同济大学

TONGJI UNIVERSITY

操作系统课程设计

作 者 姓 名：雷思雨

学　　号：2354181

指 导 教 师：王冬青

学院、 专业：软件学院 软件工程

同济大学

Tongji University

目录

第 0 章 环境配置

配置镜像源：

sudo apt update
sudo apt upgrade -y

配置编译工具：

通过下面的命令进行安装基本的编译和构建软件工具可以，build-essential 软件包组包含了 gcc、g++、make 等编译工具和构建工具，也包含了标准 C 库和头文件：

sudo apt install build-essential -y

安装 git 工具：
sudo apt install git-all -y

安装编译工具
sudo apt install cmake

第 1 章 Lab: Xv6 and Unix utilities

1.1 Boot xv6

1.1.1 实验目的

使用 make qemu 运行 xv6。

1.1.2 实验步骤

进入的虚拟机后，通过“git clone git://g.csail.mit.edu/xv6-labs-2021”克隆远端的仓库。

然后进行如下操作：

$ cd xv6-labs-2021-1（此处对文件夹进行了重命名）

$ git checkout util

Branch 'util' set up to track remote branch 'util' from 'origin'.

Switched to a new branch 'util'

$ make qemu

1.1.3 实验中遇到的问题和解决方案

无。

1.1.4 实验心得

初步了解 xv6 实验的具体操作步骤以及各指令的使用。

1.2 sleep

1.2.1 实验目的

为 xv6 实现 sleep 指令，该 sleep 指令需要暂停相应的 tick 数，该数由用户指定。

注：tick 是由 xv6 定义的时间概念，即定时器芯片两个中断间的经过的时间。

1.2.2 实验步骤

在编写 user 文件夹中新建 sleep.c 文件，并编写相应代码。

在 Makefile 的 UPROGS 中增加$U/_sleep\。

1.2.3 实验中遇到的问题和解决方案

对实验编写代码的方式不熟悉，经过查阅资料和学习了解相关知识。

1.2.4 实验心得

在实验中通过编写 sleep 函数初步掌握 xv6 的文件结构。

1.3 pingpong

1.3.1 实验目的

掌握 xv6 操作系统中 pipe 的工作原理，理解进程间基于管道的通信机制；熟悉 fork()、read()/write()、close()等核心系统调用的使用场景；理解父子进程的同步与数据传递逻辑，验证进程独立地址空间下的通信方式。

1.3.2 实验步骤

1. 创建管道：调用 pipe(p)创建管道，获取读端 p[0]和写端 p[1]文件描述符；
2. 创建子进程：调用 fork()，父进程继续执行，子进程复制父进程的文件描述符表；
3. 父进程发送数据：父进程关闭读端 p[0]，通过 write(p[1], "ping", 4)向管道写入数据，写完后关闭写端 p[1]；
4. 子进程接收并回传：子进程关闭写端 p[1]，通过 read (p[0], buf, 4)读取管道数据，打印 “received ping”；之后向管道写入 “pong”，关闭读端 p[0]；
5. 父进程接收并验证：父进程通过 wait()等待子进程结束，读取管道中的 “pong”，打印 “received pong”，完成通信闭环。

（在编写 user 文件夹中新建 pingpong.c 文件，并编写相应代码。在 Makefile 的 UPROGS 中增加$U/_ pingpong \。）

1.3.3 实验中遇到的问题和解决方案

不熟悉 fork、write、read 具体的使用方式，通过查看相应文件解决。

1.3.4 实验心得

管道是半双工通信，需明确关闭无用端（如父进程关读端、子进程关写端），否则会导致进程阻塞。在实验中系统调用的顺序至关重要（如先创建管道再 fork，先关闭无用端再读写），错误顺序会导致通信失败或资源泄漏。

1.4 primes

1.4.1 实验目的

使用管道编写 prime 筛的并发版本。

1.4.2 实验步骤

1. 初始化管道与进程：主进程创建初始管道，写入 2~35 的整数，随后 fork() 创建子进程；
2. 子进程筛选逻辑：子进程从管道读第一个数（素数）并打印，再创建新管道，过滤掉能被该素数整除的数，将剩余数写入新管道；
3. 递归创建子进程：子进程重复“读素数→创新管道→过滤数据→fork 新子进程” 流程，直至管道无数据可读。

（在编写 user 文件夹中新建 pingpong.c 文件，并编写相应代码。在 Makefile 的 UPROGS 中增加$U/_ pingpong \。）

1.4.3 实验中遇到的问题和解决方案

在实验过程中未及时关闭无用管道端，导致进程因等待数据陷入无限阻塞；后修改在子进程退出后父进程通过 wait()回收资源。

1.4.4 实验心得

在实验中逐渐理解分布式计算中 “任务拆分与协作”，需要注意多进程协作需严格控制管道读写顺序与文件描述符生命周期，避免死锁或资源浪费。

1.5 find

1.5.1 实验目的

编写 find 程序用于查找目录树中具有特定名称的所有文件；理解 xv6 文件系统层次结构，掌握目录与文件的遍历逻辑。

1.5.2 实验步骤

1. 解析命令参数：获取用户输入的查找路径（默认当前目录）和目标文件名；
2. 遍历目录初始化：调用 opendir()打开目标目录，通过 readdir()读取目录项（文件 / 子目录）；

3. 文件类型判断：对每个目录项调用 stat()获取文件属性，区分普通文件与子目录；
4. 文件名匹配与输出：若为普通文件且文件名匹配，打印文件路径；若为子目录（排除.和..），递归调用自身遍历子目录；

（在编写 user 文件夹中新建 find.c 文件，并编写相应代码。在 Makefile 的 UPROGS 中增加$U/_ find \。）

1.5.3 实验中遇到的问题和解决方案

对实验编写代码的方式不熟悉，经过查阅资料和学习了解相关知识。

在实验过程中仅匹配文件名而非全路径，导致多目录下同名文件漏查。后存储完整文件路径，基于全路径进行匹配判断。

1.5.4 实验心得

在实验的递归遍历过程中，需严格控制边界，避免无限循环或资源浪费，在全部遍历的基础上不重复遍历。

1.6 xargs

1.6.1 实验目的

编写 xaegs 文件，理解 xargs 命令“读取标准输入作为命令行参数”的核心逻辑，掌握进程间标准输入重定向。

1.6.2 实验步骤

1. 解析命令行参数：获取 xargs 后的基础命令（如 echo）及固定参数，区分“固定参数”与“待从标准输入读取的参数”；
2. 读取标准输入数据：从 stdin 读取数据（如通过管道传递的文件名列表），按空格或换行分割为参数列表；
3. 创建子进程并执行命令：调用 fork()创建子进程，在子进程中拼接 “基础命令 + 固定参数 + 标准输入参数”，通过 execvp()执行命令。

（在编写 user 文件夹中新建 xargs.c 文件，并编写相应代码。在 Makefile 的 UPROGS 中增加$U/_ xargs \。）

1.6.3 实验中遇到的问题和解决方案

在调用 exec()时参数数组格式错误，后调整代码确保参数数组最后一位为 NULL。

1.6.4 实验心得

Xargs 实验的核心是“标准输入转命令行参数”，本质是进程间数据传递与命令执行的协作，需明确 stdin 的继承与重定向逻辑，同时需注意回收资源。

1.7 实验结果

```
make[1]: Leaving directory '/home/cecilia/xv6-labs-2021-1'
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (6.1s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (0.4s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (0.7s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (1.1s)
== Test primes ==
$ make qemu-gdb
primes: OK (1.2s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (1.1s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (1.2s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (0.7s)
== Test time ==
time: OK
Score: 100/100
cecilia@cecilia-virtual-machine:~/xv6-labs-2021-1$
```

第 2 章 Lab: system calls

2.1 System call tracing

2.1.1 实验目的

理解 xv6 系统调用的执行流程，掌握在内核层添加追踪功能的方法。熟悉并编写 trace，学会通过内核参数控制追踪范围（帮助了解系统调用日志打印逻辑）。

2.1.2 实验步骤

1. 添加系统调用声明：在 syscall.h 定义 SYS_trace 编号，user.h 声明 trace() 函数原型；
2. 实现 trace 系统调用：在 syscall.c 添加 sys_trace 函数，将用户传入的追踪掩码存入进程控制块（proc）的 trace_mask 字段；
3. 修改系统调用入口：在 syscall()函数（syscall.c）中，检查当前进程 trace_mask，若包含当前系统调用编号，打印调用信息（进程 ID、系统调用名、返回值）；
4. 用户态测试验证：编写 trace 用户程序，接收命令行参数（如 trace 32 sh，32 对应 exec），调用 trace()后执行目标程序，验证日志输出。

（在编写 user 文件夹中新建 trace.c 文件，并编写相应代码。在 Makefile 的 UPROGS 中增加$U/_ trace\。）

2.1.3 实验中遇到的问题和解决方案

打印时系统调用编号与名称对应表索引不匹配，后发现 syscalls 数组顺序与 SYS_*不一致，修改后打印正常。

2.1.4 实验心得

在实验中我了解到系统调用追踪的核心是内核态对进程行为的监控，通过进程控制块存储状态（trace_mask）实现进程级追踪控制；在编写代码的过程中须保持全局的一致性。

2.2 Sysinfo

2.2.1 实验目的

添加一个系统调用 sysinfo，用于收集关于运行系统的信息。

2.2.2 实验步骤

1. 定义数据结构与系统调用声明：在 user.h 声明 struct sysinfo（含 freemem、nproc 字段）和 sysinfo()函数；在 syscall.h 定义 SYS_sysinfo 编号；
2. 统计空闲内存：遍历空闲页链表（kmem.freelist），累加空闲页大小（每页 4096 字节）得到 freemem；
3. 统计活跃进程：遍历 proc 数组，计数状态为 RUNNABLE/RUNNING/SLEEPING 的进程得到 nproc；
4. 实现 sysinfo 系统调用：在 syscall.c 添加 sys_sysinfo 函数，通过 argaddr() 获取用户态 struct sysinfo 地址，调用 copyout 将内核收集的信息写入用户态内存；
5. 用户态测试验证：编写 sysinfo 用户程序，调用 sysinfo()获取并打印系统空闲内存与活跃进程数，验证数据准确性。

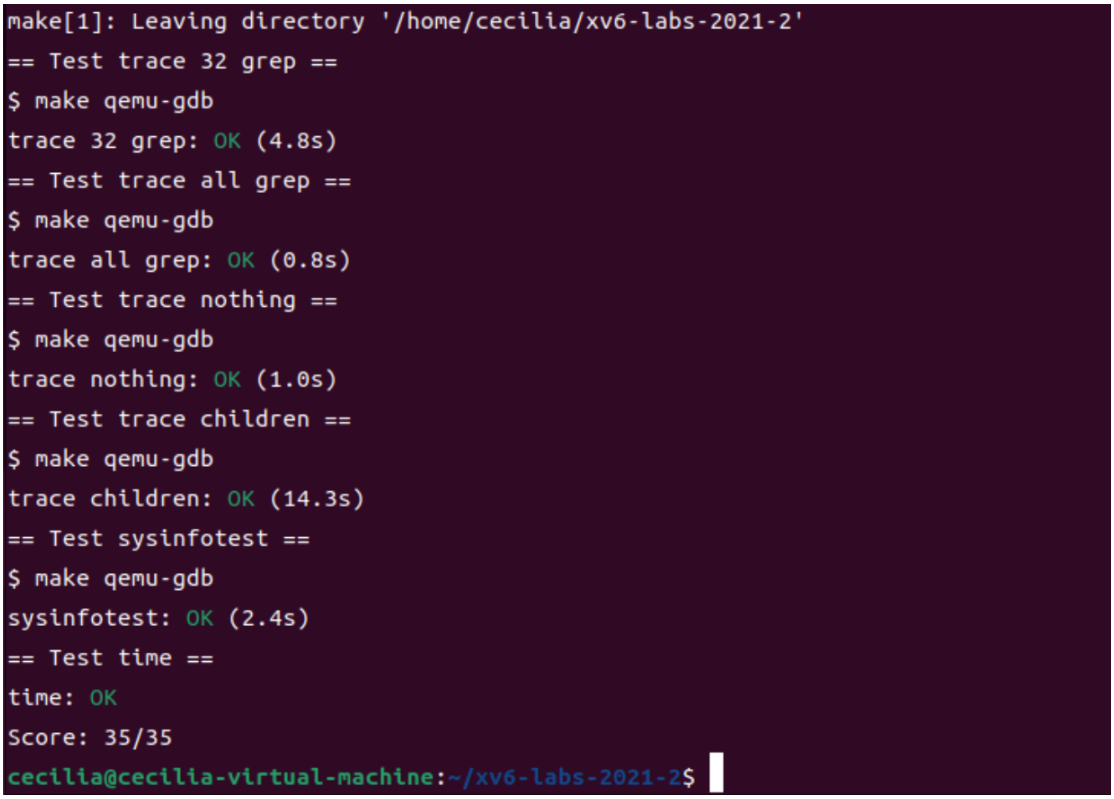
2.2.3 实验中遇到的问题和解决方案

用户态与内核 struct sysinfo 字段顺序和类型存在不一致，导致数据错乱。调整确保两端结构体定义完全相同，消除问题。

2.2.4 实验心得

在跨态数据传递需严格保证数据格式一致性与内存地址合法性，此外还意识到需要利用 copyout()函数将一块内存从内核态 copy 到用户态。

2.3 实验结果


make[1]: Leaving directory '/home/cecilia/xv6-labs-2021-2'
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (4.8s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (0.8s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.0s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (14.3s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (2.4s)
== Test time ==
time: OK
Score: 35/35
cecilia@cecilia-virtual-machine:~/xv6-labs-2021-2$

第 3 章 Lab: page tables

3.1 Speed up system calls

3.1.1 实验目的

理解用户态与内核态切换的开销，掌握通过“共享内存页”减少切换次数的优化思路。

3.1.2 实验步骤

1. 创建共享内存页：在内核初始化时（如 procinit()），为每个进程分配 1 页共享内存，将其同时映射到用户地址空间（如 USYSCALL）和内核地址空间；
2. 定义共享数据结构：在共享页中定义 struct usyscall，包含 pid、uptime 等需快速获取的系统信息字段；
3. 内核更新共享数据：修改内核相关模块（如进程调度 schedule()、时钟中断 timerintr()），实时更新共享页中的 pid（当前进程 ID）、uptime（系统运行时间）等数据；
4. 实现快速系统调用：在用户态编写 fastgetpid()、fastuptime()，直接读取共享页地址（USYSCALL）的数据，无需触发传统系统调用（避免 ecall 切换）；
5. 性能验证：对比传统系统调用与快速调用的执行时间（如循环调用计数耗时），验证优化效果。

3.1.3 实验中遇到的问题和解决方案

这部分需要查看和学习的源代码相对前面实验较多 ，初步进行时因为不了解不知道如何入手。待理解后，实际操作毕竟想象中简单。

3.1.4 实验心得

在该实验中，系统调用优化的核心是减少用户态与内核态的切换次数，共享内存通过“一次映射、多次读取”避免频繁 ecall 开销，适合高频简单调用。

3.2 Print a page table

3.2.1 实验目的

定义一个名为 vmprint()的函数实现页表打印功能，掌握内核态访问进程页表的方法，理解页表与进程地址空间的关联机制。

3.2.2 实验步骤

1. 添加打印函数声明：在 vm.h 声明 vmprint(pagetable_t pagetable)函数，用于遍历并打印页表
2. 实现页表遍历逻辑：从页目录（一级页表）开始，遍历每个页目录项，跳过无效项（PTE_V 未置位）；对有效页目录项，获取二级页表物理地址，遍历二级页表项，同样过滤无效项；打印页表层级（一级 / 二级）、虚拟地址、物理地址及 PTE 字段（如 PTE_R/PTE_W/PTE_U）
3. 调用打印函数：在 exec()（进程加载程序时）或 fork()（创建子进程时）中调用 vmprint，打印目标进程的页表
4. 验证输出结果：运行 xv6，执行测试程序（如 echo），查看页表打印日志，确认映射关系正确

3.2.3 实验中遇到的问题和解决方案

打印页表时，509 和 511 一个缺失，一个重复打印。后发现是遍历过程中出现问题，未正确判断 PTE_R/PTE_W/PTE_U 等标志位，导致打印错误。

3.2.4 实验心得

实验中页表打印的核心是按层级遍历有效页表项，需准确处理地址转换与标志位解析，避免内存访问错误，在此过程中需要理解二级页表的映射链。

3.3 Detecting which pages have been accessed

3.3.1 实验目的

实现页面访问检测功能（pgaccess()），熟悉内核中遍历页表、清除与检查访问位的操作逻辑。

3.3.2 实验步骤

1. 添加系统调用与数据结构：在 syscall.h 定义 SYS_pageaccess 编号，user.h 声明 pageaccess(void *addr, int *accessed)函数，用于检测指定页面是否被访问；
2. 实现内核检测逻辑：在 sys_pageaccess 函数中，通过 walk()函数（vm.c）遍历进程页表，找到目标虚拟地址对应的页表项（PTE）；

3. 检查 PTE 的 PTE_A 位：若置 1，说明页面已被访问，将 accessed 设为 1；反之设为 0；
4. 清除 PTE_A 位（pte &= ~PTE_A），为下一次检测做准备；
5. 用户态测试程序：编写测试程序，分配内存页并访问部分页面，调用 pageaccess()检测，打印结果验证是否正确识别访问状态；
6. 验证内核逻辑：运行 xv6，执行测试程序，检查页表项 PTE_A 位的置位与清除是否正常，确保检测结果准确。

3.3.3 实验中遇到的问题和解决方案

对 walk()的实验不够熟悉，通过查阅资料了解。

3.3.4 实验心得

在本实验中进一步加深了对页表理解，遍历页表时需严格区分用户态与内核态地址，避免越权访问。walk()函数是内核操作页表的核心工具，需理解其参数与返回值含义，从而更好地使用。

3.4 实验结果

```
make[1]: Leaving directory '/home/cecilia/xv6-labs-2021-3'
== Test pgtbltest ==
$ make qemu-gdb
(4.2s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.4s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
(57.4s)
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
cecilia@cecilia-virtual-machine:~/xv6-labs-2021-3$
```

第 4 章 Lab: traps

4.1 RISC-V assembly

4.1.1 实验目的

理解 RISC-V 架构的基础汇编指令集（如算术运算、加载存储、分支跳转），掌握指令格式与操作数规则；学会通过调试工具查看汇编指令执行过程，分析程序底层执行逻辑。

4.1.2 实验步骤

按要求回答。

4.1.3 实验中遇到的问题和解决方案

无。

4.1.4 实验心得

在实验中进一步了解 RISC-V 架构。

4.1 Backtrace

4.2.1 实验目的

在 kernel/printf.c 中实现 backtrace()函数并在 sys_sleep 中插入对这个函数的调用。

4.2.2 实验步骤

1. 添加 backtrace 函数声明：在 defs.h 声明 void backtrace(void)，在 proc.h 的进程控制块（struct proc）中添加 uint64 fp 字段（保存当前进程栈基址）
2. 实现栈帧遍历逻辑：在 backtrace 函数中，从当前进程的 fp（栈基址）开始，获取栈帧中保存的返回地址；过滤无效返回地址，打印有效地址。读取当前栈帧中保存的上一级栈基址，更新 fp 为该值，循环遍历直至 fp 为 0（栈底）。
3. 保存栈基址：在进程切换（switch 函数，swtch.S）时，将当前 fp 寄存器值存入进程 proc 的 fp 字段，确保切换后能正确追踪栈帧

4. 测试验证：在 sys_sleep 或 panic 等函数中调用 backtrace，运行 xv6 触发该函数，查看打印的函数调用链是否正确

4.2.3 实验中遇到的问题和解决方案

在实验中未正确判断栈底，导致遍历越界访问非法内存，后修改判断语句，在越界后停止遍历。

4.2.4 实验心得

Backtrace 的核心是利用栈帧链的结构性，这要求我们精准掌握 RISC-V 栈帧布局，避免因偏移错误导致内存访问异常。

4.3 Alarm

4.3.1 实验目的

实现用户态 Alarm 功能，理解 xv6 中定时器中断（Timer Interrupt） 的触发与处理流程，掌握内核态向用户态发送“闹钟信号”的机制。

4.3.2 实验步骤

1. 扩展进程控制块：在 proc.h 的 struct proc 中添加闹钟相关字段（alarm_interval：间隔时间、alarm_handler：处理函数地址、alarm_ticks：计时计数器、alarm_in_handler：是否在处理函数中）；
2. 实现 sigalarm 系统调用：在 syscall.c 添加 sys_sigalarm，接收用户传入的 “间隔时间” 与 “处理函数地址”，赋值给进程控制块对应字段，初始化 alarm_ticks 为 0；
3. 处理定时器中断：修改 timerintr（trap.c），对每个活跃进程，若已设置闹钟（alarm_interval>0），则 alarm_ticks 累加；当 alarm_ticks 达到 alarm_interval 时，触发闹钟处理；
4. 切换到用户态处理函数：在 usertrap（trap.c）中，若触发闹钟且不在处理函数中（!alarm_in_handler），保存当前用户态上下文（ra、sp 等），设置 alarm_in_handler=1，将 pc 指向 alarm_handler；
5. 实现 sigreturn 系统调用：用户态处理函数执行完毕后，调用 sigreturn 恢复之前保存的上下文，设置 alarm_in_handler=0，重置 alarm_ticks 为 0，让进程继续执行原逻辑。

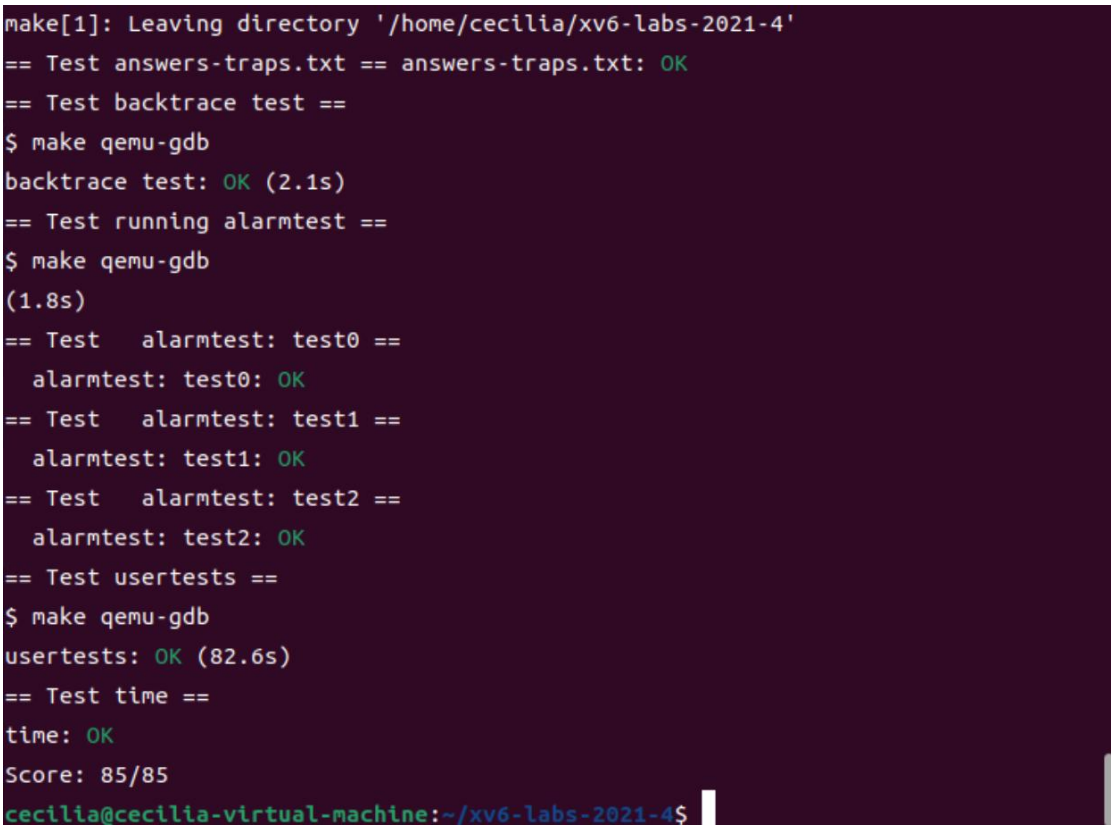
4.3.3 实验中遇到的问题和解决方案

不太清楚这一部分具体需要实现什么，以及该怎么实现，深入了解后得知许修改的部分后，还是挺好实现的。

4.3.4 实验心得

Alarm 功能的核心是定时器中断触发、上下文切换、信号处理，实现这一部分主要还是需要理解系统函数的运行方式。

4.4 实验结果



第 5 章 Lab: Copy-on-Write Fork for xv6

5.1 Implement copy-on write

5.1.1 实验目的

理解写时复制（Copy-On-Write, COW） 机制的核心原理，掌握通过延迟页复制优化 fork()系统调用内存开销的方法（实验网页有详细介绍）。

5.1.2 实验步骤

1. 扩展物理页与进程结构：在 kalloc.c 的物理页元数据（如 struct run）中添加 refcnt（引用计数）字段，记录页被共享的进程数；在 proc.h 的 struct proc 中，标记页表项是否为 COW 共享页（可通过页表项保留位或单独标记）；
2. 修改 fork()实现 COW 页共享：fork()时不再复制父进程物理页，而是让父子进程页表映射同一物理页，同时清除双方页表项的 PTE_W 位（设为只读）；递增共享物理页的 refcnt（父子共享时 refcnt=2）；
3. 处理 COW 缺页中断：在 usertrap（trap.c）中，判断缺页中断是否因写入 COW 只读页触发（r_scause()为存储页错误，且页表项为 COW 共享）；若触发 COW 缺页：分配新物理页，复制原页数据，更新当前进程页表为新页（恢复 PTE_W），递减原页 refcnt（若 refcnt 为 0 则释放原页）；
4. 完善引用计数管理：kfree()时先递减 refcnt，仅当 refcnt 为 0 时才真正释放物理页；kalloc()时初始化 refcnt=1；处理进程退出（exit()）时，遍历页表递减所有共享页的 refcnt，避免内存泄漏。

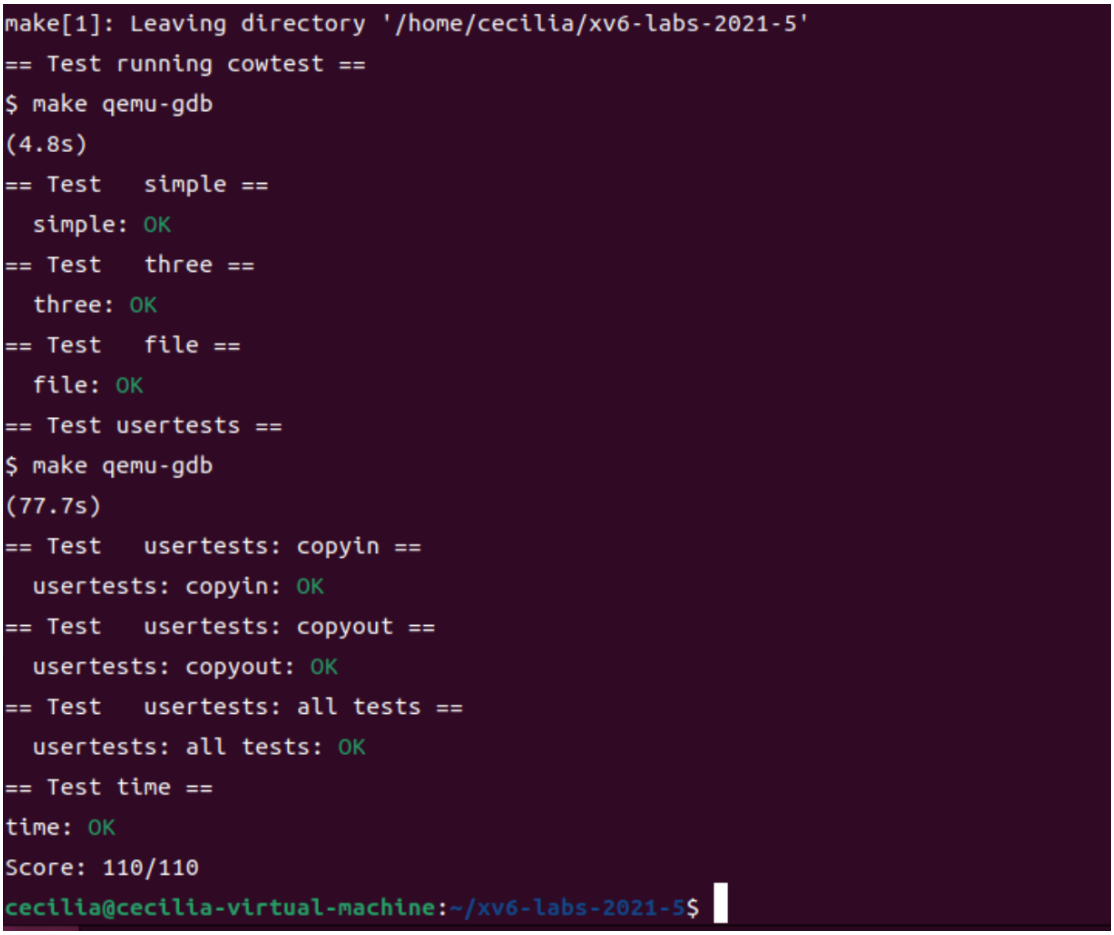
5.1.3 实验中遇到的问题和解决方案

除注意页表项权限设置外没有什么问题。

5.1.4 实验心得

本实验中 COW 的核心是延迟复制，其通过共享只读页减少 fork()时的内存开销，仅在实际写入时复制页。

5.2 实验结果


make[1]: Leaving directory '/home/cecilia/xv6-labs-2021-5'
== Test running cowtest ==
$ make qemu-gdb
(4.8s)
== Test   simple ==
  simple: OK
== Test   three ==
  three: OK
== Test   file ==
  file: OK
== Test usertests ==
$ make qemu-gdb
(77.7s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyout ==
  usertests: copyout: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 110/110
cecilia@cecilia-virtual-machine:~/xv6-labs-2021-5$

第 6 章 Lab: Multithreading

6.1 Uthread: switching between threads

6.1.1 实验目的

实现用户态线程切换机制，熟悉线程控制块（TCB）设计与线程调度的基础流程。

6.1.2 实验步骤

1. 设计线程控制块（TCB）：在 uthread.c 中定义 struct thread，包含线程状态（RUNNING/READY）、栈指针（sp）、线程函数（func）等字段，维护线程链表 all_threads；
2. 实现线程创建：编写 thread_create(func, arg)，为新线程分配栈空间，初始化栈帧（压入 arg、线程退出函数地址、func 地址），设置 TCB 的 sp 指向栈顶，将线程加入 READY 队列；
3. 实现线程切换上下文：编写汇编函数 thread_switch（uthread_switch.S），保存当前线程的寄存器（ra、sp、s0-s11）到其 TCB，从目标线程 TCB 恢复寄存器，完成上下文切换；
4. 实现线程调度：编写 schedule()函数，遍历 READY 队列，选择下一个线程，调用 thread_switch 切换；在 thread_yield()中触发调度，让当前线程放弃 CPU，进入 READY 队列；
5. 测试验证：编写多线程测试程序（如 3 个线程循环打印），调用 thread_create 创建线程，thread_yield 触发切换，验证线程是否交替执行。

6.1.3 实验中遇到的问题和解决方案

在实验中需注意在 thread_switch 中保存 s0-s11，避免线程切换后局部变量错乱。

6.1.4 实验心得

在理论课中了解后，实验中更加深入了解 TCB 是线程管理的核心数据结构，线程状态、栈指针等信息的准确维护，是调度器正确工作的基础。

6.2 Using threads

6.2.1 实验目的

使用哈希表的线程和锁，理解线程间共享资源竞争问题，掌握通过互斥锁（mutex）、条件变量（condition variable）解决同步问题的方法。

6.2.2 实验步骤

1. 搭建线程环境：基于 Uthread 实验的线程库，扩展 mutex_init（初始化锁）、mutex_lock（加锁）、mutex_unlock（解锁）等同步接口；
2. 实现共享资源场景：定义共享数据结构（如环形缓冲区），创建生产者线程（向缓冲区写入数据）与消费者线程（从缓冲区读取数据）；
3. 添加同步控制：用互斥锁保护共享缓冲区的访问，避免多线程同时读写导致数据错乱；用条件变量处理 “缓冲区满”（生产者等待）与 “缓冲区空”（消费者等待）的阻塞逻辑；
4. 测试验证：启动多个生产者与消费者线程，观察是否无数据丢失、重复读取或死锁，验证同步机制有效性。

6.2.3 实验中遇到的问题和解决方案

哈希表（Hash Table）其实也叫散列表，是一个数据结构。哈希表本质上就是一个数组，只不过数组存放的是单一的数据，而哈希表中存放的是键值对（key - value pair）。key 通过哈希函数（hash function）得到数组的索引，进而存取索引位置的值。不同的 key 通过哈希函数可能得到相同的索引值，此时，产生了哈希碰撞。通过在数组中插入链表或者二叉树，可以解决哈希碰撞问题。

需注意多线程哈希表的安全问题。

6.2.4 实验心得

在实验中进一步理解线程同步的核心是控制临界区访问，互斥锁确保 “同一时间仅一个线程进入临界区”，条件变量解决 “线程等待特定条件” 的阻塞问题，二者需配合使用。

6.3 Barrier

6.3.1 实验目的

理解屏障（Barrier） 的同步原理，掌握多线程在执行到指定点时 “等待所有线程到达后再继续” 的实现逻辑。

6.3.2 实验步骤

1. 设计屏障数据结构：定义 struct barrier，包含 mutex（保护屏障状态）、cv（线程等待条件）、num_threads（总线程数）、arrived（已到达屏障的线程数）、round（屏障轮次，实现复用）；
2. 实现屏障初始化：编写 barrier_init(b, n)，初始化 mutex、cv，设置 num_threads = n，arrived = 0，round = 0；
3. 实现屏障等待逻辑：编写 barrier_wait(b)；
4. 解锁 mutex，线程继续执行后续逻辑。

6.3.3 实验中遇到的问题和解决方案

最开始不太理解 Barrier 的具体作用，后面了解相关概念和工具后，解决了之前的问题。

6.3.4 实验心得

实验中理解到屏障的核心是计数等待以及广播唤醒，需通过 arrived 计数判断是否所有线程到达，cv_broadcast 确保唤醒全部等待线程，避免遗漏。

6.4 实验结果

```
make[1]: Leaving directory '/home/cecilia/xv6-labs-2021-6'
== Test uthread ==
$ make qemu-gdb
uthread: OK (3.1s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/cecilia/xv6-labs-2021-6'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/cecilia/xv6-labs-2021-6'
ph_safe: OK (7.9s)
== Test ph_fast == make[1]: Entering directory '/home/cecilia/xv6-labs-2021-6'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/cecilia/xv6-labs-2021-6'
ph_fast: OK (17.7s)
== Test barrier == make[1]: Entering directory '/home/cecilia/xv6-labs-2021-6'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/cecilia/xv6-labs-2021-6'
barrier: OK (3.4s)
== Test time ==
time: OK
Score: 60/60
cecilia@cecilia-virtual-machine:~/xv6-labs-2021-6$
```

第 7 章 Lab: locks

7.1 Memory allocator

7.1.1 实验目的

实现内存分配（kalloc）与释放（kfree）功能，熟悉物理内存页的管理逻辑（页帧标记、空闲状态维护）

7.1.2 实验步骤

1. 设计空闲内存数据结构：选择空闲链表（如单链表、双向链表）或伙伴系统，定义页帧元数据（如 struct run，包含下一页指针、引用计数等）；
2. 实现内存初始化：在 kinit 函数中，扫描物理内存区域（PHYSTOP 以下），将空闲页帧加入空闲链表，初始化分配器状态；
3. 实现 kalloc 函数：遍历空闲链表，找到满足大小的页帧（通常为 1 页，4096 字节），从链表中移除并返回其物理地址，标记为已分配；
4. 实现 kfree 函数：接收物理地址，将对应页帧标记为空闲，清理页内数据（避免敏感信息残留），并插入空闲链表（按需合并相邻空闲页减少碎片）；
5. 测试验证：通过内核模块（如创建进程、分配内核栈）调用 kalloc/kfree，检查内存是否正确分配与回收，无内存泄漏或重复释放。

7.1.3 实验中遇到的问题和解决方案

所有锁必须指定以“kmem”开头。

在频繁分配/释放后，空闲页分散成小块，无法满足连续页需求。需要检查前后相邻页是否空闲，合并为大页后再插入空闲链表。

7.1.4 实验心得

在实验总，内存分配器的核心是高效管理空闲页帧，数据结构的选择直接影响分配和释放的效率，没有绝对好的选择，只有相对平衡的选择。

7.2 Buffer cache

7.2.1 实验目的

实现缓冲区的分配、查找、同步（读 / 写磁盘）与回收机制，熟悉 xv6 中磁盘块与内存缓冲区的映射关系

7.2.2 实验步骤

1. 设计缓冲区数据结构：定义 struct buf（包含磁盘块号、数据缓冲区、状态标志（如 VALID/DIRTY）、锁、LRU 链表指针），维护全局缓存链表与空闲缓冲区池；
2. 实现缓冲区查找与分配；
3. 实现缓冲区读写与同步；
4. 实现 LRU 替换算法：当缓存满时，淘汰 LRU 链表尾部的“最近最少使用”缓冲区（需确保缓冲区未被占用），为新块腾出空间。

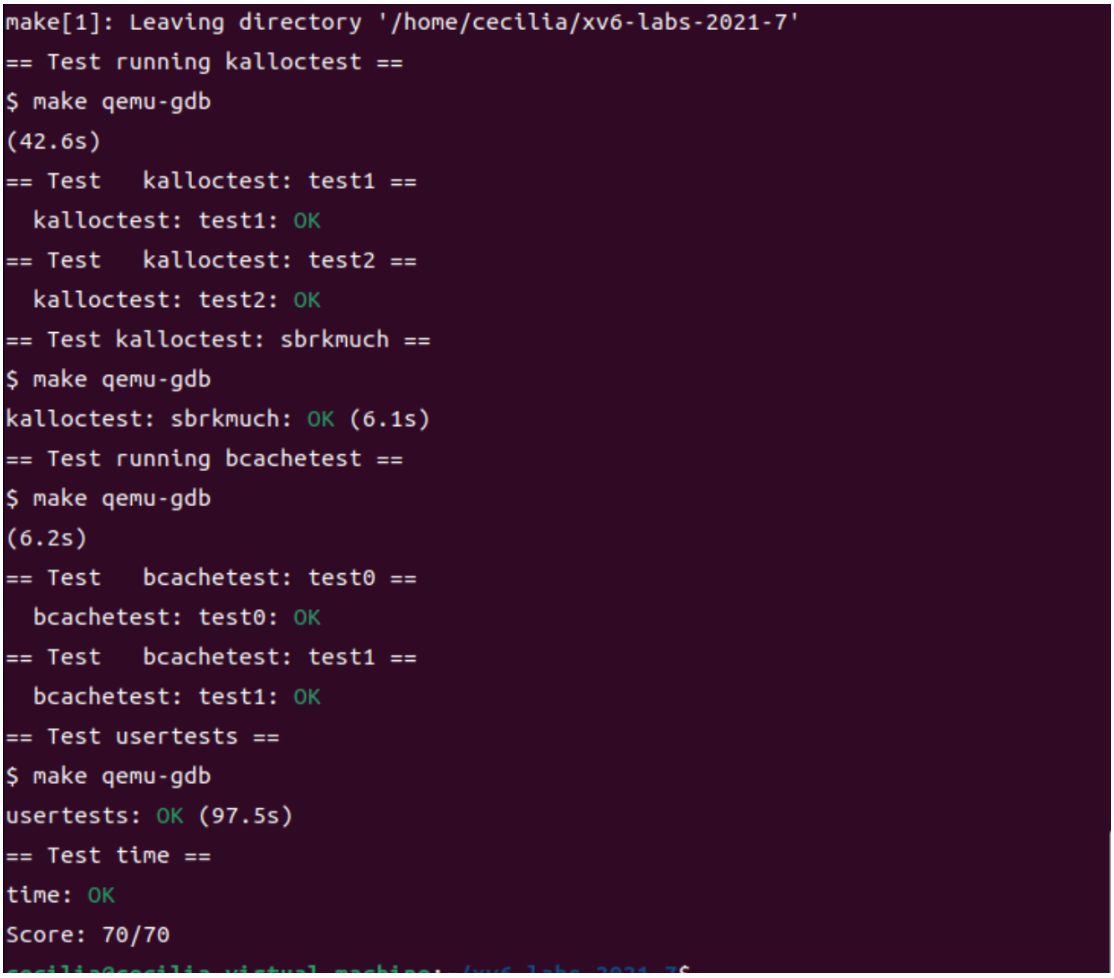
7.2.3 实验中遇到的问题和解决方案

多线程操作时需注意互斥锁。

7.2.4 实验心得

Buffer Cache 为以内存换 I/O 性能存在，通过缓存热点数据减少磁盘访问（机械磁盘 I/O 远慢于内存），是文件系统性能优化的关键组件。

7.3 实验结果


make[1]: Leaving directory '/home/cecilia/xv6-labs-2021-7'
== Test running kalloctest ==
$ make qemu-gdb
(42.6s)
== Test   kalloctest: test1 ==
  kalloctest: test1: OK
== Test   kalloctest: test2 ==
  kalloctest: test2: OK
== Test kalloctest: sbrkmuch ==
$ make qemu-gdb
kalloctest: sbrkmuch: OK (6.1s)
== Test running bcachetest ==
$ make qemu-gdb
(6.2s)
== Test   bcachetest: test0 ==
  bcachetest: test0: OK
== Test   bcachetest: test1 ==
  bcachetest: test1: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (97.5s)
== Test time ==
time: OK
Score: 70/70

第 8 章 file system

8.1 Large files

8.1.1 实验目的

实现用户态线程切换机制，熟悉线程控制块（TCB）设计与线程调度的基础流程。

8.1.2 实验步骤

1. 修改 inode 数据结构：在 fs.h 的 struct dinode 中，扩展块地址数组：保留原有的直接块（如 12 个），增加二级间接块指针（原一级间接块基础上，再嵌套一层间接块）；调整 inode 中块地址的索引规则：直接块索引 0~11，一级间接块索引 12，二级间接块索引 13；
2. 实现多级间接块的地址解析：编写 bmap 函数（文件块号→磁盘块号映射）：若块号在直接块范围，直接返回地址；若在一级 / 二级间接块范围，递归读取间接块中的地址，直至找到目标磁盘块号；编写 itrunc 函数（回收文件数据块）：除回收直接块外，需逐层释放一级、二级间接块及其指向的数据块，避免内存泄漏；
3. 适配文件读写逻辑：确保 readi（读文件）、writei（写文件）函数能正确处理多级间接块映射的磁盘块，在分配新块时支持间接块的创建（如无间接块则先分配间接块）。

8.1.3 实验中遇到的问题和解决方案

需注意混淆一级、二级间接块的地址偏移。

8.1.4 实验心得

这一部分实验进一步加深了我对文件系统的理解，文件系统的块映射逻辑具有严格的层级性，每一层间接块都是对下一层地址的 “索引扩展”，需精准处理地址计算与块分配的依赖关系。

8.2 Symbolic links

8.2.1 实验目的

实现符号链接的创建（symlink）、解析（路径转换）与删除逻辑，熟悉 xv6 文件系统中 inode 类型的扩展（新增符号链接类型）。

8.2.2 实验步骤

1. 扩展 inode 类型与数据结构：在 fs.h 的 enum inode_type 中新增 T_SYMLINK（符号链接类型），修改 struct dinode，确保 inode 能存储符号链接的目标路径（利用原数据块存储路径字符串）；
2. 实现 symlink 系统调用：编写 sys_symlink 函数：接收 “符号链接名” 与 “目标文件路径”，创建类型为 T_SYMLINK 的 inode，将目标路径写入 inode 数据块，完成符号链接创建；
3. 适配路径解析与文件打开逻辑：修改 namei 函数（路径解析核心）：当遇到 T_SYMLINK 类型的 inode 时，读取其目标路径，递归解析该路径（需限制递归深度，避免循环链接）；
4. 调整 open 函数：确保打开符号链接时，自动跟随到目标文件（而非打开链接本身），支持 O_NOFOLLOW 标志（可选，用于禁止跟随）。

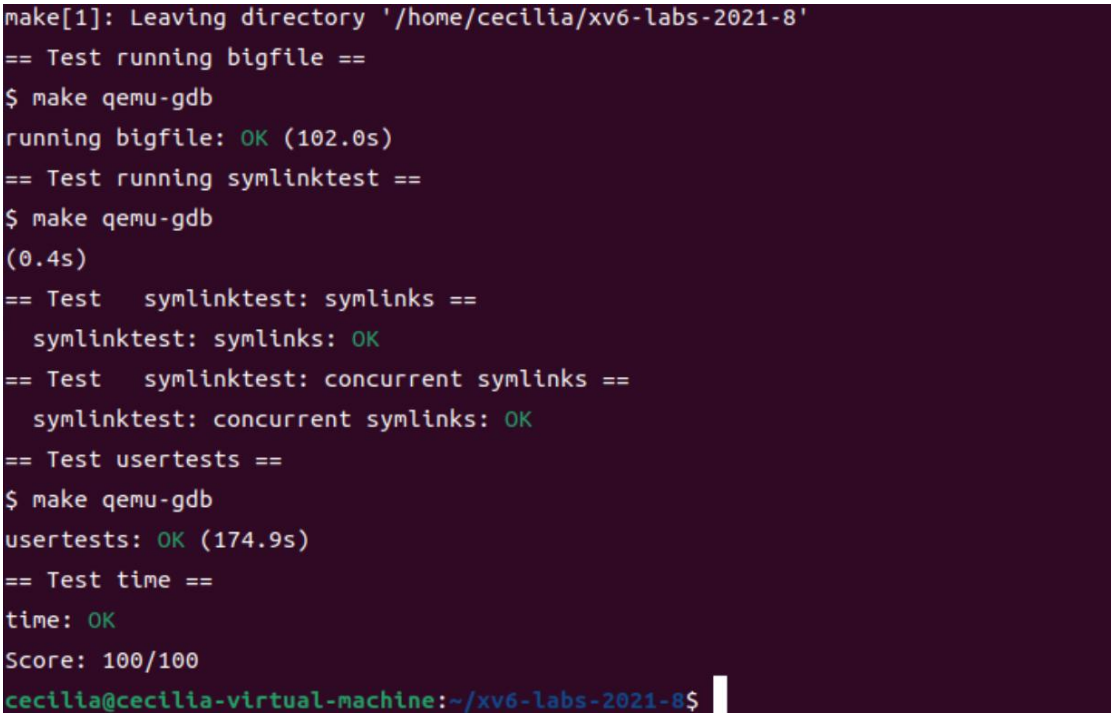
8.2.3 实验中遇到的问题和解决方案

实验中出现过很多问题，部分不清楚产生原因，猜测是路径解析时无限递归造成的。

8.2.4 实验心得

在实验中应分清符号链接与硬链接，两者的区别在于：符号链接依赖路径解析，硬链接依赖 inode 编号。

8.3 实验结果


make[1]: Leaving directory '/home/cecilia/xv6-labs-2021-8'
== Test running bigfile ==
$ make qemu-gdb
running bigfile: OK (102.0s)
== Test running symlinktest ==
$ make qemu-gdb
(0.4s)
== Test   symlinktest: symlinks ==
  symlinktest: symlinks: OK
== Test   symlinktest: concurrent symlinks ==
  symlinktest: concurrent symlinks: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (174.9s)
== Test time ==
time: OK
Score: 100/100
cecilia@cecilia-virtual-machine:~/xv6-labs-2021-8$