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 4](#_Toc172217231)

[Lab1 Xv6 and Unix utilities 6](#_Toc172217232)

[1.1 Boot xv6 (easy) 6](#_Toc172217233)

[1.1.1实验目的 6](#_Toc172217234)

[1.1.2实验步骤 6](#_Toc172217235)

[1.1.3实验心得 6](#_Toc172217236)

[1.2 sleep (easy) 7](#_Toc172217237)

[1.2.1实验目的 7](#_Toc172217238)

[1.2.2实验步骤 7](#_Toc172217239)

[1.2.3问题与解决方案 7](#_Toc172217240)

[1.2.4实验心得 8](#_Toc172217241)

[1.3 pingpong (easy) 8](#_Toc172217242)

[1.3.1实验目的 8](#_Toc172217243)

[1.3.2实验步骤 8](#_Toc172217244)

[1.3.3问题与解决方案 9](#_Toc172217245)

[1.3.4实验心得 9](#_Toc172217246)

[1.4 primes (moderate)/(hard) 9](#_Toc172217247)

[1.4.1实验目的 9](#_Toc172217248)

[1.4.2实验步骤 9](#_Toc172217249)

[1.4.3问题与解决方案 10](#_Toc172217250)

[1.4.4实验心得 10](#_Toc172217251)

[1.5 find (moderate) 10](#_Toc172217252)

[1.5.1实验目的 10](#_Toc172217253)

[1.5.2实验步骤 11](#_Toc172217254)

[1.5.3问题与解决方案 11](#_Toc172217255)

[1.5.4实验心得 11](#_Toc172217256)

[1.6 xargs (moderate) 11](#_Toc172217257)

[1.6.1实验目的 11](#_Toc172217258)

[1.6.2实验步骤 12](#_Toc172217259)

[1.6.3问题与解决方案 12](#_Toc172217260)

[1.6.4实验心得 12](#_Toc172217261)

[1.7 实验小结 12](#_Toc172217262)

[Lab2 System Calls 13](#_Toc172217263)

[2.1 System call tracing (moderate) 13](#_Toc172217265)

[2.1.1实验目的 13](#_Toc172217266)

[2.1.2实验步骤 13](#_Toc172217267)

[2.1.3问题与解决方案 14](#_Toc172217268)

[2.1.4实验心得 15](#_Toc172217269)

[2.2 Sysinfo (moderate) 15](#_Toc172217270)

[2.2.1实验目的 15](#_Toc172217271)

[2.2.2实验步骤 15](#_Toc172217272)

[2.2.3问题与解决方案 16](#_Toc172217273)

[2.2.4实验心得 16](#_Toc172217274)

[2.3 实验小结 17](#_Toc172217275)

[Lab3 Page Tables 18](#_Toc172217276)

[3.1 Speed up system calls (easy) 18](#_Toc172217280)

[3.1.1实验目的 18](#_Toc172217281)

[3.1.2实验步骤 18](#_Toc172217282)

[3.1.3问题与解决方案 19](#_Toc172217283)

[3.1.4实验心得 19](#_Toc172217284)

[3.2 Speed up system calls (easy) 19](#_Toc172217285)

[3.2.1实验目的 19](#_Toc172217286)

[3.2.2实验步骤 19](#_Toc172217287)

[3.2.3问题与解决方案 20](#_Toc172217288)

[3.2.4实验心得 20](#_Toc172217289)

[3.3 实验小结 21](#_Toc172217290)

[Lab4 Traps 22](#_Toc172217291)

[Lab5 Lazy allocation 23](#_Toc172217292)

[Lab6 Copy-on-Write 24](#_Toc172217293)

[Lab7 Multithreading 25](#_Toc172217294)

[Lab8 Locks 26](#_Toc172217295)

[Lab9 File System 27](#_Toc172217296)

[Lab10 Mmap 28](#_Toc172217297)

[Lab11 Network Driver 29](#_Toc172217298)

[2.1.1实验目的 29](#_Toc172217299)

[2.1.2实验步骤 29](#_Toc172217300)

[2.1.3问题与解决方案 29](#_Toc172217301)

[2.1.4实验心得 29](#_Toc172217302)

Lab0 Program execution logic

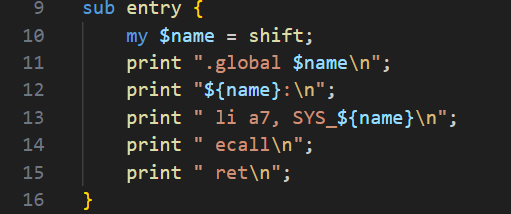
0.1用户态程序发起系统调用

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

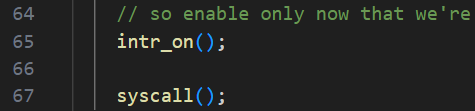
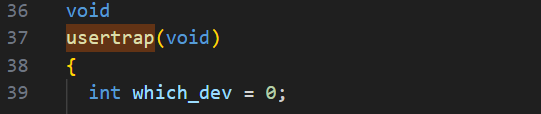
库函数会将系统调用号和参数放入指定的位置。在 xv6 中，系统调用号通常放在寄存器 a7，参数放在寄存器 a0, a1, a2, a3, a4, a5。

0.3触发系统调用指令

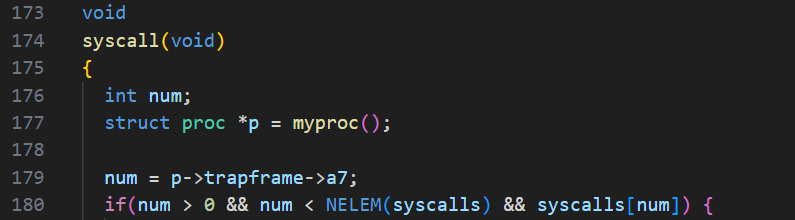
在 xv6 中，触发系统调用的指令是ecall。

0.4内核接收系统调用

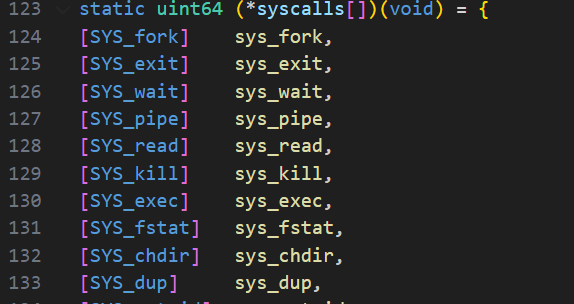
触发ecall指令后，CPU切换到内核态，跳转到内核的系统调用处理入口。在 xv6 中，这通常是 usertrap() 函数。



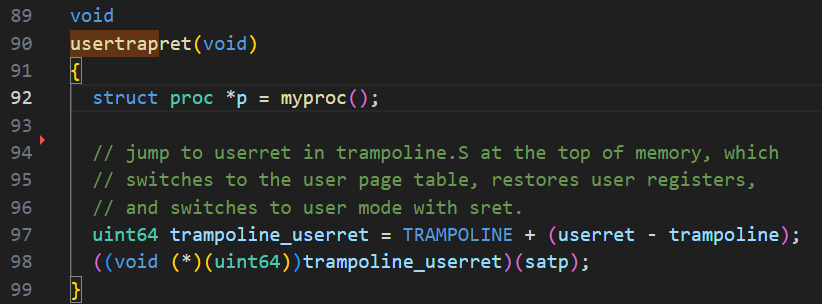
0.5处理系统调用

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。

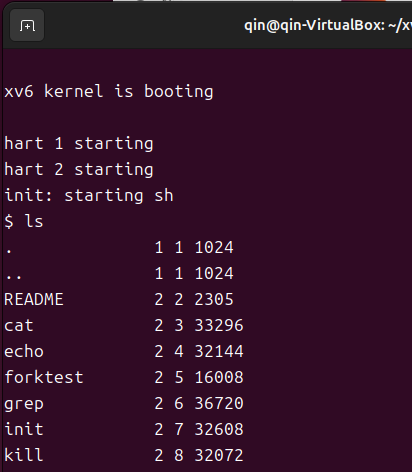
2. 运行xv6系统

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

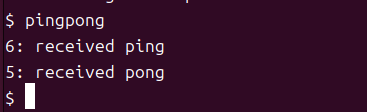
创建管道子进程：使用pipe函数创建一个管道，使用fork函数创建一个子进程。

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

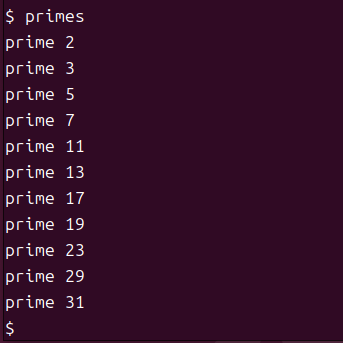
实现一个使用管道的并发质数筛选程序。。目标是使用 pipe 和 fork 创建管道，第一个进程将数字 2 到 35 输入管道。对于每个质数，创建一个进程，从左邻居的管道读取数据，并写入到右邻居的管道。

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。

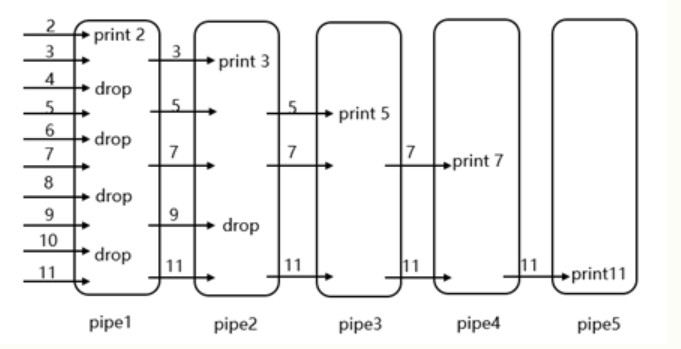
筛选质数：对于每个质数，创建一个进程，读取从左邻居传来的数字，筛选出质数，并将剩余的数字写入右邻居。质数筛原理简单的来说就是从头开始选取未被筛去的数，将它的倍数筛去，不断重复。

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)

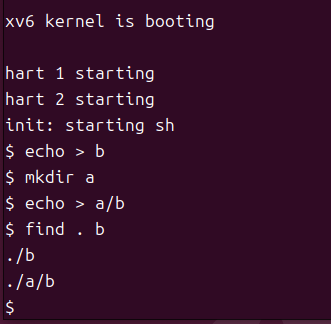
1.5.1实验目的

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

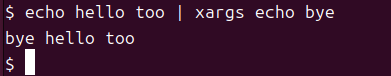
编写一个简单版本的 UNIX xargs 程序，要求从标准输入读取行，并为每一行运行一个命令，将该行作为命令的参数。

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。

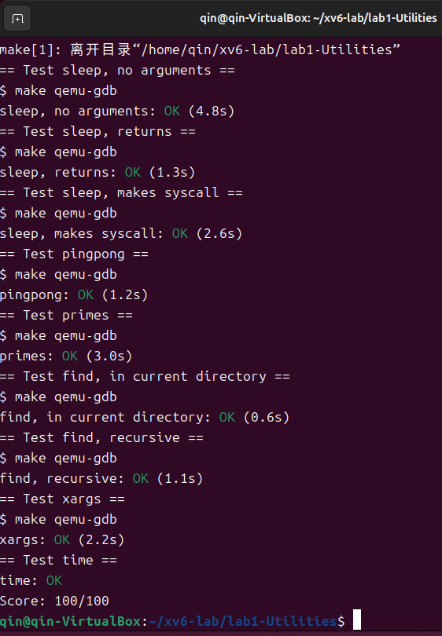
1.6.4实验心得

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。

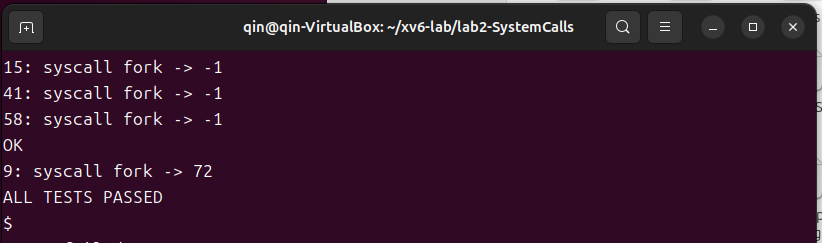
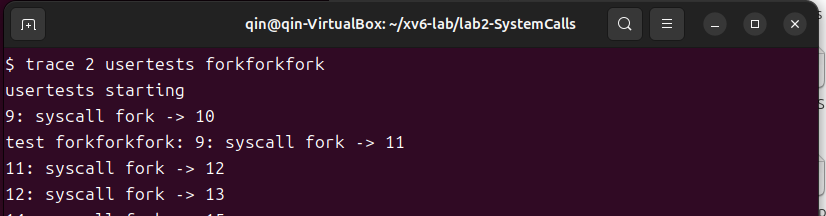
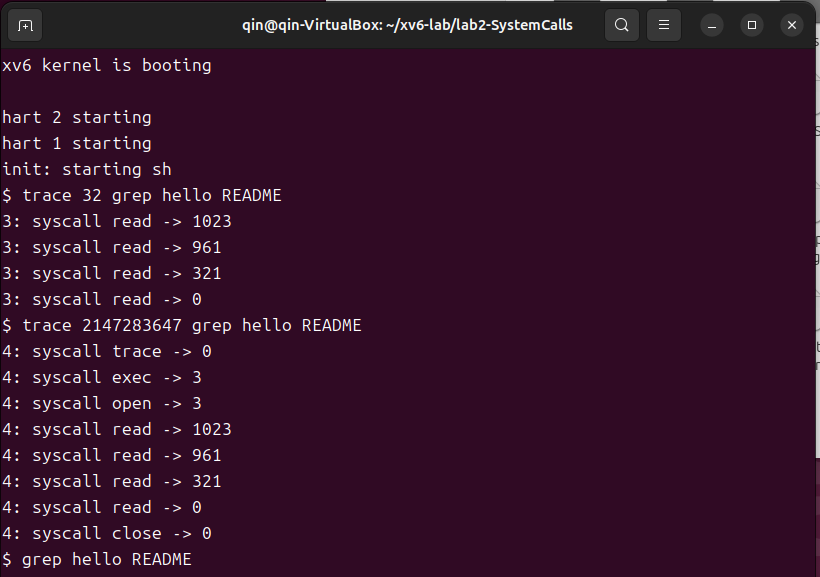
修改kernel/proc.c：修改fork()函数，确保在创建子进程时将父进程的跟踪掩码复制到子进程中，以实现跟踪信息在进程间的继承，即np->trace\_mask=p->trace\_mask;。

**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。

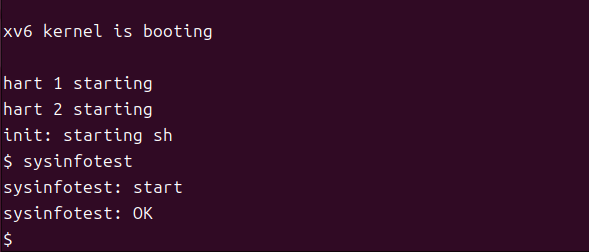
修改 kernel/kalloc.c：添加一个函数uint64 kfreemem(void)来收集可用内存的字节数。

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：

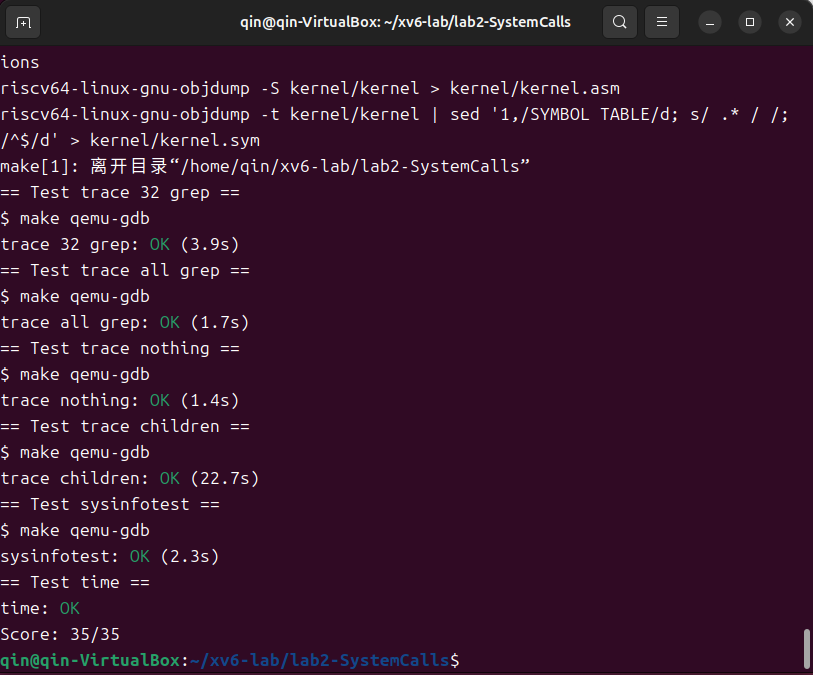
系统调用的实现：了解了如何在 xv6 内核中添加和实现新的系统调用。

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。

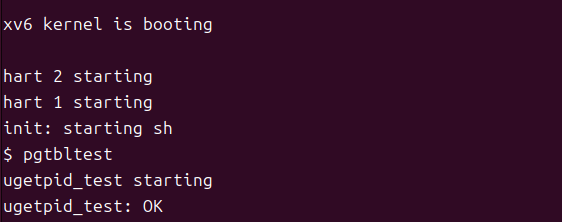
修改 kernel/proc.c：在 allocproc() 函数中初始化 struct usyscall 以存储当前进程的 PID。

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

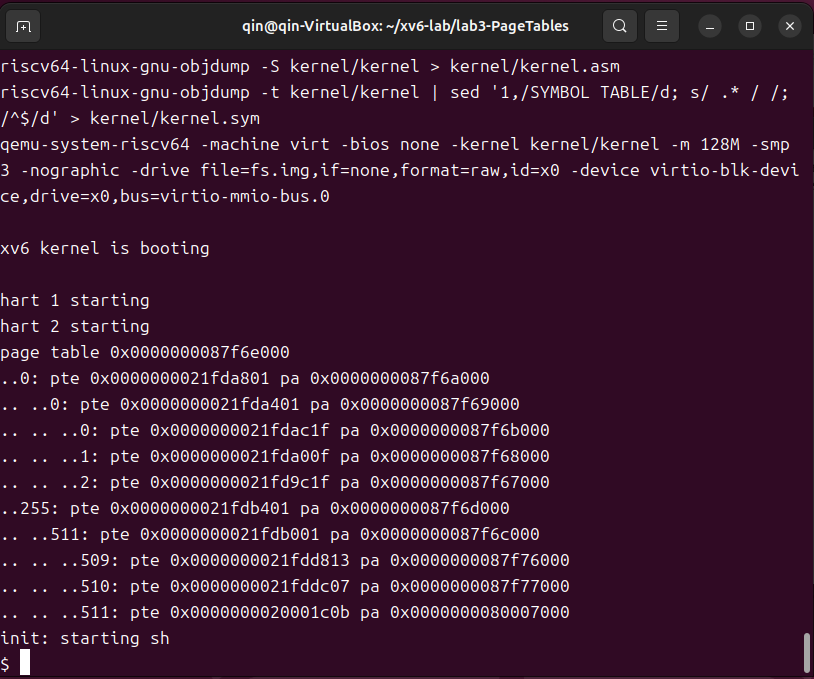
本实验的目的是实现一个函数 vmprint()，用于打印 RISC-V 页表的内容。通过这个函数，可以帮助我们可视化页表结构，便于调试和理解页表的工作原理。

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。

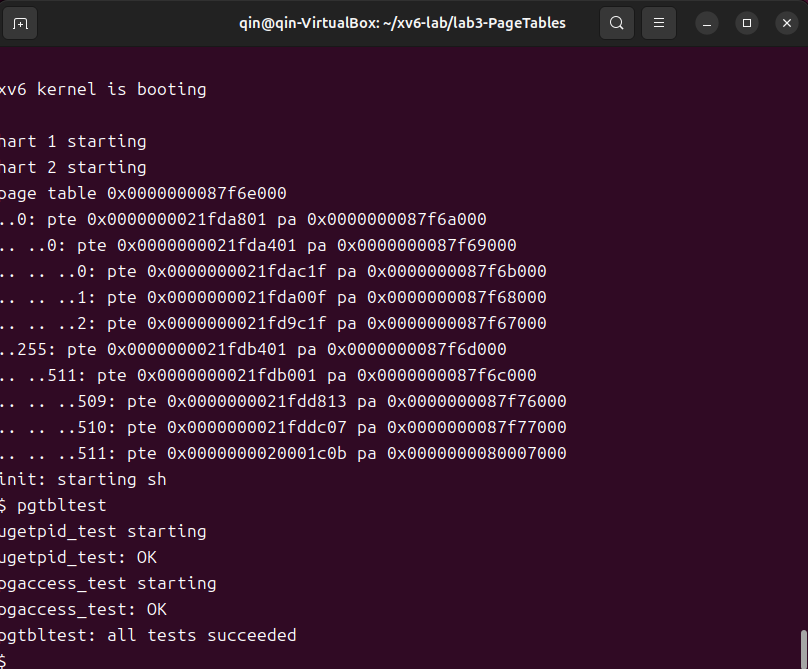
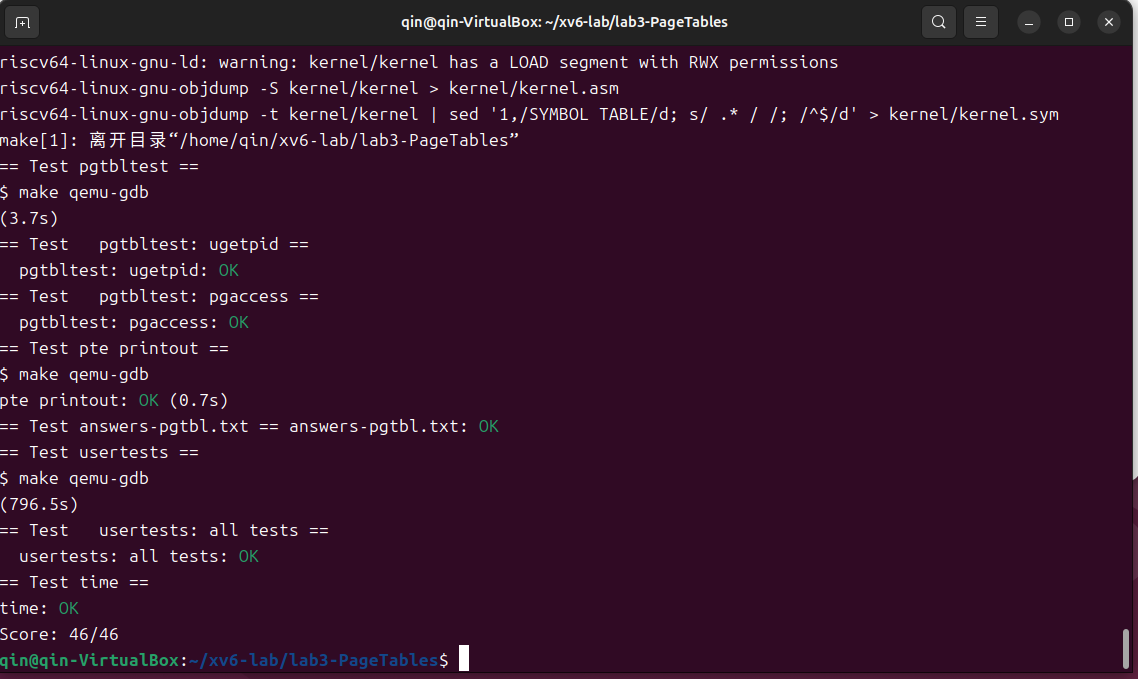
内核与用户空间的交互：在 exec() 函数中插入 vmprint() 调用，使我理解了内核中不同模块之间的交互和调用关系。

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. RISC-V assembly (easy)

4.1.1实验目的

4.1.2实验步骤

4.1.3问题与解决方案

4.1.4实验心得

Lab5 Lazy allocation

Lab6 Copy-on-Write

Lab7 Multithreading

Lab8 Locks

Lab9 File System

Lab10 Mmap

Lab11 Network Driver

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