操作系统课程设计

2151121 高羽

目录

[一、 远程仓库与环境搭建 4](#_Toc143489757)

[远程仓库 4](#_Toc143489758)

[1.安装并配置虚拟机VMware 4](#_Toc143489759)

[2.下载Ubuntu22.04 4](#_Toc143489760)

[3.下载Visual Studio Code 5](#_Toc143489761)

[4.在Ubuntu安装相关包 5](#_Toc143489762)

[5.成功搭建 6](#_Toc143489763)

[二、运行示例 6](#_Toc143489764)

[1.切换到xv6-labs-2020代码库的lab1分支 6](#_Toc143489765)

[2.启动xv6 6](#_Toc143489766)

[3.新建源文件 6](#_Toc143489767)

[4.配置Makefile编译运行 7](#_Toc143489768)

[5.执行sleep 10 7](#_Toc143489769)

[实验⼀ Xv6 and Unix utilities 8](#_Toc143489770)

[概述 8](#_Toc143489771)

[Sleep 8](#_Toc143489772)

[实验目的 8](#_Toc143489773)

[实验步骤 8](#_Toc143489774)

[问题和解决方法 9](#_Toc143489775)

[Pingpong 9](#_Toc143489776)

[实验目的 9](#_Toc143489777)

[实验步骤 9](#_Toc143489778)

[运行截图： 10](#_Toc143489779)

[问题和解决方法 10](#_Toc143489780)

[Primes 10](#_Toc143489781)

[实验目的 10](#_Toc143489782)

[实验步骤 10](#_Toc143489783)

[问题和解决方法： 11](#_Toc143489784)

[Find 11](#_Toc143489785)

[实验目的 11](#_Toc143489786)

[实验步骤 12](#_Toc143489787)

[问题和解决方法： 12](#_Toc143489788)

[Xargs 13](#_Toc143489789)

[实验目的 13](#_Toc143489790)

[实验步骤 13](#_Toc143489791)

[问题和解决方法： 13](#_Toc143489792)

[小结 14](#_Toc143489793)

[实验⼆ System calls 14](#_Toc143489794)

[概述 14](#_Toc143489795)

[Trace 14](#_Toc143489796)

[实验目的 14](#_Toc143489797)

[实验步骤 14](#_Toc143489798)

[问题和解决办法 15](#_Toc143489799)

[Sysinfo 16](#_Toc143489800)

[实验目的 16](#_Toc143489801)

[实验步骤 16](#_Toc143489802)

[问题和解决办法 16](#_Toc143489803)

[小结 16](#_Toc143489804)

[实验三 Page Tables 17](#_Toc143489805)

[概述 17](#_Toc143489806)

[Speed up system calls 17](#_Toc143489807)

[实验目的 17](#_Toc143489808)

[实验步骤 17](#_Toc143489809)

[实验结果 18](#_Toc143489810)

[Print a page table 19](#_Toc143489811)

[实验目的 19](#_Toc143489812)

[实验步骤 19](#_Toc143489813)

[实验结果 19](#_Toc143489814)

[Detecting which pages have been accessed 20](#_Toc143489815)

[实验目的 20](#_Toc143489816)

[实验步骤 20](#_Toc143489817)

[实验结果 20](#_Toc143489818)

[小结 21](#_Toc143489819)

[实验四 traps 21](#_Toc143489820)

[概述 21](#_Toc143489821)

[RISC-V assembly 22](#_Toc143489822)

[实验目的 22](#_Toc143489823)

[实验步骤 22](#_Toc143489824)

[问题 22](#_Toc143489825)

[Backtrace 23](#_Toc143489826)

[实验目的 23](#_Toc143489827)

[实验步骤 23](#_Toc143489828)

[Alarm 25](#_Toc143489829)

[实验目的 25](#_Toc143489830)

[实验步骤 25](#_Toc143489831)

[实验问题和解决 25](#_Toc143489832)

[小结 25](#_Toc143489833)

[实验五 Copy on write 26](#_Toc143489834)

[概述 26](#_Toc143489835)

[Implement copy-on write 26](#_Toc143489836)

[实验目的 26](#_Toc143489837)

[实验步骤 26](#_Toc143489838)

[实验问题和解决 27](#_Toc143489839)

[小结 29](#_Toc143489840)

[实验六 Multithreading 29](#_Toc143489841)

[概述 29](#_Toc143489842)

[Uthread 29](#_Toc143489843)

[实验目的 29](#_Toc143489844)

[实验步骤 29](#_Toc143489845)

[Using threads 31](#_Toc143489846)

[实验目的 31](#_Toc143489847)

[实验步骤 31](#_Toc143489848)

[加锁并在main函数中初始化锁 31](#_Toc143489849)

[实验问题和解决 32](#_Toc143489850)

[Barrier 32](#_Toc143489851)

[实验目的 32](#_Toc143489852)

[实验步骤 32](#_Toc143489853)

[实验问题和解决 33](#_Toc143489854)

[小结 33](#_Toc143489855)

[实验七 Network driver 34](#_Toc143489856)

[实验目的 34](#_Toc143489857)

[实验步骤 34](#_Toc143489858)

[实验结果 35](#_Toc143489859)

[实验八 Locks 35](#_Toc143489860)

[概述 35](#_Toc143489861)

[Memory allocator 35](#_Toc143489862)

[实验目的 35](#_Toc143489863)

[实验步骤 35](#_Toc143489864)

[实验结果 37](#_Toc143489865)

[Buffer cache 38](#_Toc143489866)

[实验目的 38](#_Toc143489867)

[实验步骤 38](#_Toc143489868)

[实验问题与解决 39](#_Toc143489869)

[小结 39](#_Toc143489870)

[实验九 File System 39](#_Toc143489871)

[概述 39](#_Toc143489872)

[Large files 40](#_Toc143489873)

[实验目的 40](#_Toc143489874)

[实验步骤 40](#_Toc143489875)

[Symbolic links 41](#_Toc143489876)

[实验目的 41](#_Toc143489877)

[实验步骤 41](#_Toc143489878)

[小结 43](#_Toc143489879)

[实验⼗ Munmap 43](#_Toc143489880)

[实验目的 43](#_Toc143489881)

[实验步骤： 43](#_Toc143489882)

[实验结果 45](#_Toc143489883)

[小结 45](#_Toc143489884)

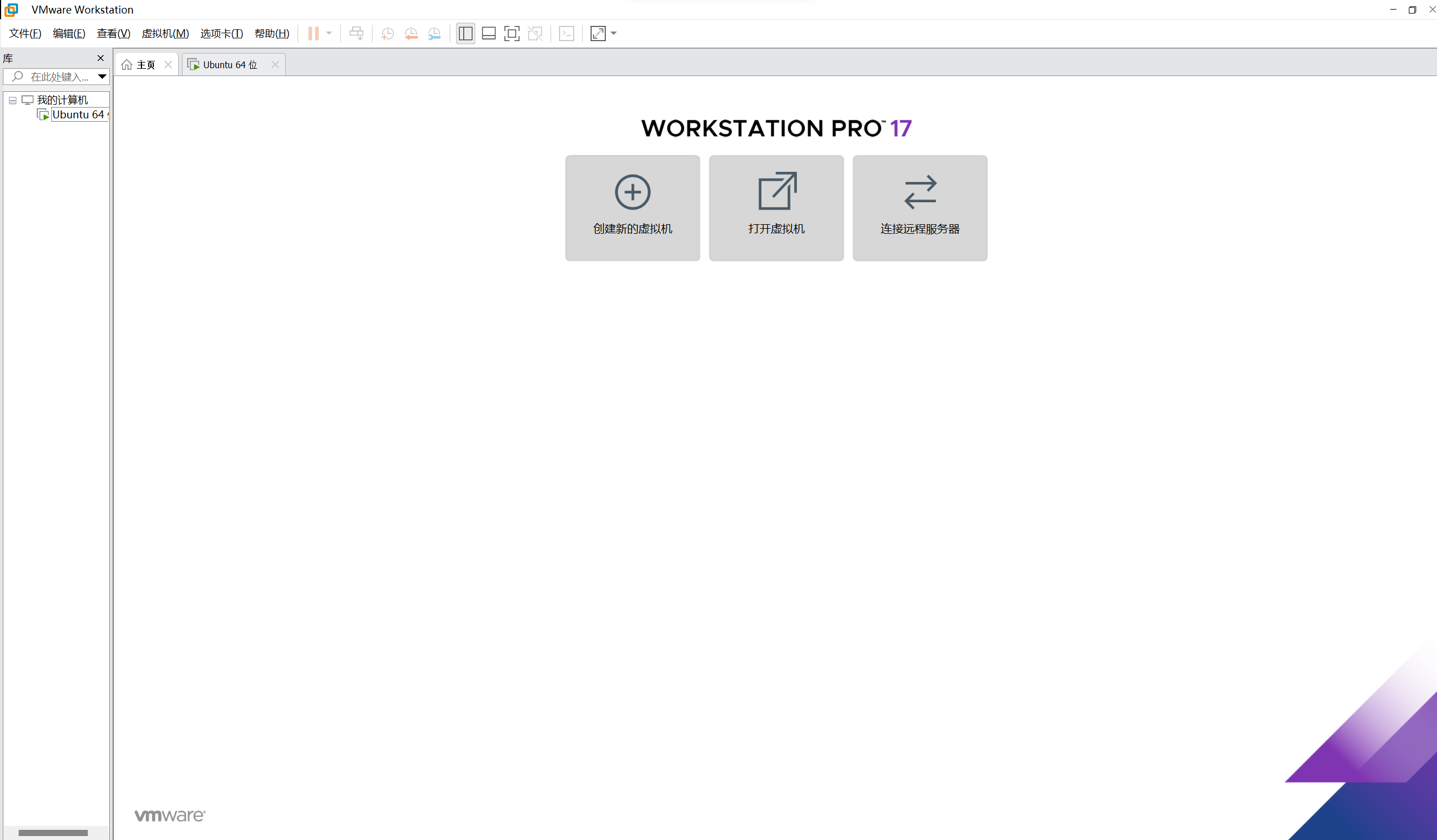
# 远程仓库与环境搭建

## 远程仓库

https://github.com/kylin370370/xv6-labs

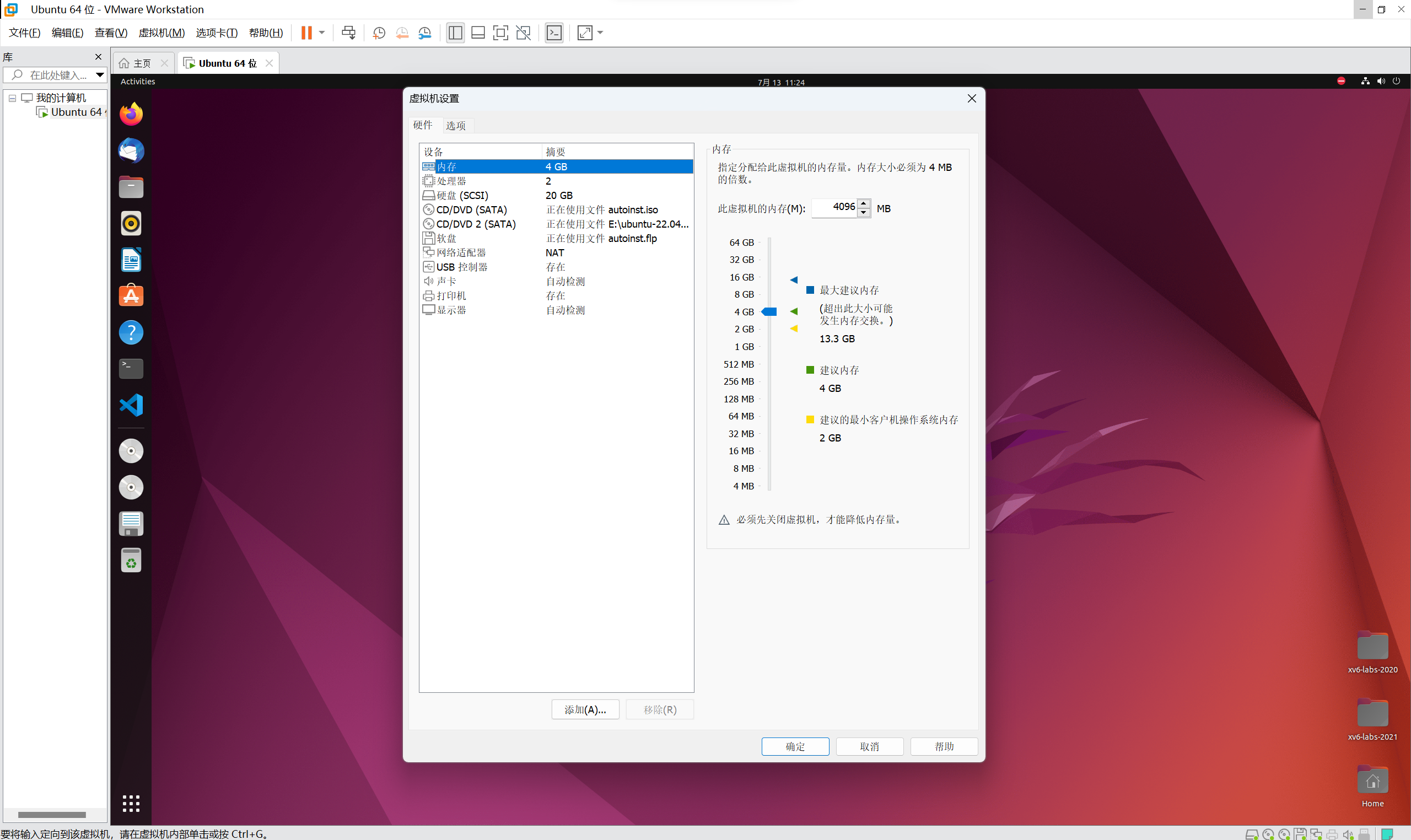
## 1.安装并配置虚拟机VMware

教程<https://blog.csdn.net/weixin_74195551/article/details/127288338>



## 2.下载Ubuntu22.04

教程<https://blog.csdn.net/weixin_45912291/article/details/108901106>

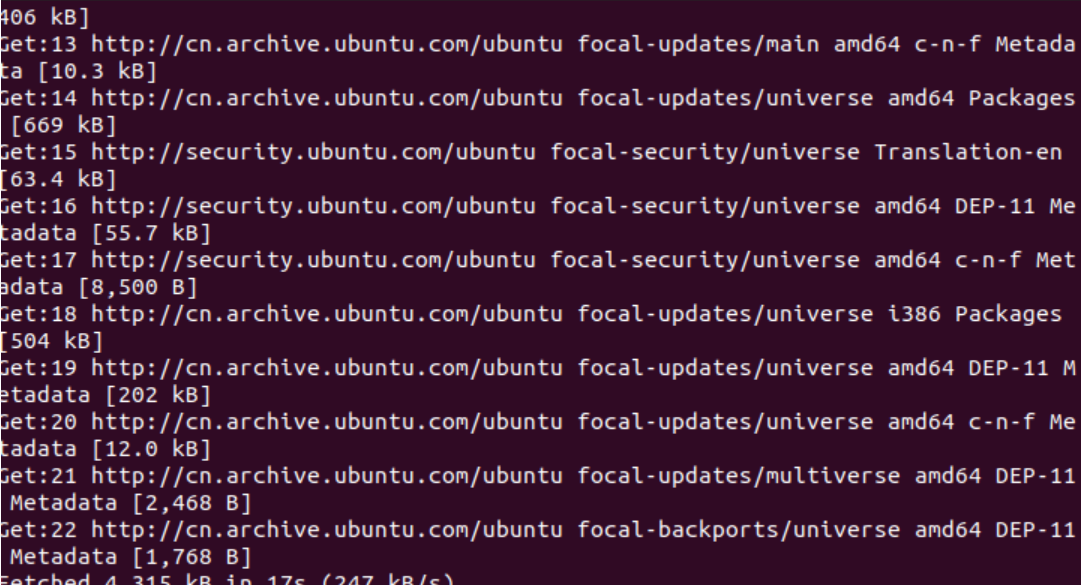


## 3.下载Visual Studio Code

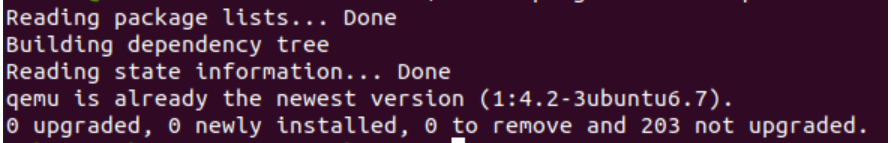
## 4.在Ubuntu安装相关包

教程https://blog.csdn.net/m0\_45291976/article/details/109121959

打开Terminal 更新/安装apt：sudo apt-get update



安装qemu：sudo apt-get install qemu



安装git：sudo apt install git

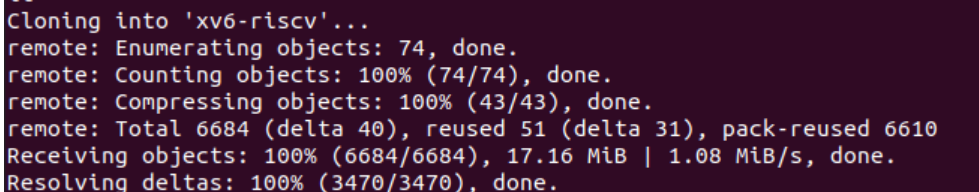
安装risc-v工具链：git clone --recursive <https://github.com/riscv/riscv-gnu-toolchain>

下载源代码文件：

完整源代码：git clone git://github.com/mit-pdos/xv6-riscv.git

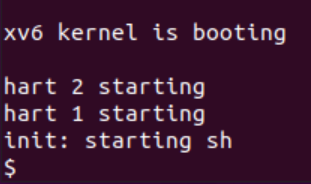
项目所用代码：git clone git://g.csail.mit.edu/xv6-labs-2021

（此处仅列举重要步骤，过程中有缺少其他包的情况，按提示下载即可）



## 5.成功搭建

在xv6目录下输入make qemu，成功运行如下



按ctrl+a松开后按x退出

# 二、运行示例

## 1.切换到xv6-labs-2020代码库的lab1分支

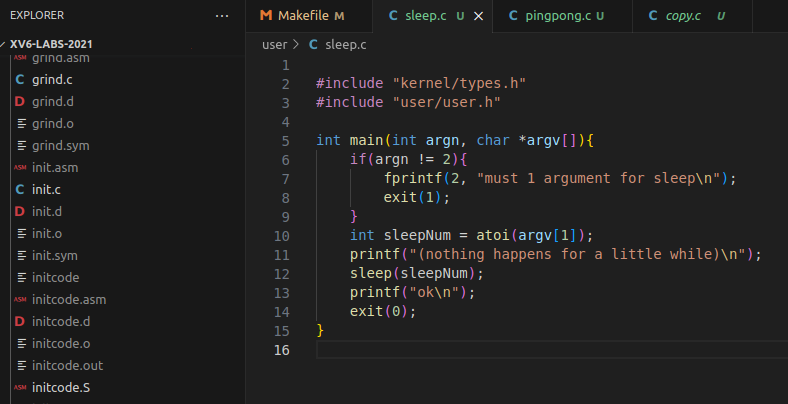
git checkout util

## 2.启动xv6

make qemu

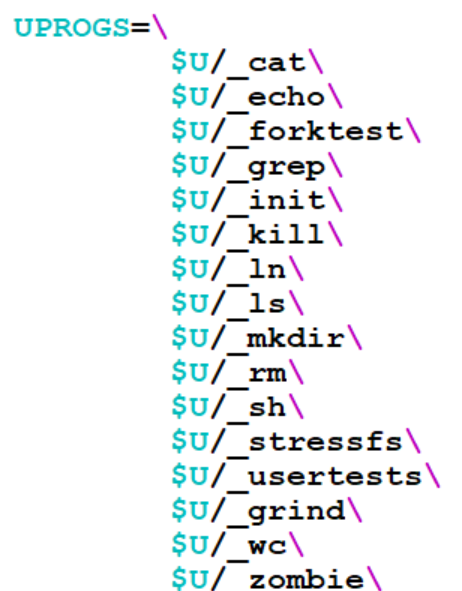
## 3.新建源文件

在xv6-labs-2021/user目录下新建sleep.c如下



## 4.配置Makefile编译运行

打开xv6-labs-2021/Makefile，添加配置：＄U/\_sleep\



## 5.执行sleep 10

# 实验⼀ Xv6 and Unix utilities

## 概述

1. 初步掌握 xv6 这⼀操作系统内核和⽤户。

2. 了解系统调⽤。

3. 了解⽗⼦进程，以及 fork()函数创建⽗⼦进程的⽅法。

4. 掌握素数筛primes，以及利⽤管道编写素数筛的⽅法。

5. 完成 find 和 xargs 程序的编写。

教程[Mit6.S081-实验1-Xv6 and Unix utilities\_mit6.s081 实验1\_解析Ta的博客-CSDN博客](https://blog.csdn.net/u013577996/article/details/108680888" \l ":~:text=2%EF%BC%8C%E6%93%8D%E4%BD%9C%E6%B5%81%E7%A8%8B%201%201%EF%BC%89%E5%88%87%E6%8D%A2%E5%88%B0xv6-labs-2020%E4%BB%A3%E7%A0%81%E5%BA%93%E7%9A%84lab1%E5%88%86%E6%94%AF%202%202%EF%BC%89%E5%90%AF%E5%8A%A8xv6,3%203%EF%BC%89%E6%B5%8B%E8%AF%95xv6%204%204%EF%BC%89%E8%BF%87%E7%A8%8B%E5%88%86%E6%9E%90%205%205%EF%BC%89%E5%85%B6%E4%BB%96%E6%93%8D%E4%BD%9C)

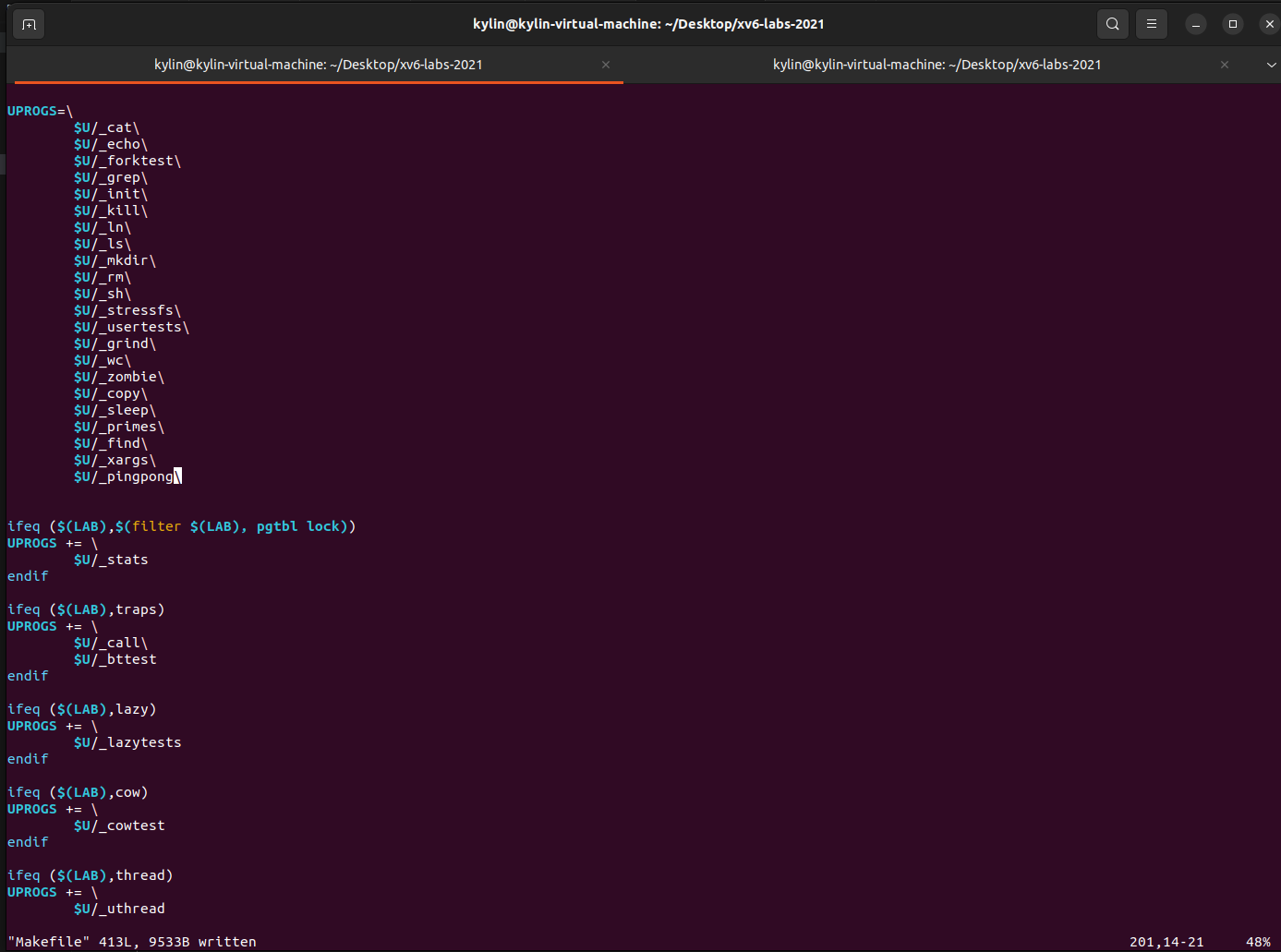
## Sleep

实验目的

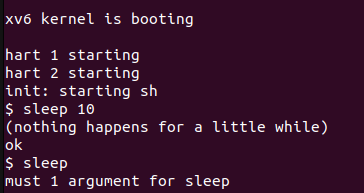
实现 UNIX 程序 sleep：sleep 应该暂停⼀段⽤户所指定的时间。

实验步骤

* 切换到util分⽀ ：git checkout util
  + 查看系统调用
  + 获取参数，利⽤系统调⽤实现sleep
  + 修改Makefile



* + 编译运⾏



问题和解决方法

遇到问题1：sleep初始值设置过大，导致运行不出；sleep初始值设置过小，导致无法观测

解决办法：反复调整sleep值，并结合秒表判断sleep 10大致为1秒

遇到问题2：sleep不加参数会报错

解决办法：判断sleep后是否有参数，若无则直接退出并给出提示

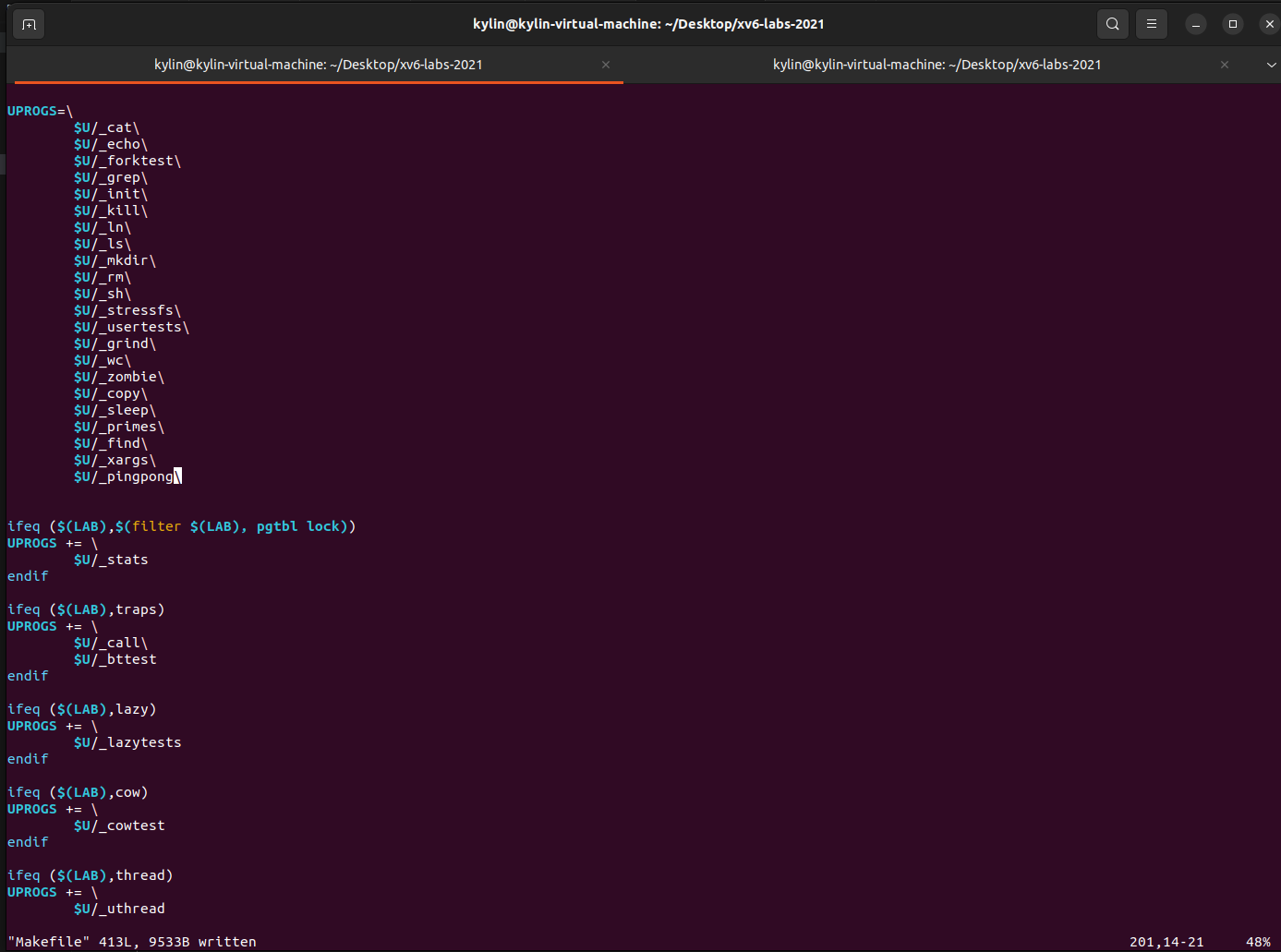
## Pingpong

实验目的

编写程序，使⽤UNIX 系统调⽤在两个进程之间通过⼀对管道“乒乓”⼀个字节。

实验步骤

* 启动qemu ：make gemu
  + 获取参数，利⽤系统调⽤实现pingpong
  + 修改Makefile



运行截图：



问题和解决方法

遇到问题：子进程先读再写，父进程先写再读，pipe同时读写会报错

解决办法：读写过后及时关闭pipe，

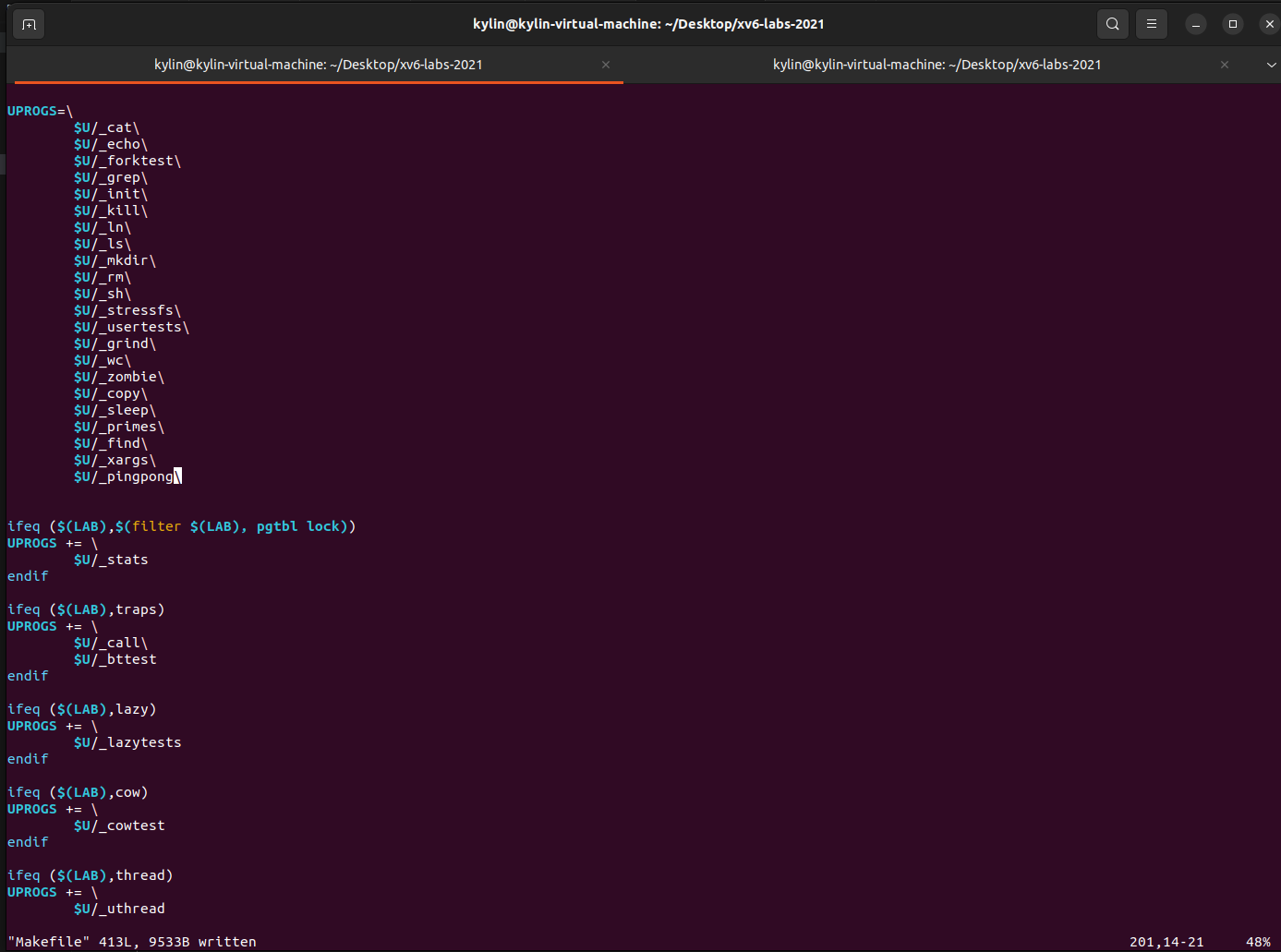
## Primes

实验目的

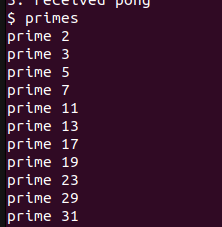
使⽤管道编写并发版本的主筛。

实验步骤

* 切换到util分⽀ ：git checkout util
* 启动qemu ：make gemu
  + 查看系统调用
  + 获取参数，利⽤系统调⽤实现pingpong
  + 修改Makefile



* + 编译运⾏



问题和解决方法：

遇到问题1：程序未停止

解决办法：pipeline最后需要close

遇到问题2：输出乱序

解决办法：write(pipeline[1])和printf需要在调用子任务之前

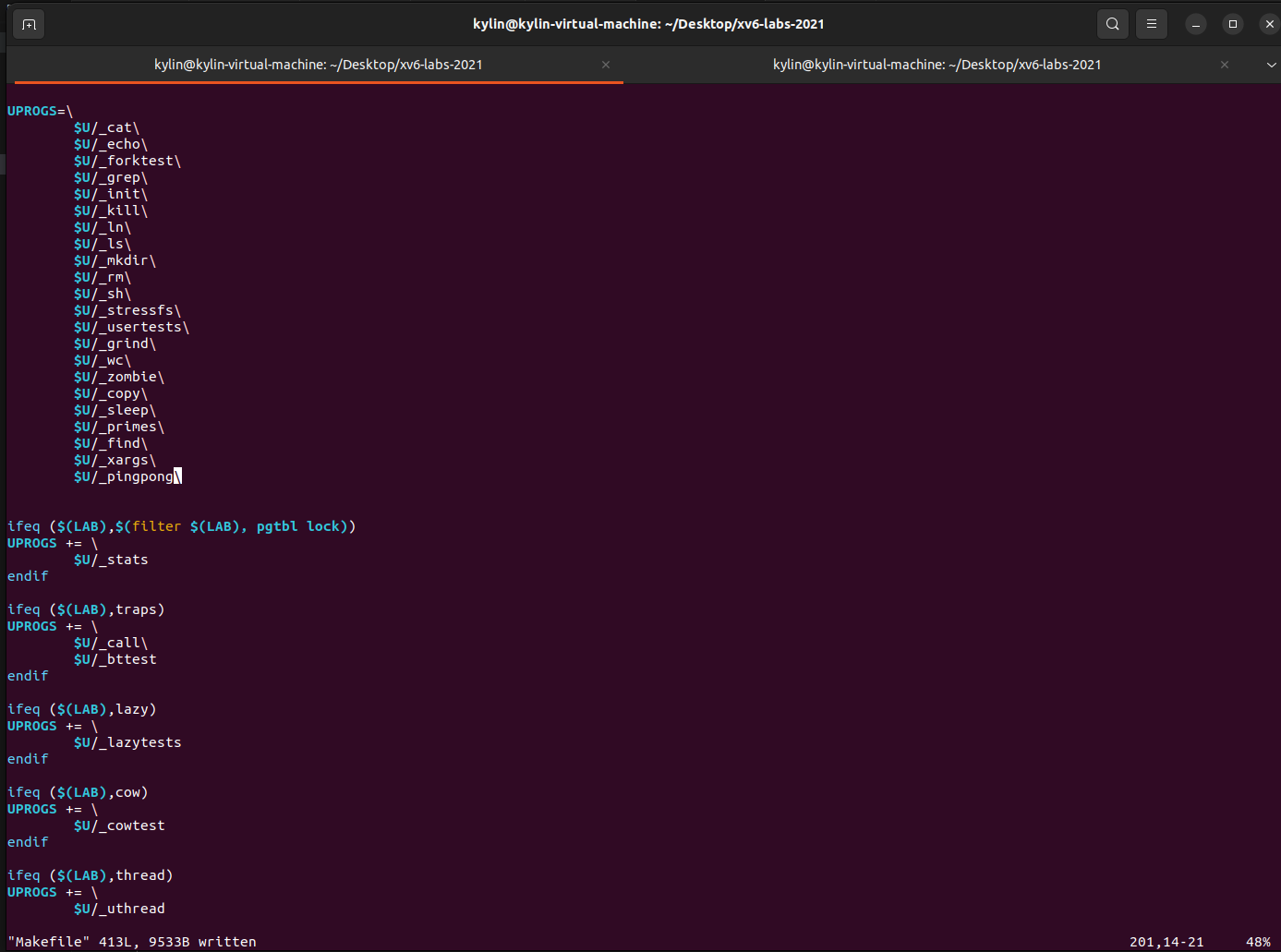
## Find

实验目的

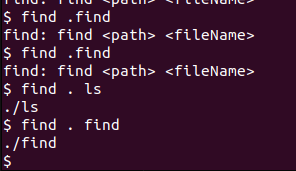
编写⼀个简单版本的UNIX 查找程序：查找具有特定名称的⽬录树中的所有⽂件。

实验步骤

* 切换到util分⽀ ：git checkout util
* 启动qemu ：make gemu
  + 获取参数，利⽤系统调⽤实现pingpong
  + 修改Makefile



* + 编译运⾏



问题和解决方法：

遇到问题：find不能对.目录和..目录进行递归，

解决办法：加入对于strcmp(de.name,".")==0||strcmp(de.name,"..")==0的特殊判断

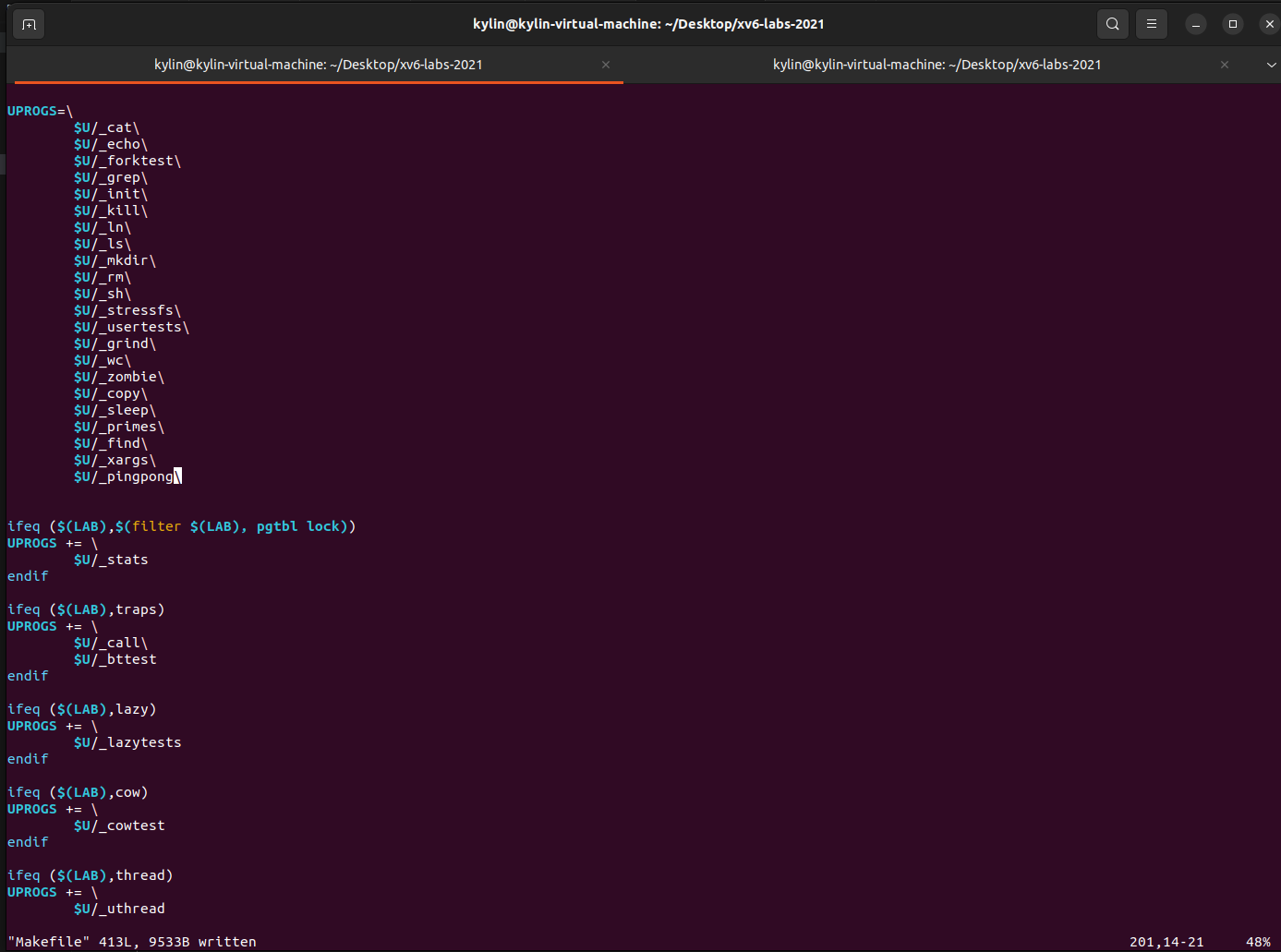
## Xargs

实验目的

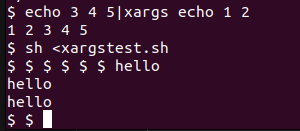
编写⼀个简单版本的UNIX xargs 程序：从标准输⼊中读取⾏并为每⼀⾏运⾏⼀个命 令，将该⾏作为命令的参数提供。

实验步骤

* 切换到util分⽀ ：git checkout util
* 启动qemu ：make gemu
  + 获取参数，利⽤系统调⽤实现pingpong
  + 修改Makefile



* + 编译运⾏



问题和解决方法：

遇到问题：exec由于无需提供argc 参数，会报错

解决办法：将lineSplit以nullptr结尾

## 小结

本实验使我初步了解xv6这⼀操作系统内核的架构和qemu模拟器的使⽤⽅法。在这个实验中， 我学到了通过虚拟机容器搭建环境和运行和系统调⽤接⼝，以及操作系统内核和⽤户区别。

在完成实验的过程中，我遇到了如下困难：

1.对vim使用不熟悉，包括但不仅限于插入修改保存等，后在runoob找到相关教程。

2.无法进行模拟器的编译运行，经查阅资料后才了解。修改makefile时需要用vim（ Linux 系统的文本编辑器）打开，输入i切换成insert（插入）模式，修改后需要esc（退出），并输入:w进行writeback（保存），每次修改后需要重新make qemu（编译运行）

3.不知道管道如何创建、创建完成后如何关闭。查阅相关代码后解决。

# 实验⼆ System calls

## 概述

1. 进⼀步了解系统调⽤。

2. 掌握添加系统调⽤的⽅法。

3. 理解系统调⽤的⼯作原理和 xv6 内核的⼯作过程。

参考教程：

[【MIT-6.S081-2020】Lab2 syscall - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/274369780)

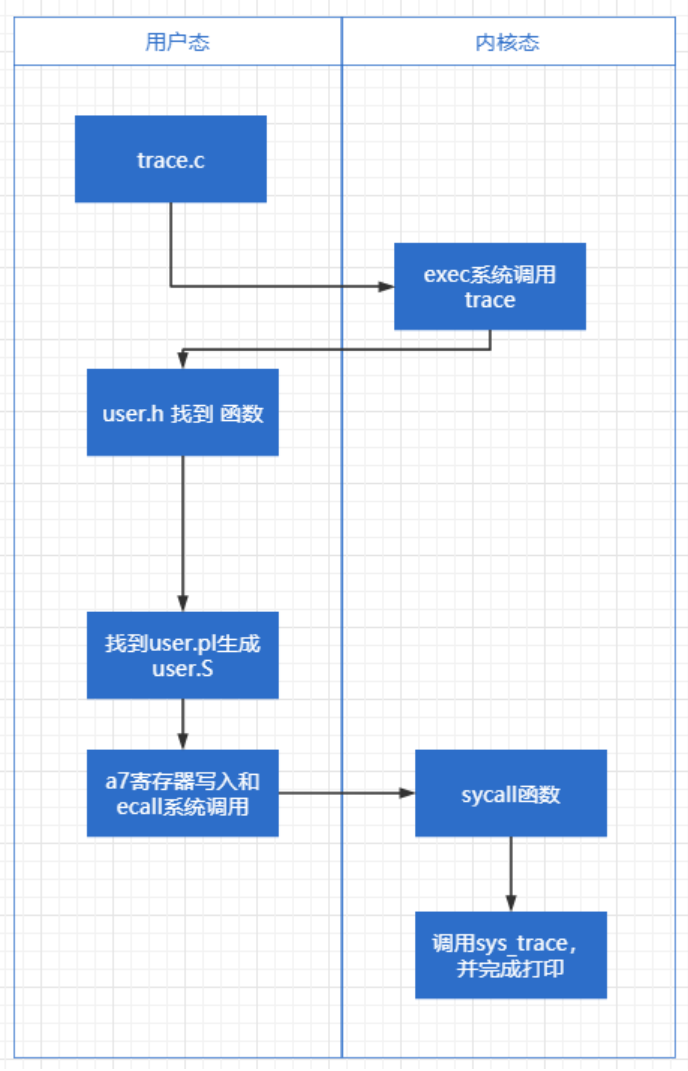
## Trace

实验目的

为 xv6 添加系统调⽤跟踪trace功能。

实验步骤

* 切换到syscall分⽀ ：git checkout syscall
* 启动qemu ：make gemu
  + 添加$U/\_trace到Makefile中的UPROGS变量里
  + 查看已有代码trace.c，发现需要传入一个int型数值（系统调用号），利⽤系统调⽤记录mask实现trace
  + 需要添加声明（系统调用原型）到user/user.h，
  + 添加一个entry（存根）到user/usys.pl
  + 添加一个syscall number（系统调用号）到kernel/syscall.h
  + 添加sys\_trace()到kernel/sysproc.c，查看kernel/sysproc.c中函数都是如何获取参数以及运行的（根据 kill模仿写出trace ，用argint读取参数），修改kernel/proc.h中的proc结构（记录当前进程信息），给proc添加一个mask值用来识别system number，
  + 因为打印的该行应该包含进程id、系统调用的名称和返回值，所以应该在fork中传递参数修改kernel/proc.c中的fork函数，添加子进程复制父进程mask的功能
  + 最后修改kernel/syscall.c，首先添加声明sys\_trace函数，然后添加到syscalls数组中，然后就是修改syscall函数添加trace识别功能

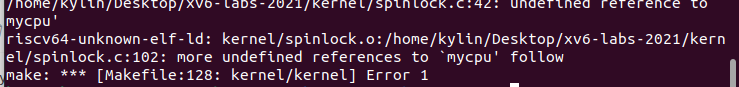


* + 编译运⾏



问题和解决办法

问题：找不到函数定义



原因：(从.cpp源代码到可执行的ELF文件，要经过预处理->编译->链接三个阶段，此时预处理和编译已经通过了)的时候，链接器因链接顺序或编译原因找不到函数定义

解决方法：在makefile中补充sysinfo，生成缺失的-o可执行文件

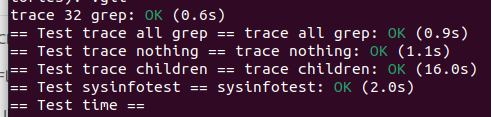
## Sysinfo

实验目的

添加系统调⽤ sysinfo，⽤于收集正在运⾏的系统的信息。

实验步骤

* + 添加$U/\_sysinfotest
  + 需要添加声明（系统调用原型）到user/user.h，
  + 添加一个entry（存根）到user/usys.pl
  + 添加一个syscall number（系统调用号）到kernel/syscall.h，然后加入到syscall函数数组中
  + 为添加sys\_sysinfo函数到kernel/sysproc.c，在kernel/proc.c和kernel/kalloc.c中分别添加函数获取正在使用的进程和可用的内存数，并将添加的函数声明在defs.h
  + 实现sys\_sysinfo函数，这里需要看一下copyout函数的使用方法(可以参考sys\_fstat（kernel / sysfile.c）和filestat（kernel / file.c）)

****

****

问题和解决办法

**问题：**最初未查询kalloc说明文档，不知道如何统计未被使⽤的进程和空间

**解决方法：**查询到kalloc中定义了 kemem，每个链表都指向了⼀个可⽤空间，并保存了最后⼀个链表

## 小结

在实验一中，我们知道进行系统调用的原理是从用户态到内核态然后调用kernel里的sys\_sleep，但对其实际运行非常模糊，经过这一实验，以trace为例，经过系统梳理得知，先由用户输入指令user/trace.c调用trace系统调用，-> markfile调用usys.pl代码通过汇编进入内核 -> 获得跟踪结果输出到屏幕。

最初，由于对进程的定义仅仅停留在课本上，因此对开始实验手足无措。但是通过查阅相关资料，我了解到 xv6 中对进程的定义，以及存储所有 进程的变量和进程状态变量，参考已有函数声明及实现，最终完成实验。

# 实验三 Page Tables

## 概述

1. 了解 RISC-V 中的⻚表机制。

2. 深⼊理解逻辑地址到物理地址的映射关系，掌握如何实现⼀个⻚表映射。

3. 为每个进程实现⼀个⻚表，以更加深刻的认识到⻚表的重要性。

4. 加深对操作系统内存管理的理解。

参考教程：[6.S081 Xv6 Lab 3: page tables\_CDFMLR的博客-CSDN博客](https://blog.csdn.net/u012419550/article/details/114701482)

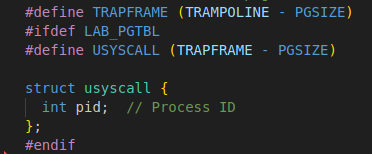
## Speed up system calls

实验目的

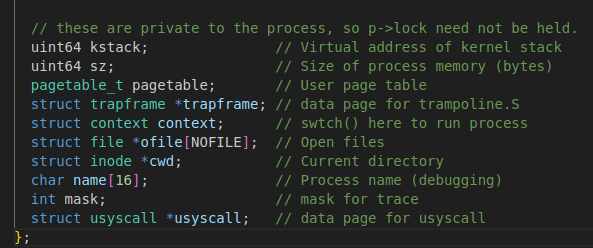
通过在用户空间和内核之间的只读区域共享数据来加速某些系统调用

实验步骤

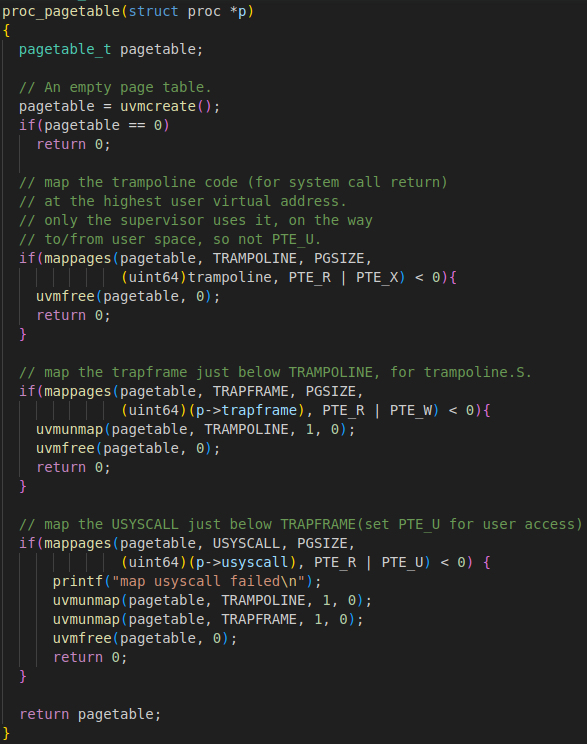
* 注意到在 memlayout.h 中为每一个进程多分配一个虚拟地址位于 USYSCALL 的页，然后这个页的开 头保存一个 usyscall 结构体，结构体中存放这个进程的 pid:



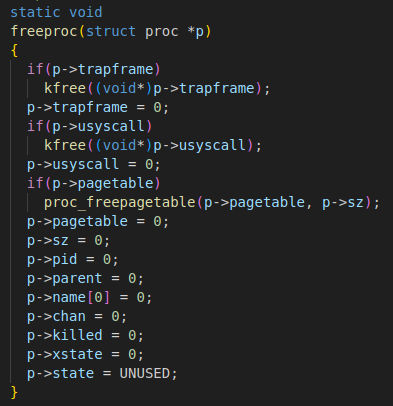
* 这个 USYSCALL 页一定是独立于进程页表的一个页，把定义加到 proc.h



* 然后去 proc.c 中初始化这个页面，allocproc() 中会分配一些页，在这个函数里面分配出需要的 usyscall

* 因为用户态寻址的时候都要经过页表硬件的翻译，所以 usyscall 也要映射在进程的 pagetable 上，刚刚加入 的位置的下面就有 proc\_pagetable(p) 用于 map page，在 proc\_pagetable() 中加入映射逻辑
* 映射完成了之后就要对其进行初始化，回到 allocproc()，
* 在进程回收的时候，这个页面也要一起释放，在 freeproc() 中加入释放即可



* 此外还要在 proc\_freepagetable() 中解除映射，不然启动的时候会报 freewalk panic

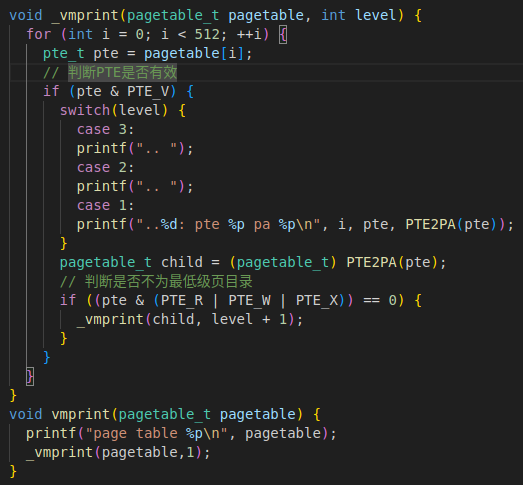
## Print a page table

实验目的

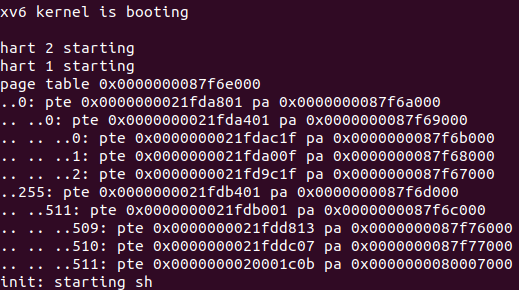
定义函数 vmprint()，接受⼀个 pagetable\_t 参数，功能是打印⻚表结构。

实验步骤

* 将 vmprint() 放在 kernel/vm.c 中，定义放在 kernel/defs.h 中，在 exec.c 中调用
* 在printf 调用中使用%p 来打印像上面示例中的完成的 64 比特的十六进制 PTE 和地址
* 在 exec.c 中的 return argc 之前插入 if(p->pid==1) vmprint(p->pagetable)，以打印第一 22 个进程的页表。

****

实验结果

****

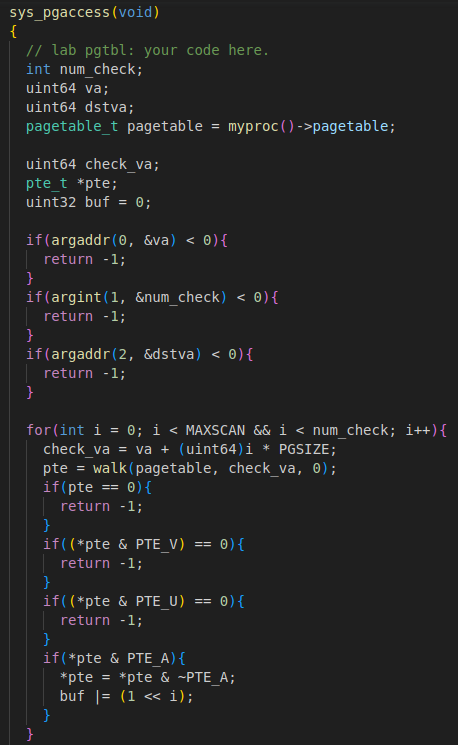
## Detecting which pages have been accessed

实验目的

内核中的 copyin()读取⽤户指针指向的内存，它会将其翻译成物理地址。给每个进 程的内核⻚表都添加⽤户映射，使得 copyin 可以直接使⽤⽤户指针。

实验步骤

* 首先在 kernel/defs.h 中添加 walk 函数的定义
* 在 riscv.h 中定义两个常数
* 在 kernel/sysproc.c 中实现 sys\_pgaccess 函数

****

* 函数具体实现逻辑为：

从参数中提取并验证需要检查的页面的范围。

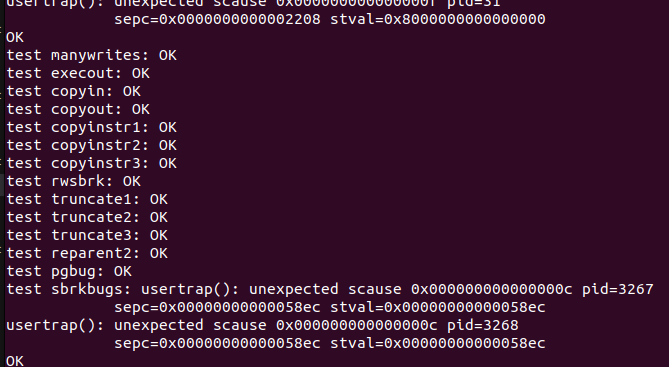
遍历这个范围内的每一个页面。

对于每个页面，查看其在页表中的条目，检查其是否已经被访问。

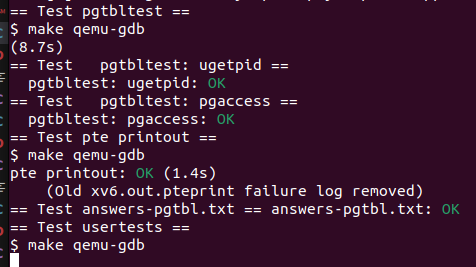
如果一个页面被访问了，则在结果缓冲区中设置相应的位，并清除页表条目的访问位。

最后，将结果缓冲区复制回用户空间。

实验结果



## 小结



本次实验涉及到了操作 系统中⼀个重要的概念，就是⻚表结构。同时，还涉及到了逻辑地址到物理地址之间的 转换和操作系统中地址的定位，较为复杂在完成的过程中还要注意越 界判断等。

# 实验四 traps

## 概述

1. 进⼀步了解系统调⽤所发挥的重要作⽤。

2. 掌握通过中断进⾏系统调⽤的过程及其所发挥的作⽤。

3. 了解简单的汇编语⾔，知道汇编语⾔是如何发挥作⽤的。

4. 理解 xv6 中的堆栈，同时试着实现⼀个⽤户级中断处理。

参考教程：

[6.S081 Xv6 Lab 4 traps\_CDFMLR的博客-CSDN博客](https://blog.csdn.net/u012419550/article/details/115279274)

[Xv6 Lab4：traps(详解) - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/624606664)

## RISC-V assembly

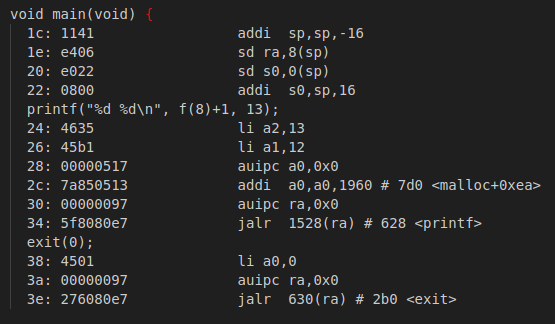
实验目的

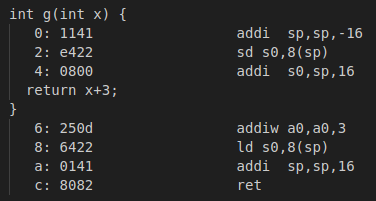
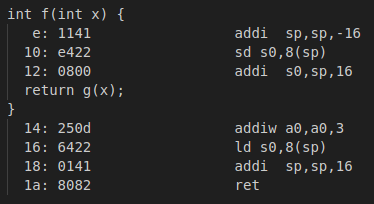
了解 RISC-V 程序集，阅读 call.asm 中的函数 g、f 和 main 的代码。

认识汇编代码，通过gdb调试user/call.c文件，回答相应问题。

实验步骤

* + 执行 make fs.img 编译，在 user/call.asm 中生成可读的汇编版本。
  + 阅读 call.asm 的源代码中函数 g、函 数 f 和 main 函数



问题

1.Which registers contain arguments to functions? For example, which register holds 13 in main's call to printf?

一共有8 个寄存器(a0-a7)

其中

a0：0

a1：12

a2：13

2. Where is the call to function f in the assembly code for main? Where is the call to g? (Hint: the compiler may inline functions.)

对于函数 f 调用是直接计算出了结果 12

G函数被编译器优化为了内联函数，内联在f中在编译期就计算得出了值。

3. At what address is the function printf located?

查看call.asm中printf函数的入口地址，

34: 5f8080e7 jalr 1528(ra) # 628 <printf>

也即跳转到 ra+1528 的位置，此时 ra 的值即为 pc 的值，

4. What value is in the register ra just after the jalr to printf in main?

查询 jalr 指令，我发现：

跳转并链接指令（jal）具有双重功能。若将下一条指令 PC + 4 的地址保存到目标寄存器中，通常是返回地址寄存器 ra，便可以用它来实现过程调用。像分支一样，jal 将其 20 位分支地址乘以 2，进行符号扩展后再添加到 PC 上，便得到了跳转地址

因此，ra 此时存储的应当是 printf 函数的返回地址，也即 0x38

5.运行以下代码。

unsigned int i = 0x00646c72;

printf("H%x Wo%s", 57616, &i);

输出结果为：Hello World。

6. 在下面的代码中，“y=”之后将打印什么 (注：答案不是一个特定的值）？为什么会发生这种情况？

printf ( ”x=%d y=%d” , 3) ; 1

输出的是寄存器 a2 的值，这是因为 printf 会从 a2 寄存器读取参数作为 y 的值。

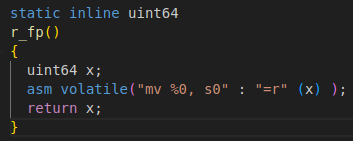
## Backtrace

实验目的

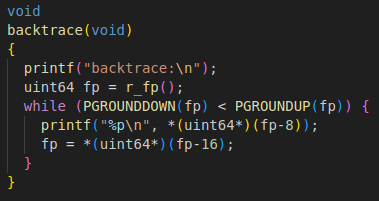
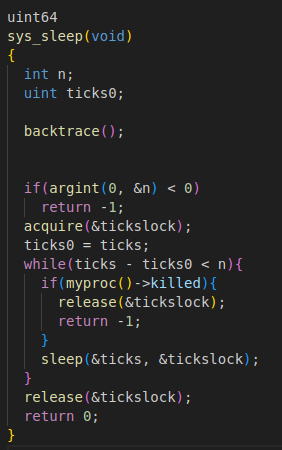
完成 Backtrace 函数，当发⽣错误的时候查看当前堆栈中的系统调⽤。

实验步骤

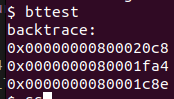
* + 在kernel/riscv.h中实现一个获取当前 frame pointer 的方法

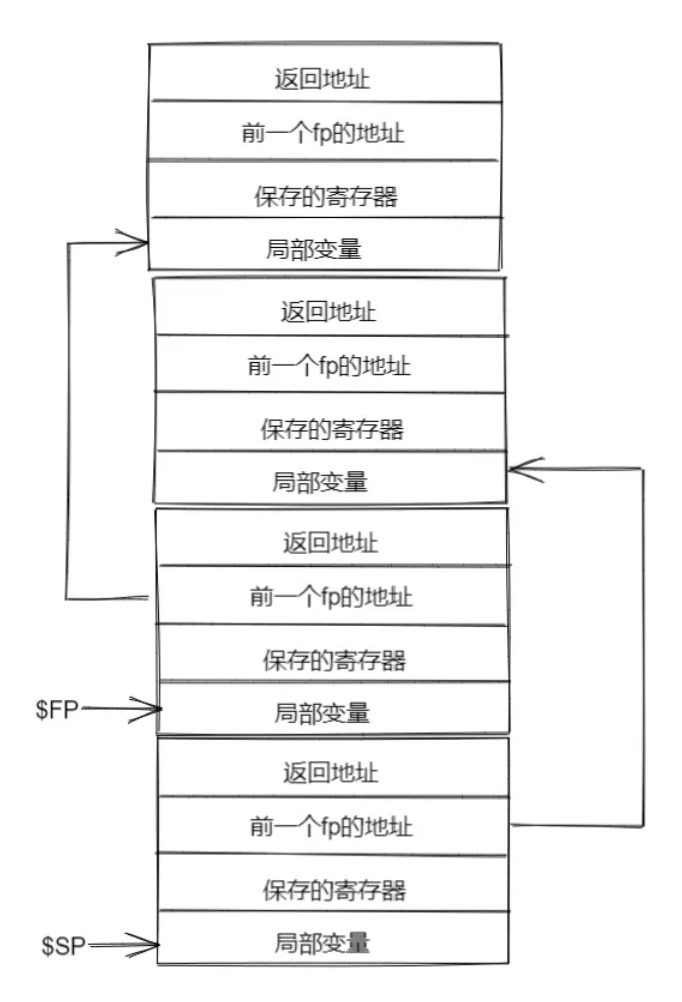


* + 在kernel/printf.c: 打印 backtrace

* + 在def.h中添加backtrace函数声明
  + 在 sys\_sleep 的开头调用一次 backtrace()
  + 编译运行并测试



****

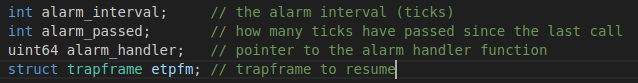
## Alarm

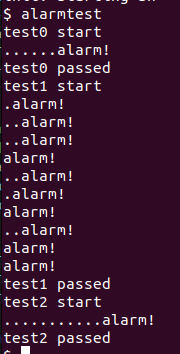
实验目的

添加系统调⽤ sigalarm，当⽤户程序运⾏了 n 个 ticks 后，触发回调函数。

实验步骤

* + kernel/proc.h: 在 proc 结构体里添加需要的字段（间隔时长、已运行时长计数、处理函数地址(vm)、保存 alarm 前的寄存器）



* + 添加 sys\_sigalarm 系统调用：接收参数，为进程设置 alarm 的间隔时长和处理函数。最终的实现在 kernel/sysproc.c
  + 在 usertrap （kernel/trap.c）中处理时钟中断时，如果进程需要 alarm 就保存当前 trapframe 到 etpfm，调用 handler （把 handler 地址放到 trapframe->epc，回到用户空间之后就会运行该函数）
  + 实现 sys\_sigalarm 系统调用，恢复 alarm 前的 trapframe（回到用户空间就会接着 alarm 之前的 PC 开始运行），把 alarm\_passed 计数器置为零（允许下一次 alarm）：
  + 编译运行
  + 

实验问题和解决

问题：sizeof调用错误

解决：把 sizeof(p->trapframe) 改成 sizeof(struct trapframe)。

## 小结

对于调试来说，反向跟踪通常是有用的:在错误发生点上方的栈上的函数调用列表。为了帮助进行回溯，编译器生成机器码，这些机器码在与当前调用链中的每个函数对应的栈上维护一个栈帧。每个栈帧由返回地址和一个指向调用方栈帧的“帧指针”组成。寄存器0包含一个指向当前栈帧的指针(它实际上指向栈上保存的返回地址的地址加上8)。回溯应该使用帧指针来遍历栈，并在每个栈帧中打印保存的返回地址。

# 实验五 Copy on write

## 概述

1. 理解什么是 Copy On Write（COW），其思想和作⽤是什么。

2. 熟悉简单 fork() 系统调⽤的⼯作机制，提出⼀种 COW 的实现⼿段。

3. 理解 xv6 中的 fork 函数，并修改内核程序，实现 COW。

参考教程：[Mit6.S081-实验6-Copy-on-Write Fork for xv6\_解析Ta的博客-CSDN博客](https://blog.csdn.net/u013577996/article/details/111972075?spm=1001.2101.3001.6650.2&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-2-111972075-blog-121418548.235%5Ev38%5Epc_relevant_sort_base3&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-2-111972075-blog-121418548.235%5Ev38%5Epc_relevant_sort_base3&utm_relevant_index=3)

## Implement copy-on write

实验目的

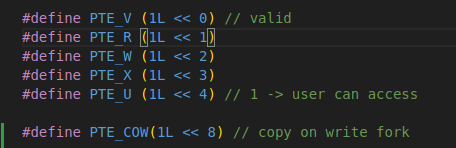
1.了解 Unix 中的 fork() 系统调⽤的功能以及实现机制，重点了解调⽤ fork()过程中的内存分配流程。

2.阅读 xv6 源码，理解 fork 函数以及⽗⼦进程拷⻉内容的机制。

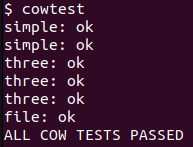
3.根据提示中给出的⽅案啊，修改 xv6 内核使系统⽀持 COW。

实验步骤

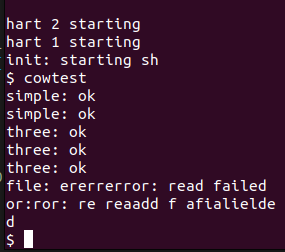
* 在 kernel/riscv.h 中定义 COW 标志位. PTE\_COW.

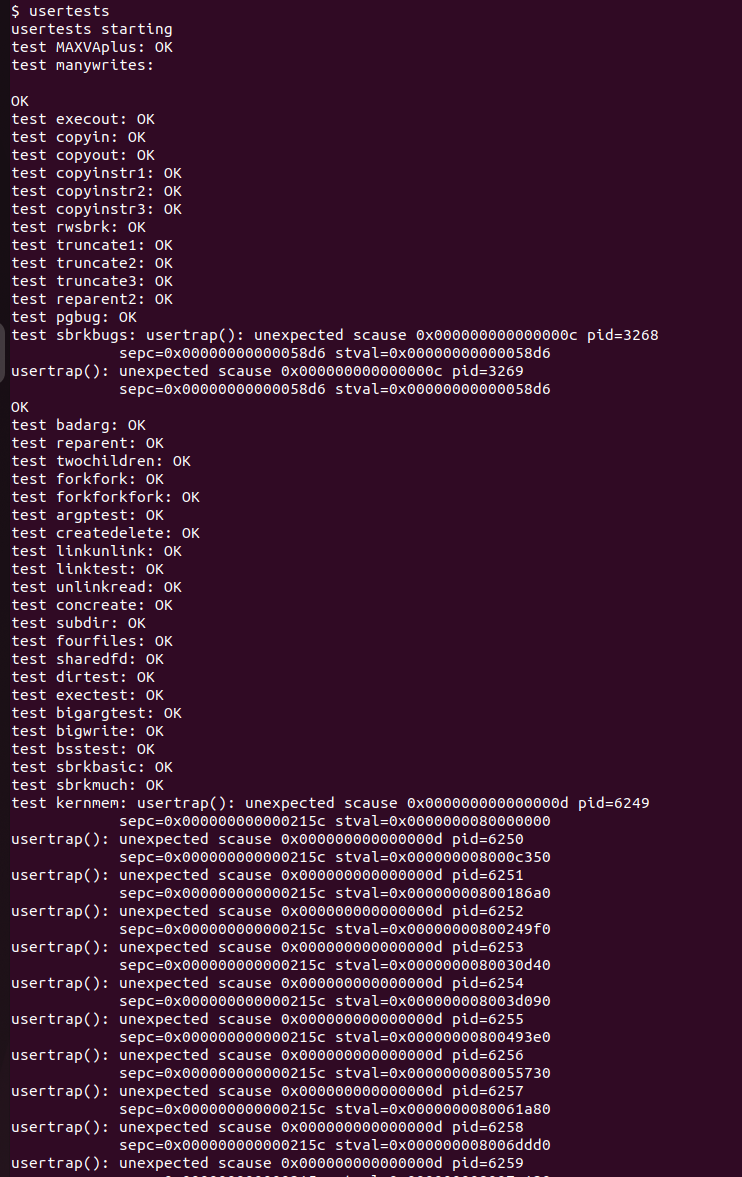


* 修改 uvmcopy() 函数。该函数仅会被 fork() 调用，原本用于将父进程的 pagetable 中所含的所有 page 复制到子进程的 pagetable 中，类似 deepcopy；修改后的 uvmcopy() 函数使用浅拷贝，子进程的 pagetable 结构与父进程完全一致。但同时，我们需要把父进程和子进两者的 pagetable 中的所有 pte 中，PTE\_W 有效的 pte 的这一位清空，具体表现为使用 \*pte &= ~PTE\_W。同时，我们需要使用 pte 中 RSW 中的其中一位，用于标记这个 pte 实现了 cow，这里我选择了使用第 8 位作为标记，定义为：#define PTE\_COWPG (1L << 8) 对 pte 做 \*pte |= PTE\_COW 操作。
* 修改 usertrap() 函数，为其添加 scause 处理入口。由 riscv 文档，我们仅需处理 15 号 code，该 code 代表 Store/AMO page fault，而 13 号代表的 Load page fault 不在我们实现 cow 的处理范围内。在这个 scause 处理分支里，我们需要通过调用 r\_stval() 得知导致 page fault 的虚拟地址，并一定要判断这个地址是否为合法的 cow 地址，若否，将进程标记为 killed 并返回，否则，复制 stval 地址中一个 page 的内容至新跑分配的空间，计算出正确的 pte 目录并重写。
* 这里是一步非常容易出错，即关于每一个 page 的引用计数 (Reference Count)。基本思路是维护一个关于每个 page 的 array，每个 page 对应其在 array 中的索引为该 page 的物理地址除以 PGSIZE(4096)，即 pa / PGSIZE，我们还能粗略计算出这个表的长度为 PHYSTOP >> PGSHIFT。对于：
* 每次 kalloc() 调用，分配新的 page，我们对该 page 所对应的 rc 加 1
* 每次 kfree() 调用，将该 page 对应的 rc 减 1，若此时 rc 为 0，则真正释放这段内存，否则什么也不做。 这里，我们有如下定义：#define RC\_SZ PHYSTOP >> PGSHIFT
* uvmcopy() 中，对每个添加了 PTE\_COWPG 的 pte 条目对应的物理地址的 rc 加 1
* usertrap() 中，在处理 cow 的分支中，对导致 page fault 的虚拟地址所对应的物理地址的 rc 减 1
* 而任何一步对 rc 的读写，需要进行加锁处理，否则会导致条件竞争，这种情况下，为 xv6 添加 CPUS=1 参数能勉强通过测试，但实际测试中无法得到正确结果。
* 编译执行



实验问题和解决





问题：make qemu 启动不了，一直卡在 xv6 kernel is booting

解决：，os 无法启动，问题出现在内存方面。

在os 启动时，要先对内存进行一个初始化，即函数 kinit，该函数主要用来初始化 freelist，调用的函数为 freerange，在追溯下去就是多次调用 kfree。在 kfree 中打印日志，发现一直在调用，死循环。kalloc 会将相应 pm\_ref[idx] 初始化为 1，但是在 kalloc 之前 pm\_ref[] 的初始值并不是 0，因此如果 kfree 只进行 == 0 判断，那很显然 os 启动时一块空闲内存都找不到，改成 <= 即可。

## 小结

这个部分的设计是 6.S081-6缺页异常Page Fault的思想。

当shell执行指令的时候会fork，shell子进程的第一件事就是调用exec，执行我们想要执行的命令。如果fork创建了shell地址空间的完整copy，而exec的第一件事就是丢弃空间。

因此fork之后，我们直接让父进程的物理内存page被共享 —— 让子进程的PTE指向父进程对应的物理内存page。

当我们需要更改内存的时候，就会触发page fault （因为现在正在向一个只读PTE的进行写数据的操作）。

page fault后的处理：

1.page fault之后重新分配一个物理内存page；

2.然后copy page fault对应的物理内存的内容到新建的page；

3.将新分配的page映射到子进程。

注意：这时候新分配的子进程的page和之前父进程的page对应的PTE都设置成可读可写。

# 实验六 Multithreading

## 概述

1. 熟悉多线程编程，理解进程切换⽅式和执⾏顺序。

2. 了解多核多线程对程序执⾏效率的影响。

3. 理解锁的含义，能够判断何时、何处应该加锁。

4. 回顾进程同步/互斥的模式并在程序中应⽤。

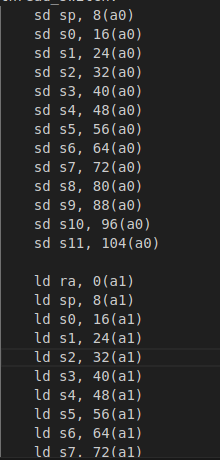
## Uthread

实验目的

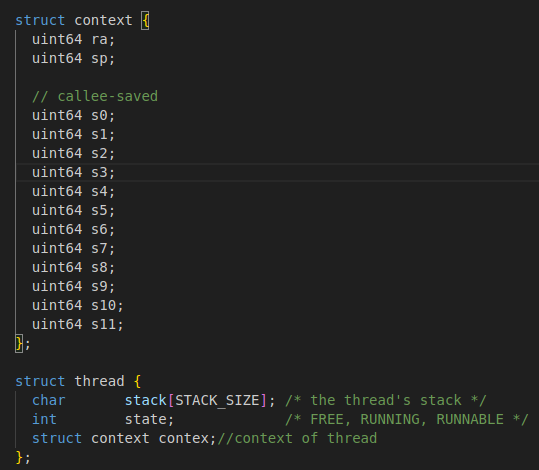
实现⼀个⽤户级进程的创建和切换。

实验步骤

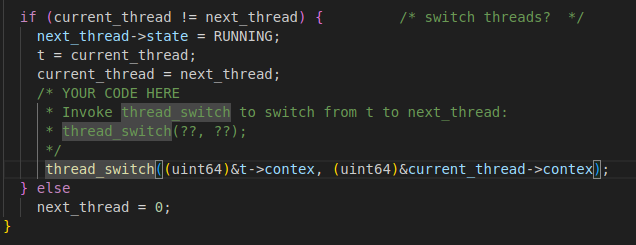
* uthread\_switch.S 中实现上下文切换，这里可以直接参考 swtch.S



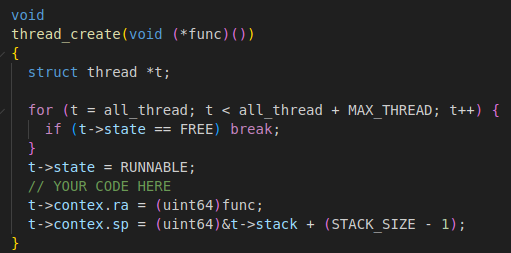
* 定义上下文字段。从 proc.h 中复制context 结构体内容，用于保存 ra、sp 以及 callee-saved registers, 将其放入 user/uthread.c中



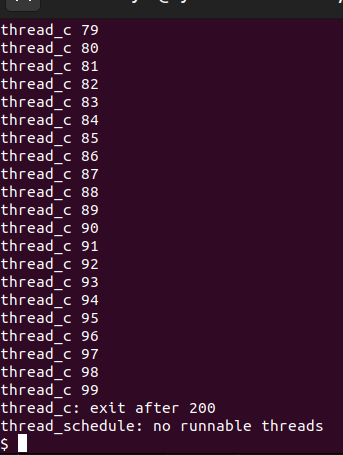
* 在 thread\_schedule 中调用 thread\_switch



* 创建并初始化线程



* 编译运行



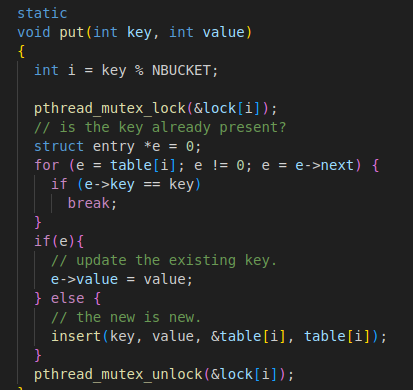
## Using threads

实验目的

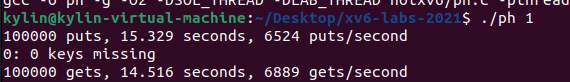
使⽤ UNIX pthread 线程库实现⼀个线程安全的哈希表。

实验步骤

加锁并在main函数中初始化锁







实验问题和解决

问题：如果只加一个锁，锁的粒度很大，会导致丢失性能。

解决：将锁的粒度减小，为每个槽位（bucket）加一个锁。解决：利用加锁操作，解决哈希表 race-condition 导致的数据丢失问题。

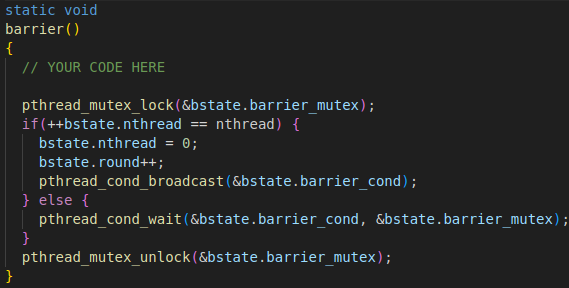
## Barrier

实验目的

实现 barrier 函数。其作⽤是：当进程到达 barrier 函数调⽤时，会开始等待其他进 程，当所有进程都到达 barrier 时，才停⽌等待

实验步骤

生产者消费者模式，如果还有线程没到达，就加入到队列中，等待唤起；如果最后一个线程到达 了，就将轮数加一，然后唤醒所有等待这个条件变量的线程

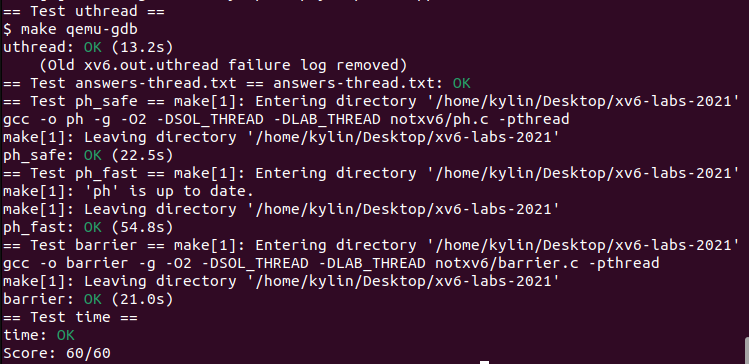


实验问题和解决

问题：「将已进入屏障的线程数量增加 1，然后再判断是否已经达到总线程数」这一步 并不是原子操作，并且这一步和后面的两种情况中的操作「睡眠」和「唤醒」之间也不是原子的

解决方法：「屏障的线程数量增加 1；判断是否已经达到总线程数；进入睡眠」这三步必须原子。所以使用一个互斥锁 barrier\_mutex 来保证原子性。

## 小结



解决键丢失的问题，需要采用上锁的策略。一个简单的思想是每次只允许一个进程对于整个哈希表操作，但是这个想法存在明显缺陷：多进程应该效率更高，如果一次只允许一个进程，那么多进程和单进程的并行性相等，且多进程因为进程切换等操作会浪费更多时间。正确的做法是分别加锁：对于每个哈希筒，每次只允许一个进程访问。barrier实现则需要线程进入同步屏障 barrier 时，将已进入屏障的线程数量增加 1，然后再判断是否已经达到总线程数。 如果未达到，则进入睡眠，等待其他线程。如果已经达到，则唤醒所有在 barrier 中等待的线程，所有线程 继续执行；屏障轮数 + 1；

# 实验七 Network driver

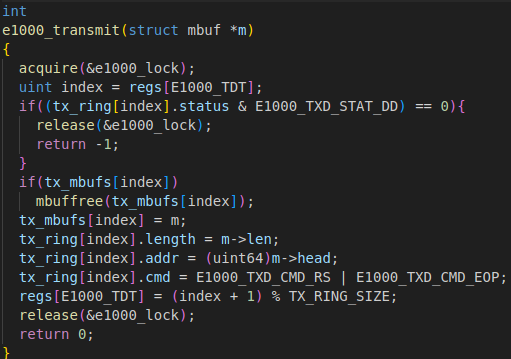
实验目的

熟悉系统驱动与外围设备的交互、内存映射寄存器与 DMA 数据传输

实验步骤

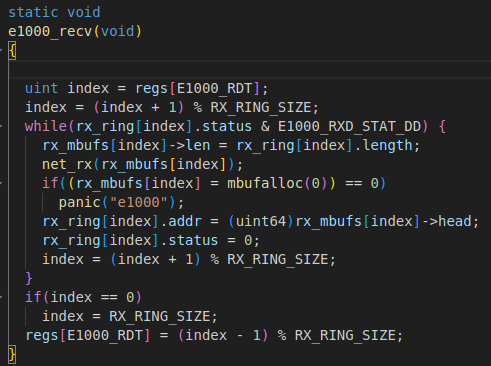
实现 e1000\_transmit

* 首先，通过读取 E1000\_TDT 控制寄存器，向 E1000 询问它期望下一个数据包的 TX 环索引。
* 然后检查环是否溢出。如果未在 E1000\_TDT 索引的描述符中设置 E1000\_TXD\_STAT\_DD ，则E1000 尚未完成相应的上一个传输请求，返回错误。
* 否则，请使用 mbuffree() 释放从该描述符传输的最后一个 mbuf（如果有的话）。
* 然后填写描述符。m->head 指向内存中数据包的内容，m->len 是数据包长度。设置必要的 cmd 标志
* 最后，通过将环位置添加到 E1000\_TDT 模 TX\_RING\_SIZE 来更新环位置。
* 如果 e1000\_transmit() 将 mbuf 成功添加到环中，则返回 0。失败时（例如，没有可用于传输 mbuf的描述符），返回 -1，以便调用方知道释放 mbuf。

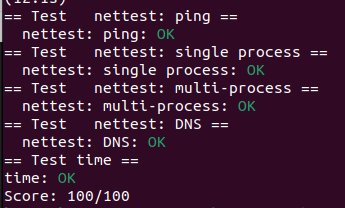


实现 e1000\_recv /

* 首先，通过获取 E1000\_RDT 控制寄存器并添加一个模 RX\_RING\_SIZE ，向 E1000 询问下一个等
* 待接收的数据包（如果有）所在的环索引。
* 然后，通过检查描述符 status 部分中的 E1000\_RXD\_STAT\_DD 位来检查新数据包是否可用。如果没有，请停止。
* 否则，请将 mbuf 的 m->len 更新为描述符中报告的长度。使用 net\_rx() 将 mbuf 传递到网络堆栈。
* 然后使用 mbufalloc() 分配一个新的 mbuf 来替换刚刚给出的 net\_rx() 的 mbuf。将其数据指针（m->head ）编程到描述符中。将描述符的状态位清除为零。
* 最后，将 E1000\_RDT 寄存器更新为最后处理的环描述符的索引。e1000\_init() 使用 mbufs 初始化RX 环



实验结果

****

# 实验八 Locks

## 概述

1. 理解锁竞争，包括锁竞争产⽣的原因以及解决⽅法。

2. 理解通过增加锁的数量来降低锁竞争的原理，并应⽤到程序中。

3. 加深对于使⽤锁来达到进程互斥⽅法的掌握程度。

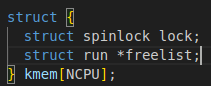
## Memory allocator

实验目的

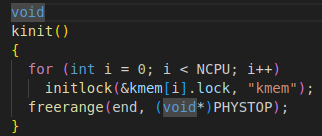
重新设计系统管理内存的⽅式，实现每个 CPU 管理⼀个空闲链表和⼀个锁。

实验步骤

* 根据题意，出现大量竞争情况是因为kalloc()只有一个被单个锁保护的freelist。为了消除锁争用，我们要重新设计内存分配器，以避免单一的锁和列表。基本思想是为每个CPU维护一个空闲列表freelist，每个列表都有自己的锁。然后根据提示，我们需要将其修改为多个，
* 首先更改内存块结构 NCPU 在 kernel/param.h 中



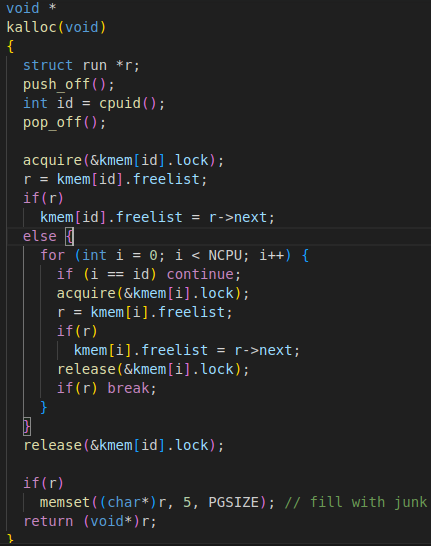
* 将所有空闲内存全部分配给当前 cpu 的 freelist。



* 获取 cpuid 的时候需要关闭中断。

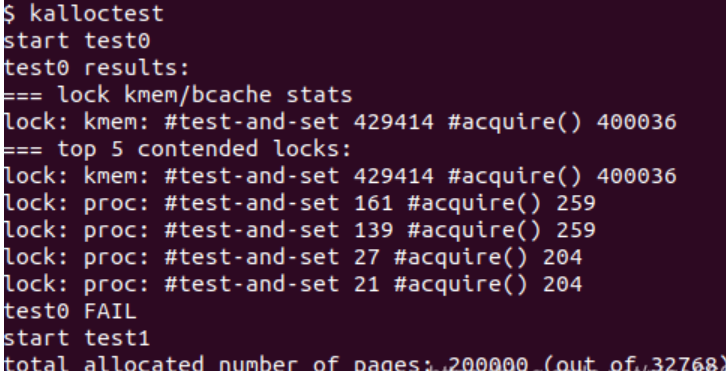


* 如果当前 cpu 有空闲内存块，就直接返回；没有的话，从其他 cpu 对应的 freelist 中“偷” 一块。

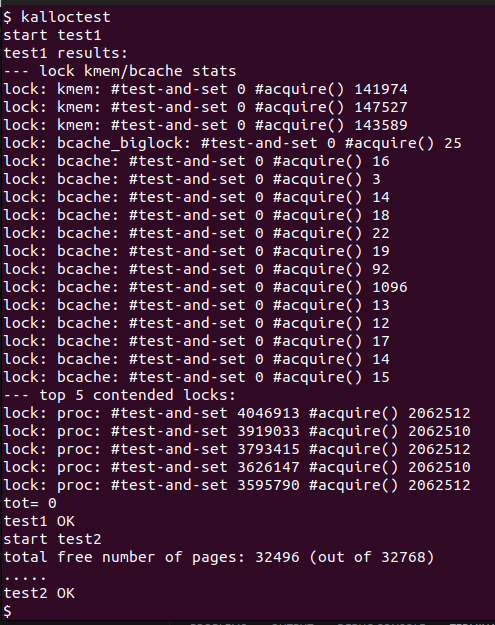


实验结果

改动之前：

****

改动之后：



## Buffer cache

实验目的

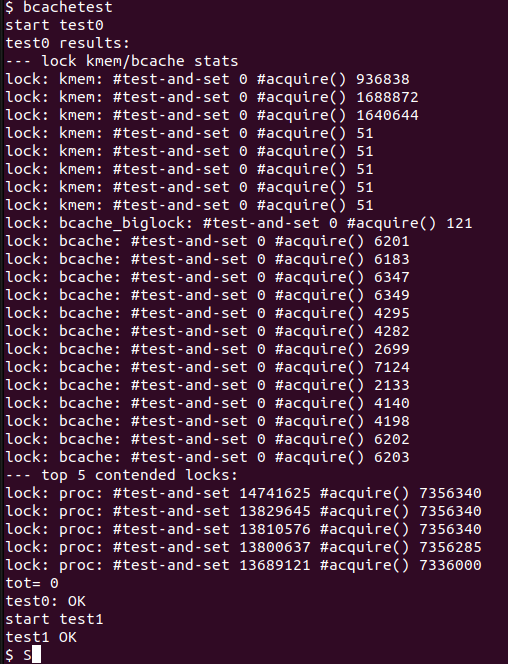
修改系统的 IO 缓冲区，通过散列的⽅式完成对缓冲区和锁的分割。

实验步骤

* 修改结构将 buf 分成 13 份（实验指导书上建议使用的质数），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。

* 初始化后对brelse，bpin 和 bunpin加锁。
* 在bget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。
* 运行结果



实验问题与解决

问题：像任务一那样直接使用 bcache 数组，这会改变 buf 的大小。

解决：拆分buf定义新的数据，数据结构

## 小结

重新设计系统管理内存的⽅式 本实验的⽬的是对每个 CPU 拆分⼀个空闲空间链表并⽤单独的锁管理，从⽽降低锁冲 突带来的额外开销，增加并⾏性。 实验的要点是增加锁和空闲链表的数量并正确地初始化，然后在函数中对不同的 CPU 使⽤不同的锁-链表组合，完成资源和锁的分割。按照要求，⼀个 CPU的空闲链表为空 时，需要向别的 CPU“借”⼀些空间以保证所有空间都能正常分配。

# 实验九 File System

## 概述

1. 理解 UNIX 系统以及 xv6 系统组织⽂件的⽅式，包括 inode 数据结构以及地址的映 射关系。

2. 掌握通过增加索引级数的⽂索引扩展⽅式。

3. 理解⽂件映射和打开的⽅式，了解 xv6 系统的⽂件系统层级架构。

4. 理解硬链接和软链接的实现⽅式。

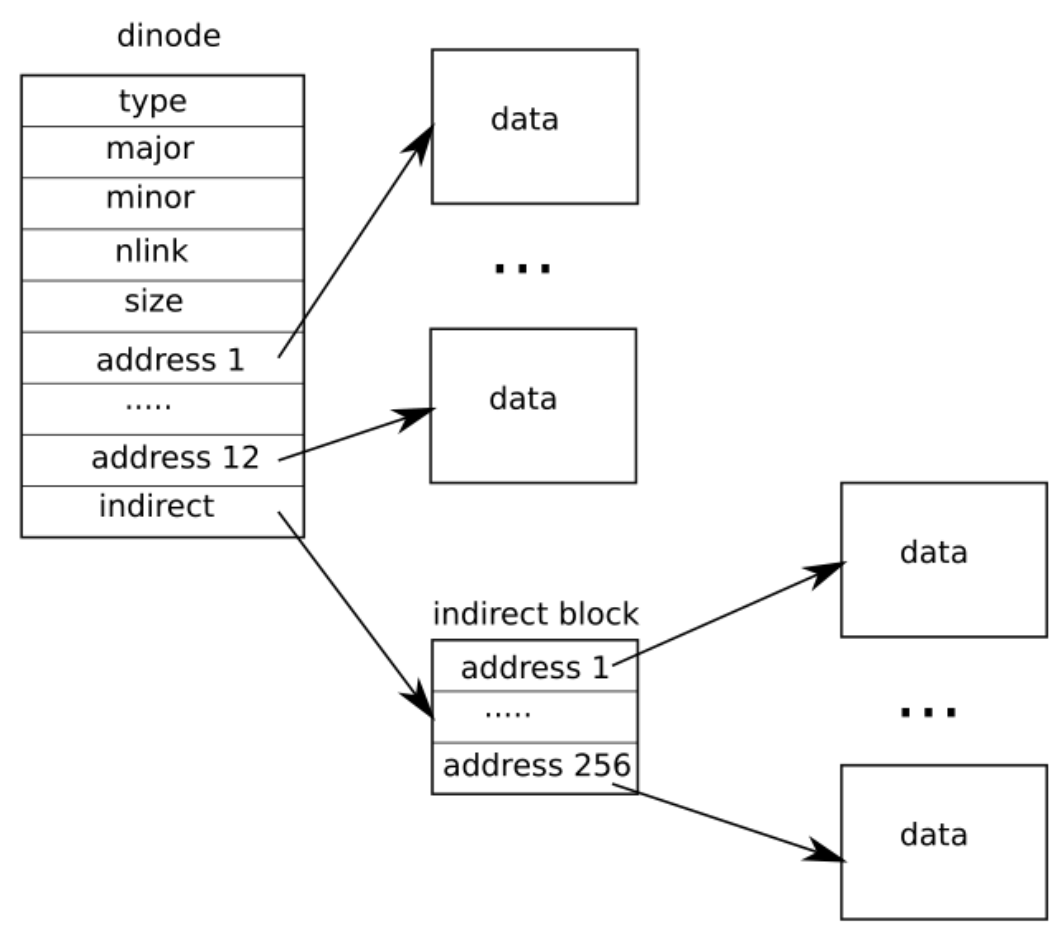
参考教程：[「实验记录」MIT 6.S081 Lab9 file system - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/609840101)

## Large files

实验目的

修改⽂件系统使其⽀持更⼤的⽂件存储。

xv6中的 inode 有 12个直接索引（直接对应了 data 区域的磁盘块），1个一级索引（存放另一个指向 data 区域的索引）。因此，最多支持 12 + 256 = 268 个数据块。如下图所示：



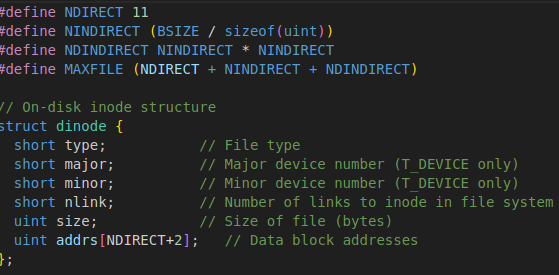
因为这个设计，xv6 中存储文件的大小受到了限制，因此本实验的第一个任务就是通过实现二级索引扩大所支持的文件大小。

具体做法是实现⼆级间接的索引节点 （inode）。

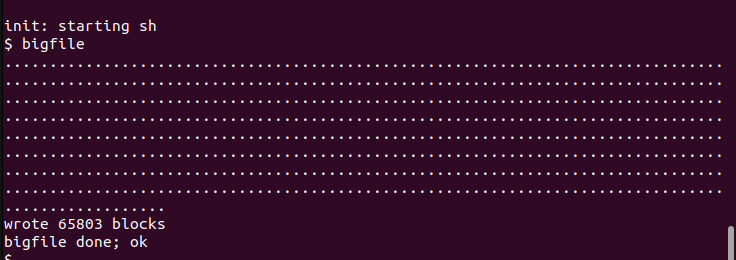
测试程序将创建一个 65803 个磁盘块大小的文件，而未修改前的 xv6 只支持最大 268 个 磁盘块大小的文件。目标是修改文件系统结构和分配磁盘块代码以支持更大的文件

实验步骤

* 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

* 在 kernel/file.h 中更改内存中的 inode 结构
* 实现 bmap 映射：仿照一级索引，写一下二级索引，在 kernel/fs.c 中添加代码
* itrunc 清理：在 kernel/fs.c 中，添加第二级索引的释放操作：

****

## Symbolic links

实验目的

增加⼀个通过符号链接的系统调⽤。将⽬标符号与系统路径链接，使对符号⽂件的 操作都同步到对相应路径⽂件的操作。

实验步骤

* 增加 symlink 函数和系统调用，具体操作和实验二四相同

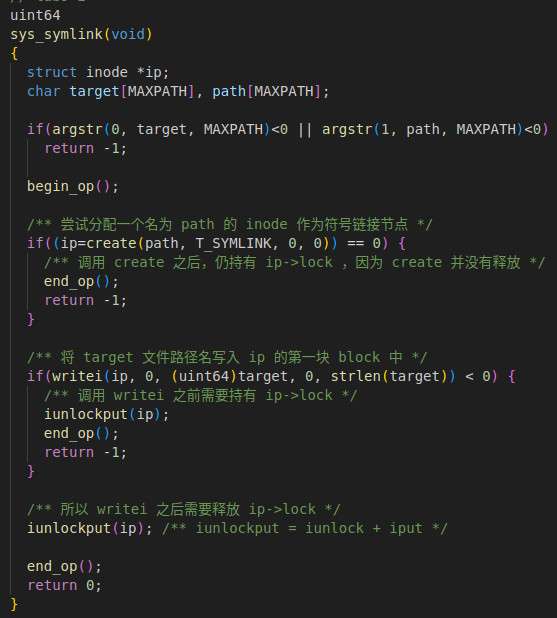
user/usys.pl添加entry("symlink");

user/user.h添加int symlink(const char\*, const char\*);

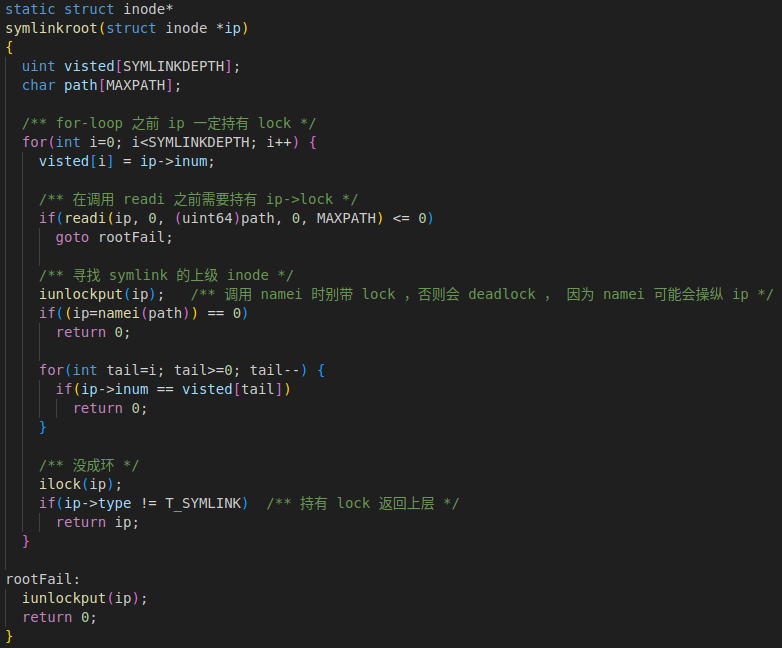
kernel/syscall.h添加#define SYS\_symlink 22

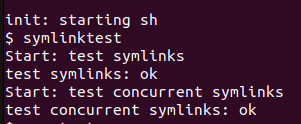
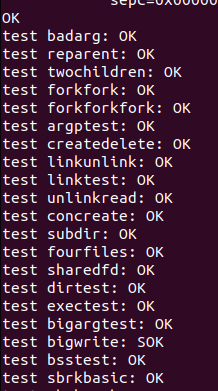
kernel/syscall.c添加extern uint64 sys\_symlink(void);和[SYS\_symlink] sys\_symlink,

symlink的实现则如下：



* knerl/sysfile.c实现系统调用函数，先从寄存器中读取参数，然后开启事务，避免提交出错；为这个符号链接新建一个 inode；在符号链接的 data 中写入被链接的文件；

****

** **

## 小结

第一个实验中的拓展文件大小是一个典型的程序数据结构设计思想，可以看作是对于一个高度为 1 的树，将 其某一些节点拓展成指向相似的节点的非叶子结点，从而扩展这个数的容量（使其高度增加）。这种思想被 运用在可拓展地址数的操作数设计，以及多级页表的设计。

第二个实验是要求添加符号链接系统调用，类比硬链接文件，如 link，首先进行参数获取，然后是文件创 建，文件写入，最后保存退出。重点是如何打开此类文件，这里需要在 open 函数里面稍作修改，使得能够 处理这类文件，这里对于这类文件，实现了一种 follow\_do 的处理函数，用于跟随符号链接的目标路径。

整体做下来较为吃力，这个实验对应的课程里没有讲太多有关实验内容的事，主要还是多看看其他文件系统函数是如何实现的（sys\_open 函数帮助很大，还有 namei, readi, writei 等跟 inode 有关的函数）。

# 实验⼗ Munmap

## 实验目的

1. 了解使⽤虚拟内存的原因和好处。

2. 理解创建和释放⽂件映射的⽅式。

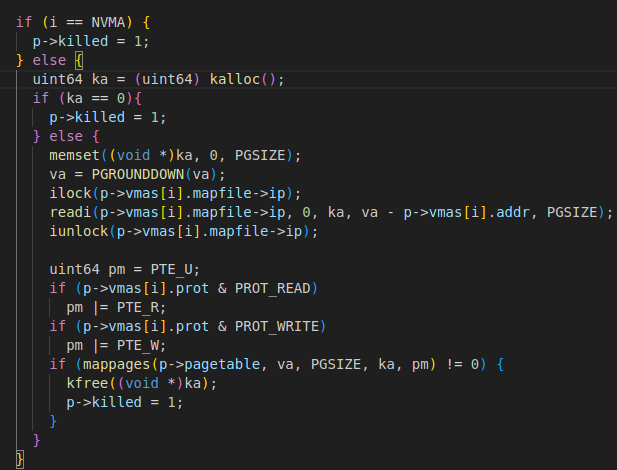
3. 梳理⽂件内存映射的⽣命周期。

## 实验步骤：

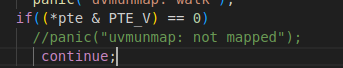
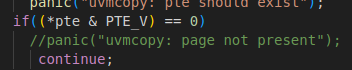
* 添加**系统调用**



* **实现 mmap。**根据传入的函数参数，构建一个 VMA 结构体数组，实验手册中，数组长度建议为 16：然后实现 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。
* **实现 trap 中断处理**。与实验五很相似，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。

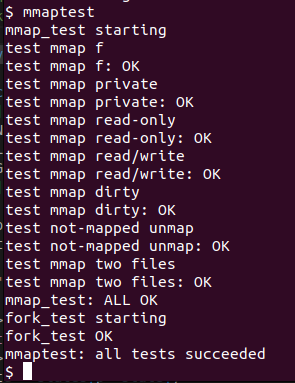
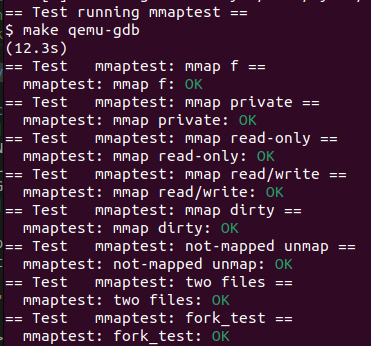


* **实现 munmap**。这部分的逻辑比较简单，主要是取消虚拟地址的映射关系，同时，设置进程 VMA 结 构体相应的 vma 为未使用状态。这部分函数，实验中做了简化，只需要取消地址与传入地址相同的文 件的映射即可。
* 此外，由于一些地址并没有进行映射，因此在 walk 的时候，需要直接跳过（vm.c中）

* 最后，需要处理一些附带的问题：**fork 和 exit 函数**。在进程创建和退出时，需要复制和清空相应的文件映射：

## 实验结果

## 小结

本实验需要添加 mmap 与 mumap 两个系统调⽤。前者将⼀个⽂件内存映射到进程的地址空间，后者取消已有地址空间的映射。理解虚拟地址映射和访问的⽅式，了解 mmap 系统调⽤的实际⼯作流程，然后分步实现mmap 与 mumap并修改相关函数，这样更利于实验的理解和报告的组织。

这个实验让我印象最深刻的地方在于对于页面懒加载的实现以及 vma 数据结构的设计。懒加载这种方 式在实验 5（Lazy allocation）和实验 6（Copy on write）中都有涉及，中心思想是不能一次性加载、拷贝 所有内存，因为大多时候用不到。这种方式的泛用性印证了懒加载思想的重要地位。而在设计 vma 数据结 构时，除了记录地址、长度、权限、文件等描述中提醒的字段之外，我还记录了该内存映射地址是否正在使 用、文件的共享标记这两个必要的字段。提出这两个字段是开始编写 mmap 系统调用时发现实现中有缺漏 或是困难，才发现有字段缺失。