // Perm of page mapping received

    // work path
    char workpath[MAXPATH];
};
```

由于env对于用户是不可以写的，所以要添加新的syscall，进入内核态改。


```
enum {
    SYS_cputs = 0,
    SYS_cgetc,
    SYS_getenv,
    SYS_env_destroy,
    SYS_page_alloc,
    SYS_page_map,
    SYS_page_unmap,
    SYS_exofork,
    SYS_env_set_status,
    SYS_env_set_trapframe,
    SYS_env_set_pgfault_upcall,
    SYS_yield,
    SYS_ipc_try_send,
    SYS_ipc_recv,
    SYS_getcwd,
    SYS_chdir,
    NSYScalls
};
```

由于MOS中用户其实可以读env中的内容，所以getcwd就不陷入内核态了，直接读取就好。

新建dir.c用于存放与目录有关的函数，实现getcwd

```
char *getcwd(char *buffer, int maxlen)
{
    if(!buffer || maxlen < 0)
        return NULL;
    return strncpy((char *)buffer, (const char*)thisenv->workpath, maxlen);
}
```

而对于修改目录，必须要陷入内核态了，新加syscall。

```
int sys_chdir(const char *path)
{
    return syscall(SYS_chdir, 0, (uint32_t)path, 0, 0, 0, 0);
}
```

刚才的dir.c中加入用户接口

```

// change work path
// Return 0 on success,
// Return < 0 on error. Errors are:
// -E_INVAL *path not exist or not a path
int chdir(const char *path)
{
    int r;
    struct Stat st;
    if ((r = stat(path, &st)) < 0)
        return r;
    if(!st.st_isdir)
        return -E_INVAL;
    return sys_chdir(path);
}

```

然后去内核添加功能

```

// change work path
// return 0 on success.
static int
sys_chdir(const char * path)
{
    strcpy((char *)curenv->workpath,path);
    return 0;
}

```

最后实现pwd

```

#include <inc/lib.h>

void umain(int argc, char **argv)
{
    char path[200];
    if(argc > 1)
        printf("%s : too many arguments\n",argv[0]);
    else
        printf("%s\n",getcwd(path,200));
}

```

make directories

发现MOS给我们预留了标识位O_MKDIR，由于与普通的file_create不一样，当有同名的文件存在的时候，但其不是目录的情况下，我们仍然可以创建，所以新写了函数

```

int dir_create(const char *path, struct File **pf)
{
    char name[MAXNAMELEN];
    int r;
    struct File *dir, *f;

    if (((r = walk_path(path, &dir, &f, name)) == 0) &&
        f->f_type == FTYPE_DIR)
        return -E_FILE_EXISTS;
    if (r != -E_NOT_FOUND || dir == 0)
        return r;
    if ((r = dir_alloc_file(dir, &f)) < 0)
        return r;

    // fill struct file
    strcpy(f->f_name, name);
    f->f_type = FTYPE_DIR;

    *pf = f;
    file_flush(dir);
    return 0;
}

```

然后在serve_open下建立新的分支

```

// create dir
else if (req->req_omode & 0_MKDIR)
{
    if ((r = dir_create(path, &f)) < 0)
    {
        if (!(req->req_omode & 0_EXCL) && r == -E_FILE_EXISTS)
            goto try_open;
        if (debug)
            cprintf("file_create failed: %e", r);
        return r;
    }
}

```

在dir.c下提供mkdir函数

```

// make directory
// Return 0 on success,
// Return < 0 on error. Errors are:
// -E_FILE_EXISTS directory already exist
int mkdir(const char *dirname)
{
    char cur_path[MAXPATH];
    int r;
    getcwd(cur_path, MAXPATH);
    strcat(cur_path, dirname);
    if ((r = open(cur_path, O_MKDIR)) < 0)
        return r;
    close(r);
    return 0;
}

```

最后提供用户程序

```

#include <inc/lib.h>
#define MAXPATH 200

void umain(int argc, char **argv)
{
    int r;

    if (argc != 2)
    {
        printf("usage: mkdir directory\n");
        return;
    }
    if((r = mkdir(argv[1])) < 0)
        printf("%s error : %e\n",argv[0],r);
}

```

Create file

创建文件直接利用open中的O_CREAT选项即可。

```
#include <inc/lib.h>
#define MAXPATH 200

void umain(int argc, char **argv)
{
    int r;
    char *filename;
    char pathbuf[MAXPATH];
    if (argc != 2)
    {
        printf("usage: touch filename\n");
        return;
    }
    filename = argv[1];
    if (*filename != '/')
        getcwd(pathbuf, MAXPATH);
    strcat(pathbuf, filename);
    if ((r = open(pathbuf, O_CREAT)) < 0)
        printf("%s error : %e\n", argv[0], r);
    close(r);
}
```

cat

这个只需要修改好支持工作路径即可

```

#include <inc/lib.h>

char buf[8192];

void cat(int f, char *s)
{
    long n;
    int r;

    while ((n = read(f, buf, (long)sizeof(buf))) > 0)
        if ((r = write(1, buf, n)) != n)
            panic("write error copying %s: %e", s, r);
    if (n < 0)
        panic("error reading %s: %e", s, n);
}

void umain(int argc, char **argv)
{
    int f, i;
    char *filename;
    char pathbuf[MAXPATH];

    binaryname = "cat";
    if (argc == 1)
        cat(0, "<stdin>");
    else
        for (i = 1; i < argc; i++)
        {
            filename = argv[i];
            if (*filename != '/')
                getcwd(pathbuf, MAXPATH);
            strcat(pathbuf, filename);
            f = open(pathbuf, O_RDONLY);
            if (f < 0)
                printf("can't open %s: %e\n", argv[i], f);
            else
            {
                cat(f, argv[i]);
                close(f);
            }
        }
}

```

SHELL

写Shell的时候发现问题：之前没有解决fork以及spawn时候的子进程的工作路径的问题。所有再一次修改了系统调用，将系统调用sys_chdir修改为能够设定指定进程的工作目录的系统调用。

```
int sys_env_set_workpath(envid_t envid, const char *path);
```

修改对应的内核处理：

```
// change work path
// return 0 on success.
static int
sys_env_set_workpath(envid_t envid, const char *path)
{
    struct Env *e;
    int ret = envid2env(envid, &e, 1);
    if (ret != 0)
        return ret;
    strcpy((char *)e->workpath, path);
    return 0;
}
```

这样就会fork出来的子进程继承父亲的工作路径。

在shell中加入built-in功能，为未来扩展shell功能提供基础

```

int builtin_cmd(char *cmdline)
{
    int ret;
    int i;
    char cmd[20];
    for (i = 0; cmdline[i] != ' ' && cmdline[i] != '\0'; i++)
        cmd[i] = cmdline[i];
    cmd[i] = '\0';
    if (!strcmp(cmd, "quit") || !strcmp(cmd, "exit"))
        exit();
    if (!strcmp(cmd, "cd"))
    {
        ret = do_cd(cmdline);
        return 1;
    }
    return 0;
}

int do_cd(char *cmdline)
{
    char pathbuf[BUFSIZ];
    int r;
    pathbuf[0] = '\0';
    cmdline += 2;
    while (*cmdline == ' ')
        cmdline++;
    if (*cmdline == '\0')
        return 0;
    if (*cmdline != '/')
    {
        getcwd(pathbuf, BUFSIZ);
    }
    strcat(pathbuf, cmdline);
    if ((r = chdir(pathbuf)) < 0)
        printf("cd error : %e\n", r);
    return 0;
}

```

修改< 与 > 支持当前工作路径

```

case '<': // Input redirection
    // Grab the filename from the argument list
    if (gettoken(0, &t) != 'w')
    {
        cprintf("syntax error: < not followed by word\n");
        exit();
    }
    // Open 't' for reading as file descriptor 0

```



```

// (which environments use as standard input).
// We can't open a file onto a particular descriptor,
// so open the file as 'fd',
// then check whether 'fd' is 0.
// If not, dup 'fd' onto file descriptor 0,
// then close the original 'fd'.

if (t[0] != '/')
    getcwd(argv0buf, MAXPATH);
strcat(argv0buf, t);
if ((fd = open(argv0buf, O_RDONLY)) < 0)
{
    cprintf("Error open %s fail: %e", argv0buf, fd);
    exit();
}
if (fd != 0)
{
    dup(fd, 0);
    close(fd);
}
break;

case '>': // Output redirection
// Grab the filename from the argument list
if (gettoken(0, &t) != 'w')
{
    cprintf("syntax error: > not followed by word\n");
    exit();
}
if (t[0] != '/')
    getcwd(argv0buf, MAXPATH);
strcat(argv0buf, t);
if ((fd = open(argv0buf, O_WRONLY | O_CREAT | O_TRUNC)) <
0)
{
    cprintf("open %s for write: %e", argv0buf, fd);
    exit();
}
if (fd != 1)
{
    dup(fd, 1);
    close(fd);
}
break;

```

创建硬盘镜像

- 利用mmap映射到内存，对内存读写。

```
if ((diskmap = mmap(NULL, nblocks * BLKSIZE, PROT_READ |  
PROT_WRITE,  
MAP_SHARED, diskfd, 0)) == MAP_FAILED)  
panic("mmap %s: %s", name, strerror(errno));
```

从diskmap开始，大小为nblocks * BLKSIZE

- alloc用于分配空间，移动diskpos

```
void *  
alloc(uint32_t bytes)  
{  
    void *start = diskpos;  
    diskpos += ROUNDUP(bytes, BLKSIZE);  
    if (blockof(diskpos) >= nblocks)  
        panic("out of disk blocks");  
    return start;  
}
```

- 块 123 在初始化的时候分配

```
alloc(BLKSIZE);  
super = alloc(BLKSIZE);  
super->s_magic = FS_MAGIC;  
super->s_nblocks = nblocks;  
super->s_root.f_type = FTYPE_DIR;  
strcpy(super->s_root.f_name, "/");  
  
nbitblocks = (nblocks + BLKBITSIZE - 1) / BLKBITSIZE;  
bitmap = alloc(nbitblocks * BLKSIZE);  
memset(bitmap, 0xFF, nbitblocks * BLKSIZE);
```

- writefile用于申请空间，写入磁盘

```
void writefile(struct Dir *dir, const char *name)  
{  
    int r, fd;  
    struct File *f;  
    struct stat st;  
    const char *last;  
    char *start;  
  
    if ((fd = open(name, O_RDONLY)) < 0)  
        panic("open %s: %s", name, strerror(errno));  
    if ((r = fstat(fd, &st)) < 0)  
        panic("stat %s: %s", name, strerror(errno));
```

```

if (!S_ISREG(st.st_mode))
    panic("%s is not a regular file", name);
if (st.st_size >= MAXFILESIZE)
    panic("%s too large", name);

last = strrchr(name, '/');
if (last)
    last++;
else
    last = name;

// 获取目录中的一个空位
f = diradd(dir, FTYPE_REG, last);
// 获取文件存放地址, 分配空间
start = alloc(st.st_size);
// 将文件读如到磁盘中刚刚分配的地址
readn(fd, start, st.st_size);
// 完成文件信息
finishfile(f, blockof(start), st.st_size);
close(fd);
}

void finishfile(struct File *f, uint32_t start, uint32_t len)
{
    int i;
    // 这个是刚才目录下传过来的地址, 直接修改目录下的相应项
    f->f_size = len;
    len = ROUNDUP(len, BLKSIZE);
    for (i = 0; i < len / BLKSIZE && i < NDIRECT; ++i)
        f->f_direct[i] = start + i;
    if (i == NDIRECT)
    {
        uint32_t *ind = alloc(BLKSIZE);
        f->f_indirect = blockof(ind);
        for (; i < len / BLKSIZE; ++i)
            ind[i - NDIRECT] = start + i;
    }
}

```

- 目录结构体与何时将目录写入

```

void startdir(struct File *f, struct Dir *dout)
{
    dout->f = f;
    dout->ents = malloc(MAX_DIR_ENTS * sizeof *dout->ents);
    dout->n = 0;
}

void finishdir(struct Dir *d)
{
    // 目录文件的大小
    int size = d->n * sizeof(struct File);
    // 申请目录文件存放空间
    struct File *start = alloc(size);
    // 将目录的文件内容放进去
    memmove(start, d->ents, size);
    // 补全目录在磁盘当中的信息
    finishfile(d->f, blockof(start), ROUNDUP(size, BLKSIZE));
    free(d->ents);
    d->ents = NULL;
}

```

- 添加bin路径，并在shell中类似path环境变量默认读取bin下的可执行文件

```

opendisk(argv[1]);

startdir(&super->s_root, &root);
f = diradd(&root, FTYPE_DIR, "bin");
startdir(f, &bin);
for (i = 3; i < argc; i++)
    writefile(&bin, argv[i]);
finishdir(&bin);
finishdir(&root);

finishdisk();

```

获取时间

又新增一个syscall，这里不再累述，利用mc146818_read获取cmos时间即可。

```

int gettimeofday(struct tm *tm)
{
    unsigned datas, datam, datah;
    int i;
    tm->tm_sec = BCD_TO_BIN(mc146818_read(0));
    tm->tm_min = BCD_TO_BIN(mc146818_read(2));
    tm->tm_hour = BCD_TO_BIN(mc146818_read(4)) + TIMEZONE;
    tm->tm_wday = BCD_TO_BIN(mc146818_read(6));
    tm->tm_mday = BCD_TO_BIN(mc146818_read(7));
    tm->tm_mon = BCD_TO_BIN(mc146818_read(8));
    tm->tm_year = BCD_TO_BIN(mc146818_read(9));
    return 0;
}

```

实机运行输出

```

check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
====Graph mode on====
    scrnx = 1024
    scrny = 768
MMIO VRAM = 0xef803000
=====
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2 4
FS is running
FS can do I/O
Device 1 presence: 1
block cache is good
superblock is good
bitmap is good

# msh in / [12: 4:28]
$ cd documents

# msh in /documents/ [12: 4:35]
$ echo hello liu > hello

# msh in /documents/ [12: 4:45]
$ cat hello
hello liu

```

```
# msh in /documents/ [12: 4:49]
```

```
$ cd /bin
```

```
# msh in /bin/ [12: 4:54]
```

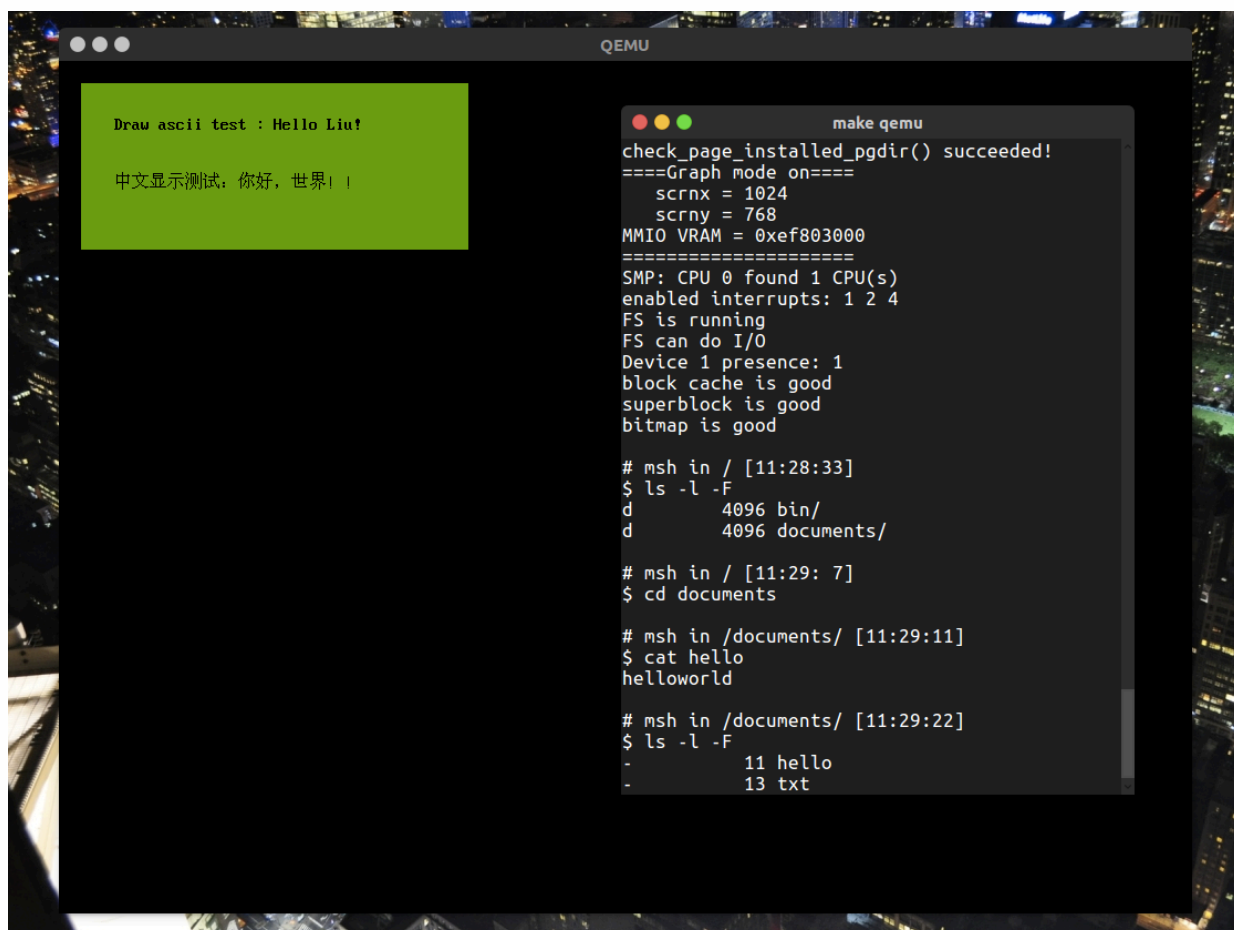
```
$ ls -l -F
```

```
-          37 newmotd
-          92 motd
-         447 lorem
-         132 script
-        2916 testshell.key
-         113 testshell.sh
-       20308 cat
-       20076 echo
-       20508 ls
-       20332 lsfd
-       25060 sh
-       20076 hello
-       20276 pwd
-       20276 mkdir
-       20280 touch
-       29208 msh
```

```
# msh in /bin/ [12: 4:57]
```

```
$
```

第五章：初探图形编程



About VESA

Video Electronics Standards Association（视频电子标准协会，简称“VESA”）是制定计算机和小型工作站视频设备标准的国际组织，1989年由NEC及其他8家显卡制造商赞助成立。创立VESA的原始目的是要制定分辨率为800x600的SVGA视频显示标准。其后，VESA公告一系列的个人电脑视频周边功能的相关标准。

VBE 功能调用

参考博客[CSDN博客](#)

VBE功能调用

- AH必须等于4FH，表明是VBE标准
- AL等于VBE功能号， $0 \leq AL \leq 0BH$
- BL等于子功能号，也可以没有子功能
- 调用INT 10H
- 返回值在AX中
 - AL=4FH：支持该功能
 - AL!=4FH：不支持该功能
 - AH=00H：调用成功
 - AH=01H：调用失败
 - AH=02H：当前硬件配置不支持该功能
 - AH=03H：当前的显示模式不支持该功能

具体功能

此部分参考VESA编程——GUI离我们并不遥远，原作者博客已关闭。

功能0x00：返回控制器信息

输入：			
	AX	= 4F00h	返回VBE控制器信息
	ES:DI	=	指向存放VbeInfoBlock结构体的缓冲区指针
输出：			
	AX	=	VBE返回状态

这个函数返回一个VbeInfoBlock结构体，该结构体定义如下：


```
// Vbe Info Block
typedef struct {
    unsigned char    vbe_signature;
    unsigned short   vbe_version;
    unsigned long    oem_string_ptr;
    unsigned char    capabilities;
    unsigned long    video_mode_ptr;
    unsigned short   total_memory;
    unsigned short   oem_software_rev;
    unsigned long    oem_vendor_name_ptr;
    unsigned long    oem_product_name_ptr;
    unsigned long    oem_product_rev_ptr;
    unsigned char    reserved[222];
    unsigned char    oem_data[256];
} VbeInfoBlock;
```

- vbe_signature是VBE标识，应该填充的是“VESA”
- vbe_version是VBE版本，如果是0300h则表示3.0版本
- oem_string_ptr是指向oem字符串的指针，该指针是一个16位的selector:offset形式的指针，在实模式下可以直接使用。
- video_mode_ptr是指向视频模式列表的指针，与oem_string_ptr类型一样
- total_memory是64kb内存块的个数
- oem_vendor_name_ptr是指向厂商名字字符串的指针
- oem_product_name_ptr是指向产品名称字符串的指针

功能01 返回VBE模式信息

输入：

AX	=	0x4F01	返回VBE模式信息
CX	=		模式号
ES:DI	=		指向VBE特定模式信息块的指针

输出：

AX	=	VBE返回值
----	---	--------

这个函数返回一个ModeInfoBlock结构体，其中重要的部分如下：

- mode_attributes字段，这个字段描述了图形模式的一些重要属性。其中最重要的是第4位和第7位。第4位为1表示图形模式(Graphics mode)，为0表示文本模式(Text mode)。第7位为1表示线性帧缓冲模式(Linear frame buffer mode)，为0表示非线性帧缓冲模式。我们主要要检查这两个位。
- xresolution，表示该视频模式的X分辨率。
- yresolution，表示该视频模式的Y分辨率。
- bits_per_pixel，表示该视频模式每个像素所占的位数。
- phys_base_ptr，这是一个非常重要的字段，它给出了平坦内存帧缓冲区的物理地址，你可以理解为显存的首地址。如果每个像素占32位的话，屏幕左上角第一个点所占的缓冲区就

是phys_base_ptr所指的第一个4个字节。按照先行后列的顺序，每个像素点所占缓冲区依次紧密排列。我们要想在屏幕上画出像素点，就得操作以phys_base_ptr为起始的物理内存空间。

功能02 设置VBE模式信息

输入：

AX	=	4F02h	设置VBE模式
BX	=		需要设置的模式
		D0 - D8	= 模式号
		D9 - D10	= 保留（必须为0）
		D11	= 0 使用当前缺省刷新率
			= 1 使用用户指定的CRTC值为刷新率
		D12 - D13	= 为VBE/AF保留（必须为0）
		D14	= 0 使用窗口帧缓冲区模式
			= 1 使用线性/平坦帧缓冲区模式
		D15	= 0 清除显示内存
			= 1 不清除显示内存
ES:DI	=		指向CRTCInfoBlock结构体的指针

输出：

AX	=	VBE返回状态
----	---	---------

MOS实现

- Qemu需要添加-vga std

```
QEMUOPTS = -drive
file=$(OBJDIR)/kern/kernel.img,index=0,media=disk,format=raw -serial
mon:stdio -gdb tcp::$(GDBPORT) -vga std
```

- 在boot中实模式获取VBE，设置VBE

```
sti
call    getvideomode
call    setvideomode
cli
```

先获取VBE模式，填充di，然后切换模式，设置VBE模式。

根据上面查的VESA资料，调用函数，实现这两个功能。

```

getvideomode:
    mov $0x4f01, %ax # get mode
    mov $0x105, %cx  # mode 0x105
    mov $0x8000, %di # mode info block address
    int $0x10        # VBE int
    ret

setvideomode:
    movw $0x4f02, %ax # set mode
    movw $0x105, %bx
    movw $0x8000, %di
    int $0x10        # VBE int
    movl 40(%di), %eax # get memory address
    movl %eax, info_vram
    movw 18(%di), %ax # get x resolution
    movw %ax, info_scrnx
    movw 20(%di), %ax # get y resolution
    movw %ax, info_scrny
    ret

```

这里设计了一个结构体来存放从boot传来的东西。

```

struct boot_info
{
    short scrnx, scrny;
    char *vram;
};

```

- init的时候，设计一个获取boot_info的模块

```

static void get_boot_info(void)
{
    struct boot_info *info = (struct boot_info *) (KADDR(0x0ff0));
    // Init Graph info
    graph.scrnx = info->scrnx;
    graph.scrny = info->scrny;
    graph.vram = info->vram;
}

```

这个地方我选择初始化memory layout之后，开启真正的页表的时候才获取信息。所以这里要用KADDR进行物理地址到KVA的转化。

- 设计一个全局用于保存图像相关信息的结构体

```

struct graph_info
{
    short scrnx,scrny;
    char *vram;
};

extern struct graph_info graph;

```

- 利用MMIO映射一片显存

```

void graph_init()
{
    int i;
    // Init Graph MMIO
    graph.vram =
        (char *)mmio_map_region((physaddr_t)graph.vram,
                                graph.scrnx * graph.scrny);

    cprintf("====Graph mode on====\n");
    cprintf("    scrnx = %d\n",graph.scrnx);
    cprintf("    scrny = %d\n",graph.scrny);
    cprintf("MMIO VRAM = %#x\n",graph.vram);
    cprintf("=====\n");
    // Draw Screen
    for (i = 0; i < graph.scrnx * graph.scrny; i++)
        *(graph.vram + i) = 0x34;
}

```

补充图像库

上面基本已经实现了图像显示基本平台。现在补充一些常用的图像库。

```

#define PIXEL(x, y) *(graph.vram + x + (y * graph.scrnx))
int draw_screen(uint8_t color)
{
    int i;
    for (i = 0; i < graph.scrnx * graph.scrny; i++)
        *(graph.vram + i) = color;
    return 0;
}

int draw_pixel(short x, short y, uint8_t color)
{
    if ((x >= graph.scrnx) || (y >= graph.scrny))
        return -1;
    *(graph.vram + x + (y * graph.scrnx)) = color;
    return 0;
}

int draw_rect(short x, short y, short l, short w, uint8_t color)
{
    int i, j;
    w = (y + w) > graph.scrny ? graph.scrny : (y + w);
    l = (x + l) > graph.scrnx ? graph.scrnx : (x + l);
    for (j = y; j < w; j++)
        for (i = x; i < l; i++)
            *(graph.vram + i + j * graph.scrnx) = color;
    return 0;
}

```

字库实现

这部分也是老生常谈了，板子上各种系统都实现过点阵字库。

```

int draw_ascii(short x, short y, char *str, uint8_t color)
{
    char *font;
    int i, j, k = 0;
    for (k = 0; str[k] != 0; k++)
    {
        font = (char *) (ascii_8_16 + (str[k] - 0x20) * 16);
        for (i = 0; i < 16; i++)
            for (j = 0; j < 8; j++)
                if ((font[i] << j) & 0x80)
                    PIXEL((x + j), (y + i)) = color;
        x += 8;
    }
    return k;
}

int draw_cn(short x, short y, char *str, uint8_t color)
{
    uint16_t font;
    int i, j, k;
    int offset;
    for (k = 0; str[k] != 0; k += 2)
    {
        offset = ((char)(str[k] - 0xa0 - 1) * 94 +
                  ((char)(str[k + 1] - 0xa0) - 1)) *
                  32;
        for (i = 0; i < 16; i++)
        {
            font = cn_lib[offset + i * 2] << 8 |
                   cn_lib[offset + i * 2 + 1];
            for (j = 0; j < 16; j++)
                if ((font << j) & 0x8000)
                    PIXEL((x + j), (y + i)) = color;
        }
        x += 16;
    }
    return 0;
}

```

直接把之前单片机的点阵字库拿过来，不过单片机当时开发环境是win，找的字库寻址模式是GB2312的。这里把这个文件的编码改为GB2312来正确编码中文即可。实现效果见文章头。

总结

至此，基本的GUI底层接口已基本实现，后面的就是各种数据结构的设计，窗口树设计之类。这里暂不打算继续深究，转而研究其余内核的东西。

第六章：内存管理进阶

在lab2中实现的内存管理只是针对单页建立freelist，list中用链表连接起来的都是代表单页的结构体struct PageInfo。且每次释放页，都是丢在这个free_list的头。这样有几个问题：

- 不能分配大于4k的连续空间（后面做frambuf的时候要用到）
- 不断地加到空闲列表的头会使内存空间十分的混乱。不利于内存管理。

所以先要设计一种能够支持分配连续空间的机制。

一种简单的实现

最简单的想法就是保持现有的不动，freelist保证从高地址到低地址。

这要求在page_free的时候做一下手脚，放到合适的位置。

在npages_alloc的时候，找到连续的空闲的页即可。

Free

既然最主要的是在free的时候需要维护freelist按照地址的大小排列，那么就先简单将page_free重新写一下，找到合适的位置再进行插入操作。

特别要注意是否刚好应该插入到free list的头的情况：

如果刚好是最高地址，那么就需要修改page_free_list

```
if (page2pa(page_free_list) < page2pa(pp))
{
    cur = page_free_list;
    page_free_list = pp;
    pp->pp_link = cur;
    return;
}
```

否则需要遍历来查找位置插入


```

cur = page_free_list;
prev = page_free_list;
while (page2pa(cur) > page2pa(pp))
{
    prev = cur;
    if ((cur = cur->pp_link) == NULL)
        break;
}
prev->pp_link = pp;
pp->pp_link = cur;

```

写完简单的free之后，我们可以确保freelist的顺序问题了。

npages_alloc

再来看主要的alloc，其核心思想则是检查是否刚好有连续的空间能够分配出去，这里用consecutive来记录累计连续的页数。

通过pageInfo在pages的数组的偏移即可知道其对应的地址，如果这个偏移是连续的，则代表着一块连续的空间：

```

(int)(cur - pages) == (int)(prev - pages) - 1

```

其中cur为当前遍历到的pageInfo，而prev是上次遍历的，通过上面的表达式可以判断是否为连续。

如果找到了合适的一块空间，则需要

- 维护freelist，将这块空间前的最后一页连接到分配走的后面一页。

同样注意是否有存在需要换头的情况

```

if (pp == page_free_list)
    page_free_list = cur;
else
    pp_prev->pp_link = cur;

```

- 初始化页属性与空间

```

if (alloc_flags & ALLOC_ZERO)
    memset(page2kva(prev), 0, n * PGSIZE);
// clear pp link
for (i = 0; i < n; i++)
    (prev + i)->pp_link = NULL;
return prev;

```

完整的npages_alloc见下:

```
struct PageInfo *npages_alloc(unsigned int n, int alloc_flags)
{
    struct PageInfo *cur;
    struct PageInfo *prev;
    struct PageInfo *pp;
    struct PageInfo *pp_prev;
    unsigned int i;
    unsigned int consecutive = 1;
    if (page_free_list == NULL)
        return NULL;
    pp = page_free_list;
    pp_prev = page_free_list;
    prev = page_free_list;
    cur = page_free_list->pp_link;

    while (consecutive < n && cur != NULL)
    {
        if ((int)(cur - pages) != (int)(prev - pages) - 1)
        {
            consecutive = 1;
            pp_prev = prev;
            pp = cur;
        }
        else
            consecutive++;
        prev = cur;
        cur = cur->pp_link;
    }
    if (consecutive == n)
    {
        // alloc flags
        if (alloc_flags & ALLOC_ZERO)
            memset(page2kva(prev), 0, n * PGSIZE);
        // update page_free_list
        if (pp == page_free_list)
            page_free_list = cur;
        else
            pp_prev->pp_link = cur;
        // clear pp link
        for (i = 0; i < n; i++)
            (prev + i)->pp_link = NULL;
        return prev;
    }
    return NULL;
}
```

kmalloc

实现了npages_alloc，再来实现malloc就简单了，主要两个问题

- 需要分配多少页？通过ROUNDUP后再除以页大小即可。
- 其对应的虚拟地址是多少？利用page2kva转换。

完整的kmalloc如下。

```
void *kmalloc(size_t size)
{
    struct PageInfo *pp;
    int npages;
    size = ROUNDUP(size, PGSIZE);
    npages = size / PGSIZE;
    if ((pp = npages_alloc(npages, 1)) == NULL)
        return NULL;
    return page2kva(pp);
}
```

此时已经可以测试是否基本正确。

npages_free

为了实现free，还需要实现npages_free，这个和之前实现的思路相同。

主要注意如何连接起freelist。

```
prev->pp_link = pp + n - 1;
pp->pp_link = cur;
for (i = 1; i < n; i++)
    (pp + i)->pp_link = pp + i - 1;
```

其中prev是合适位置的之前一个，cur是合适位置的下一个。

npages_free完整实现见下。

```

void npages_free(struct PageInfo *pp, unsigned int n)
{
    struct PageInfo *cur, *prev;
    unsigned int i;
    for (i = 0; i < n; i++)
    {
        if ((pp + i)->pp_ref)
            panic("npages_free error: (pp+%d)->pp_ref != 0", i);
        if ((pp + i)->pp_link != NULL)
            panic("npages_free error: (pp+%d)->pp_link != NULL", i);
    }
    if (page2pa(page_free_list) < page2pa(pp))
    {
        cur = page_free_list;
        page_free_list = pp + n - 1;
        pp->pp_link = cur;
        for (i = 1; i < n; i++)
            (pp + i)->pp_link = pp + i - 1;
        return;
    }
    cur = page_free_list;
    prev = page_free_list;
    while (page2pa(cur) > page2pa(pp))
    {
        prev = cur;
        if ((cur = cur->pp_link) == NULL)
            break;
    }
    // test use
    cprintf("find prev %d cur %d\n", (int)(prev - pages), (int)(cur -
pages));

    prev->pp_link = pp + n - 1;
    pp->pp_link = cur;
    for (i = 1; i < n; i++)
        (pp + i)->pp_link = pp + i - 1;
    return;
}

```

kfree

和kmalloc类似，算出大小释放即可

```

void kfree(void *kva, size_t size)
{
    struct PageInfo *pp = pa2page(PADDR(kva));
    int npages;
    size = ROUNDUP(size, PGSIZE);
    npages = size / PGSIZE;
    npages_free(pp, npages);
}

```

兼容单页分配

为了和已经写的兼容，直接调用npages_xx即可

```

struct PageInfo *page_alloc(int alloc_flags)
{
    return npages_alloc(1, alloc_flags);
}
void page_free(struct PageInfo *pp)
{
    npages_free(pp, 1);
}

```

测试

首先取消有关内存的几个check，有几个原因导致：

- free后不是放在头的
- check page中做了mmio映射，从而kern pgdir中对应的pd就是PTE_P

所以check除了第一个全部取消。

freelist free后顺序

首先是单页free后的顺序测试

```

struct PageInfo *alloc1 = page_alloc(1);
struct PageInfo *alloc2 = page_alloc(1);
struct PageInfo *alloc3 = page_alloc(1);
cprintf("alloc 1 at %d\n", (int)(alloc1 - pages));
cprintf("alloc 2 at %d\n", (int)(alloc2 - pages));
cprintf("alloc 3 at %d\n", (int)(alloc3 - pages));
page_free(alloc1);
page_free(alloc3);

```

刚才free中写了测试语句，输出如下：

```
alloc 1 at 1023
alloc 2 at 1022
alloc 3 at 1021
find prev 1023 cur 1020
```

malloc/free 测试

之前在写显存的双缓冲的时候，委曲求全选择了静态分配。这里重新使用动态分配并进行测试：

```
void init_framebuffer(){
    void *malloc_free_test;
    if((framebuffer = (uint8_t *) kmalloc((size_t)
(graph.scrnx*graph.scrny)))== NULL)
        panic("kmalloc error!");
    malloc_free_test = framebuffer;
    kfree(framebuffer, (size_t)(graph.scrnx*graph.scrny));
    if((framebuffer = (uint8_t *) kmalloc((size_t)
(graph.scrnx*graph.scrny)))== NULL)
        panic("kmalloc error!");
    if(malloc_free_test == framebuffer)
        cprintf("kmalloc/kfree check success\n");
    else
        panic("kmalloc/kfree error!\n");

    // framebuffer = tmpbuf;
    if(framebuffer == NULL)
        panic("Not enough memory for framebuffer!");
}
```

测试输出正确。

更高效的空间链表设计

上面说过，原来的空闲链表是连接的一个一个页的信息。但是由于MOS在设计的时候希望能够每个页有一个对应的页信息。并利用此来从pageinfo找到kva。所以设计的新的pageinfo结构体仍然是每个页拥有一个。

Free Area

新增一个概念：Free Area。所谓Free Area则是空闲页连接起来的一片区域，直到下一个被使用的页。

这将涉及到两个重要的信息：

- FreeArea的第一个页是谁
- FreeArea的大小是多少

以及分配，释放的策略的不同。这里使用First fit的策略来分配页。

新的PageInfo结构体

设计的新的PageInfo结构体：

```
#define FIRSTPAGE 0x1
struct PageInfo {
    // Old:Next page on the free list.
    // New:Fist page in next free area.
    struct PageInfo *pp_link;
    // some infomation about this page
    uint8_t flags;
    // size of this free area.
    uint32_t freesize;
    // pp_ref is the count of pointers (usually in page table
    entries)
    // to this page, for pages allocated using page_alloc.
    // Pages allocated at boot time using pmap.c's
    // boot_alloc do not have valid reference count fields.
    uint16_t pp_ref;
};
```

具体分配策略与释放策略

npages_alloc

分配一个大小为n pages的页，要遍历free list。

这里的free list保存的不再是一页页的链表，而是Free area的链表。

找到第一个freesize > n的区域，分配出去，并且设置后面的一页为新的Area头。

npages_free

释放一个大小为n pages的页，也要遍历free list，找到地址在其前的进的看能不能加入它的area，找到地址在其后的，看看能否加入。不能的话需要插入一个新的独立的area。

类似的思路在ucore中写过了，在MOS中由于init的时候比较复杂，链表的链接问题比较严重，这里就不尝试了。关于ucore的实现可见我的[CSDN 博客](#)。

第七章：低耦合度图形界面与API设计



本文将介绍在本人MOS中实现的简单图形界面应用程序接口，应用程序启动器，以及一些利用了图形界面的示例应用程序。

本文主要涉及以下部分：

- 内核/用户RW/RW调色板framebuffer共享区域
- 8bit颜色深度BMP格式图片读取与绘制
 - 读BMP头总是出现问题？不合理的数据？
 - 为啥读出来的图片颜色怪怪的！！
 - 为啥是倒的，还有的运气不好出错了
 - 如果是想绘制多个图片在一页上，调色板问题？？

- 如果读到一个32位色的图片咋办?
- 图形化界面数据结构，框架以及接口设计
- 利用图形化接口实现应用程序：
 - 日历程序（实时时钟刷新）
 - 系统信息获取
 - 终端CGA模拟器

PART1

framebuffer

在图形库中，已经将图形模式打开，将显存映射到内存中的一段空间。并进行了简单的测试。

实际上，直接对显存写是很不负责任的行为。很早之前在写java的界面的时候，就接触了双缓冲技术，其实与显示有关的思想都是差不多的，我们应该提供一个framebuffer。当完成一个frame后，再将这个frame update到显存中。

```
uint8_t *framebuffer;
void init_framebuffer(){
    if((framebuffer = (uint8_t *) kmalloc((size_t)
    (graph.scrnx*graph.scrny)))== NULL)
        panic("Not enough memory for framebuffer!");
}

void update_screen(){
    memcpy(graph.vram, framebuffer, graph.scrnx*graph.scrny);
}
```

经过实现kmalloc与kfree，已经可以分配这个缓冲区，并直接向缓冲区写入，最后再进行update

```
#define PIXEL(x, y) *(framebuffer + x + (y * graph.scrnx))
int draw_xx()
{
    xxx;
    update_screen();
}
```

canvas (这种思路已经废弃)

从一个单一的应用程序角度来看，应分配一个单独的画布，然后选择在一个位置显示。

```
typedef struct canvas
{
    uint16_t width;
    uint16_t height;
    uint8_t *data;
} canvas_t;
```

设计的模式是，与文件系统服务器类似，提供一个图形系统服务器，用于接收从其他的程序发来的请求。请求包括显示的位置，以及canvas。该服务器将canvas写入frambuffer并update。其他程序与图形服务器通过IPC进行通讯。

剩余的事情就可以交给用户空间了。包括对canvas的处理，更新显示，添加各种元件。之前写的字库也可以不用写在内核了...

首先实现绘制canvas。

```
int draw_canvas(uint16_t x, uint16_t y, canvas_t *canvas)
{
    int i, j;
    int width = (x + canvas->width) > graph.scrnx ? graph.scrnx : (x
+ canvas->width);
    int height = (y + canvas->height) > graph.scrny ? graph.scrny :
(y + canvas->height);
    cprintf("width %d height %d\n",width,height);
    for (j = y; j < height; j++)
        for (i = x; i < width; i++)
            PIXEL(i, j) = *(canvas->data + (i - x) + (j - y) *
canvas->width);
    update_screen();
    return 0;
}
```

然后在lib中新建canvas的相关方法：

```
int canvas_init(uint16_t width, uint16_t height, canvas_t *canvas);
int canvas_draw_bg(uint8_t color, canvas_t *canvas);
int canvas_draw_ascii(uint16_t x, uint16_t y, char *str, uint8_t
color, canvas_t *canvas);
int canvas_draw_cn(uint16_t x, uint16_t y, char *str, uint8_t color,
canvas_t *canvas);
int canvas_draw_rect(uint16_t x, uint16_t y, uint16_t l, uint16_t w,
uint8_t color, canvas_t *canvas);
```

其中只需要将原来的PIXAL宏换为

```
#define CANVAS_PIXEL(canvas, x, y) *(canvas->data + x + (y * canvas->width))
```

测试canvas

```
canvas_t canvas_test;
canvas_init(300, 200, &canvas_test);
uint8_t testcanvas[60000];
canvas_test.data = (uint8_t *)testcanvas;
canvas_draw_bg(0x22, &canvas_test);
canvas_draw_ascii((uint16_t)2, (uint16_t)2, test_ascii,
(uint8_t)0xff, &canvas_test);
canvas_draw_cn((uint16_t)2, (uint16_t)50, test_cn, (uint8_t)0xff,
&canvas_test);
draw_canvas(500, 500, &canvas_test);
```

图像处理的两种设计与遇到的问题

- 第一种设计与之前描述的一致：

提供一个图像服务器，接收请求，从用户进程传来需要画的画布和显示位置，并在位置上进行绘画。这种方式遇到的问题是画布过大，一页可能装不下。需要mmap（还没写）

- 第二种设计是一个launcher和application两个单独的单页面切换制度。

这样就是launcher提供应用启动界面，application提供应用界面。

重新回顾了一下内存分配，内核与用户态数据共享的方法后，决定先就第二个思路实现一个简单的用户内核均可见可读写的Framebuffer。

实现RW/RW的Framebuffer

分析如何做才能内核用户均可读写

首先分析一个之前做过的pages，是如何做到用户态可以读，内核态可以写的。

- 在mem_init的时候在在内核空间中分配指定的空间给pages

```
pages = boot_alloc(sizeof(struct PageInfo) * npages);
memset(pages, 0, sizeof(struct PageInfo) * npages);
```

- 利用boot_map_region将其映射到内核页表中的UPAGES的位置。

```
boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U | PTE_P);
```

- 这样内核中依然可以通过pages访问页表，而用户程序在entry的时候通过给pages变量赋予存储位置

```
.globl pages
.set pages, UPAGES
```

也可以通过pages变量进行访问。

预留内存用于framebuffer

再思考如果需要这么一个framebuffer，我们需要放到哪里。仿造上面的UVPD，UPAGES，等，决定就放在接近ULIM的位置。一个PTSIZE也远超我们需要的空间，为以后扩展也留下了余量。

```
/*
 * ULIM, MMIOBASE --> +-----+ 0xef800000
 *                   | Cur. Page Table (User R-) | R-/R- PTSIZE
 *   UVPT          ----> +-----+ 0xef400000
 *                   |           RO  PAGES       | R-/R- PTSIZE
 *   FRAMEBUF      ----> +-----+ 0xef000000
 *                   |           FRAME BUFFER    | RW/RW PTSIZE
 *   UPAGES        ----> +-----+ 0xeec00000
 *                   |           RO  ENVVS       | R-/R- PTSIZE
 *   UTOP, UENVVS  -----> +-----+ 0xee800000
 */

// User read-only virtual page table (see 'uvpt' below)
#define UVPT (ULIM - PTSIZE)
// Read-only copies of the Page structures
#define UPAGES (UVPT - PTSIZE)
// Read-write framebuffer
#define FRAMEBUF (UPAGES - PTSIZE)
// Read-only copies of the global env structures
#define UENVVS (FRAMEBUF - PTSIZE)
// #define UENVVS (UPAGES - PTSIZE)
```

什么时候映射到内核的页表?

由于图像初始化在内存初始化之后，需要留一个接口来进行映射。（boot_map是隐式函数）

```
void map_framebuffer(void *kva)
{
    boot_map_region(kern_pgdir, FRAMEBUF, PTSIZE, PADDR(kva), PTE_W |
PTE_U | PTE_P);
}
```

在分配好内核中的Framebuffer就可以开始映射了

```
void init_framebuffer()
{
    if ((framebuffer = (uint8_t *)kmalloc((size_t)(graph.scrnx *
graph.scrny))) == NULL)
        panic("Not enough memory for framebuffer!");
    map_framebuffer(framebuffer);
}
```

用户程序如何访问?

在libmain的时候初始化即可

```
framebuffer = (uint8_t *)FRAMEBUF;
```

用户态刷新屏幕?

用户程序在写完framebuffer后，如何才能刷新屏幕？这又需要一个新的内核调用

```
static int sys_updatescreen()
{
    update_screen();
    return 0;
}
```

配套的一些代码就不解释了。

PART2

上一个部分已经将一个用户与内核均可读写的缓冲区域，并提供了一个系统调用，用于将显示缓存内容拷贝至MMIO显存。从理论上来说，用户空间的程序现在已经可以直接在这块Framebuffer上绘制任何图形。

但是对于一个友好的用户界面，至少要支持一种格式的图片显示。这里选择一种最简单的，没有压缩过的位图显示实现。推荐各位想自己写图形界面的小伙伴也从这里入手。

关于 BMP 的读取可以参考这篇文章 [256-Color VGA Programming in C Bitmaps & Palette Manipulation](#)。要注意详细读其中的每一个细节，直接扫一眼看代码写的话会遇到很多问题，下面会提到我遇到的问题与解决方案。

Bitmap 图片显示

There are many file formats for storing bitmaps, such as RLE, JPEG, TIFF, TGA, PCX, BMP, PNG, PCD and GIF. The bitmaps studied in this section will be 256-color bitmaps, where eight bits represents one pixel.

One of the easiest 256-color bitmap file format is Windows' BMP. This file format can be stored uncompressed, so reading BMP files is fairly simple.

Windows' BMP 是没有压缩过的，所以读这种 BMP 会非常方便。这里也准备就支持这种格式的图片。

There are a few different sub-types of the BMP file format. The one studied here is Windows' RGB-encoded BMP format. For 256-color bitmaps, it has a 54-byte header (Table III) followed by a 1024-byte palette table. After that is the actual bitmap, which starts at the lower-left hand corner.

BMP 的文件格式如下：

Data	Description
WORD Type;	File type. Set to "BM".
DWORD Size;	Size in BYTES of the file.
DWORD Reserved;	Reserved. Set to zero.
DWORD Offset;	Offset to the data.
DWORD headerSize;	Size of rest of header. Set to 40.
DWORD Width;	Width of bitmap in pixels.
DWORD Height;	Height of bitmap in pixels.
WORD Planes;	Number of Planes. Set to 1.
WORD BitsPerPixel;	Number of bits per pixel.
DWORD Compression;	Compression. Usually set to 0.

DWORD SizeImage;	Size in bytes of the bitmap.
DWORD XPixelsPerMeter;	Horizontal pixels per meter.
DWORD YPixelsPerMeter;	Vertical pixels per meter.
DWORD ColorsUsed;	Number of colors used.
DWORD ColorsImportant;	Number of "important" colors.

下面就我遇到的四个严重的问题，来实现BMP格式的图片读取。

Q1：读头总是出现问题？不合理的数据？

这里要注意GCC默认4字节对齐！！！！！！

但是Bitmap的文件头是14Bytes，如果不加特殊标记，其会变成16Bytes，导致文件偏移错误

```
typedef struct bitmap_fileheader
{
    uint16_t bfType;
    uint32_t bfSize;
    uint16_t bfReserved1;
    uint16_t bfReserved2;
    uint32_t bfOffBits;
}__attribute__((packed)) bitmap_fileheader;

typedef struct bitmap_infoheader
{
    uint32_t biSize;
    uint32_t biWidth;
    uint32_t biHeight;
    uint16_t biPlanes;
    uint16_t biBitCount;
    uint32_t biCompression;
    uint32_t biSizeImage;
    uint32_t biXPelsPerMeter;
    uint32_t biYPelsPerMeter;
    uint32_t biClrUsed;
    uint32_t biClrImportant;
} bitmap_infoheader;
```

这里添加的__attribute__((packed))关键字用于告诉编译器，最小单位进行对齐，而不使用默认的四单位进行对齐。

Q2：为啥读出来的图片颜色怪怪的！！

最开始设置VBE的时候，我以为所谓8位色就是真8位色，之前徒手撸FPGA的显卡的时候也是这么设计的，直接读取后分位后丢给一个D/A输出给VGA变成各自的颜色信号。但是实际系统没有这么简单，其实现了一个8位到32位的对应关系，提供了256位色的调色板。这样能支持更自由的调色方案，显示更加定制化的颜色。所以之前我没有初始化调色板，利用了系统默认的调色板，所以显示才出现问题。

但是理解BMP又出现了偏差，以为大体上是遵循RGB3bit3bit2bit的配色方案，先写了一个初始化调色板的函数：

```
void init_palette()
{
    int i;
    outb(0x03c8, 0);
    for (i = 0; i < 256; i++)
    {
        outb(0x03c9, (i & 0xe0) >> 2); //| 0xA);
        outb(0x03c9, (i & 0x1c) << 1); //| 0xA);
        outb(0x03c9, (i & 0x03) << 3); //| 0xA);
    }
}
```

其选择了最接近想表达的颜色32位颜色并给端口输出。但是颜色还是不大对劲，调色板应该不是这么简单的对应关系。

重新读之前文章的介绍，发现每一个图片文件都有自己的调色板，这种调色板还不太一样，之后使用PS绘制系统图标的时候深有感触，后面再说。

现在面临的主要问题是，我们需要从用户空间读取文件后，才能取出调色板的具体内容，但是通过端口与VGA调色板的通讯在我的设计里面是不能够通过用户空间实现的。那么又要进入内核。那么这个调色板的信息如何传给内核？动态分配的话不能通过栈来传，内核没有用户的页表，也就无法通过地址进行访问。

为了能够从用户空间读取调色板配置文件，并在内核中修改调色板，在原来设计framebuffer的地址上又重新设计了一块专门用于保存调色板的区域，与之前的framebuffer一样，都是RW/RW的。

计算一下占用的空间： $256 * \text{sizeof}(\text{uint8_t}) + \text{sizeof}(\text{uint8_t}) * \text{SCRNSIZE}$ 还是比PTSIZE小，没关系，继续用之前分配的memorylayout，只需要定义一个结构体方便我们来算偏移即可。

所以对于一个BMP图片浏览器，显示图片的整个流程是这样的：

- 用户读BMP文件头
- 用户读BMP调色板
- 放入与内核共享的调色板空间

- 系统调用内核修改调色板
- 用户读文件内容
- 用户写入与内核共享的显示缓存空间
- 系统调用更新屏幕

到这里还有误解，认为BMP的调色板可能大致一致“然后发现几个文件的调色基本一致，于是单独设计了一个用于保存调色板信息的文件，用以下工具导出”。当时的记录是这样，naive！但是这个程序对于其后导出PS调色板有帮助，所以也放在这里。

```
void read_bmp_palette(char *file)
{
    FILE *fp;
    long index;
    int x;
    /* open the file */
    if ((fp = fopen(file, "rb")) == NULL)
    {
        printf("Error opening file %s.\n", file);
        exit(1);
    }
    uint8_t buf[1000];
    bitmap_fileheader head;
    bitmap_infoheader info;
    uint16_t width, height;
    bitmap_image image;
    bitmap_infoheader *infoheader = &info;
    fread(&head, sizeof(bitmap_fileheader), 1, fp);
    fread(&info, sizeof(bitmap_infoheader), 1, fp);
    struct palette palette[256];
    FILE *fout = fopen("palette.plt", "wb");

    for (int i = 0; i < 256; i++)
    {
        fread(&palette[i], sizeof(struct palette), 1, fp);
        palette[i].rgb_red >>= 2;
        palette[i].rgb_green >>= 2;
        palette[i].rgb_blue >>= 2;
        fwrite(&palette[i], sizeof(struct palette), 1, fout);
    }
    fclose(fout);
    fclose(fp);
}
```

好了，到这里运气好的话，应该可以正常颜色绘制出来一个位图了。（那啥读取位图内容显示在屏幕上的代码实在太简单了，就不单独说了）

Q3：为啥是倒的，还有的运气不好出错了

之前之所以说运气好，是因为刚好这个图片信息中的高为正的，那么按照基本逻辑，可以画出来一个倒的图片。还是太naive，不好好看文档中的头文件具体参数描述，想当然的给了图片高为一个无符号数。

在BMP的文件头中，高为一个有符号数。正表示下面的位图像素信息是倒着来的，负表示下面的位图像素信息是正着的.....这个设计，好吧...

Q4：如果是想绘制多个图片在一页上，调色板问题？

在Q2中提到，想用一个调色板文件预配置后就不管其他图片的调色板的思路太单纯了...当使用一些比较fancy的素材进来的时候，发现其颜色根本完全不一样，失真的可怕。

为了更加理解调色板这个设定，我们需要一个photoshop。设置图片为图像->模式->索引模式后，就可以生成我们需要的位图了。注意这里的设置页面：



可以发现系统有自己的调色板，可能用于绘制所有的图标使用的。（当然可能也已经是历史的产物了）后面我将用相同的思路实现图标的绘制。还有一些局部的选项，这样就会利用一个颜色更加单一，但是转化出来的图片更接近32位色的图片的调色板来生产了。

打开图像->模式->颜色表可以看到当前图片使用的调色板：



可以看到它完全不按照套路出牌，并没有之前说的R3G3B2的影子。

所以对于一个页面，如何选择调色板？我的方案是把这个页面所有的素材丢到一个ps文件中，并生成针对这个页面还原度最高的调色板方案。在绘制这个页面的时候先载入这个页面的调色板，再进行绘制。

PS可以导出调色板，按照官方的文档，也是一个简单的二进制的堆叠，与上面的思路类似写一个调色板转系统plt文件的导出即可。

Q5：如果读到一个32位色的图片咋办？

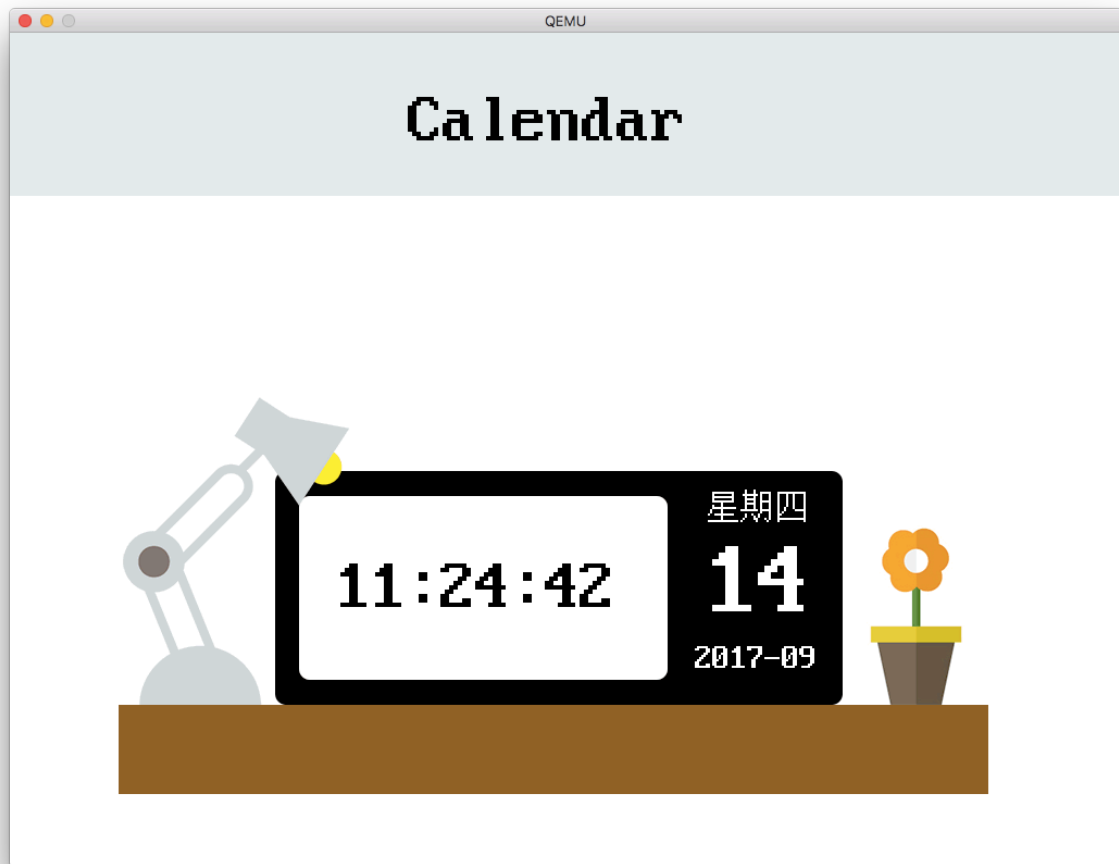
好吧，我的选择不读，可以在网上找找32位色妥协到8位色的算法，然而实在效果非常糟糕，单独生成调色板算法就复杂了，不如交给PS。毕竟这不是操作系统的重点。

Part3

本部分将解释我设计的图形化界面数据结构，框架以及接口。

其实这部分设计的比较乱，也就只能支持单页面切换的需求了。作为一个技术试探是足够了，但是扩展性很差，想继续在这上面做文章可能需要推倒重来。

先看效果图：



界面由标题和内容组成，界面是应用程序请求屏幕资源的基本单位。界面的数据结构如下：

```
struct interface
{
    uint8_t titletype;
    char title[MAX_TITLE];
    uint8_t title_textcolor;
    uint8_t title_color;
    uint8_t content_type;
    uint8_t content_color;

    // about the size and buff of interface
    uint16_t scrnx;
    uint16_t scrny;
    uint8_t *framebuffer;
};
```

其包含了这个界面的基本信息，以及当前屏幕的各项参数，各种函数将直接向framebuffer上操作。

```

void draw_interface(struct interface *interface);
void draw_title(struct interface *interface);
// if color == back means transparent
int draw_cn(uint16_t x, uint16_t y, char *str, uint8_t color, uint8_t
back, uint8_t fontmag, struct interface *interface);
int draw_ascii(uint16_t x, uint16_t y, char *str, uint8_t color,
uint8_t back, uint8_t fontmag, struct interface *interface);
void draw_fontpixel(uint16_t x, uint16_t y, uint8_t color, uint8_t
fontmag, struct interface *interface);
void interface_init(uint16_t scrnx, uint16_t scrny, uint8_t
*framebuffer, struct interface *interface);
void add_title(char *title, uint8_t title_textcolor, uint8_t
title_color, struct interface *interface);
int init_palette(char *plt_filename, struct frame_info *frame);
void draw_content(struct interface *interface);
int draw_screen(uint16_t x, uint16_t y, struct screen *screen,
uint8_t color, uint8_t back, uint8_t fontmag);

```

提供了以上基本操作，实现都很简单，没有做错误处理。

值得一提的是字体的设置。由于用的点阵字库，放大后会马赛克。这里使用的方法为打包具体绘制像素方法至draw_fontpixel，其提供了多个像素抽象为一个字体像素进行统一绘制的方法。

Part 4

本部分终于到了图形界面的程序应用了。具体应用如何使用上面设计的接口呢？

首先看一个最简单的例子：

```

#include <inc/lib.h>
#define BACKGROUND 0x00
struct interface interface;

void input_handler();
void display_info();

void umain(int argc, char **argv)
{
    int r;
    // 初始化本界面使用的调色板
    if ((r = init_palette("/bin/sysinfo.plt", frame)) < 0)
        printf("Open palette fail %e\n", r);
    // 初始化界面信息
    interface_init(graph.scrnx, graph.scrny, graph.framebuffer,
&interface);

```

```

interface.titletype = TITLE_TYPE_TXT;
strcpy(interface.title, "System information");
interface.title_color = 0x5a;
interface.title_textcolor = 0xff;
interface.content_type = APP_NEEDBG;
interface.content_color = BACKGROUND;
// 绘制界面
draw_interface(&interface);
// 绘制Bitmap
if ((r = draw_bitmap("/bin/sysinfo.bmp", 100, 160, &interface)) <
0)
    printf("Open clock back fail %e\n", r);
// 显示信息
display_info();
// 绘制结束, 刷新屏幕
sys_updatescreen();
// 处理按键中断
input_handler();
}

void input_handler()
{
    unsigned char ch;
    ch = getchar();
    while (1)
    {
        switch (ch)
        {
            case KEY_ESC:
                exit();
            }
        ch = getchar();
    }
}

void display_info()
{
    ...
    struct sysinfo info;
    // 通过系统调用获取一些系统信息
    sys_getinfo(&info);
    draw_ascii(display_x, display_y, "Sys      : Hello_Liu's MOS
version 0.1", 0xff, 0x00, fontmeg, &interface);

    display_y += font_height;
    draw_ascii(display_x, display_y, "Github :
https://github.com/HelloLiu/MOS", 0xff, 0x00, fontmeg, &interface);
    display_y += font_height;
    draw_ascii(display_x, display_y, "Blog    :

```

```
http://blog.csdn.net/hello_liu", 0xff, 0x00, fontmeg, &interface);  
    ...  
}
```

一个简单的具有图像界面的程序由以下步骤：

- 初始化本界面使用的调色板
- 初始化界面信息
- 绘制界面，绘制内容
- 更新屏幕
- 开启按键处理，具体事件具体处理

关于具体应用程序的一些要点

应用启动器



启动器算比较复杂的一个部分，专门设计了一个单独的数据结构和绘制方法：


```

struct launcher_content
{
    int app_num;
    int app_sel;
    uint8_t background;
    char icon[MAX_APP][MAX_PATH];
    char app_bin[MAX_APP][MAX_PATH];
};

void draw_launcher(struct interface *interface, struct
launcher_content *launcher);

```

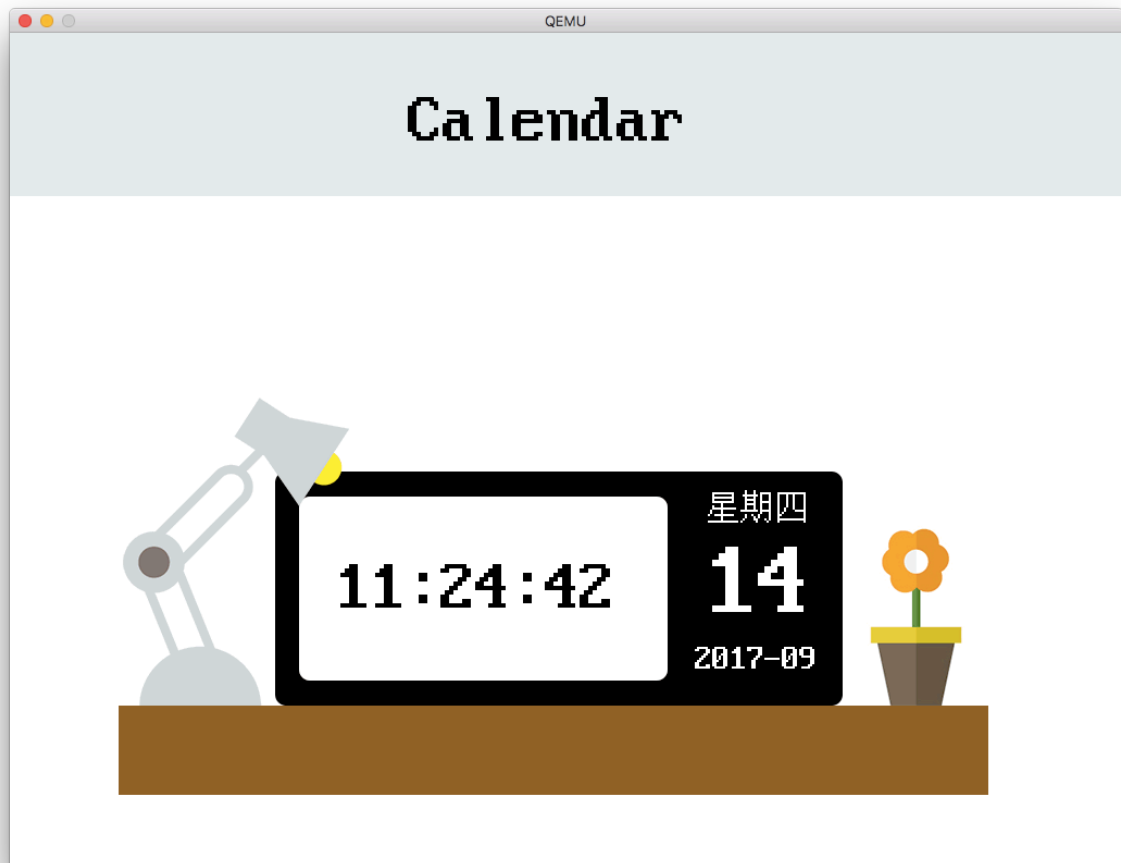
用icon来保存对应的app的图标文件路径，用app_bin来保存对应的程序的路径。当选择了对应的程序的时候spawn这个程序，并等待其运行结束后回收进程并重绘启动器：

```

void launch_app()
{
    char *app_bin = launcher.app_bin[launcher.app_sel];
    int r;
    char *argv[2];
    argv[0] = app_bin;
    argv[1] = 0;
    printf("[launcher] Launching %s\n", app_bin);
    if ((r = spawn(app_bin, (const char **)argv)) < 0)
    {
        printf("App %s not found!\n", app_bin);
        return;
    }
    wait(r);
    printf("[launcher] %s normally exit\n", app_bin);
    init_palette("/bin/palette.plt", frame);
    refresh_interface();
}

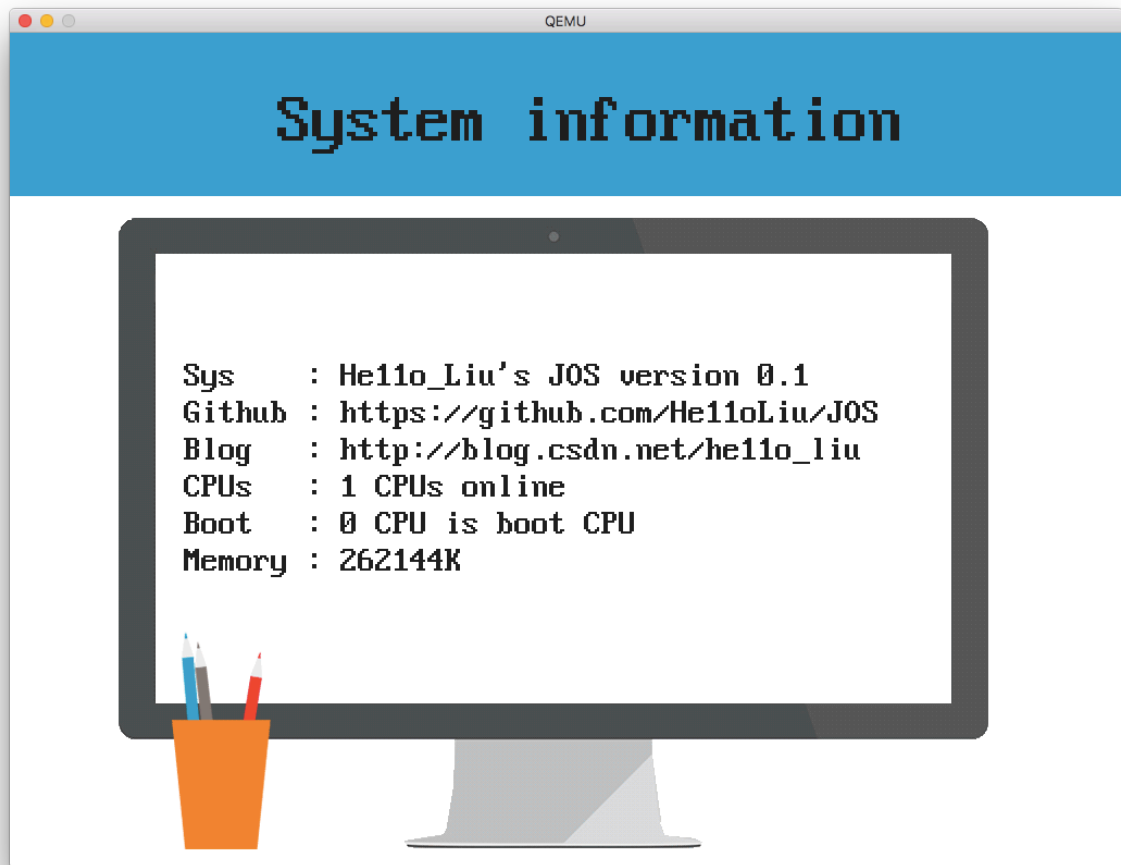
```

日历



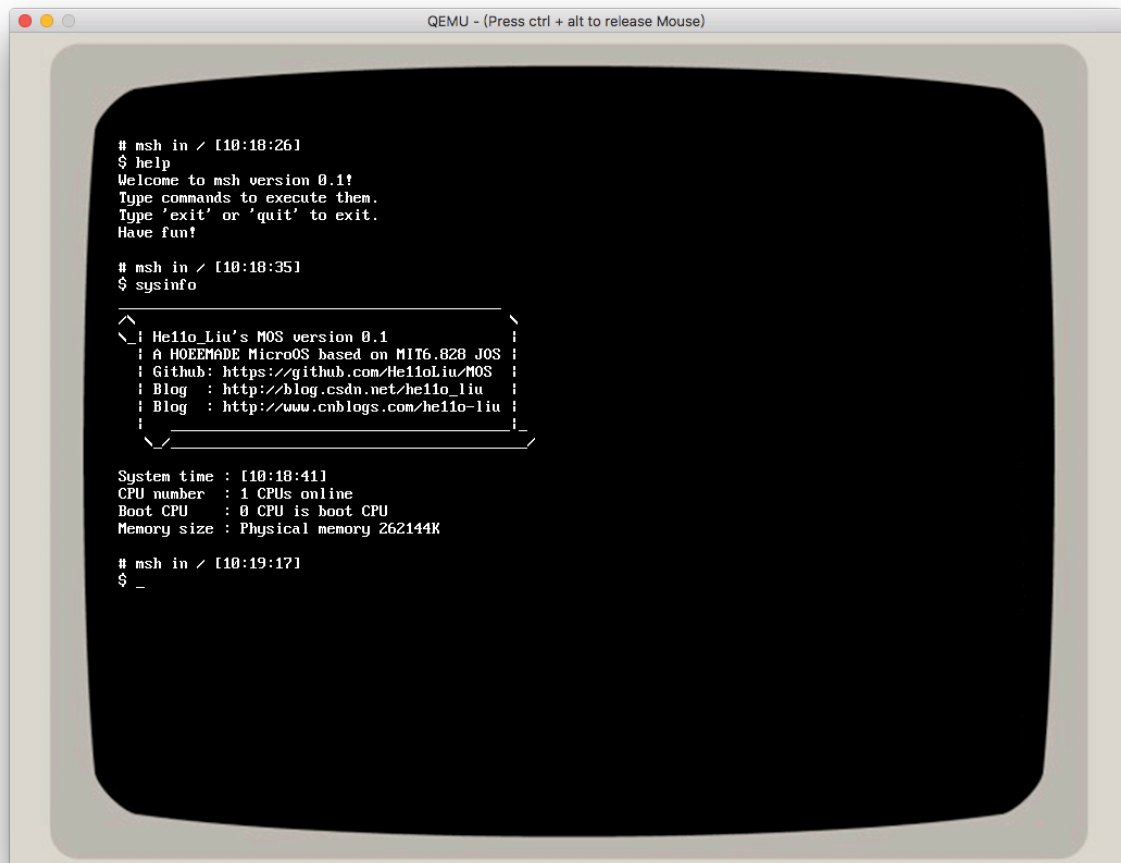
由于没有写系统时钟，只提供了对于RTC的系统调用。这里实现Fork了一个进程用于监控RTC的更新，并在适当时候更新屏幕，主进程用于监听键盘，并在退出的时候摧毁子进程。

系统信息



这个程序的代码已经放在上面了，主要是设计了一个新的syscall，用于从内核中返回一些基本系统信息。

Part4 模拟CGA显示模式终端应用



```
QEMU - (Press ctrl + alt to release Mouse)

# msh in / [10:18:26]
$ help
Welcome to msh version 0.1!
Type commands to execute them.
Type 'exit' or 'quit' to exit.
Have fun!

# msh in / [10:18:35]
$ sysinfo

  ^
  | Hello_Liu's MOS version 0.1 |
  | A HOEEMADE MicroOS based on MIT6.828 JOS |
  | Github: https://github.com/HelloLiu/MOS |
  | Blog : http://blog.csdn.net/hello_liu |
  | Blog : http://www.cnblogs.com/hello-liu |
  |
  v

System time : [10:18:41]
CPU number : 1 CPUs online
Boot CPU : 0 CPU is boot CPU
Memory size : Physical memory 262144K

# msh in / [10:19:17]
$ _
```

终端程序与普通程序的设计思路完全不同，本部分将根据我的思考来一步步阐述如何实现终端APP。

思路

作为一个终端程序，

- 终端模拟器应该支持一种printf能显示到屏幕的功能。
 - printf是向文件描述符1进行输出。
 - 查看之前写的console代码，openconsole的操作就是分配一个文件描述符，设置文件操作为(键盘输入，串口输出)的策略。
 - 所以我们这个终端模拟器应该提供一种新的device，这种device提供了(键盘输入，屏幕输出)的功能。

```

struct Dev devscreen =
{
    .dev_id = 's',
    .dev_name = "screen",
    .dev_read = devscreen_read,
    .dev_write = devscreen_write,
    .dev_close = devscreen_close,
    .dev_stat = devscreen_stat};

```

- 直接在屏幕上显示出来并不是一个很好的选择，参考CGA的显示，设计了一个屏幕字符缓冲区。

```

struct screen
{
    uint8_t screen_col;
    uint8_t screen_row;
    uint16_t screen_pos;
    char screen_buf[SCREEN_SIZE];
};

```

- 提供新的bprintf方法，方便screen device调用。
- 作为终端模拟器，其需要集成fork出来的各种进程的输出。
 - 首先对于其他的程序，其输出也是printf
 - 其他的程序会继承其父进程的文件描述符表。
 - 父进程中的文件描述符1号，则应该指向上面定义的screen（这条思路最后没走通）

思路中的part1

这个部分的实现还是比较顺利的。上面已经定义了新的device。

- 这个新的device的read策略，还是从键盘读，无须进行修改。
- 而这个device的写策略，则需要写入到屏幕了。这里新写了一个bprintf的函数与其配套方法。(bprintf a.k.a printf to buf)

bprintf的基本实现与之前在CGA模式的输出类似，所以才叫仿CGA模式。主要是bputchar的实现：

```

void bputchar(char c)
{
    switch (c)
    {
        case '\b': /* backspace */
            if (screen.screen_pos > 0)
            {
                screen.screen_pos--;
                // delete the character
                screen.screen_buf[screen.screen_pos] = ' ';
            }
            break;
        case '\n': /* new line */
            screen.screen_pos += SCREEN_COL;
            /* fallthru */
        case '\r': /* return to the first character of cur line */
            screen.screen_pos -= (screen.screen_pos % SCREEN_COL);
            break;
        case '\t':
            bputchar(' ');
            bputchar(' ');
            bputchar(' ');
            bputchar(' ');
            break;
        default:
            screen.screen_buf[screen.screen_pos++] = c; /* write the
character */
            break;
    }

    // When current pos reach the bottom of the creen
    // case '\n' : screen.screen_pos -= SCREEN_COL will work
    // case other: screen.screen_pos must equal to SCREEN_SIZE
    if (screen.screen_pos >= SCREEN_SIZE)
    {
        int i;
        // Move all the screen upward (a line)
        memmove(screen.screen_buf, screen.screen_buf + SCREEN_COL,
(SCREEN_SIZE - SCREEN_COL) * sizeof(uint8_t));
        // Clear the bottom line
        for (i = SCREEN_SIZE - SCREEN_COL; i < SCREEN_SIZE; i++)
            screen.screen_buf[i] = ' ';
        screen.screen_pos -= SCREEN_COL;
    }
    screen.screen_col = SCREEN_COL;
    screen.screen_row = SCREEN_ROW;
    draw_screen(100, 80, &screen, 0x00, 0xff, 1);
}

```

bputchar实现了对特殊描述符，换行，翻页的情况的处理，并将打印的内容放入屏幕字符缓冲区。

最后要实现的就是把屏幕缓冲区的内容放倒屏幕上。这个实现起来就比较简单了，遍历字符串，然后一个个字从字库中获取显示信息显示出来即可。

思路中的part2

part2才是设计终端中需要动脑子的地方。正如思路中所说，我一开始的想法是：

老思路

父进程中的文件描述符1号，则应该指向上面定义的screen

然而没有考虑这个问题：

interface与**screen**参数均属于与之平等的另一个用户程序！在调用bprintf的时候，没有初始化screen，也不知道interface在哪里。

之所以**CGA**模式可以使用这个思路是因为**CGA**的文字缓冲区是在内核中，可以看为这项服务是内核提供的，是一个上下级的关系，而不是平行的

如果必须要走这条路，有以下解决方法：

- 在库中判断没有参数时，直接初始化screen与interface，可以做到直接新建一个输出页的效果。
- 提升终端到内核中运行（无法忍受）
- 使终端成为如同文件系统的非普通程序，接收输出请求。这个也很复杂，不够优雅！！！！

新思路：利用Pipe!!!!!!!

老思路中的第一条解决方法走通后又思考了一会儿，实在不想走第二第三条路。

换个思路一想，原来这个事可以这么简单。申请一个pipe，读取端给（输出到屏幕的）服务进程作为输入来源，输出端给用户程序作为输出。程序输出的内容会通过pipe发送给服务进程，最终服务进程显示到屏幕上即可。

整个程序的流程如下：

- 终端程序打开屏幕输出设备，给文件描述符1（默认输出）
- 终端程序申请pipe
- 终端程序fork子进程
 - 子进程关闭读的pipe，保留写的pipe，并将写的pipe给默认输出1，后面的程序输出都会写进pipe中。子进程开始运行shell。
 - 父进程关闭写的pipe，保留读的pipe，并将读的pipe给默认输入0，后面程序的输入都会读pipe中的内容。父进程进入循环，服务所有的输入输出到屏幕的功能。

来看核心代码：

```
void umain(int argc, char **argv)
{
    ...
    close(1);
    // 打开屏幕CGA输出到文件描述符1
    if ((r = openscreen(&interface)) < 0)
        panic("fd = %e\n", r);
    cprintf("fd = %d\n", r);
    // 申请一个pipe
    if ((r = pipe(p)) < 0)
    {
        cprintf("pipe: %e", r);
        exit();
    }
    readfd = p[0];
    writefd = p[1];
    r = fork();
    if (r == 0)
    {
        close(readfd);
        close(1);
        // 写入端给子进程作为其输出默认文件描述符1
        dup(writefd, 1);
        // 运行shell（修改过，没有文件描述符操作版本）
        msh(argc, argv);
        printf("msh exit\n");
        exit();
    }
    else
    {
        close(writefd);
        close(0);
        // 读入端作为其默认读取文件描述符0
        if (readfd != 0)
            dup(readfd, 0);
        close(readfd);
    }
}
```



```

e = &envs[ENVX(r)];
while(1)
{
    // 获取所有的pipe中的数据并显示在模拟CGA屏幕上
    r = getchar();
    printf("%c", r);
    // 当shell退出的时候退出
    if(e->env_status == ENV_FREE)
        break;
}
}

```

Part 收获

这个部分将记录这几天写图形界面的收获。

之前看知乎上的大佬们的论调：图形界面和操作系统有啥关系？没啥好写的，简单！其实写写简单的图形界面一个是转换一下思路，有简单可见的产出激励，另一个是进一步理解体会操作系统的设计，并实际修改一些MOS中的设计，并实现一些类似之前照着任务指南写出来的功能。In another words, get your hands dirty.

用户与内核的关系

按照MOS的思路，还是希望保持一个相对小的内核，提供最基本的服务，剩下的交给用户空间玩耍。但是到实际的问题上，包括了

- 功能的划分，用户需要的服务能否在用户态实现大部分，内核实现小部分（如上次的用户空间的页错误处理与这次的Framebuffer与屏幕更新）。这样的设计更加flexable，并且保证了呆在内核中的时间非常短暂（毕竟还用着big kernel lock...）
- 内核与用户空间的信息交换。栈或者固定地址的交互。

栈比较灵活，可以在每次的系统调用的时候直接压进去，交换完了后再取回来。但是只能传值，传的内容比较少。

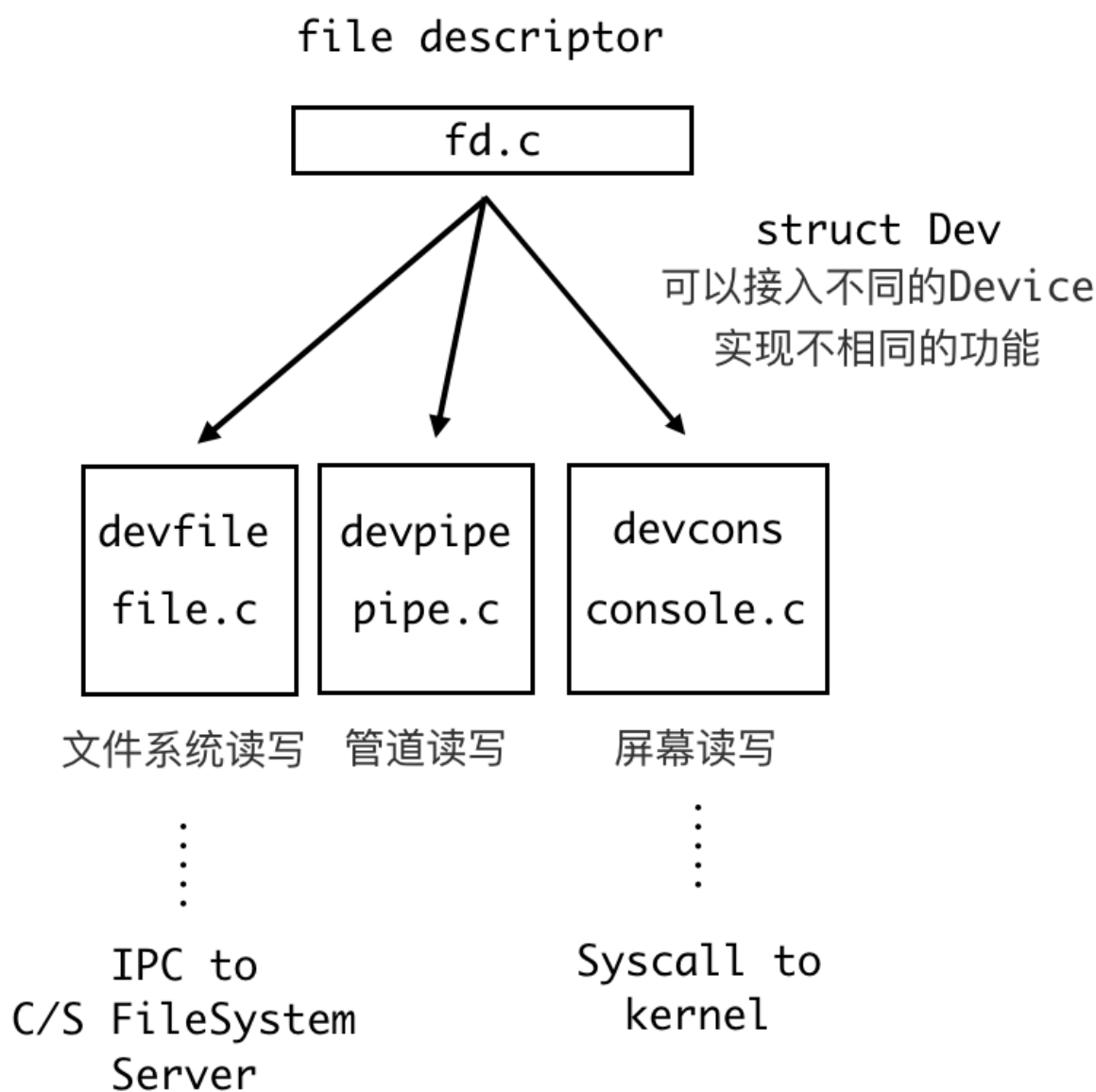
固定地址则使得系统变得不那么灵活，不利于扩展与移植。但是可以高效的大量数据交换。在写这块的时候实现了kmalloc，并理解了之前mem_init时做的各种映射的意义。

用户与用户的关系（用户空间程序）

理想的用户与用户的关系是平齐的（不提供服务的用户），在写用户程序的时候不知道其父进程是谁，也不会要求子进程知道自己的存在。但是跨进程之间的服务需求仍然存在，如对于一个进程输出的获取，或图形界面中的界面重叠。这就需要一个服务提供者的存在，来抽象用户之间的需求。比如之前设计的文件系统服务器，比如这回本来准备实现的图形界面服务器。

文件描述符 | 句柄 | FD

这部分感触最深的是unix中的文件描述符的设计，简直太妙。将所有的内容全部抽象成文件，就可以灵活的在不同的需求之前切换。最简单的读写文件，读写串口，读写屏幕，pipe均使用的这种抽象。



这种抽象将抽象层和具体实现层分开，降低耦合度的同时提供了非常高的灵活性。

写在最后

至此，所有目前已经实现特性的记录就结束了。MOS在最初规划的时候还有更多想要实现的内容，可惜时间短暂，马上又有投入大四的学习之中了。

然而MOS的生命并没有结束，作为一个完全知晓其内部设计，拥有了添加各种特性的经验，MOS对于我来说，更像一个工作台。如已经实现的PRWLock，MOS完全有能力胜任新想法基本设计与功能验证，作为早期的一个实验环境，使用起来相较于linux更加方便。

开发MOS中，经历了绝望，死磕，最后又柳暗花明的阶段，最终也解决了其中大部分的问题。最后记录一点与具体技术无关的内容，给同样在自己写OS的同学的一些建议吧。

■ 首先自制操作系统资料

对于自制OS的资料中文内容非常少，大部分是英文的。除了MIT给的xv6以及各种ref之外，还有：

- [OSDev](#)：OSDev上主要包含了操作系统中各个功能实现的基本思路，一般会提供简要的伪代码。OSDev下还有一个自制操作系统的索引，找一个类似的项目进行参考也是不错的选择。（不过一般没文档，看悟性了）
- [256-Color VGA Programming in C](#)：这个系列完整介绍了C语言的VGA图像编程，包括了BMP的读取，双缓冲技术，动画等。非常有参考价值，附带了源码资料，不过是DOS环境的。

除此，对于网络上的资料，不能全信。特别是中文博客中各种转来转去的内容，很有可能就是错的！最靠谱的解决方案就是找到各种协议的源头（而非写好的代码），要么查指令手册，要么直接看协议具体内容。

■ 系统查错 Debug

这个部分应该是自己写OS中最痛苦的地方了。可能很多底层中隐藏的错误，在很久之后写完全无关的内容的时候才会暴露出来。但是毕竟还是不是高并发情况下不容易重现的错误。一般固定好情况，算好正确的流程，再用GDB一个个走，还是能够找到的。还可以利用核间中断，系统调用等技术，适时dump出系统状态。

写OS的体验肯定和写安卓写前端画画的体验不一样，注定要死磕一些细节，在各种内部错误中完善，耐心和信心都是必不可少的。

如对MOS中的实现有异议或者建议，欢迎邮箱(he11oliu830@gmail.com)与我讨论！