

操作系统课程设计实验报告

姓名：余建桅

学号：2052345

同济大学

Tongji University

目 录

[1 Lab1: Lab Utilities 1](#_Toc495668153)

[1.1 实验目的 1](#_Toc495668154)

[1.2 实验步骤 1](#_Toc495668155)

[1.2.1 boot xv6 1](#_Toc495668163)

[1.2.2 sleep 1](#_Toc495668163)

[1.2.3 pingpong 2](#_Toc495668163)

[1.2.4 primes 2](#_Toc495668163)

[1.2.5 find 3](#_Toc495668163)

[1.2.6 xargs 4](#_Toc495668163)

[1.2.7 Lab1所有实验测试 5](#_Toc495668163)

[1.3 实验中遇到的问题和解决方法 5](#_Toc495668155)

[1.4 实验心得 5](#_Toc495668155)

[2 Lab2: Lab System calls 6](#_Toc495668153)

[2.1 实验目的 6](#_Toc495668154)

[2.2 实验步骤 6](#_Toc495668155)

[2.2.1 System call tracing 6](#_Toc495668163)

[2.2.2 Sysinfo 8](#_Toc495668163)

[2.3 实验中遇到的问题和解决方法 9](#_Toc495668155)

[2.4 实验心得 1](#_Toc495668155)0

[3 Lab3: Lab Page tables 1](#_Toc495668153)0

[3.1 实验目的 1](#_Toc495668154)0

[3.2 实验步骤 1](#_Toc495668155)0

[3.2.1 打印一个页表 1](#_Toc495668163)0

[3.2.2 给每一个进程制作一个kernel\_pagetable 1](#_Toc495668163)1

[3.2.3 简化copyin/copyinstr 1](#_Toc495668163)4

[3.3 实验中遇到的问题和解决方法 1](#_Toc495668155)4

[3.4 实验心得 1](#_Toc495668155)4

[4 Lab4: Lab Traps 1](#_Toc495668153)5

[4.1 实验目的 1](#_Toc495668154)5

[4.2 实验步骤 1](#_Toc495668155)5

[4.2.1 RISC-V assembly 1](#_Toc495668163)5

[4.2.2 Backtrace 1](#_Toc495668163)5

[4.2.3 Alarm 1](#_Toc495668163)6

[4.3 实验中遇到的问题和解决方法 2](#_Toc495668155)0

[4.4 实验心得 2](#_Toc495668155)0

[5 Lab5: Lab Lazy allocation 2](#_Toc495668153)0

[5.1 实验目的 2](#_Toc495668154)0

[5.2 实验步骤 2](#_Toc495668155)0

[5.2.1 Eliminate allocation from sbrk() 2](#_Toc495668163)0

[5.2.2 Lazy allocation 2](#_Toc495668163)1

[5.2.3 Lazytests and Usertests 2](#_Toc495668163)1

[5.3 实验中遇到的问题和解决方法 2](#_Toc495668155)2

[5.4 实验心得 2](#_Toc495668155)2

[6 Lab6: Lab Copy on-write 2](#_Toc495668153)2

[6.1 实验目的 2](#_Toc495668154)2

[6.2 实验步骤 2](#_Toc495668155)3

[6.3 实验中遇到的问题和解决方法 2](#_Toc495668155)4

[6.4 实验心得 2](#_Toc495668155)4

[7 Lab7: Lab Multithreading 2](#_Toc495668153)4

[7.1 实验目的 2](#_Toc495668154)4

[7.2 实验步骤 2](#_Toc495668155)4

[7.2.1 Uthreads:switching between threads 2](#_Toc495668163)4

[7.2.2 Using threads 2](#_Toc495668163)5

[7.2.3 Barrier 2](#_Toc495668163)6

[7.3 实验中遇到的问题和解决方法 2](#_Toc495668155)6

[7.4 实验心得 2](#_Toc495668155)6

[8 Lab8: Lab Locks 2](#_Toc495668153)7

[8.1 实验目的 2](#_Toc495668154)7

[8.2 实验步骤 2](#_Toc495668155)7

[8.2.1 Memory allocator 2](#_Toc495668163)7

[8.2.2 Buffer cache 2](#_Toc495668163)8

[8.3 实验中遇到的问题和解决方法 3](#_Toc495668155)0

[8.4 实验心得 3](#_Toc495668155)0

[9 Lab9: Lab File system 3](#_Toc495668153)0

[9.1 实验目的 3](#_Toc495668154)0

[9.2 实验步骤 3](#_Toc495668155)0

[9.2.1 Large files 3](#_Toc495668163)0

[9.2.2 Symbolic links 3](#_Toc495668163)2

[9.3 实验中遇到的问题和解决方法 3](#_Toc495668155)3

[9.4 实验心得 3](#_Toc495668155)3

[10 Lab10: Lab Mmap 3](#_Toc495668153)3

[10.1 实验目的 3](#_Toc495668154)3

[10.2 实验步骤 3](#_Toc495668155)4

[10.3 实验中遇到的问题和解决方法 3](#_Toc495668155)6

[10.4 实验心得 3](#_Toc495668155)6

[11 Lab11: Lab Network driver 3](#_Toc495668153)6

[11.1 实验目的 3](#_Toc495668154)6

[11.2 实验步骤 3](#_Toc495668155)6

[11.3 实验中遇到的问题和解决方法 3](#_Toc495668155)8

[11.4 实验心得 3](#_Toc495668155)8

# Lab1:Lab Utilities

## **1.1 实验目的**

1.配置xv6系统运行环境，了解并熟悉xv6环境和系统调用命令

2.切换到xv6-labs-2020代码的util分支，并利用QEMU模拟器启动xv6系统

3.通过阅读代码及进行实验加深对于操作系统中基本概念的理解

## **1.2 实验步骤**

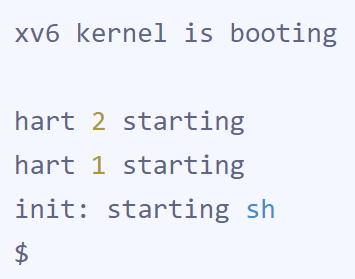
### **1.2.1 boot xv6**

1.使用命令git clone git://g.csail.mit.edu/xv6-labs-2020克隆 xv6-labs-2020 代码到本地

2.使用命令cd xv6-labs-2020进入 xv6-labs-2020 代码目录

3.使用命令git checkout util切换到 util 分支

4.使用命令make qemu命令编译并运行 xv6 系统

5.xv6 通过 QEMU 模拟器启动后，启动 shell 进程，最后可以看到如下的输出

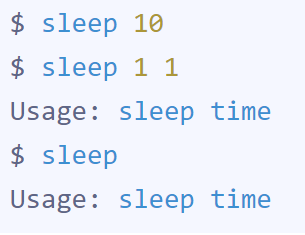
6.在 xv6 中按 Ctrl + a ，然后按 x 即可退出 xv6 系统

### **1.2.2 sleep**

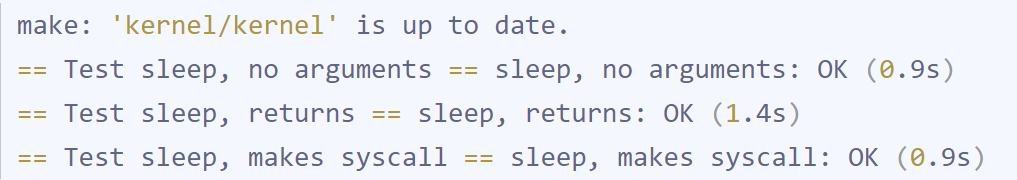
1.在user目录下创建并编写好sleep.c程序

2.在 Makefile 文件中添加配置，在 UPROGS 项中最后一行添加 $U/\_sleep\

3.在xv6中输入如下命令



1. 退出xv6运行单元测试，输入命令./grade-lab-util sleep
2. 通过测试样例

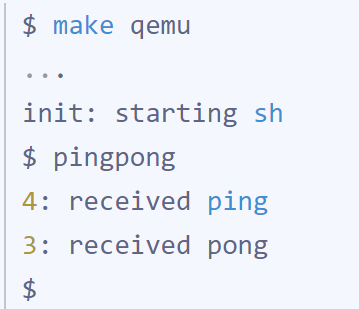


### **1.2.3 pingpong**

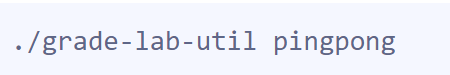
1.在user目录下创建并编写好pingpong.c程序

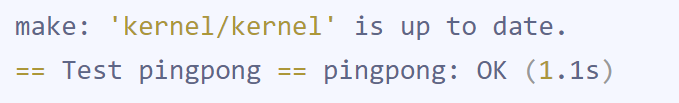
2.在 Makefile 文件中添加配置，在 UPROGS 项中最后一行添加 $U/\_pingpong\

3.在xv6中输入如下命令



1. 退出 xv6 ，运行单元测试检查结果是否正确，通过测试案例



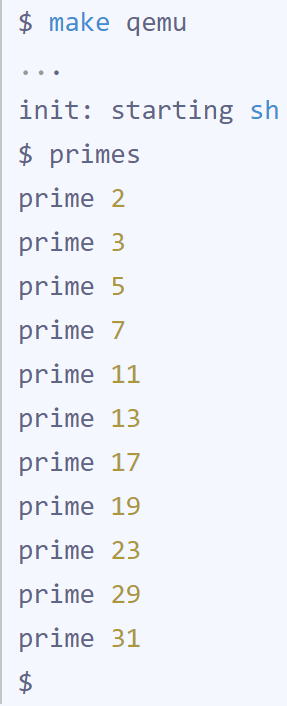


### **1.2.4 primes**

1.在user目录下创建并编写好primes.c程序

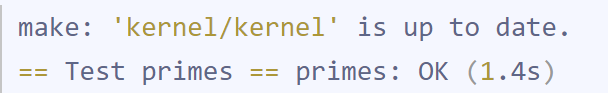
2.在 Makefile 文件中添加配置，在 UPROGS 项中最后一行添加 $U/\_primes\

3.在xv6中输入如下命令进行测试



4.退出 xv6 ，运行单元测试检查结果是否正确，通过测试案例



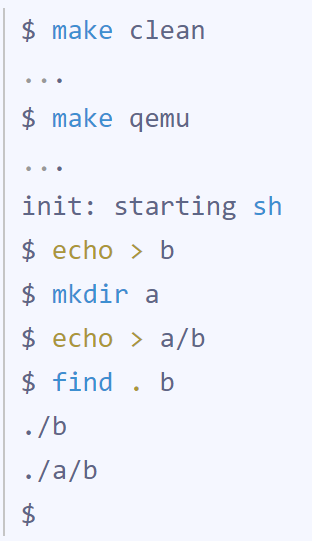


### **1.2.5 find**

1.在user目录下创建并编写好find.c程序

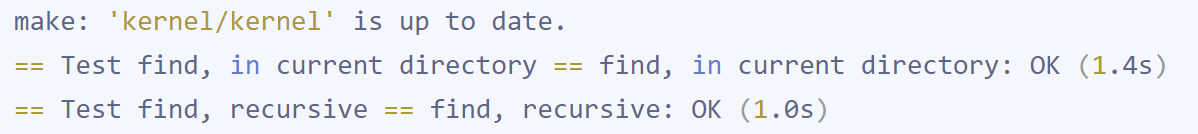
2.在 Makefile 文件中添加配置，在 UPROGS 项中最后一行添加 $U/\_find\

3.在xv6中输入如下命令进行测试



4.退出 xv6 ，运行单元测试检查结果是否正确，通过测试案例



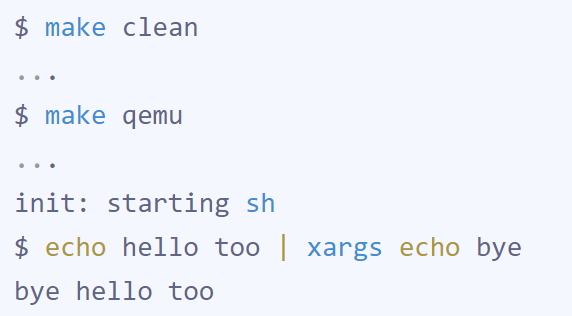


### **1.2.6 xargs**

1.在user目录下创建并编写好xargs.c程序

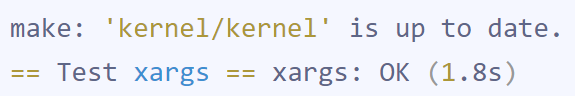
2.在 Makefile 文件中添加配置，在 UPROGS 项中最后一行添加 $U/\_xargs\

3.在xv6中输入如下命令进行测试



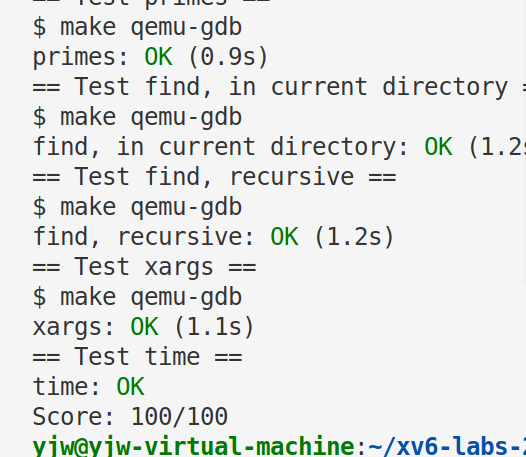
4.退出 xv6 ，运行单元测试检查结果是否正确，通过测试案例





### **1.2.7 Lab1所有实验测试**

退出xv6，运行整个 Lab 1 测试，检查结果是否正确，结果如下



## **1.3 实验中遇到的问题和解决方法**

本次实验遇到的最大问题还是实验环境的搭建。由于这是首次在虚拟机上使用linux系统，在安装虚拟机及软件之初由于不够熟练而遇到了些许问题，如无法启动xv6，无法下载编译器等，但在网上查询了相应的资料过后也都能得到解决，也通过这一系列操作熟练掌握了构建虚拟机以及使用linux操作系统的方法。

## **1.4 实验心得**

Lab1的实验总体而言难度并不大，主要还是帮助我们熟悉和掌握xv6环境与操作，通过实验与阅读源码我也对于系统调用等知识有了更进一步的理解。

# Lab2:Lab System calls

## **2.1 实验目的**

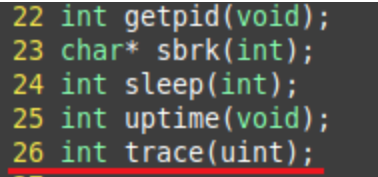
在本实验中，通过将向xv6添加一些新的系统调用，来了解它们是如何工作的，同时进而了解xv6内核的一些内部结构，在后续的实验中我们可能会更频繁地用到更多的系统调用。

## **2.2 实验步骤**

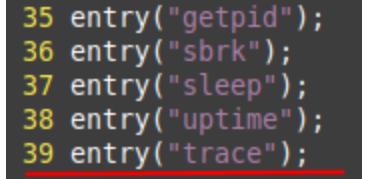
### **2.2.1 System call tracing**

一、用户接口代码的修改

1.在user/user.h中添加系统调用函数的定义

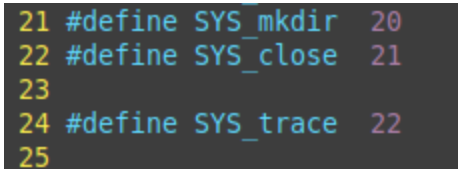


2.在user/usys.pl中添加入口entry("trace")

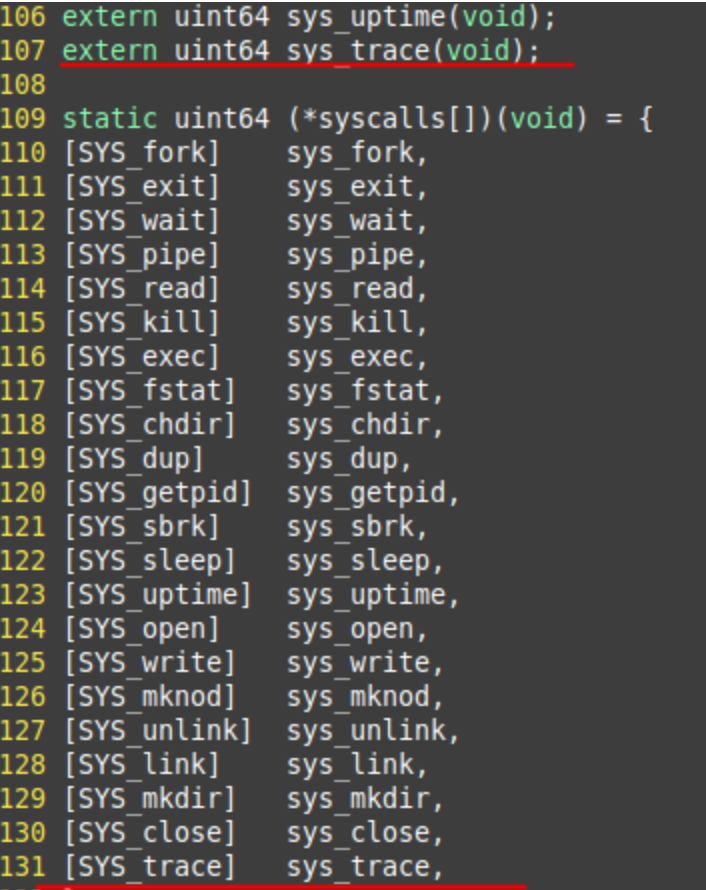


二、内核代码的修改

1. 在kernel/syscall.h中定义系统调用号

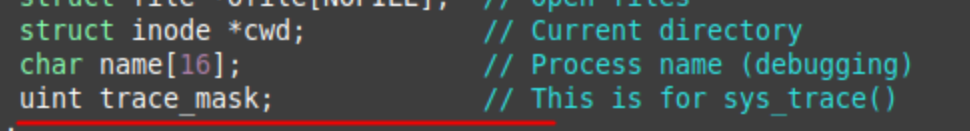


1. 在kernel/syscall.c的syscalls函数指针数组中添加对应的函数

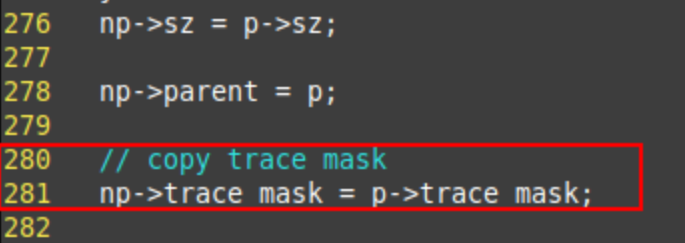


1. 在proc结构体中添加一个trace\_mask字段，之后在创建子进程的fork函数中复制该字段到新进程

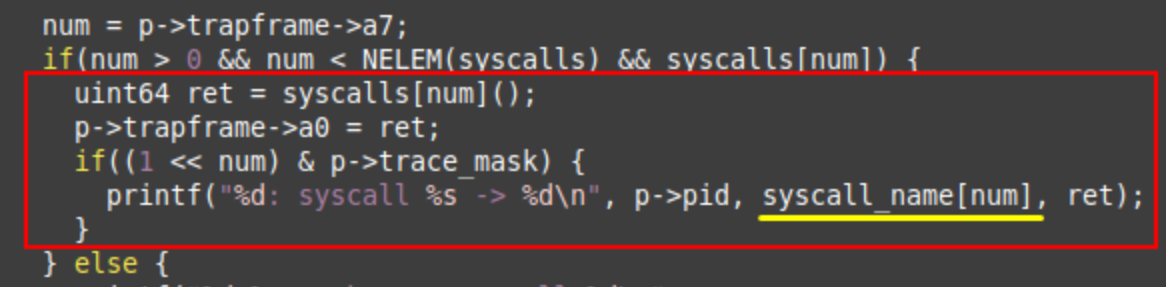
kernel/proc.h



Kernel/proc.c



1. 系统调用sys\_trace的实现。在sysproc.c中添加函数uint64 sys\_trace(void)，该函数通过argint函数读取参数赋值给mask变量，然后与trace\_mask字段位或即可
2. 修改syscall函数，当系统调用号和trace\_mask匹配时输出相关信息

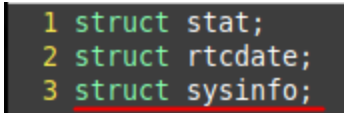


1. 编写user/trace.c
2. 修改Makefile,将$U/\_trace添加到Makefile的UPROGS中
3. make qemu,进行测试

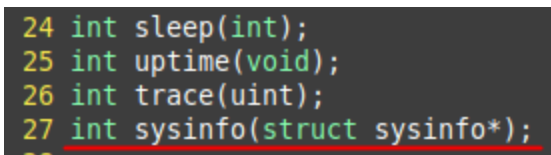
### **2.2.2 Sysinfo**

一、用户接口代码的修改

1.在user/user.h中声明struct sysinfo，其中struct sysinfo在kernel/sysinfo.h中定义



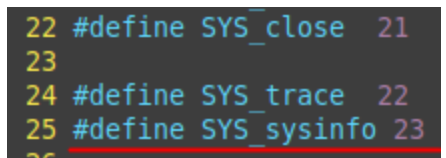
1. 在user/user.h中添加系统调用函数的定义



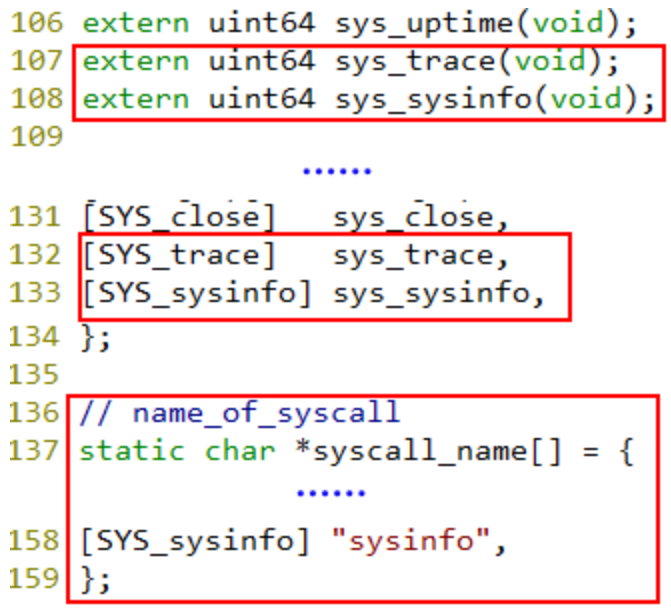
1. 在user/usys.pl中添加入口 entry("sysinfo")

二、内核代码的修改

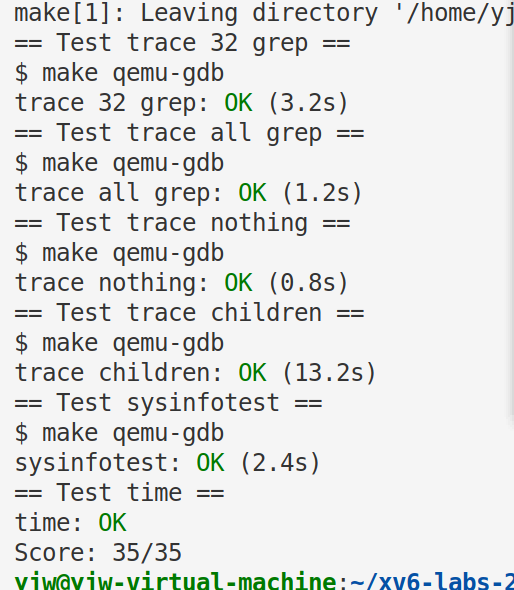
1.在kernel/syscall.h中定义系统调用号



1. 在kernel/syscall.c的syscalls函数指针数组中添加对应的函数及函数名

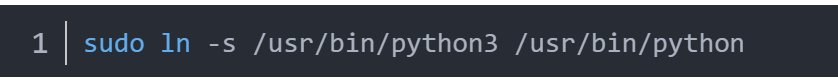


1. 在kernel/kalloc.c中添加freemem函数，在kernel/defs.h中声明freemem函数
2. 在kernel/proc.c中添加nproc函数，在kernel/defs.h中声明nproc函数
3. 在kernel/proc.c中添加头文件#include "sysinfo.h"
4. 修改Makefile,将$U/\_sysinfotest添加到Makefile的UPROGS中
5. make qemu,进行测试
6. 最后进行make grade,对这两个实验进行测试，结果如下



## **2.3 实验中遇到的问题和解决方法**

这个实验本身难度并不大，但做这个实验的时候还是遇到了不少麻烦。首先是进行实验前没有切换到syscall分支，导致在进行测试的时候终端一直显示“bash: ./grade-lab-syscall: No such file or directory”,后来在切换分支的时候又显示“当前分支有未跟踪的文件，checkout 命令会覆盖它们，请缓存( stash )或者提交( commit )”，最后通过查找资料得到的解决方法是将其存在暂存区，取出的时候使用。后来又出现“/usr/bin/env: ‘python’: No such file or directory”的错误提示，同样通过查找资料，为python创建如下符号连接得以解决。



## **2.4 实验心得**

做完这个实验，我对系统调用的理解更深刻了，并且进一步了解了xv6内核的一些内部结构。

# Lab3:Lab Page tables

## **实验目的**

该实验有三个任务，第一个任务是要求我们打印一个页表，第二个任务是给每一个进程制作一个kernel\_pagetable，第三个任务是要求我们简化 copyin/copyinstr。通过这三个实验，让我们掌握页、页表、虚拟内存、虚拟内存、地址空间等概念。

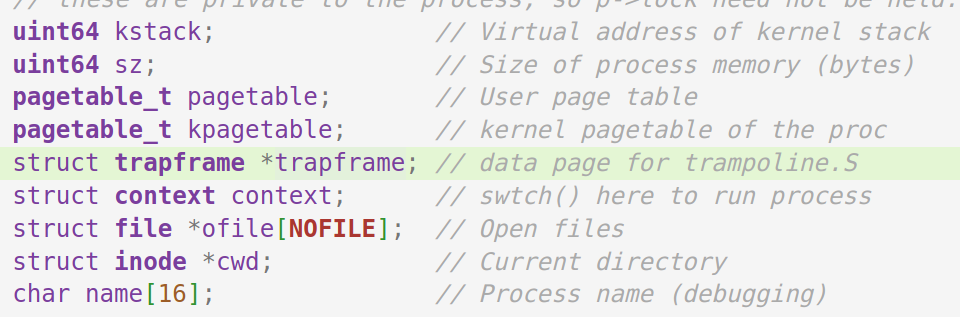
## **3.2 实验步骤**

### **3.2.1 打印一个页表**

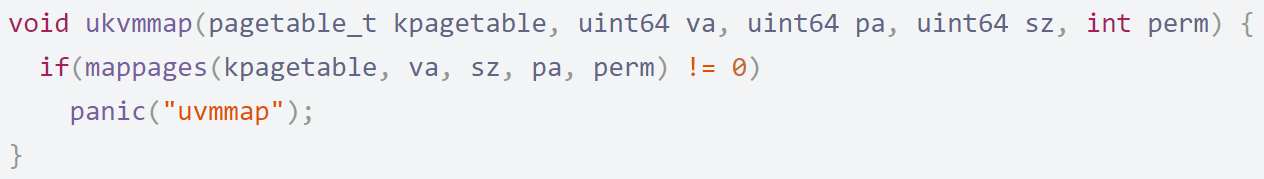
1. 在 kernel/vm.c 中编写函数 vmprint()。此处可以参考 freewalk() 函数的实现. 对于输出, 此处可以采用循环和递归两种方式实现。
2. 在 kernel/defs.h 文件中添加函数原型。
3. 在kernel/exec.c的exec函数的return argc前插入if(p->pid==1) vmprint(p->pagetable)
4. 启动xv6，进行编译测试

### **3.2.2 给每一个进程制作一个kernel\_pagetable**

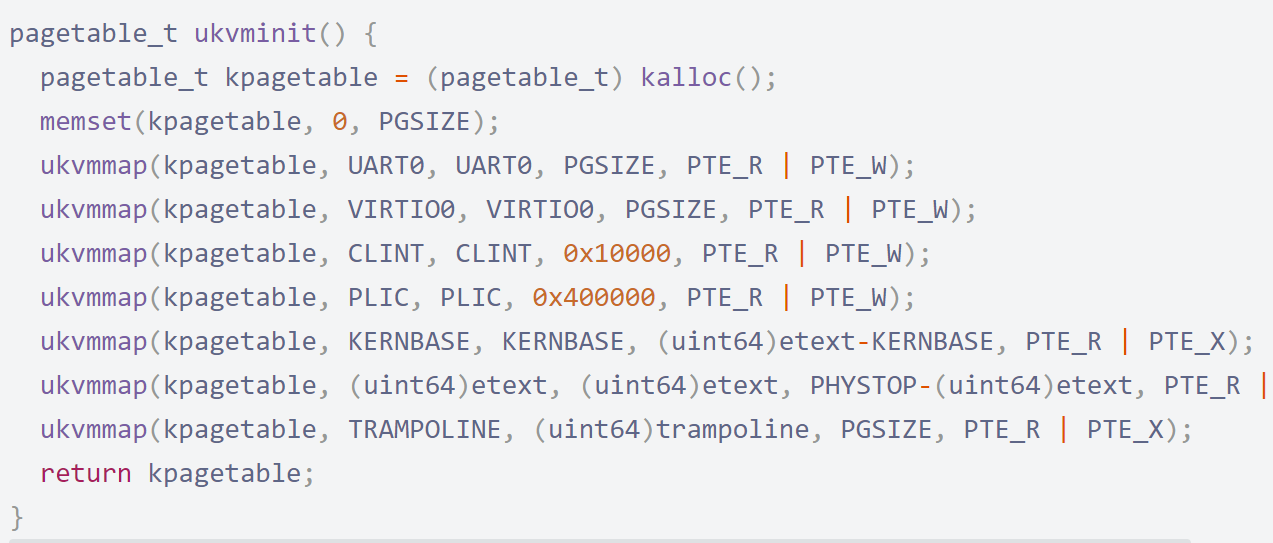
1.在 kernel/proc.h 中的 struct proc 结构体中添加成员变量 kpagetable, 代表进程的内核页表。



1. 初始化内核态页表。这部分主要参考 kernel/vm.c 里 kvminit 函数。由于其中的 kvmmap 函数是为 kernel\_pagetable 添加映射的，因此仿照它重新写一个映射函数：



然后按照 kvminit 函数的方式进行初始化。



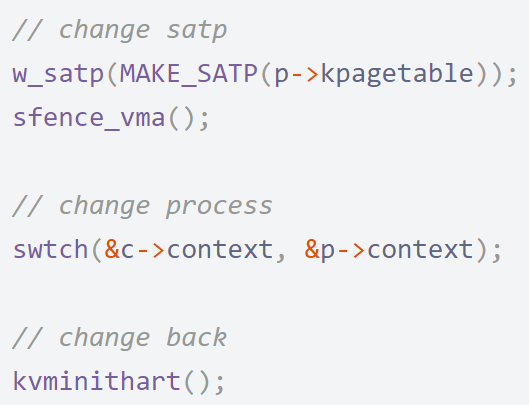
在 kernel/proc.c 中的 allocproc 函数里添加调用函数的代码：



1. 初始化内核栈。内核栈的初始化原来是在 kernel/proc.c 中的 procinit 函数内，这部分要求将函数内的代码转移到 allocproc 函数内，因此在上一步初始化内核态页表的代码下面接着添加初始化内核栈的代码：



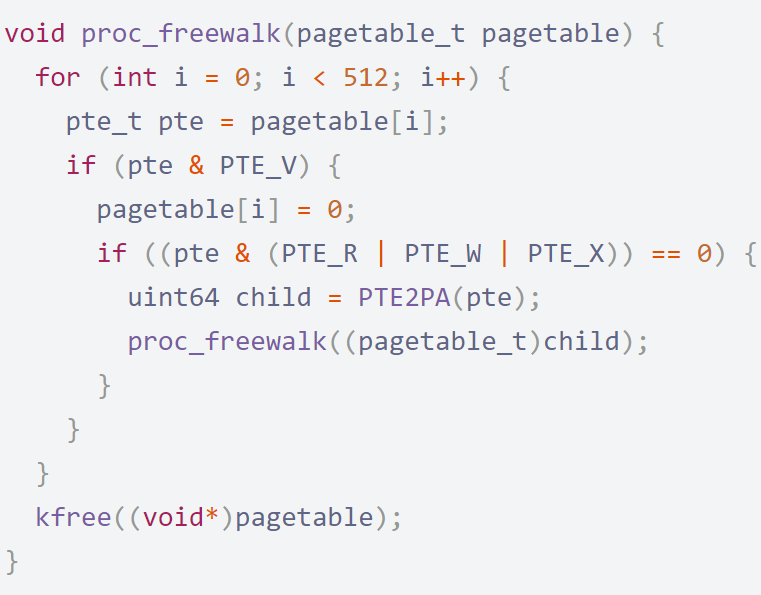
1. 进程调度时，切换内核页。内核页的管理使用的是SATP寄存器，在 kernel/proc.c 的调度函数 scheduler 中添加切换 SATP 寄存器的代码，并在调度后切换回来。



1. 释放内核栈。释放页表的第一步是先释放页表内的内核栈，因为页表内存储的内核栈地址是一个虚拟地址，需要先将这个地址指向的物理地址进行释放。



1. 释放内核页表。释放页表，直接遍历所有的页表，释放所有有效的页表项即可。



再在freeproc中进行调用：



1. 添加头文件，更改kvmpa().



1. 启动xv6，进行编译测试

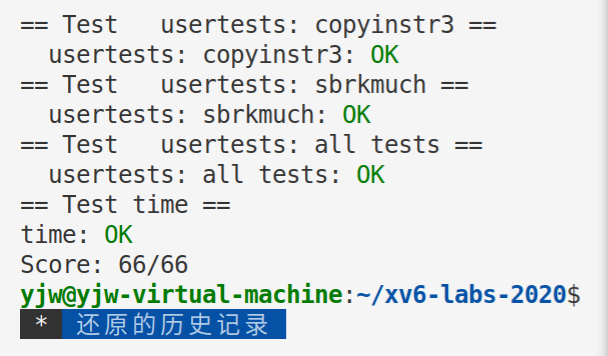
### **3.2.3 简化 copyin/copyinstr**

1.构造用于复制页表内容的函数u2kvmcopy().

2.修改fork()，sbrk()，exec() 函数，在其中添加代码.

3.调用 copyin\_new 和 copyinstr\_new。将copyin和copyinstr函数内部全部注释掉，改为调用copyin\_new和copyinstr\_new函数即可。

4. 启动xv6，进行编译测试。再运行整个lab的测试，结果如下：



## **3.3 实验中遇到的问题和解决方法**

这次的实验遇到的问题并不大，都是些小问题，例如函数定义的先后顺序、包含头文件的先后顺序、定义新变量的位置（于函数前或函数后）等，跟着实验指导步骤来走基本上都可以预期的结果，这些小问题也可以通过编译器的提示来得以解决。

## **3.4 实验心得**

和前面的两个实验相比，本次的实验难度比较大。第一个任务就是要我们定义一个名为vmprint()的函数，它用于接收一个参数并打印该可分页对象；第二个实验要修改内核，让每个进程在内核中执行时使用自己的内核页表副本，修改proc来维护每个进程的内核页表，修改scheduler在切换进程时切换内核页表；第三个实验要在刚刚创建的每个进程的内核页表中添加用户映射，以便允许copyin和copyinstr直接解除对用户指针的引用。总的来说，跟着实验指导的步骤一步一步进行操作，还是能有不小的收获。

# Lab4:Lab Traps

## 实验目的

这次实验主要是写与 traps（中断陷阱）有关的代码，分为三个实验。第一个实验主要就是要求我们输入指令“make fs.img”进行编译；第二个实验中，编译器在每个栈帧中放入一个包含调用者帧指针地址的帧指针，反向跟踪应该使用这些帧指针遍历栈帧，并在打印每个栈帧中保存的返回地址；第三个实验要求我们实现一个定时调用的函数，每过一定数目的 cpu 的切片时间，调用一个用户函数，同时，在调用完成后，需要恢复到之前没调用时的状态。

## 4.2 实验步骤

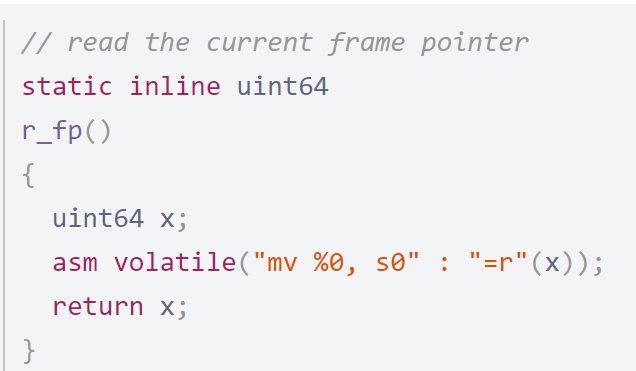
### **4.2.1 RISC-V assembly**

执行make fs.img指令，进行编译，在user目录下生成all.asm文件即可。

### **4.2.2 Backtrace**

1.在 kernel/defs.h 中添加 backtrace 的声明：void backtrace(void);

2.读取fp.由于我们需要先获取到当前函数的栈帧 fp 的值，该值存放在 s0 寄存器中，因此需要写一个能够读取 s0 寄存器值得函数。按照实验指导书给的方法，在 kernel/riscv.h 添加读取 s0 寄存器的函数：

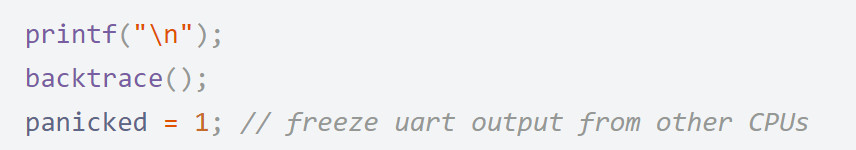


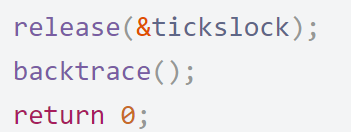
3.在 kernel/printf.c 中构造 backtrace() 函数。使用迭代方法，不断循环，输出当

前函数的返回地址，直到到达该页表起始地址为止。



4.在panic()和sys\_sleep中添加函数调用

(panic())

（sys\_sleep）

5.编译测试

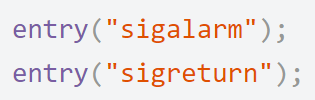
### **4.2.3 Alarm**

1.声明系统调用.

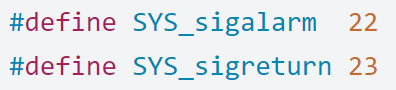
在 user/user.h 中添加声明：



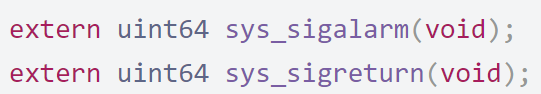
在 user/usys.pl 添加 entry，用于生成汇编代码：

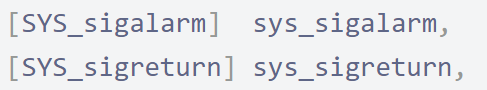


在 kernel/syscall.h 中添加函数调用码：

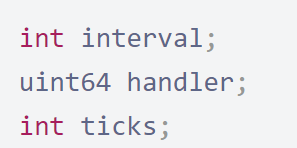


在 kernel/syscall.c 添加函数调用代码：





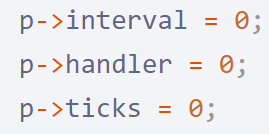
1. 实现test0.首先，写一个 sys\_sigreturn函数，直接返回 0即可；然后在 kernel/proc.h 中的 proc 结构体添加字段，用于记录时间间隔，经过的时钟数和调用的函数信息：



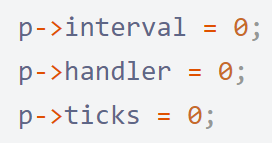
编写 sys\_sigalarm() 函数，给 proc 结构体赋值：



在allocproc()中初始化这些新加的字段：



并在freeproc()中在进程结束后释放内存：



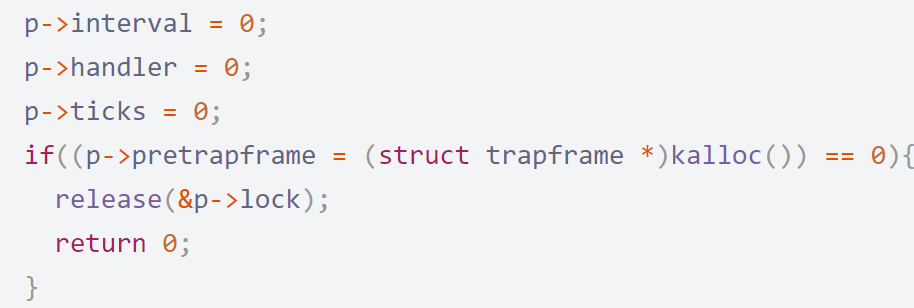
最后在时钟中断时，添加相应的处理代码：



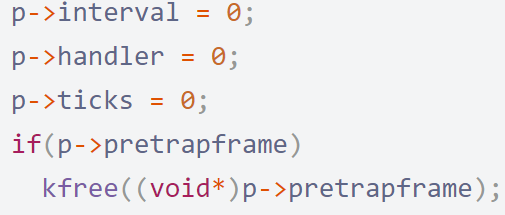
1. 实现test1/test2.在 kernel/proc.h 中添加一个指向 trapframe 结构体的指针：



在进程初始化时，为该指针进行赋值：



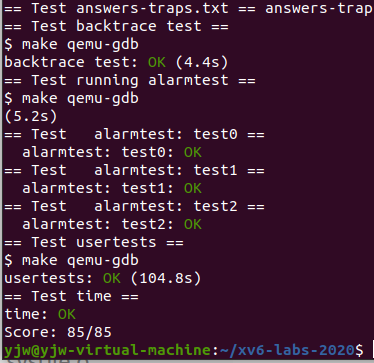
进程结束后，释放该指针：



在每次时钟中断处理时，判断是否调用 handler，如果调用了，就存储当前的 trapframe，用于调用之后的恢复：



最后，实现 sys\_sigreturn 恢复执行 handler 之前的状态：

## 4.3 实验中遇到的问题和解决方法

在这个实验中并没有遇到什么大问题，都是些小细节上没处理好的错误。

## 4.4 实验心得

内核为每个进程分配了一段栈帧内存页，用于存放栈，函数调用就是在该位置处进行的。我们可以通过当前函数的栈顶指针来找到调用该函数的函数栈帧，然后递归地进行下去，直到到达当前页的开始地址。总的来说，实验的难度并不大，主要还是帮助我们了解中断陷阱的概念以及它的过程。

# Lab5:Lab Lazy allocation

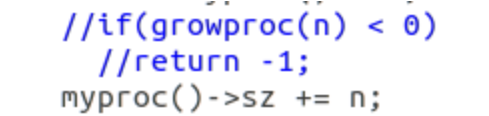
## **5.1实验目的**

Xv6应用程序使用sbrk（）系统调用向内核请求堆内存。在我们的内核中，sbrk（）分配物理内存并将其映射到进程的虚拟地址空间。但是，有些程序使用sbrk（）来请求大量内存，但从未使用大部分内存。为了对这种情况进行优化，复杂的内核延迟地分配用户内存。在本实验中，我们要向xv6添加这个延迟分配特性。

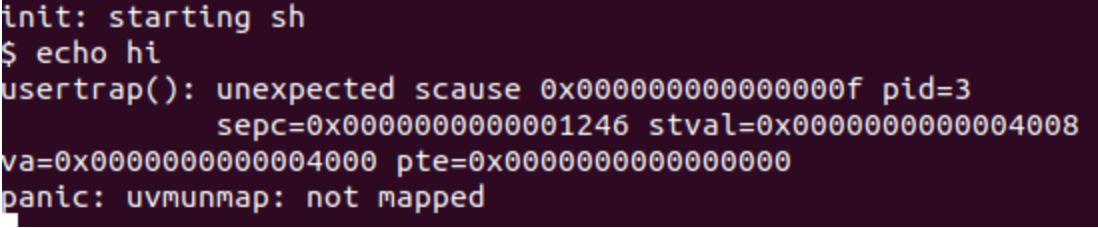
## **5.2实验步骤**

### **5.2.1 Eliminate allocation from sbrk()**

1.根据实验要求，让sbrk(n)系统调用时，增长n byte的内存空间

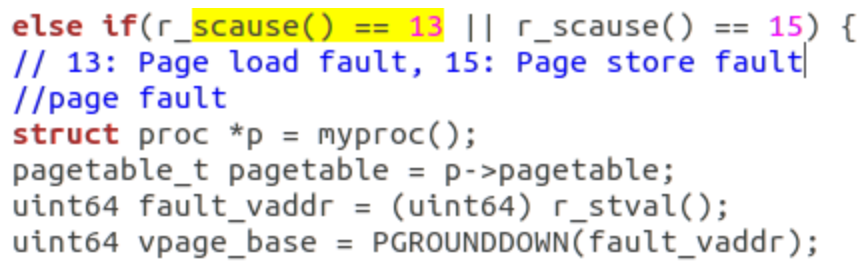


1. 输入echo hi，进行测试

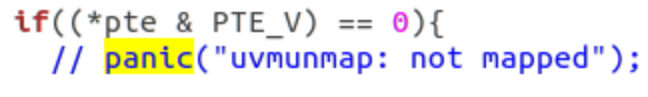


### **5.2.2 Lazy allocation**

1.修改trap.c中的usertrap()



2.对vm.c的uvmunmap()进行修改，同时我们需要引入两个头文件，“proc.h”和“spinlock.h”



1. 重新执行echo hi，执行成功
2. 为了成功通过usertests，还需要补充以下修改



1. make qemu，进行测试，与预期结果相同

### **5.2.3 Lazytests and Usertests**

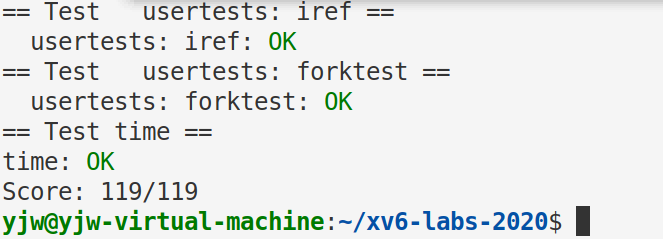
1.处理sbrk()为负数，减少相应的内存，即dealloc相应的内存n，注意n不能大于p->sz

2.修改fork()，正确处理父到子内存拷贝

3.在walkaddr里修改

4.进行测试

5.运行make grade，对上述所有实验进行测试



## **5.3实验中遇到的问题和解决方法**

这次的实验相较而言难度并不大，但在测试的时候有报错，后来仔细检查过后发现是其中一个文件漏加加头文件声明，补加之后便通过了测试。

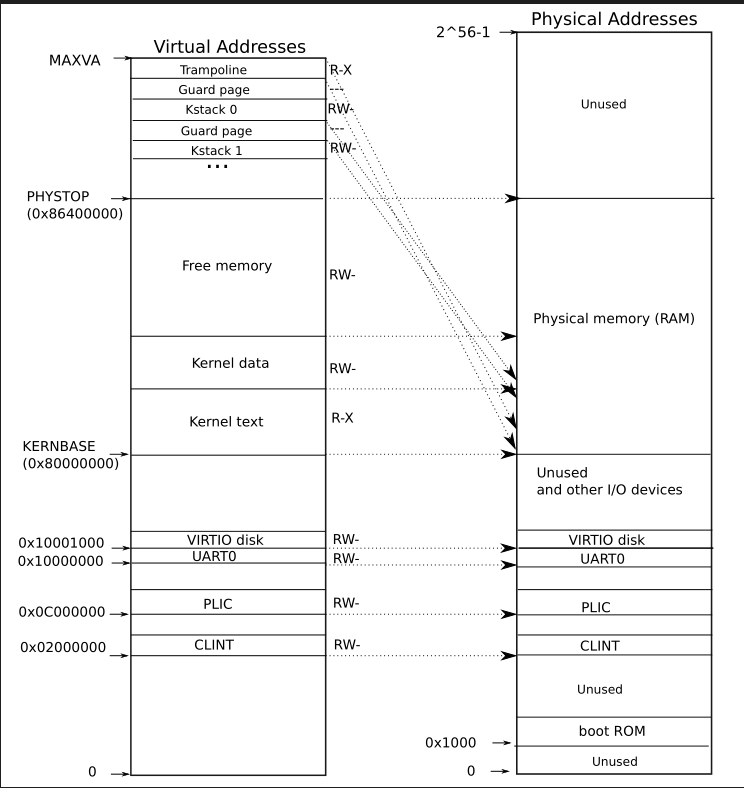
## **5.4实验心得**

很多时候，一个进程可能使用sbrk申请了大量的内存，但是其中的大部分并没有使用，造成了浪费。我们要做的是修改sbrk的实现，sbrk不分配物理内存。当进程第一次试图使用任何内存时，产生一个page fault，再让kernel分配内存，清零，映射。

# Lab6:Lab Copy on-write

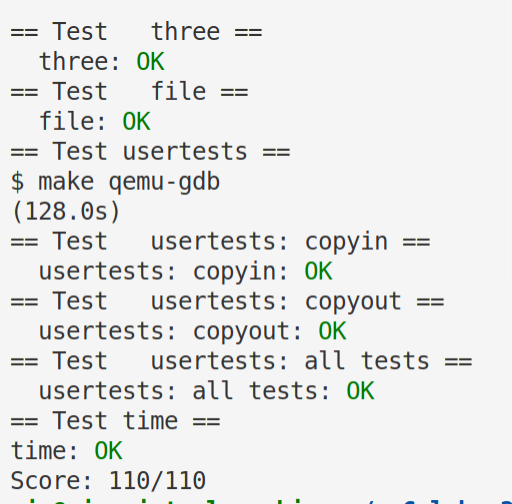
## **6.1实验目的**

在 shell 中执行指令时，首先会 fork 一个子进程，然后在子进程中使用 exec 执行 shell 中的指令。在这个过程中，fork 需要完整的拷贝所有父进程的地址空间，但在 exec 执行时，又会完全丢弃这个地址空间，创建一个新的，因此会造成很大的浪费。本次实验的任务便是就是实现一个内存的写时复制机制（copy-on-write fork），也称为 COW。具体的映射如下图：



## **6.2实验步骤**

1. 创建page的计数数组。首先对每个物理页面创建一个计数变量，保存在一个数组中，页面的数目就是数组的长度。这里有一个知识点：不是所有的物理内存都可以被用户进程映射到的，这里有一个范围，即 KERNBASE 到 PHYSTOP。
2. uvmcopy.在创建好计数数组后，在 fork 时，直接使用原来的物理页进行映射.在 kernel/vm.c 中修改 uvmcopy 函数。
3. 处理中断 usertrap.在 usertrap 中添加中断处理逻辑.其中的 cow\_alloc 函数，在 kalloc.c 中实现，并在 defs.h 中进行声明。
4. 内核写内存 copyout.直接调用上述的 cow\_alloc 函数即可。最后输入make grade进行测试。



## **6.3实验中遇到的问题和解决方法**

这个实验的问题主要就是当时在做函数的时候，对于有些定义在kernel中的函数忽略了将其在def.h中进行声明，导致一直报错，后来注意到这个细节之后就可以正常跑通了，也为我做后面的实验积累了重要经验与教训。

## **6.4实验心得**

本次的实验和前面的懒分配实验很相近，通过实验、查资料等方式，对于cow机制也有了更深的理解。

# Lab7:Lab Multithreading

## **7.1 实验目的**

这个实验的主要内容是多线程，分为三个实验，第一个实验是在用户模式模拟一个有多个用户线程的进程，第二个实验是利用加锁操作来解决数据丢失问题，第三个实验涉及互斥锁和条件变量配合达到线程同步。

## **7.2 实验步骤**

### **7.2.1 Uthread: switching between threads**

1.上下文切换.首先是 uthread\_switch.S 中实现上下文切换

2.定义上下文字段.

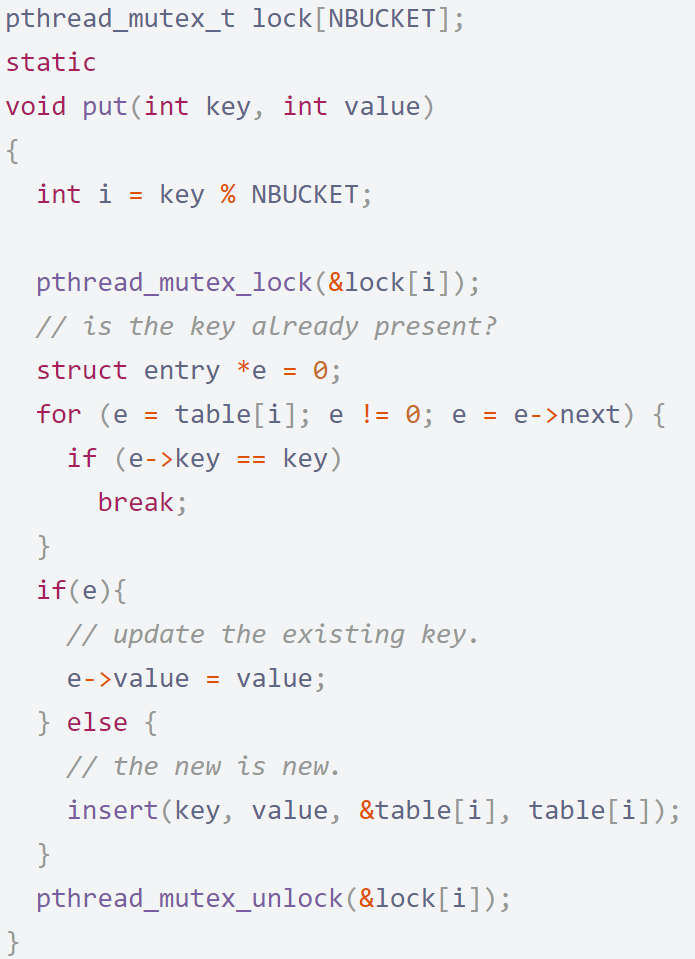
3.调度 thread\_schedule.在 thread\_schedule 中调用 thread\_switch.

4.创建并初始化线程.

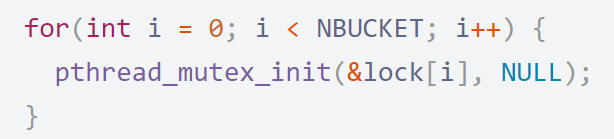


### **7.2.2 Using threads**

为每个bucket加一个锁



并在main函数中初始化锁：



### **7.2.3 Barrier**

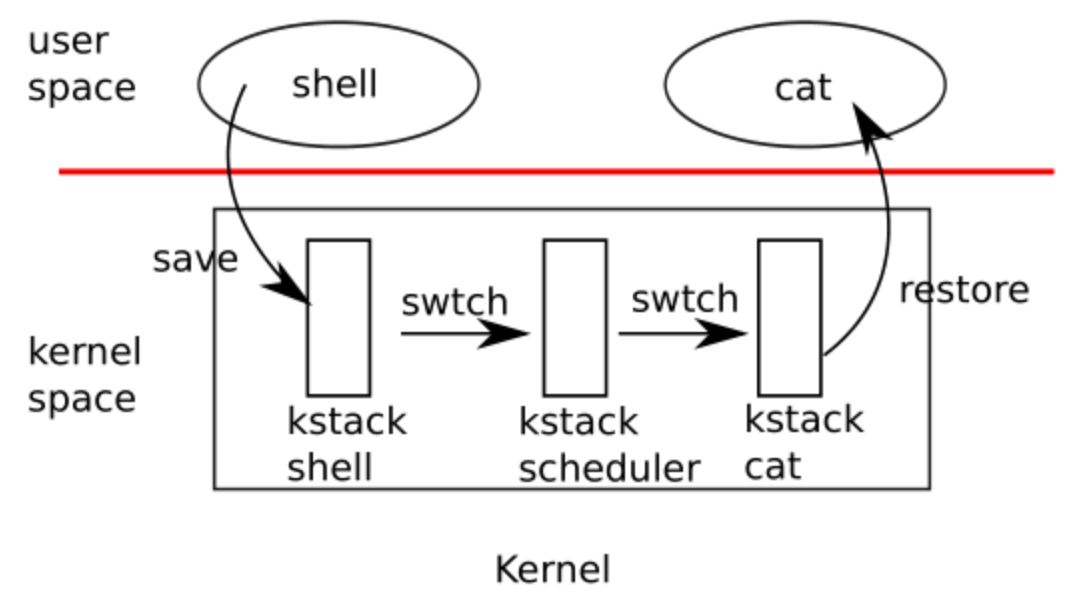


## **7.3 实验中遇到的问题和解决方法**

本次实验中用到了较多上学期课程内的知识，如进程、线程、用户态等概念，而第一个实验的内容和协程有点相近，在做实验的时候这些概念有些模糊了，后来又重新回去复习了一遍。

## **7.4 实验心得**

这个实验主要用到的是多线程中的基础知识，通过实验复习了进程调度和进程切换的相关过程。进程调度的过程如下：



在xv6中，一个进程只有一个线程，线程的切换即进程的切换。进程从用户态进入内核，保存页表、PC、寄存器，然后调用yield，之后调用swtch切换上下文，这样便完成了线程的切换。实验难度并不算太大，但对于我们理解进程还是很有帮助的。

# Lab8:Lab Locks

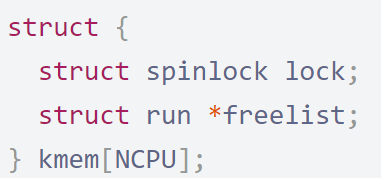
## **8.1 实验目的**

这次实验的目的是设计锁，主要目的是降低多线程情况下对锁的竞争，分为两个实验，第一个实验是解决内存块的竞争问题，第二个实验是要解决cache缓存竞争的问题。

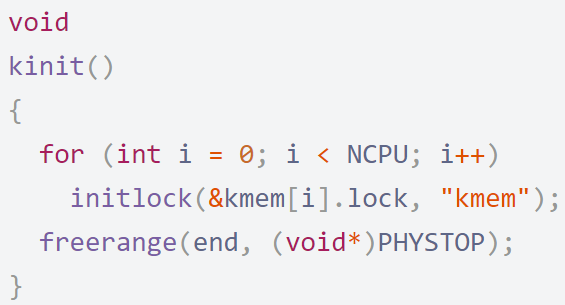
## **8.2 实验步骤**

### **8.2.1 Memory allocator**

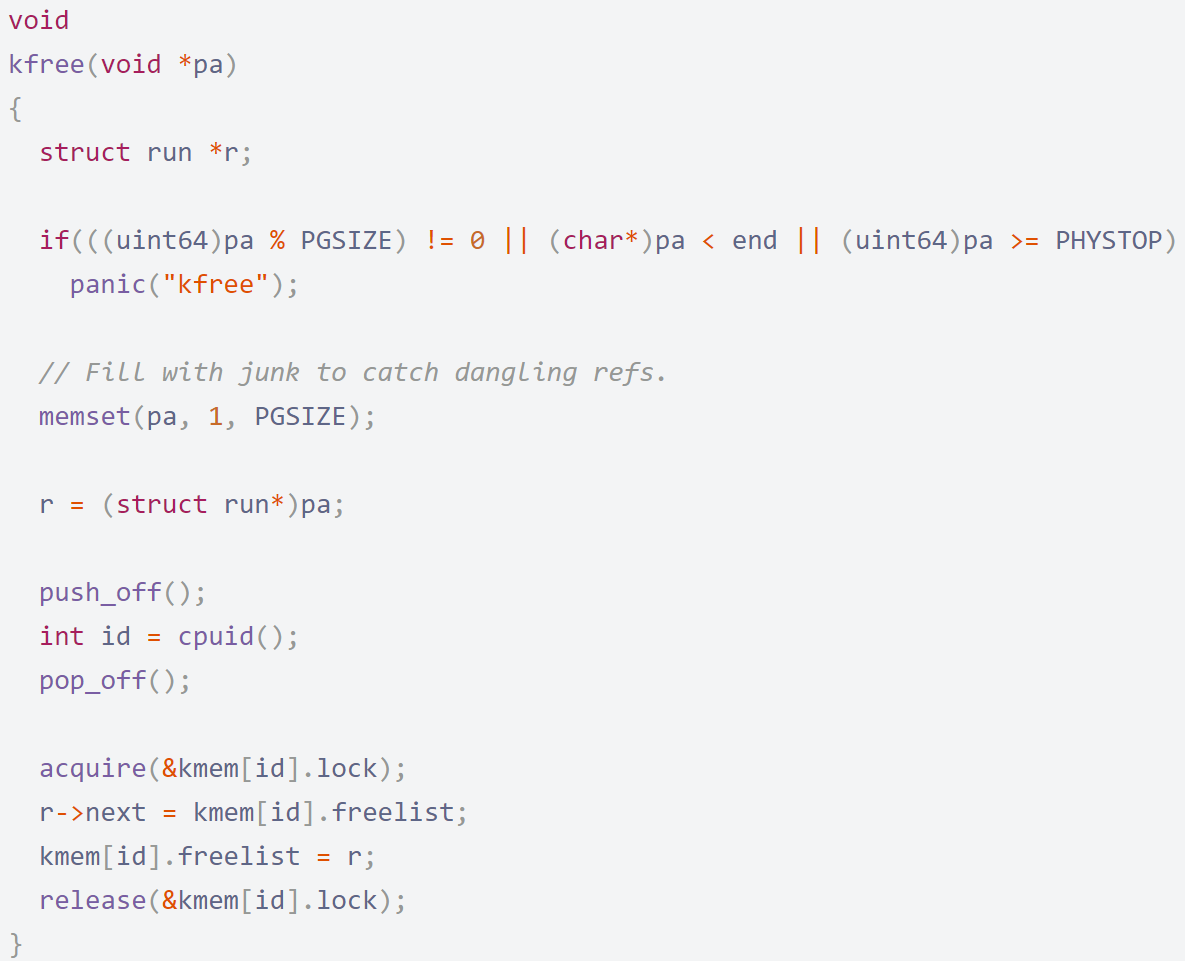
1.更改内存块结构.



1. 修改kinit.将所有空闲内存全部分配给当前 cpu 的 freelist。



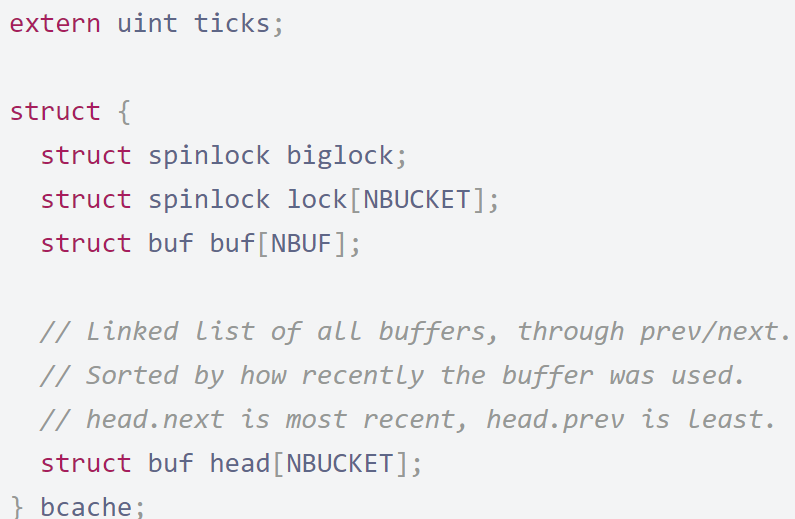
1. 修改kfree.获取 cpuid 的时候需要关闭中断。



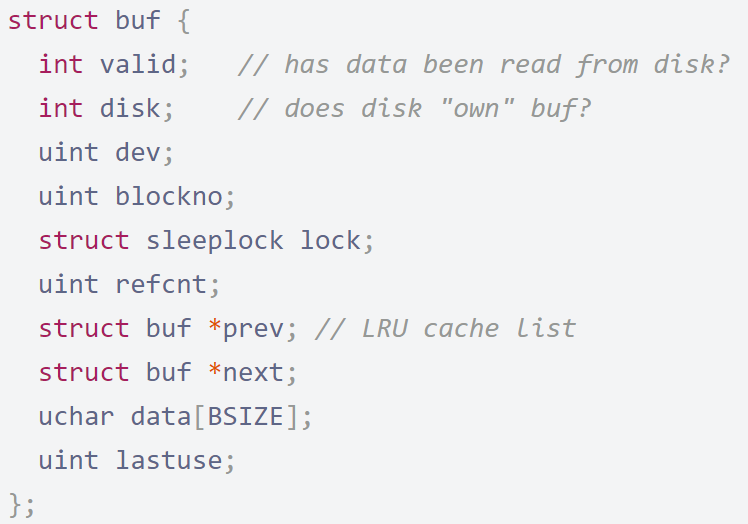
1. 修改kalloc.如果当前 cpu 有空闲内存块，就直接返回；没有的话，从其他 cpu 对应的 freelist 中“偷”一块。

### **8.2.2 Buffer cache**

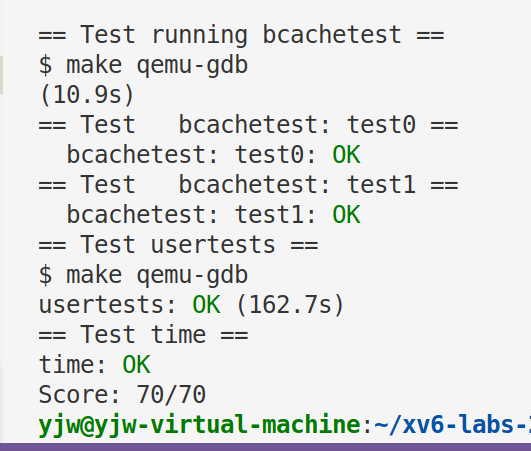
1.修改结构.将buf分成13份，同时获取 trap.c 中的 ticks 变量。这里不仅需要所有 bucket 的小锁，还需要一个大的锁防止死锁。不能像任务一那样直接使用 bcache 数组，这会改变 buf 的大小。



为 kernel/buf.h 中添加 lastuse 字段，便于使用 LRU 机制：



1. 初始化.构造binit().
2. 修改 brels，bpin和bunpin,这些函数直接加锁即可。brelse 中，由于不使用之前的方式实现 LRU，因此当遇到空闲块的时候，直接设置它的使用时间即可。
3. 构造bget.首先还是判断是否命中，如果已经缓存好，直接返回；如果没找到，释放锁，按顺序先获取大锁，再获取小锁，避免死锁；这时由于可能释放锁后，又可能会有缓存，因此再遍历一遍；如果还没命中，就去寻找当前 bucket 对应的 LRU 的空闲块，使用 ticks 的方式寻找，如果找到了，就返回； 如果还没找到，需要向其他 bucket 中拿内存块。最后输入make grade进行测试。



## **8.3 实验中遇到的问题和解决方法**

在实验前对于锁竞争的概念还不是很清晰，因此遇到最大的问题就是概念的不理解，后来通过学习知道多核计算机上并行性差的一个常见症状是高锁争用。为了减少争用，提高并行性通常需要同时改变数据结构和锁定策略。这需要我们为xv6内存分配器和块缓存执行此操作。在这个思路下进行实验就比较顺利了。

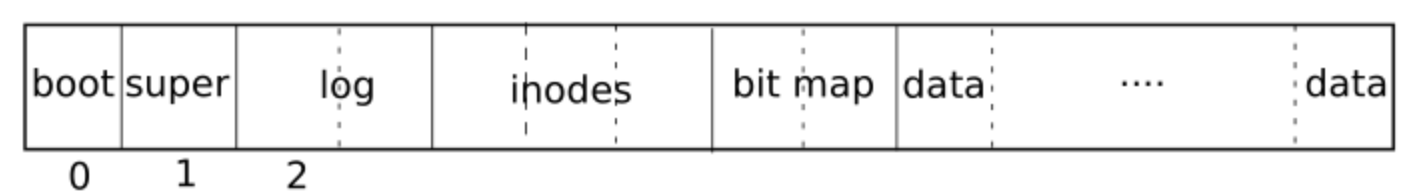
## **8.4 实验心得**

实验的整体思路比较清晰，就是降低锁的粒度，将大锁更换为粒度小的锁，从而大幅度降低锁的竞争。通过本次实验，对于锁的理解和认知更进了一步，学习到新知识的同时也复习了操作系统课上的知识。

# Lab9:Lab File system

## **9.1 实验目的**

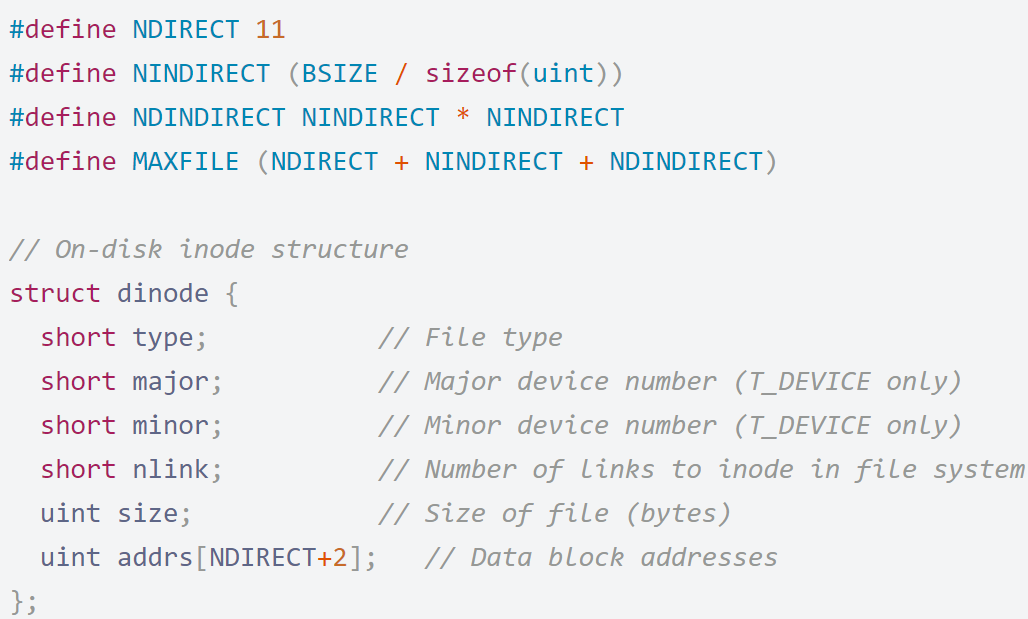
本实验的目的是向xv6文件系统添加大文件和符号链接。xv6存储文件的方式是以 block 的形式，物理磁盘在读写时，是以扇区为单位的。xv6是两个扇区，即一个 block 为 1024 个字节。磁盘需要按照操作系统读写的标准来存储数据，格式如下：



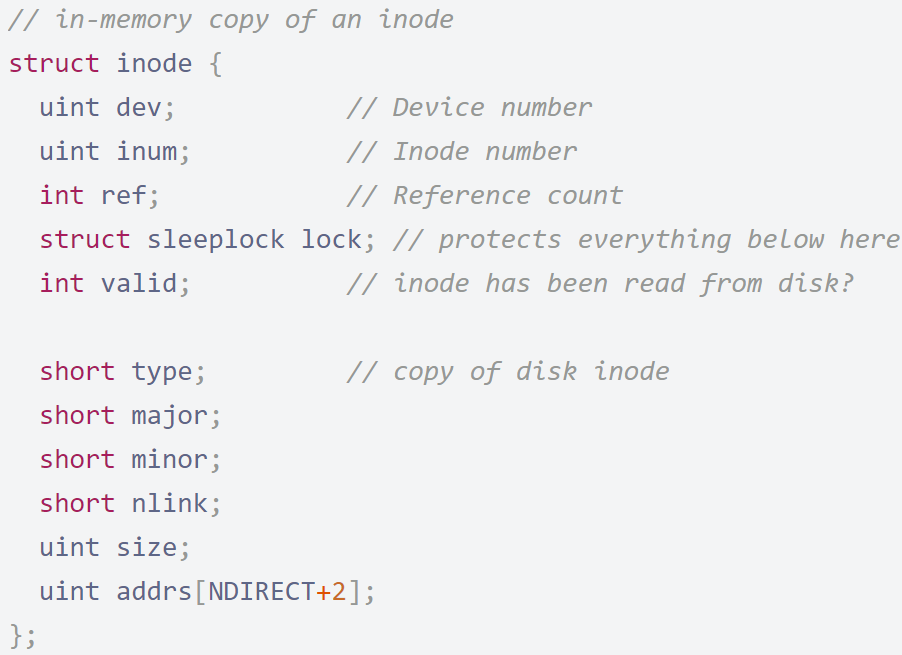
## **9.2 实验步骤**

### 9.2.1 Large files

1.修改inode数据结构.kernel/fs.h 文件中减小 NDIRECT 的值，为二级索引留一个位置：



上面的是磁盘中的 inode 结构，还需要在 kernel/file.h 中更改内存中的 inode 结构：



1. 实现bmap映射.仿照一级索引，定义二级索引
2. itrunc清理.在 kernel/fs.c 中，添加第二级索引的释放操作：

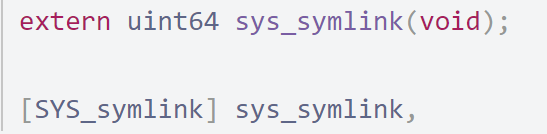


### 9.2.2 Symbolic links

1.增加symlink系统调用.user/usys.pl：

user/user.h：

kernel/syscall.h：

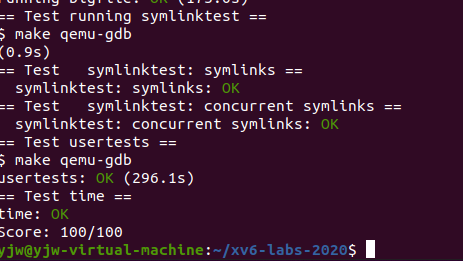
kernel/syscall.c：

最后实现系统调用函数

2.增加标志位：kernel/stat.h中添加T\_SIMLINK ,kernel/fcntl.h中添加O\_NOFOLLOW

3.修改 sys\_open 函数.在打开文件时，如果遇到符号链接，直接打开对应的文件。这里为了避免符号链接彼此之间互相链接，导致死循环，设置了一个访问深度，如果到达该访问次数，则说明打开文件失败。每次先读取对应的 inode，根据其中的文件名称找到对应的 inode，然后继续判断该 inode 是否为符号链接.

4.在终端运行make grade，进行检测



## **9.3 实验中遇到的问题和解决方法**

最初做这个实验在最后make grade进行测试时发现出现超时，并且报错

qemu-system-riscv64: terminating on signal 15 from pid 45444 (make)，后来通过增大文件grade-lab-fs中的timeout时间得以解决。

## **9.4 实验心得**

这次实验的内容和操作系统课程上所对应的索引分配等部分关联较大，第一个实验就是通过实现二级索引扩大所支持的文件大小，第二个实验是实现符号链接。总体来说实验难度适中，主要还是复习了上学期所学的知识。

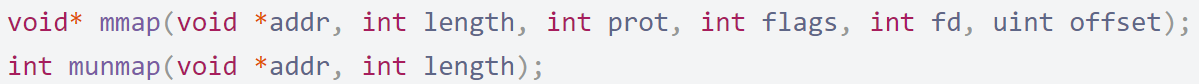
# Lab10:Lab Mmap

## **10.1 实验目的**

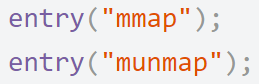
这次的任务主要是要求我们实现两个系统调用：mmap与munmap，前者主要是进行内存映射，将一个内存区域映射到当前用户进程的区域，而后者则是取消映射。该实验主要考察虚拟内存和文件系统的相关知识以及两者的综合使用。

## **10.2 实验步骤**

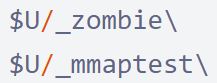
1.添加系统调用.在 user/user.h 中添加系统调用声明：



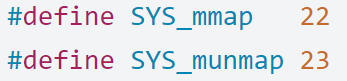
在 user/usys.pl 中添加 entry：



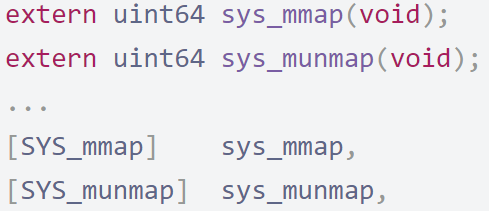
在 Makefile 中添加编译声明：



在 kernel/syscall.h 中添加系统调用码：



在 kernel/syscall.c 中添加系统调用函数引用：



在 kernel/sysfile.c 中添加sys\_mmap和sys\_munmap函数的声明和实现。

2.实现 mmap 函数.首先接收传来的参数，判断参数的合法性，然后遍历VMA数组，找到还没有使用的 vma，将参数信息添加进去。这里映射的虚拟地址，可以直接填写堆的最高地址，然后让堆继续生长。

3.实现trap中断处理.与实验五相似，由于是懒加载，在读取或写入相应的虚拟地址时，会存在地址未映射的情况。这时需要将物理地址上的数据读到虚拟地址中，然后重新进行读取或写入操作。

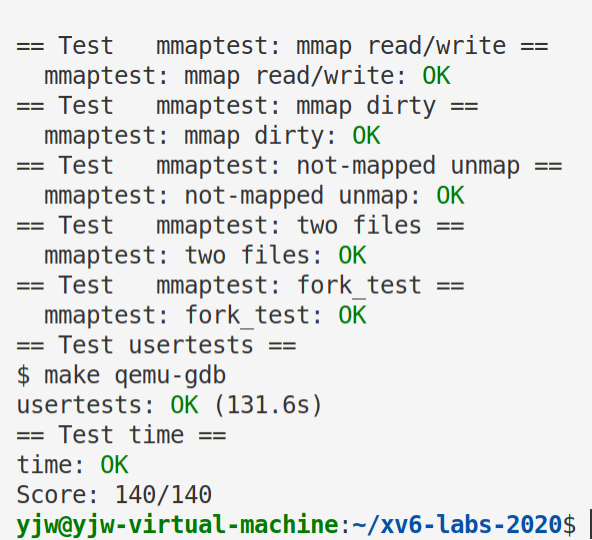
4.实现munmap函数.这部分主要是取消虚拟地址的映射关系，同时，设置进程 VMA 结构体相应的 vma 为未使用状态。这部分函数，实验中做了简化，只需要取消地址与传入地址相同的文件的映射即可。

5.fork/exit.在进程创建和退出时，需要复制和清空相应的文件映射：





6.终端输入make grade进行测试.



## **10.3 实验中遇到问题和解决方法**

实验中也没有遇到太大的问题，主要还是得要理清两个函数的作用：mmap函数是将文件地址映射到虚拟内存中，返回映射后的地址，同时记录该进程映射到的文件信息。 munmap函数就是取消进程地址空间中，文件地址某一部分的映射。

## **10.4 实验心得**

这个实验有点像前几个实验的一个综合版，关键点还是在于关注 mmap 系统调用的各个参数如何使用，以及其返回值。操作系统在很多方面采用的策略如懒分配、cow等帮助了性能的提升，同时也减少了内存的分配。这个实验是虚拟内存的一个应用，也让我看到了虚拟内存的高效性。

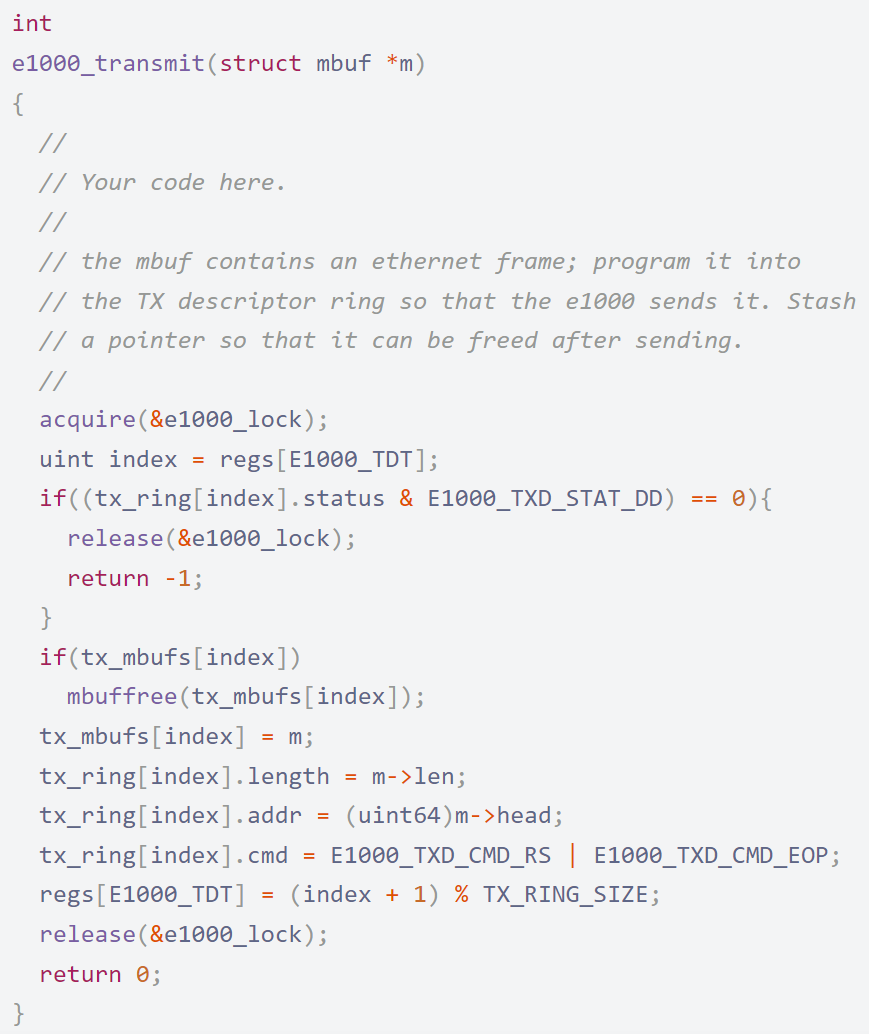
# Lab11:Lab Network driver

## **11.1 实验目的**

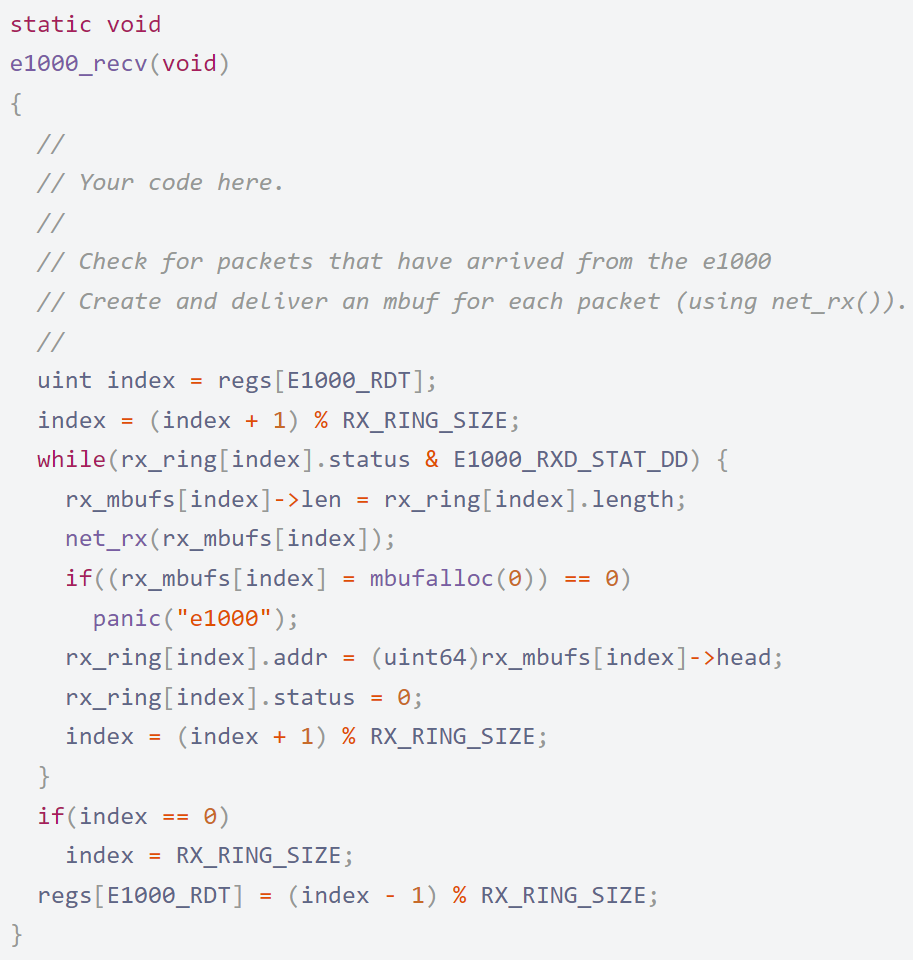
几乎所有的外部设备的机制都是产生中断，发送/接收信息，磁盘是很复杂的一种外设。不同设备的发送和接收接口是不同的，因此需要写相应设备的驱动才可以正常使用。本次实验是写两个函数，一个用于发送，另一个用于接收。本次实验参考的是 e1000 的硬件手册，并根据手册内容书写发送和接收函数，主要是利用一个循环列表不断读取和写入。

## **11.2 实验步骤**

e1000\_transmit 函数：



e1000\_recv 函数：



## **11.3 实验中遇到的问题和解决方法**

最后一个实验难度并不大，没有遇到什么问题。

## **11.4 实验心得**

作为mit的最后一个实验，主要就是写一个网卡驱动，主要还是在网上资源的帮助下完成了。但是网络上的教程资源非常多而且质量差距很大，一开始也是跟着教程一步一步地走，但从第三个开始就几乎每个实验都会遇到或大或小的问题，而由于之前也没遇到过诸如此类的问题，就会让我感觉很不知所措，后来也需要不断地去查询网络上的资料以及询问老师和同学。总体来说，做完这11个实验，我的收获很大。