操作系统课程设计项目说明文档

xv6 MIT操作系统实验课程项目(2021版)

姓 名： 秦 成

学 号： 2250397

学院、 专业： 软件学院 软件工程



同济大学

Tongji University

目录

[Lab0 Program execution logic 5](#_Toc172564859)

[Lab1 Xv6 and Unix utilities 7](#_Toc172564860)

[1.1 Boot xv6 (easy) 7](#_Toc172564861)

[1.1.1实验目的 7](#_Toc172564862)

[1.1.2实验步骤 7](#_Toc172564863)

[1.1.3实验心得 7](#_Toc172564864)

[1.2 sleep (easy) 8](#_Toc172564865)

[1.2.1实验目的 8](#_Toc172564866)

[1.2.2实验步骤 8](#_Toc172564867)

[1.2.3问题与解决方案 8](#_Toc172564868)

[1.2.4实验心得 9](#_Toc172564869)

[1.3 pingpong (easy) 9](#_Toc172564870)

[1.3.1实验目的 9](#_Toc172564871)

[1.3.2实验步骤 9](#_Toc172564872)

[1.3.3问题与解决方案 10](#_Toc172564873)

[1.3.4实验心得 10](#_Toc172564874)

[1.4 primes (moderate)/(hard) 10](#_Toc172564875)

[1.4.1实验目的 10](#_Toc172564876)

[1.4.2实验步骤 10](#_Toc172564877)

[1.4.3问题与解决方案 11](#_Toc172564878)

[1.4.4实验心得 11](#_Toc172564879)

[1.5 find (moderate) 11](#_Toc172564880)

[1.5.1实验目的 11](#_Toc172564881)

[1.5.2实验步骤 12](#_Toc172564882)

[1.5.3问题与解决方案 12](#_Toc172564883)

[1.5.4实验心得 12](#_Toc172564884)

[1.6 xargs (moderate) 12](#_Toc172564885)

[1.6.1实验目的 12](#_Toc172564886)

[1.6.2实验步骤 13](#_Toc172564887)

[1.6.3问题与解决方案 13](#_Toc172564888)

[1.6.4实验心得 13](#_Toc172564889)

[1.7 实验小结 13](#_Toc172564890)

[Lab2 System Calls 14](#_Toc172564891)

[2.1 System call tracing (moderate) 14](#_Toc172564893)

[2.1.1实验目的 14](#_Toc172564894)

[2.1.2实验步骤 14](#_Toc172564895)

[2.1.3问题与解决方案 15](#_Toc172564896)

[2.1.4实验心得 16](#_Toc172564897)

[2.2 Sysinfo (moderate) 16](#_Toc172564898)

[2.2.1实验目的 16](#_Toc172564899)

[2.2.2实验步骤 16](#_Toc172564900)

[2.2.3问题与解决方案 17](#_Toc172564901)

[2.2.4实验心得 17](#_Toc172564902)

[2.3 实验小结 18](#_Toc172564903)

[Lab3 Page Tables 19](#_Toc172564904)

[3.1 Speed up system calls (easy) 19](#_Toc172564908)

[3.1.1实验目的 19](#_Toc172564909)

[3.1.2实验步骤 19](#_Toc172564910)

[3.1.3问题与解决方案 20](#_Toc172564911)

[3.1.4实验心得 20](#_Toc172564912)

[3.2 Speed up system calls (easy) 20](#_Toc172564913)

[3.2.1实验目的 20](#_Toc172564914)

[3.2.2实验步骤 20](#_Toc172564915)

[3.2.3问题与解决方案 21](#_Toc172564916)

[3.2.4实验心得 21](#_Toc172564917)

[3.3 实验小结 22](#_Toc172564918)

[Lab4 Traps 23](#_Toc172564919)

[4.1 RISC-V assembly (easy) 23](#_Toc172564921)

[4.1.1实验目的 23](#_Toc172564922)

[4.1.2实验步骤 23](#_Toc172564923)

[4.1.3问题与解决方案 23](#_Toc172564924)

[4.1.4实验心得 24](#_Toc172564925)

[4.2 Backtrace (moderate) 24](#_Toc172564926)

[4.2.1实验目的 24](#_Toc172564927)

[4.2.2实验步骤 24](#_Toc172564928)

[4.2.4实验心得 25](#_Toc172564929)

[4.3 Alarm (hard) 26](#_Toc172564930)

[4.3.1实验目的 26](#_Toc172564931)

[4.3.2涉及函数及作用 26](#_Toc172564932)

[4.3.3实验步骤 26](#_Toc172564933)

[4.3.5实验心得 28](#_Toc172564934)

[4.4 实验小结 29](#_Toc172564935)

[Lab5 Copy-on-Write 30](#_Toc172564936)

[5.1 Implement copy-on write(hard) 30](#_Toc172564938)

[5.1.1实验目的 30](#_Toc172564939)

[5.1.2实验步骤 30](#_Toc172564940)

[5.1.3问题与解决方案 31](#_Toc172564941)

[5.1.4实验心得 31](#_Toc172564942)

[5.2 实验小结 32](#_Toc172564943)

[Lab6 Multithreading 33](#_Toc172564944)

[6.1 Uthread: switching between threads (moderate) 33](#_Toc172564946)

[6.1.1实验目的 33](#_Toc172564947)

[6.1.2实验步骤 33](#_Toc172564948)

[6.1.3问题与解决方案 33](#_Toc172564949)

[6.1.4实验心得 33](#_Toc172564950)

[6.2 Using threads (moderate) 33](#_Toc172564951)

[6.1.1实验目的 33](#_Toc172564952)

[6.1.2实验步骤 33](#_Toc172564953)

[6.1.3问题与解决方案 34](#_Toc172564954)

[6.1.4实验心得 34](#_Toc172564955)

[6.3 Barrier(moderate) 34](#_Toc172564956)

[6.1.1实验目的 34](#_Toc172564957)

[6.1.2实验步骤 34](#_Toc172564958)

[6.1.3问题与解决方案 34](#_Toc172564959)

[6.1.4实验心得 34](#_Toc172564960)

[Lab7 Network Driver 35](#_Toc172564961)

[Lab8 Lock 36](#_Toc172564962)

[Lab9 File System 37](#_Toc172564963)

[Lab10 Mmap 38](#_Toc172564964)

[2.1.1实验目的 39](#_Toc172564965)

[2.1.2实验步骤 39](#_Toc172564966)

[2.1.3问题与解决方案 39](#_Toc172564967)

[2.1.4实验心得 39](#_Toc172564968)

Lab0 Program execution logic

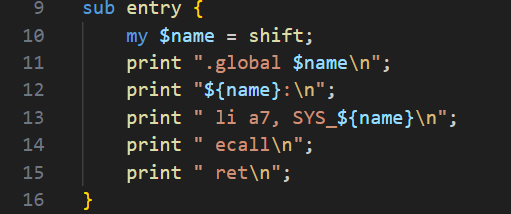
0.1用户态程序发起系统调用

用户态程序调用库函数，如 fork(), read(), write() 等，这些库函数会准备好必要的参数，并发起系统调用。

0.2准备系统调用号和参数

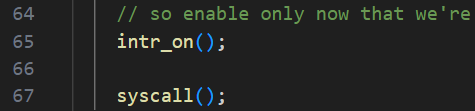
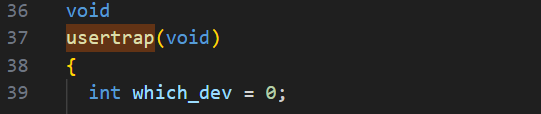
库函数会将系统调用号和参数放入指定的位置。在 xv6 中，系统调用号通常放在寄存器 a7，参数放在寄存器 a0, a1, a2, a3, a4, a5。

0.3触发系统调用指令

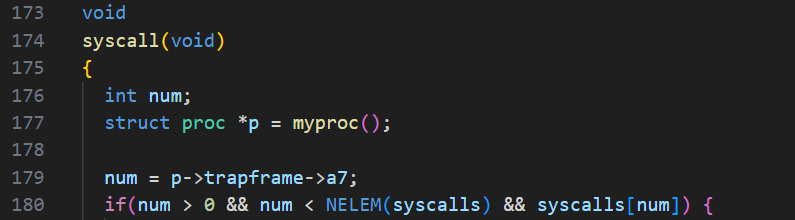
在 xv6 中，触发系统调用的指令是ecall。

0.4内核接收系统调用

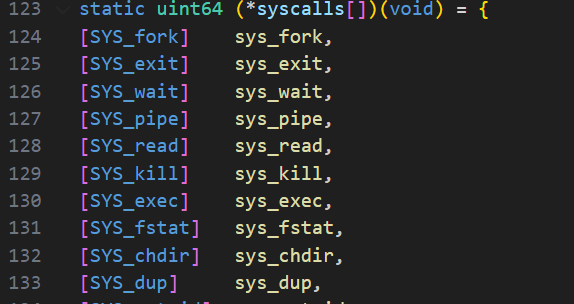
触发ecall指令后，CPU切换到内核态，跳转到内核的系统调用处理入口。在 xv6 中，这通常是 usertrap() 函数。



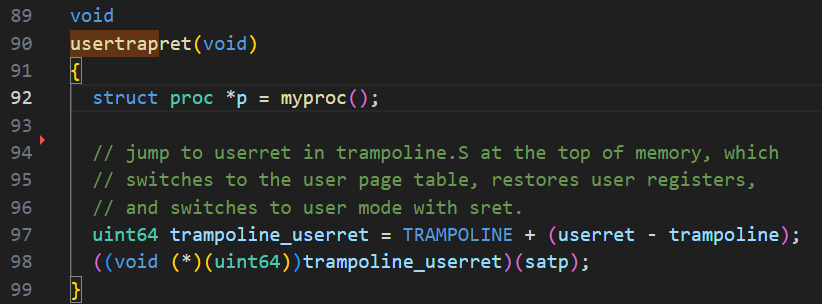
0.5处理系统调用

syscall() 函数是系统调用的核心处理函数。它根据系统调用号调用相应的处理函数。

0.6系统调用处理函数

syscalls 数组包含了所有系统调用处理函数的指针。例如，对于 fork 系统调用，它指向 sys\_fork 函数。

0.7返回用户态

系统调用处理完毕后，内核会将结果放入 a0 寄存器，并准备返回用户态。内核态处理完成后，usertrapret() 函数将恢复用户态的寄存器状态，返回到用户程序继续执行。

Lab1 Xv6 and Unix utilities

* 1. Boot xv6 (easy)

1.1.1实验目的

1、引导学生获取并运行 xv6 操作系统的源代码，并且了解基本的系统调用和文件系统的操作。

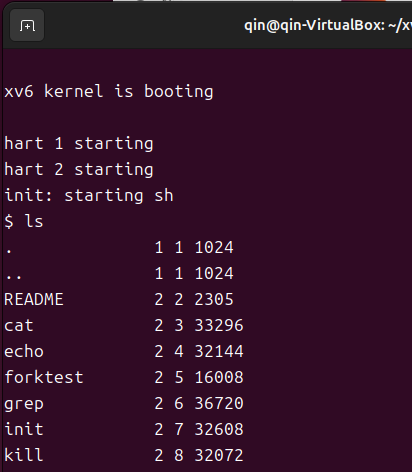
2. 运行xv6系统

1.1.2实验步骤

克隆 xv6 实验的仓库：git clone git://g.csail.mit.edu/xv6-labs-2020。

切换到 util 分支：cd xv6-labs-2020 git checkout util。

查看 Git 日志：git log。

构建和运行 xv6：make qemu。

1.1.3实验心得

通过这个实验，我学习到了以下几点：

Git 的基本操作：如克隆仓库、切换分支和提交更改等。

xv6 操作系统的构建和运行：理解了如何从源码构建一个简单的操作系统，并在模拟器中运行它。

基本的 Unix 命令和文件系统操作：在 xv6 环境中使用 ls 命令查看文件系统结构。

调试和解决构建过程中的问题：通过查找文档和错误信息，解决了在构建和运行 xv6 过程中遇到的问题。

这些技能和知识对于理解操作系统的内部工作原理非常有帮助，并且提供了实际动手操作的机会，从而加深了对操作系统概念的理解。

* 1. sleep (easy)

1.2.1实验目的

1、实现一个 UNIX 程序 sleep，用于 xv6 操作系统。

2、程序应当根据用户指定的时间间隔（以时钟滴答数为单位）暂停执行。时钟滴答是由 xv6 内核定义的时间概念，即两次定时器中断之间的时间。

1.2.2实验步骤

编写 sleep 程序。如果用户忘记传递参数，sleep应该打印一条错误信息。然后，由于命令行参数以字符串形式传递，我们使用 atoi 函数将其转换为整数。使用系统调用 sleep 来实现暂停功能。

添加sleep程序到UPROGS。添加 sleep 程序到 UPROGS。

运行程序并确保它按照预期暂停指定的时间。

1.2.3问题与解决方案

问题：命令行参数处理错误。

解决方案：使用 atoi 函数将字符串转换为整数，并确保正确传递给 sleep 系统调用。

问题：编译错误。

解决方案：检查 Makefile 确保 sleep 程序已正确添加到 UPROGS 中。

1.2.4实验心得

通过这个实验，我学习到了以下几点：

命令行参数处理：了解了如何在 xv6 中获取和处理命令行参数，并进行基本的错误检查。

系统调用的使用：学会了如何调用和使用 xv6 中的系统调用，尤其是 sleep 调用。

编译和调试：熟悉了 xv6 的编译和运行过程，学会了如何添加新的用户程序到 xv6 系统中。

C 语言编程：通过实践进一步巩固了 C 语言编程的基础知识。系统调用的使用：学会了如何调用和使用 xv6 中的系统调用，尤其是 sleep 调用。

* 1. pingpong (easy)

1.3.1实验目的

1、编写一个使用 UNIX 系统调用的程序，通过一对管道在两个进程之间传递一个字节。

2、父进程应向子进程发送一个字节；子进程应打印 "<pid>: received ping"，其中 <pid> 是其进程 ID，然后将字节写回给父进程并退出；父进程应从子进程读取字节，打印 "<pid>: received pong"，然后退出。程序代码放在 user/pingpong.c 文件中。

1.3.2实验步骤

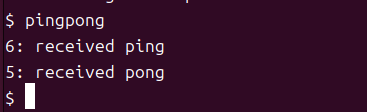
创建管道子进程：使用pipe函数创建一个管道，使用fork函数创建一个子进程。

父进程发送字节：父进程向管道写入一个字节。

子进程处理：子进程从管道读取字节，打印 "<pid>: received ping"，并将字节写回给父进程。

父进程接收字节：父进程从管道读取子进程写回的字节，打印 "<pid>: received pong"，然后退出。

将程序添加到 UPROGS：在Makefile中将pingpong程序添加到 UPROGS。

1.3.3问题与解决方案

问题：管道创建失败。

解决方案：检查 pipe 函数的返回值，确保管道创建成功。如果失败，打印错误信息并退出。

1.3.4实验心得

通过这个实验，我学习到了以下几点：

进程间通信：了解了如何使用管道在父进程和子进程之间进行数据传递。

进程创建与管理：学会了使用fork函数创建子进程，并掌握了父进程和子进程的基本操作。

系统调用使用：熟悉了pipe、fork、read、write和getpid等系统调用的使用方法。

调试技巧：在遇到问题时，通过逐步调试和打印日志，解决了进程间通信失败等问题。

* 1. primes (moderate)/(hard)

1.4.1实验目的

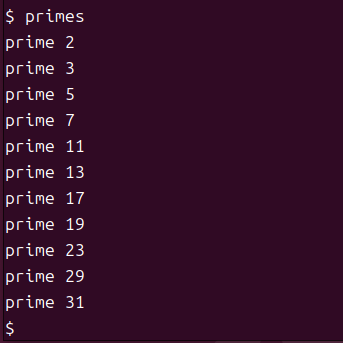
实现一个使用管道的并发质数筛选程序。。目标是使用 pipe 和 fork 创建管道，第一个进程将数字 2 到 35 输入管道。对于每个质数，创建一个进程，从左邻居的管道读取数据，并写入到右邻居的管道。

1.4.2实验步骤

创建第一个进程：创建一个进程，将数字 2 到 35 写入管道。

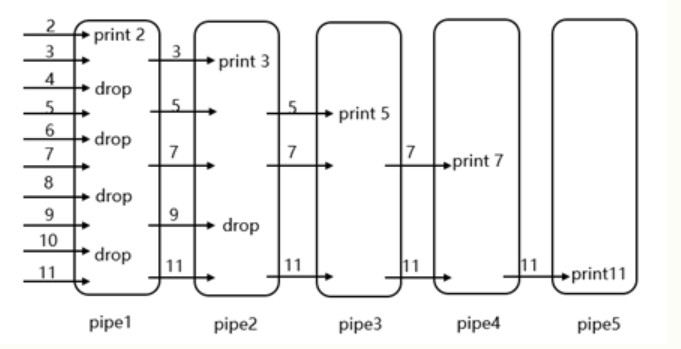
筛选质数：对于每个质数，创建一个进程，读取从左邻居传来的数字，筛选出质数，并将剩余的数字写入右邻居。质数筛原理简单的来说就是从头开始选取未被筛去的数，将它的倍数筛去，不断重复。

等待所有子进程结束：确保主进程在所有子进程完成后才退出。

1.4.3问题与解决方案

问题：子进程未正确结束。

解决方案：在主进程中等待所有子进程结束，确保主进程在所有子进程完成后才退出。

 问题：质数筛实现的原理。解决方案：如下图所示。

1.4.4实验心得

通过这个实验，我学习到了以下几点：

管道和进程创建：掌握了使用pipe和fork函数创建管道和进程的方法。

进程间通信：了解了如何在进程间传递数据，并通过筛选质数来实现并发处理。

资源管理：学会了在并发程序中管理文件描述符和进程资源，避免资源耗尽问题。

* 1. find (moderate)

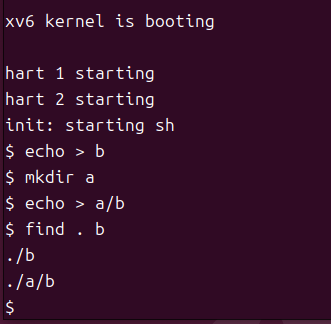
1.5.1实验目的

编写一个简单版本的 UNIX find 程序，遍历目录树中的所有文件，找到具有特定名称的文件。

1.5.2实验步骤

读取目录：使用类似 user/ls.c 的方法读取目录内容。

递归遍历子目录：使用递归方法遍历目录树，查找匹配的文件。同时，确保不递归进入 . 和 .. 目录。

添加到 Makefile：将程序添加到 Makefile 中的 UPROGS，以便编译和运行。

1.5.3问题与解决方案

问题：字符串比较。

解决方案：strcmp函数返回值如果返回值小于 0，则表示 str1 小于 str2。如果返回值大于 0，则表示 str1 大于 str2。如果返回值等于 0，则表示 str1 等于 str2。

1.5.4实验心得

通过这个实验，我学习到了以下几点：

目录读取：了解了如何使用系统调用 open 和 fstat 打开和读取目录内容，并使用 read 函数读取目录项。

递归遍历：掌握了如何使用递归方法遍历目录树，查找匹配的文件名。

字符串处理：学会了使用 strcmp() 函数比较字符串，而不是使用 == 运算符。

* 1. xargs (moderate)

1.6.1实验目的

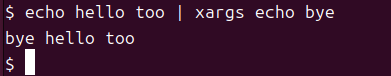
编写一个简单版本的 UNIX xargs 程序，要求从标准输入读取行，并为每一行运行一个命令，将该行作为命令的参数。

1.6.2实验步骤

读取标准输入：从标准输入逐行读取字符，直到遇到换行符（\n）。

创建子进程并执行命令：使用 fork 创建子进程，并在子进程中使用 exec 执行命令。在父进程中使用 wait 等待子进程完成命令执行。

处理命令行参数：将标准输入的每一行作为命令的附加参数传递给指定命令。

添加到 Makefile：将程序添加到 Makefile 中的 UPROGS，以便编译和运行。

1.6.3问题与解决方案

问题：子进程无法正确执行命令。解决方案：认真学习exec的用法，在子进程中使用exec函数执行命令，并检查返回值以确保命令执行成功。

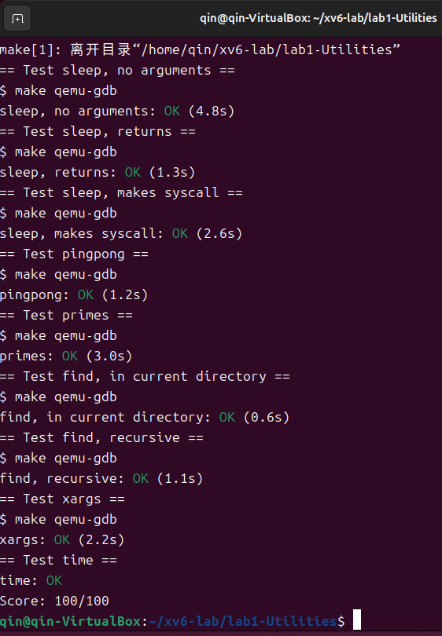
1.6.4实验心得

通过这个实验，我学习到了以下几点：

进程创建与管理：了解了如何使用 fork 创建子进程，并在子进程中使用 exec 执行命令，以及在父进程中使用 wait 等待子进程完成命令执行。

标准输入读取：掌握了如何从标准输入逐字符读取输入，并处理换行符以拆分每一行。

* 1. 实验小结

通过 make grade 进行本实验综合测试：

Lab2 System Calls

1. 1. System call tracing (moderate)

2.1.1实验目的

本实验的目的是向 xv6 内核中添加一个系统调用跟踪功能。该功能允许通过一个新的 trace 系统调用控制跟踪哪些系统调用，当系统调用返回时，在控制位被设置的情况下打印相关信息。具体要求包括：

1、实现一个新的 trace 系统调用，接受一个整数参数 mask，用于指定要跟踪的系统调用。

2、当指定的系统调用执行完成时，打印进程 ID、系统调用名称和返回值。

3、trace 系统调用应该适用于调用它的进程以及它所衍生的所有子进程，但不应影响其他进程。

2.1.2实验步骤

**修改用户空间代码：**

修改Makefile： 将trace程序添加到UPROGS中，确保能够编译用户空间的trace.c。

修改user/user.h：添加新系统调用的声明，例如int trace(int)。

修改user/usys.pl：添加新系统调用的entry(“trace”)，这将生成user/usys.s，包含系统调用的汇编接口。

**修改内核空间代码：**

修改kernel/syscall.h：分配新的系统调用号码，例如#define SYS\_trace 22。

修改kernel/syscall.c：在syscall() 函数中根据系统调用号码分派到相应的处理函数，即添加extern uint64 sys\_trace(int)和调用[SYS\_trace] sys\_trace。

在适当的位置添加跟踪信息的打印功能，这里可以使用一个预定义的系统调用名称数组来获取系统调用的名称。

修改kernel/sysproc.c：实现新的系统调用处理函数sys\_trace()。在sys\_trace() 中，获取并保存跟踪掩码到进程控制块（proc结构）中。

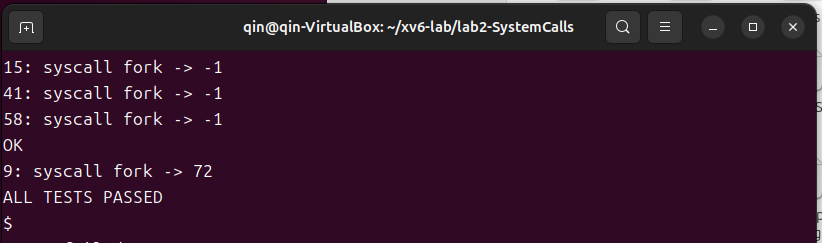
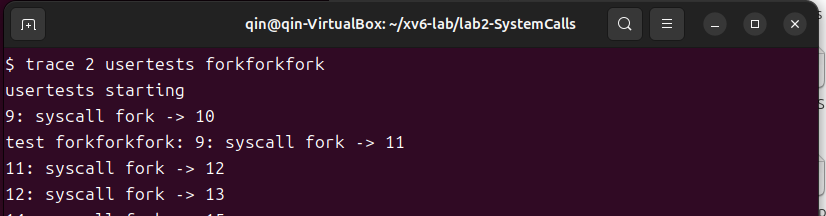
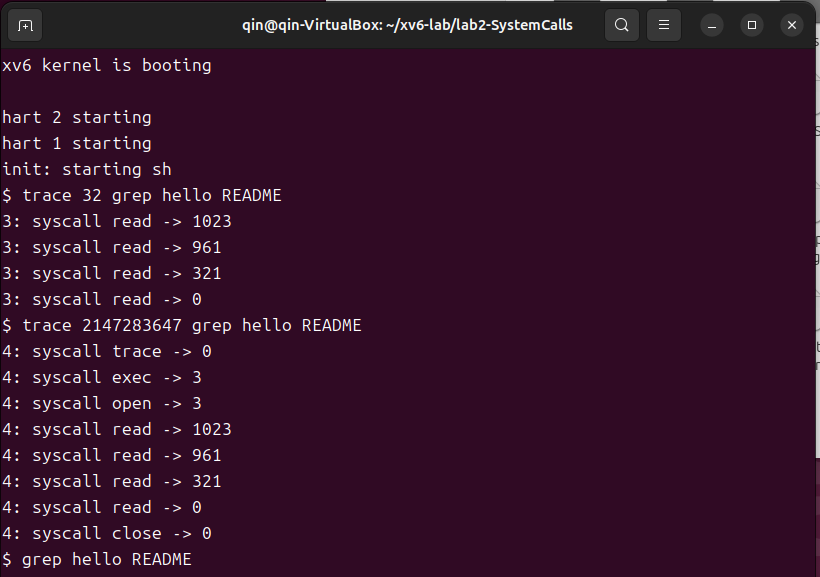
修改kernel/proc.c：修改fork()函数，确保在创建子进程时将父进程的跟踪掩码复制到子进程中，以实现跟踪信息在进程间的继承，即np->trace\_mask=p->trace\_mask;。

**编译和测试：**

使用make命令编译修改后的xv6内核和用户空间程序。

运行make qemu启动xv6系统，并通过trace程序测试新的系统调用跟踪功能。

在trace 程序中调用新的系统调用，例如trace 32 grep hello README，确保系统调用跟踪输出符合预期。



2.1.3问题与解决方案

问题1：如何添加新的系统调用？

解决方案：首先，在用户空间的user/user.h 中添加新系统调用的声明。然后，在user/usys.pl中添加新系统调用的stub。在内核空间的kernel/syscall.h中分配新的系统调用号码。最后，编写实际的系统调用处理函数，放置在kernel/sysproc.c中的sys\_trace()中。

问题2：如何在系统调用返回时打印跟踪信息？

解决方案：修改 kernel/syscall.c中的syscall()函数，在适当的地方添加跟踪信息的打印。可以利用系统调用号码查找系统调用的名称，这些名称可以定义在一个数组中，以便根据系统调用号码快速获取名称。

问题3：如何确保跟踪信息正确地传播到子进程？

解决方案：在kernel/proc.c 中的fork()函数中，确保在创建子进程时将父进程的跟踪掩码复制到子进程中，以便子进程继承父进程的跟踪设置。

2.1.4实验心得

通过这个实验，我学习到了以下内容：

本实验涵盖了系统调用的实现、进程管理以及内核与用户空间之间的交互。通过实现系统调用跟踪功能，不仅加深了对操作系统内核的理解，还学习了如何在内核中实现新的系统调用以及如何确保这些功能能够正确地在进程之间传播和继承。

* 1. Sysinfo (moderate)

2.2.1实验目的

添加一个 sysinfo 系统调用，用于收集系统运行信息，包括空闲内存和非 UNUSED 状态的进程数量。验证测试程序 sysinfotest 的运行结果是否为 "sysinfotest: OK"。

2.2.2实验步骤

**修改用户空间代码：**

修改Makefile：将\_sysinfotest添加到UPROGS中，以确保可以编译用户空间的 sysinfotest.c。

修改user/user.h：添加新系统调用的声明，并预先声明struct sysinfo的存在。

修改user/usys.pl：添加新系统调用的 entry("sysinfo")，这将生成 user/usys.s，包含系统调用的汇编接口。

**修改内核空间代码：**

修改 kernel/syscall.h：分配新的系统调用号码，例如 #define SYS\_sysinfo 23。

修改 kernel/syscall.c：在 syscall() 函数中根据系统调用号码分派到相应的处理函数，即添加 extern uint64 sys\_sysinfo(void); 并在 syscall() 中添加调用 [SYS\_sysinfo] sys\_sysinfo。

修改 kernel/sysproc.c：实现新的系统调用处理函数 sys\_sysinfo()。在 sys\_sysinfo() 中，使用 copyout() 将 sysinfo 结构复制回用户空间。

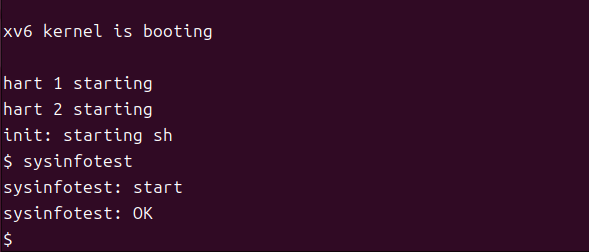
修改 kernel/kalloc.c：添加一个函数uint64 kfreemem(void)来收集可用内存的字节数。

修改 kernel/proc.c：添加一个函数int nproc(void)来收集状态不是 UNUSED 的进程数。

**编译和测试：**

使用 make 命令编译修改后的 xv6 内核和用户空间程序。

运行 make qemu 启动 xv6 系统，并通过 sysinfotest 程序测试新的系统调用功能。

在 sysinfotest 程序中调用新的系统调用，确保系统调用输出符合预期。

2.2.3问题与解决方案

问题：在sys\_sysinfo实现中，无法正确获取用户空间传递的指针。

解决方案：使用argaddr函数正确获取指针，并使用copyout函数将数据从内核空间复制到用户空间。

2.2.4实验心得

通过这个实验，我学习到了以下内容：

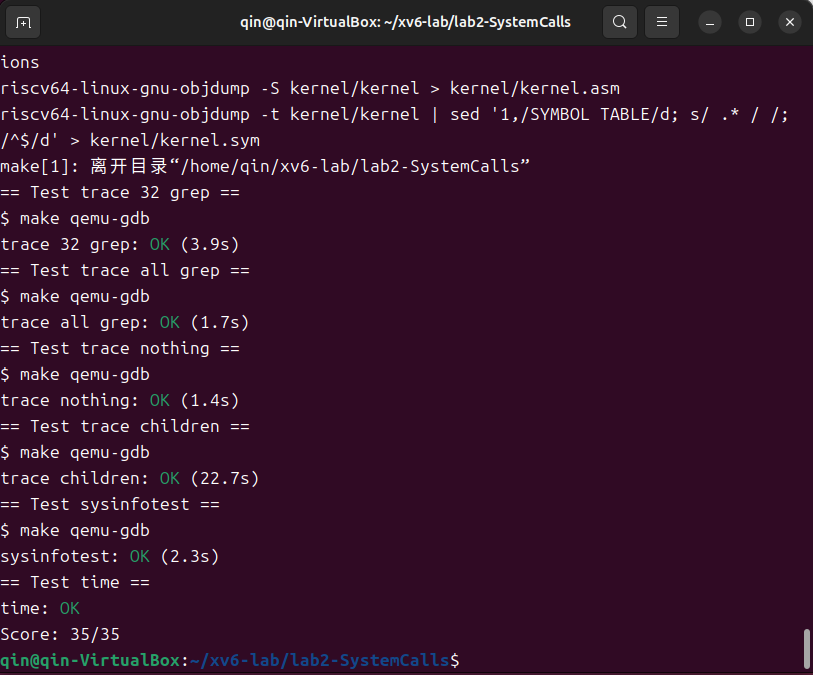
系统调用的实现：了解了如何在 xv6 内核中添加和实现新的系统调用。

数据传递：掌握了如何在内核和用户空间之间传递数据，特别是结构体。

内存管理和进程管理：学习了如何收集系统的内存使用情况和进程状态信息。

本实验涵盖了系统调用的实现、内核与用户空间之间的数据传递以及内存和进程管理。通过实现 sysinfo 系统调用，不仅加深了对操作系统内核的理解，还学习了如何收集和处理系统运行时信息。

* 1. 实验小结

通过 make grade 进行本实验综合测试：

Lab3 Page Tables

3. 1. Speed up system calls (easy)

3.1.1实验目的

本实验的目的是通过在用户空间和内核之间共享数据来加速某些系统调用。具体来说，我们将优化 xv6 的 getpid() 系统调用，避免每次调用时都需要内核切换。实现这个优化的方法是为每个进程在 USYSCALL（在 memlayout.h 中定义）处映射一个只读页面，并在该页面的起始位置存储一个包含当前进程 PID 的 struct usyscall。我们提供了用户空间的 ugetpid() 函数，它会自动使用 USYSCALL 映射。通过这个实验，可以学习如何将映射插入页表。

3.1.2实验步骤

**修改内核空间代码：**

修改 kernel/proc.c：在 proc\_pagetable() 函数中，为每个进程映射一个只读页面到 USYSCALL 地址。使用 mappages() 函数来执行实际的映射。

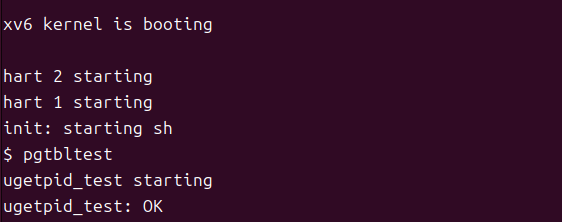
修改 kernel/proc.c：在 allocproc() 函数中初始化 struct usyscall 以存储当前进程的 PID。

修改 kernel/proc.c：在 freeproc() 函数中释放 struct usyscall 所占用的内存。

**编译和测试：**

使用 make 命令编译修改后的 xv6 内核和用户空间程序。

运行 make qemu 启动 xv6 系统，并通过 pgtbltest 程序测试新的系统调用功能。

在 pgtbltest 程序中调用 ugetpid，确保系统调用输出符合预期。

3.1.3问题与解决方案

问题1：如何在用户空间和内核之间共享数据？

解决方案：通过在 USYSCALL 处映射一个只读页面，并在该页面的起始位置存储 struct usyscall。

问题2：如何确保只读页面的正确权限？

解决方案：在 mappages() 函数中设置适当的权限位（PTE\_R | PTE\_U）以确保页面是只读的且用户空间可以访问。

3.1.4实验心得

通过这个实验，我学习到了以下内容：

页表管理：了解了如何在 xv6 内核中管理和修改页表。

数据共享：掌握了如何在用户空间和内核之间共享只读数据，以避免不必要的内核切换。

内存管理：学习了如何为进程分配和释放内存，特别是只读页面的管理。

本实验涵盖了页表管理、数据共享以及内存管理。通过实现 getpid() 系统调用的优化，不仅加深了对操作系统内核的理解，还学习了如何通过共享只读页面来提高系统调用的效率。其它系统调用，如 getppid() 和 getuid() 也可以通过类似的共享页面来加速，方法是将这些系统调用的信息也存储在 struct usyscall 中，并在进程创建时初始化相应的字段。

* 1. Speed up system calls (easy)

3.2.1实验目的

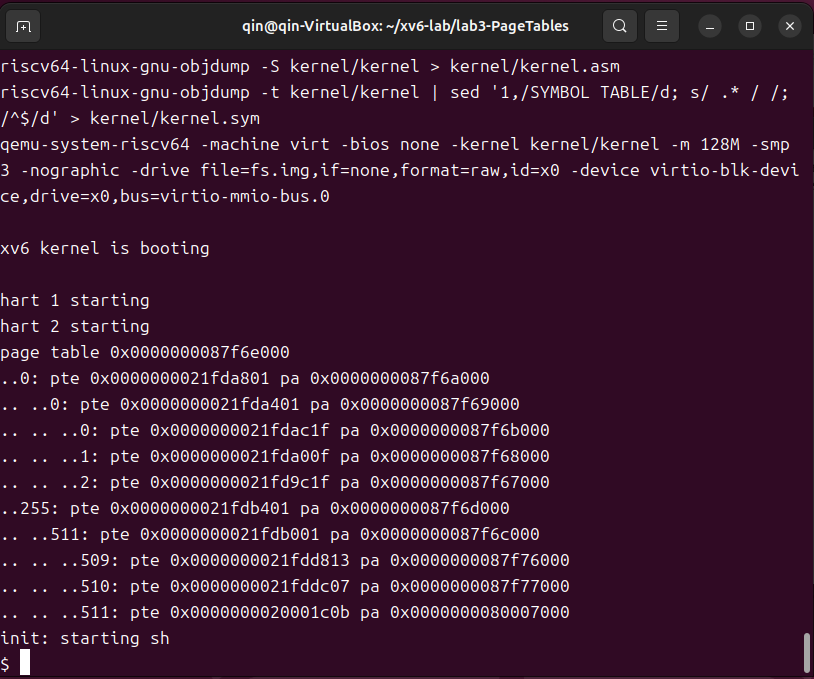
本实验的目的是实现一个函数 vmprint()，用于打印 RISC-V 页表的内容。通过这个函数，可以帮助我们可视化页表结构，便于调试和理解页表的工作原理。

3.2.2实验步骤

定义 vmprint() 函数：vmprint() 函数接受一个 pagetable\_t 类型的参数，并按照指定的格式打印该页表的内容。

在 exec.c 中插入 vmprint() 调用：在 exec() 函数中，找到适当的位置调用 vmprint() 函数。我们选择在返回 argc 之前调用它。

添加 vmprint() 的声明：在 kernel/defs.h 中添加 vmprint() 函数的声明，以便在 exec.c 中调用它。

编译和测试：使用 make 命令编译修改后的 xv6 内核和用户空间程序。运行 make qemu 启动 xv6 系统，并观察输出。

3.2.3问题与解决方案

实验提出的问题：

页面 0：包含第一级页表条目。

页面 2：包含第三级页表条目。

用户模式下，进程能否读取/写入页面 1 映射的内存？ 不可以，因为该页面仅在页表中描述，没有映射到实际的物理内存。

倒数第三个页面：包含第三级页表条目，指向实际的物理内存地址。

问题：如何遍历并打印页表？

解决方案：通过DFS遍历页表的每一层，检查每个页表项 (PTE) 是否有效 (PTE\_V)。如果有效，打印页表项的索引、PTE 值和物理地址 (PA)。如果 PTE 指向下一层页表，则递归调用 vmprint () 函数继续打印。

3.2.4实验心得

通过这个实验，我学习到了以下内容：

页表结构与遍历：通过实现 vmprint() 函数，我更深入地理解了 RISC-V 的页表结构以及如何递归遍历页表以打印其内容。

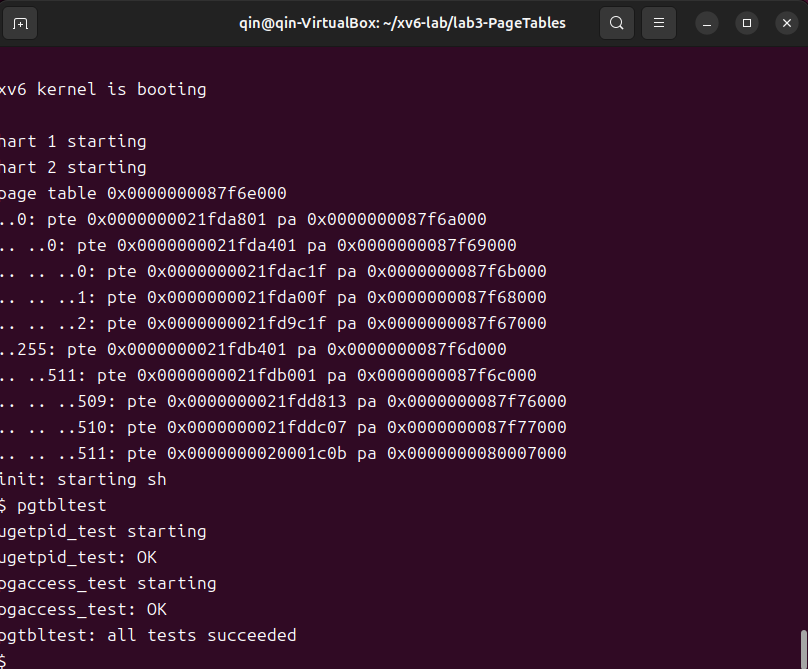
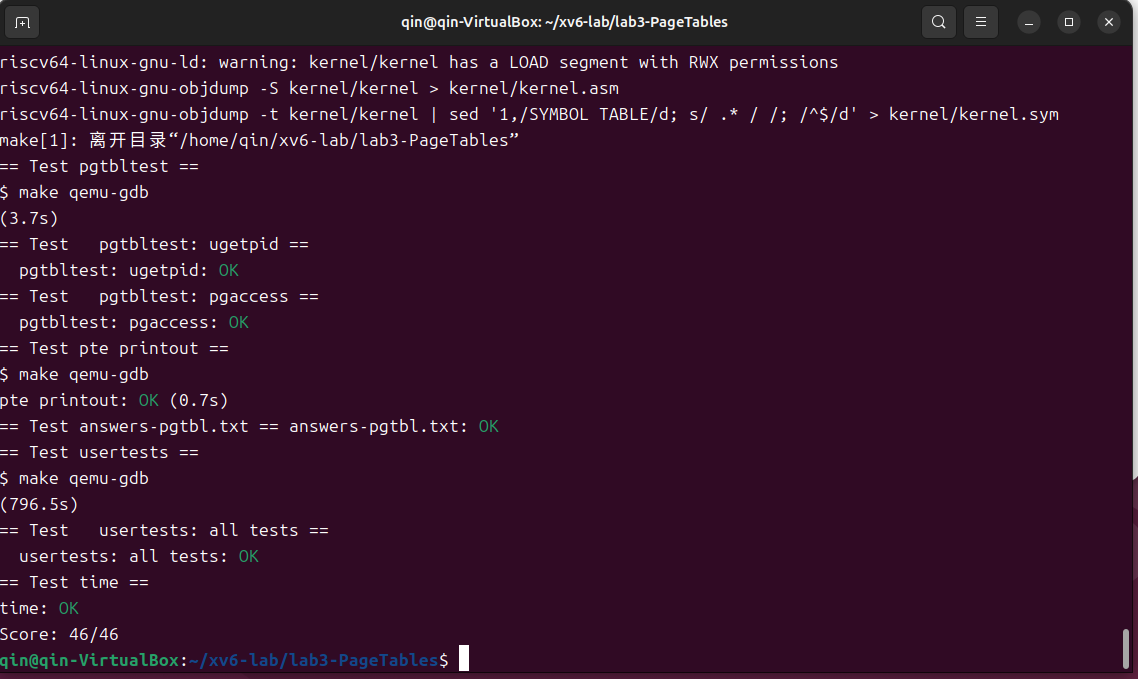
内核与用户空间的交互：在 exec() 函数中插入 vmprint() 调用，使我理解了内核中不同模块之间的交互和调用关系。

调试技巧：通过打印页表，可以更直观地观察和调试页表的内容和结构，帮助我们快速定位和解决页表相关的问题。

代码结构与可读性：在编写代码时，保持良好的代码结构和可读性非常重要。这不仅有助于调试和维护代码，也使得代码更易于理解和扩展。

总之，这个实验不仅帮助我加深了对 RISC-V 页表结构的理解，还提高了我在操作系统内核开发中的实践技能。

* 1. 实验小结

通过 make grade 进行本实验综合测试：

Lab4 Traps

1. 1. RISC-V assembly (easy)

4.1.1实验目的

本实验的目的是通过阅读和分析 RISC-V 汇编代码，理解函数调用和参数传递的细节，并回答一些关于 RISC-V 汇编代码的问题。

4.1.2实验步骤

阅读 user/call.asm 文件：查看 call.asm 文件中 g, f, 和 main 函数的汇编代码。

**根据汇编代码和 RISC-V 指令手册回答以下问题：**

函数参数存储在哪些寄存器中？例如，main 中调用 printf 时，哪个寄存器保存了值 13？

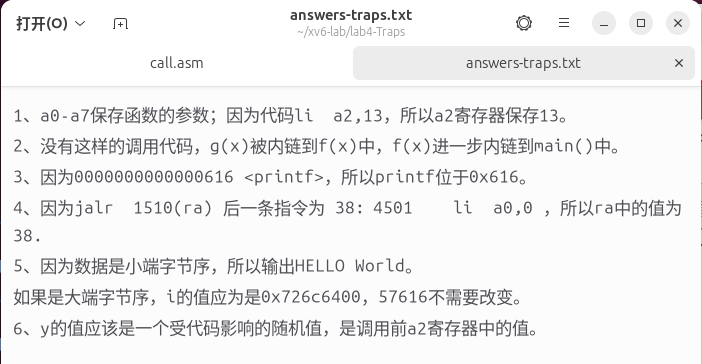
在 main 的汇编代码中，调用函数 f 和 g 的位置在哪里？

printf 函数位于什么地址？

在 main 中执行 jalr 到 printf 之后，ra 寄存器中保存的值是什么？

运行以下代码时，输出是什么？unsigned int i = 0x00646c72; printf("H%x Wo%s", 57616, &i);

在以下代码中，'y=' 后将打印什么内容？为什么会发生这种情况？printf("x=%d y=%d", 3);

4.1.3问题与解决方案

4.1.4实验心得

通过本实验，我学习到了以下内容：

RISC-V 汇编代码分析：通过阅读和分析 RISC-V 汇编代码，了解函数调用和参数传递的细节。

汇编指令和寄存器：理解了 RISC-V 汇编指令和寄存器的用法，尤其是函数参数在寄存器中的存储方式。

小端序和大端序：了解了小端序和大端序的区别，并且掌握了如何在不同字节序下设置变量值以得到相同的输出。

调试技巧：通过分析和回答问题，增强了在阅读和调试汇编代码方面的能力。

本实验不仅加深了对 RISC-V 汇编的理解，还提升了分析和解决问题的能力，为将来调试和优化代码提供了宝贵的经验。

* 1. Backtrace (moderate)

4.2.1实验目的

本实验的目的是实现一个 backtrace() 函数，用于打印当前函数调用栈的回溯信息。在调试时，这种回溯信息可以帮助开发者了解错误发生时的调用链，从而更快速地定位和解决问题。

4.2.2实验步骤

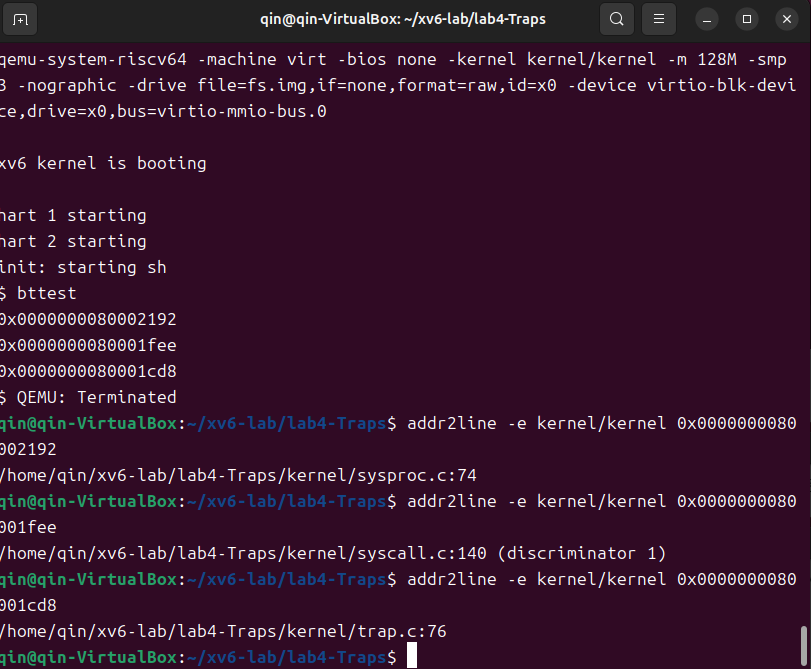
定义backtrace()函数：在 kernel/printf.c 中实现 backtrace() 函数，利用帧指针遍历栈帧并打印每个栈帧的返回地址。

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

在sys\_sleep中调用backtrace()函数：在sys\_sleep函数中插入对backtrace()函数的调用，以便在bttest运行时打印回溯信息。

编译和测试：使用make命令编译修改后的xv6内核和用户空间程序。运行make qemu启动xv6系统，并运行bttest程序。

验证回溯信息：退出QEMU，并使用addr2line工具验证打印的地址对应的源代码位置。

4.2.3问题与解决方案

问题 1：如何读取当前帧指针？

解决方案：通过在 kernel/riscv.h 中定义 r\_fp() 函数，利用内联汇编从 s0 寄存器中读取当前帧指针。

问题 2：如何遍历栈帧并打印返回地址？

解决方案：在 backtrace() 函数中，利用帧指针遍历每个栈帧，并打印帧指针- 8位置（即uint64型指针的一个单位）的返回地址。同时，检查帧指针是否越界或到达栈的顶部，以确定何时停止遍历。

4.2.4实验心得

通过这个实验，我学习到了以下内容：

帧指针和栈帧结构：通过实现 backtrace() 函数，我深入理解了帧指针在函数调用栈中的作用，以及如何利用帧指针遍历栈帧以获取回溯信息。

内联汇编的应用：在 r\_fp() 函数中使用内联汇编读取帧指针，使我了解了如何在 C 代码中嵌入汇编指令，并通过这种方式与底层硬件进行交互。

调试技巧：实现并使用 backtrace() 函数，可以帮助我们在调试时快速了解错误发生时的调用链，从而更有效地定位和解决问题。这对于操作系统的开发和调试非常有帮助。

系统调用的修改和扩展：在 sys\_sleep 函数中插入 backtrace() 调用，使我熟悉了如何修改和扩展系统调用的功能。

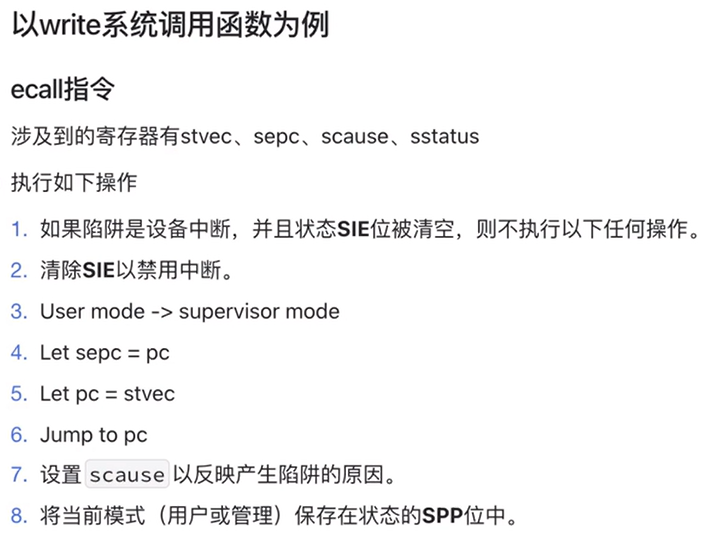
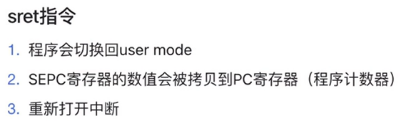
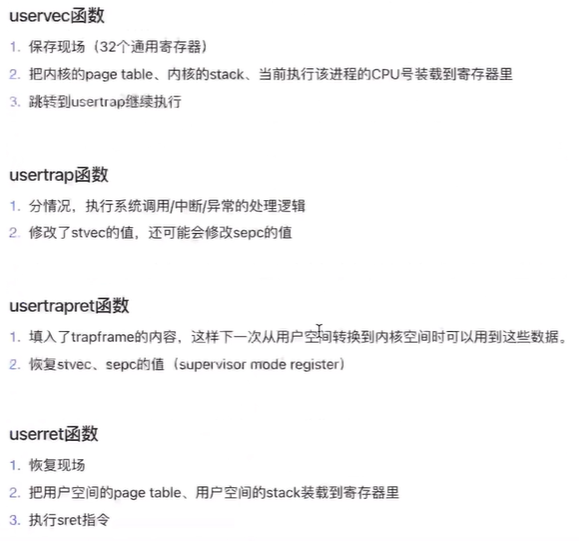
总之，这个实验不仅帮助我加深了对函数调用栈和帧指针的理解，还提高了我在操作系统内核开发中的调试技能。

* 1. Alarm (hard)

4.3.1实验目的

本实验的目的是在xv6中实现一个功能，使得进程在使用CPU时间时周期性地收到提醒。这对于需要限制CPU使用时间的计算密集型进程或希望定期执行某些操作的进程可能很有用。更一般地，我们将实现用户级中断/故障处理的基本形式。

4.3.2涉及函数及作用



4.3.3实验步骤

**添加** sigalarm **和** sigreturn **系统调用**

修改 Makefile：将 user/alarmtest.c 添加到 Makefile 中，使其编译为 xv6 用户程序。

在 user/user.h 中添加声明：int sigalarm(int ticks, void (\*handler)()); int sigreturn(void);

更新 user/usys.pl、kernel/syscall.h 和 kernel/syscall.c：使 alarmtest 可以调用 sigalarm 和 sigreturn 系统调用。

修改 sys\_sigreturn，目前只返回 0。

修改 sys\_sigalarm：在 proc 结构中添加新的字段，用于存储警报间隔和处理函数的指针。

初始化 proc 结构中的新字段：在 proc.c 中的 allocproc() 中初始化这些字段。

**修改** usertrap **函数**

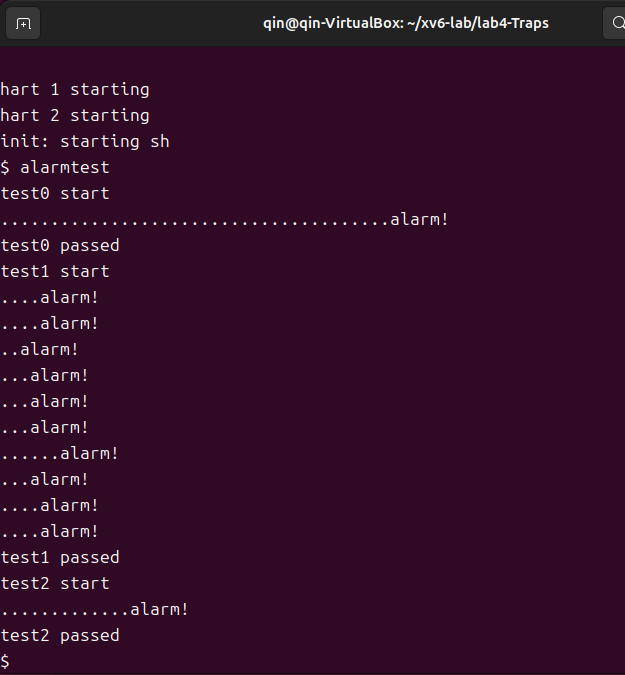
处理时钟中断：在 usertrap() 中，只有在设备号为 2 时才处理时钟中断。

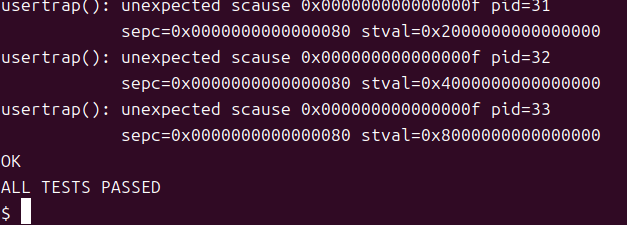
调用用户空间的警报处理函数：修改 usertrap()，在进程的警报间隔到期时，让用户进程执行处理函数。

**恢复被中断的代码**

保存和恢复寄存器：在用户中断发生时保存足够的状态，以便 sigreturn 可以正确地返回到被中断的用户代码。

防止重入调用处理函数：如果处理函数尚未返回，内核不应再次调用它。

重新设置警报计数器：在处理函数每次调用后重新设置警报计数器，使处理函数被定期调用。

4.3.4问题与解决方案

问题1: 处理函数调用失败或导致程序崩溃

解决方案:确保在 usertrap() 中正确调用处理函数，并在处理函数返回后恢复被中断的代码。修改 usertrap() 以保存当前的寄存器状态，并在 sigreturn 中恢复这些寄存器。

问题 2: 处理函数的重入调用

解决方案:在 proc 结构中添加一个标志，指示处理函数是否正在执行。如果标志为真，则不再次调用处理函数。

问题 3: 恢复用户进程的状态

解决方案:在用户中断发生时，保存用户进程的寄存器状态，并在 sigreturn 中恢复这些状态。

4.3.5实验心得

通过本实验，我学到了如何在 xv6 中添加新的系统调用，并且理解了用户级中断/故障处理的基本实现。以下是一些具体的心得：

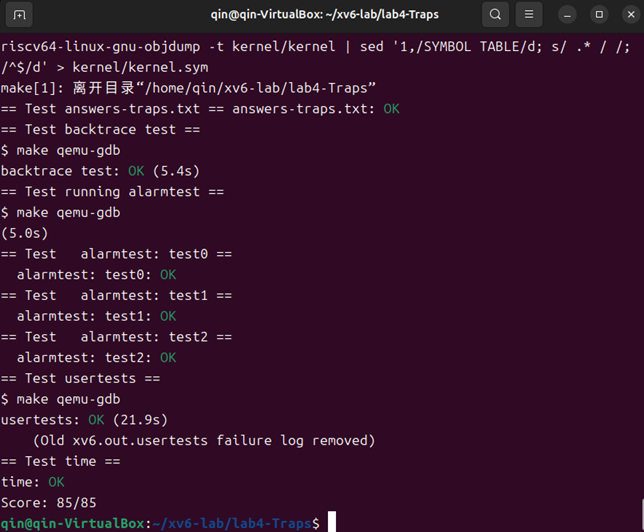
系统调用的添加：了解了如何在 xv6 中添加新的系统调用，包括更新相关的头文件和实现具体的系统调用函数。

进程结构体的扩展：学会了如何在 xv6 的进程结构体中添加新的字段，以支持新的功能。

中断处理：理解了如何在 xv6 中处理时钟中断，并利用时钟中断实现定期提醒功能。

上下文切换：掌握了如何保存和恢复进程的上下文，以确保进程在处理完中断后能继续执行之前的代码。

* 1. 实验小结

通过 make grade 进行本实验综合测试：

Lab5 Copy-on-Write

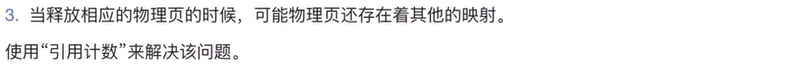
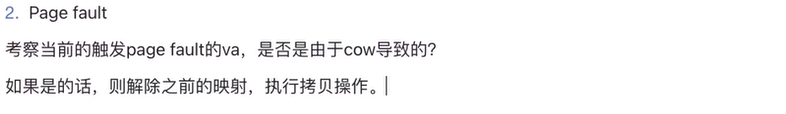
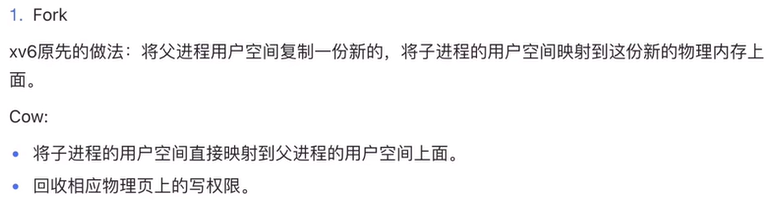
1. 1. Implement copy-on write(hard)

5.1.1实验目的

本实验的目的是在 xv6 内核中实现写时复制（Copy-on-Write, COW）的fork操作。通过使用写时复制技术，父进程和子进程在fork时共享相同的物理内存页，只有在需要写入这些页时才分配新的内存页。这将显著减少fork操作时的内存开销。

5.1.2实验步骤

简要的实验原理如下图所示：

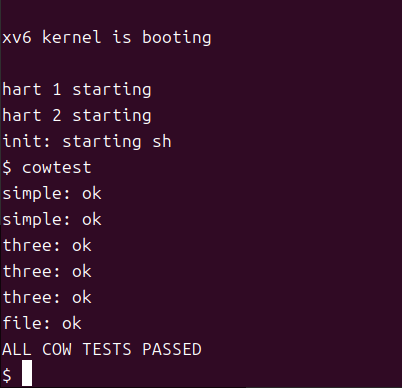
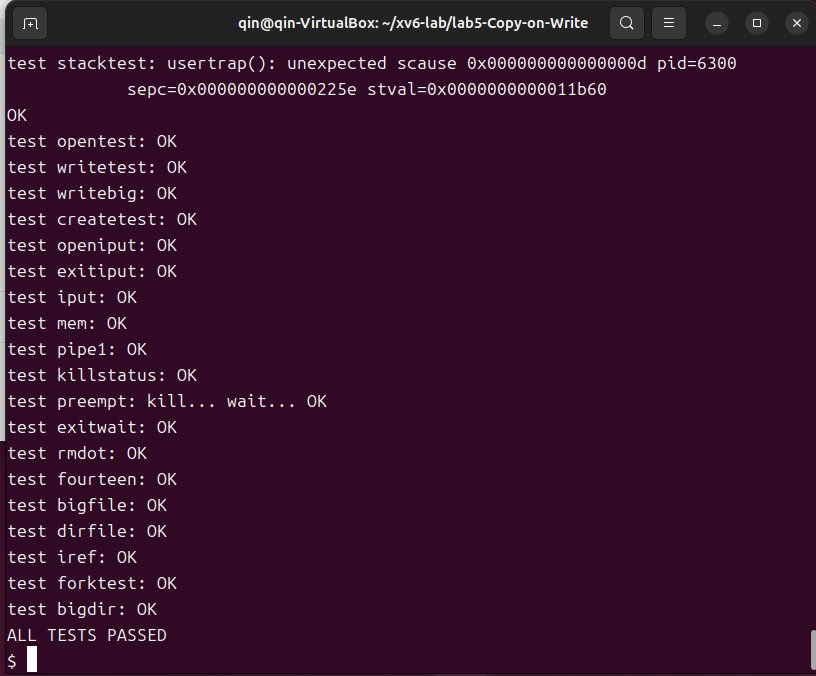


Riscv.h中定义#define PTE\_COW (1L << 8) // cow page

修改uvmcopy：修改 uvmcopy 函数，使其将父进程的物理页映射到子进程的页表中，而不是分配新的页。同时，在父进程和子进程的页表项（PTEs）中清除 PTE\_W 位，以实现只读共享。

处理页面错误：修改 usertrap 函数，以识别页面错误。当写时复制的页面错误发生时，分配一个新的页，将旧页复制到新页，并在页表中设置新的页。同时，在新页表项中设置 PTE\_W 位，使其成为可写的页。

管理物理页引用计数：首先，为每个物理页维护一个引用计数，记录有多少用户页表引用该页。可以使用物理页地址除以 4096 作为索引，将计数值存储在一个固定大小的数组中。其次，增加和减少引用计数。在 fork 导致子进程共享页时，增加页的引用计数。在进程删除页表中的页时，减少引用计数。最后，释放物理页。当引用计数减为零时，调用 kfree 将页放回空闲列表。

处理 COW 页面：当 copyout 函数遇到写时复制的页时，使用与页面错误相同的机制来处理它。

5.1.3问题与解决方案

问题1: 引用计数管理不当

解决方案:在 kalloc 分配页时，初始化引用计数为 1。在 fork 中增加引用计数，在进程删除页时减少引用计数。当引用计数减为零时，释放物理页。

5.1.4实验心得

通过本实验，我学习到了以下内容：

写时复制技术：通过实现写时复制功能，了解了如何在操作系统中实现高效的进程创建和内存管理。

页面错误处理：通过处理写时复制的页面错误，增强了对页面错误和内存管理的理解。

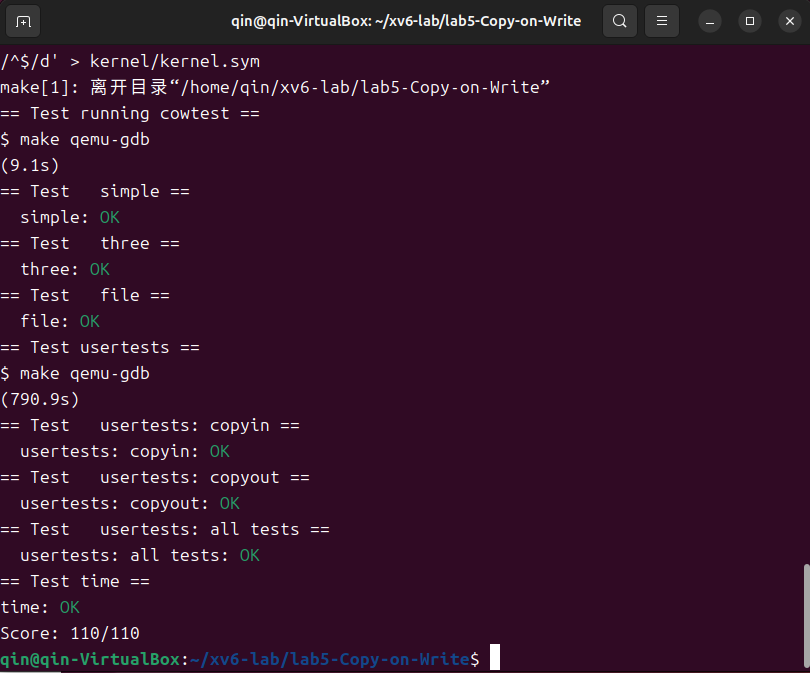
引用计数管理：通过管理物理页的引用计数，了解了如何确保物理页在没有引用时被正确释放。

调试技巧：在实现过程中，通过调试和测试不同的场景，提升了在操作系统内核级别调试和解决问题的能力。

本实验不仅加深了对 xv6 内核的理解，还提升了设计和实现复杂内存管理功能的能力，为将来开发和调试操作系统内核提供了宝贵的经验。

* 1. 实验小结

通过 make grade 进行本实验综合测试：

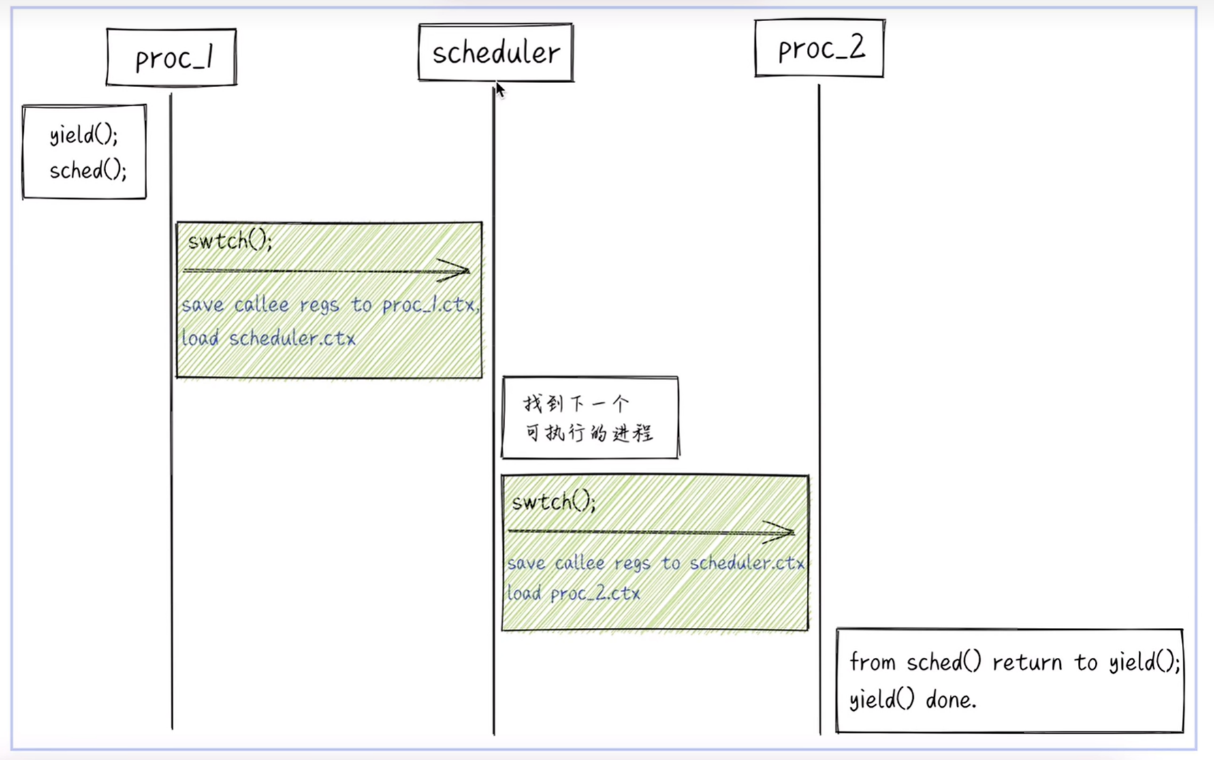


Lab6 Multithreading

1. 1. Uthread: switching between threads (moderate)

6.1.1实验目的

本实验的目的是设计并实现一个用户级线程系统中的上下文切换机制。具体来说，我们需要完成 uthread.c 和 uthread\_switch.S 文件中的部分代码，以便能够创建线程并在它们之间切换。最终目标是通过执行 uthread 程序来验证线程创建和上下文切换的正确性，确保多个线程能够并发执行并正确切换。

6.1.2实验步骤

**修改** thread\_create **函数**

初始化线程结构：为新线程分配栈空间，并设置栈指针。初始化线程的上下文，使其在第一次运行时调用传递的函数。

设置线程栈：将传递的函数地址和线程的参数压入栈中。设置返回地址，使其指向线程函数的入口。

**修改** thread\_schedule **函数**

选择下一个可运行的线程：遍历线程列表，找到下一个状态为 RUNNABLE 的线程。

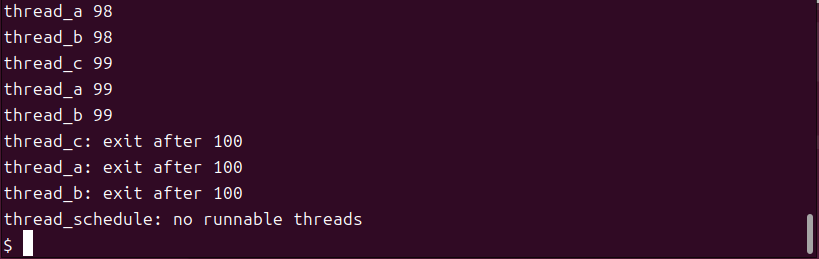
调用thread\_switch进行上下文切换：在thread\_schedule中调用thread\_switch，传递当前线程和下一个线程的指针。

**实现** thread\_switch **函数**

保存当前线程的寄存器：使用汇编代码保存当前线程的所有寄存器（主要是被调用者保存的寄存器）。

恢复下一个线程的寄存器：使用汇编代码恢复下一个线程的所有寄存器。

返回到下一个线程执行的地方。恢复栈指针并返回，从而切换到下一个线程继续执行。



6.1.3问题与解决方案

问题1: 上下文切换时寄存器保存和恢复错误

解决方案:在 thread\_switch 中正确保存和恢复寄存器。特别是要确保被调用者保存的寄存器（例如 s0 到 s11）在切换时正确保存和恢复。

6.1.4实验心得

通过本实验，我学到了以下内容：

上下文切换机制：理解了用户级线程系统中的上下文切换机制，包括如何保存和恢复线程的寄存器状态。

线程调度：实现了基本的线程调度算法，确保多个线程能够并发执行并正确切换。

汇编代码与C代码结合：通过在 thread\_switch 中编写汇编代码，实现了寄存器的保存和恢复，增强了对汇编语言的理解。

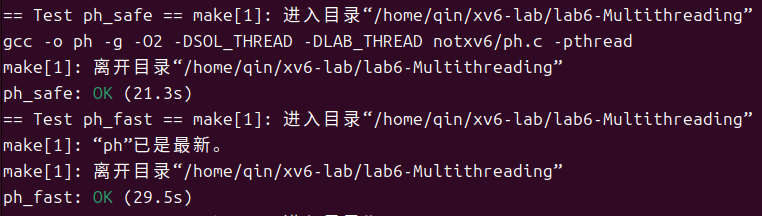
调试技巧：在调试过程中，使用 GDB 单步执行和查看寄存器状态，提升了在复杂系统中调试和解决问题的能力。

本实验不仅加深了对用户级线程系统的理解，还提高了设计和实现多线程并发程序的能力，为将来处理并发编程问题提供了宝贵的经验。

* 1. Using threads (moderate)

6.2.1实验目的

本实验旨在通过使用线程和锁实现一个并行哈希表，探索并行编程。通过使用 UNIX pthread 线程库，我们将确保哈希表在多线程环境下的正确性和性能。实验的目标是让多线程同时执行哈希表的插入（put）和获取（get）操作，并保持哈希表的一致性，同时实现一定的并行加速。

6.2.2实验步骤

6.2.3问题与解决方案

6.2.4实验心得

通过本实验，我学到了以下内容：

线程安全和竞态条件：理解了多线程环境下竞态条件的产生原因，以及如何通过使用锁来避免这些问题。

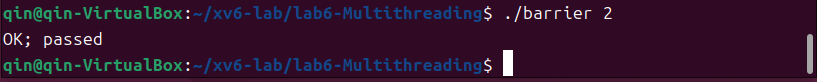
并行编程技巧：通过将全局锁优化为桶锁，提高了并行性能，学会了如何在保证线程安全的同时尽可能地提高并行效率。

实际操作和调试：在实际的 Linux 环境下操作和调试多线程程序，增强了实际编程和问题解决的能力。

* 1. Barrier(moderate)

6.3.1实验目的

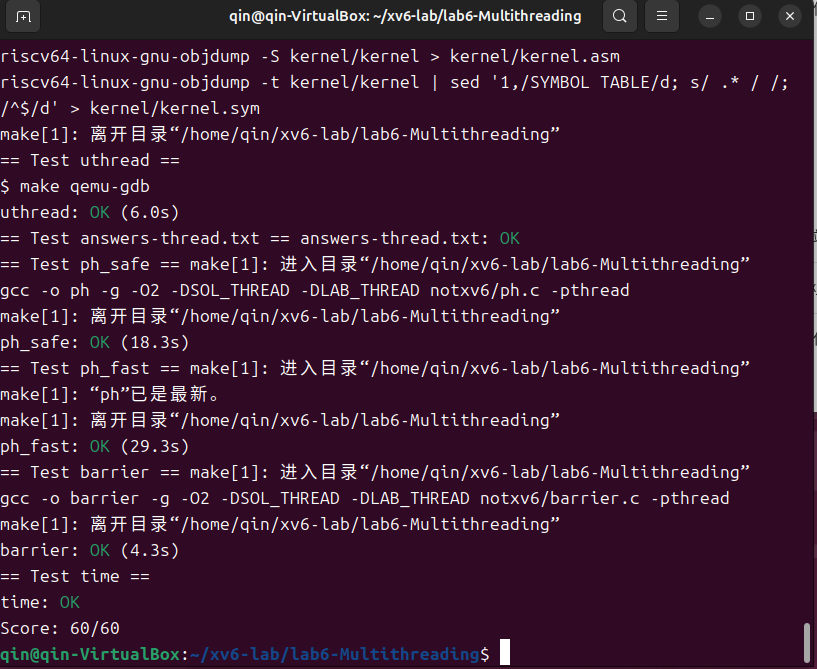
6.3.2实验步骤



6.3.3问题与解决方案

6.1.4实验心得

* 1. 实验小结



Lab7 Network Driver

Lab8 Lock

Lab9 File System

Lab10 Mmap

2.1.1实验目的

2.1.2实验步骤

2.1.3问题与解决方案

2.1.4实验心得