操作系统课程设计

Xv6实验报告

Lab2 system calls

|  |  |
| --- | --- |
| Name | 朱从周 |
| ID | 2351893 |
| Number | 42028701 |
| major | 软件工程 |



同济大学

Tongji University

1. **环境搭建**
2. 下载VMware虚拟机，并且下载 Ubuntu 镜像并安装 Ubuntu 系统。
3. 启动Ubuntu，安装本项目所需的所有软件，运行：

$ sudo apt-get update && sudo apt-get upgrade

$ sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu

1. 下载xv6内核源码

$ git clone git://github.com/mit-pdos/xv6-riscv.git

1. 更新镜像源

$ sudo nano /etc/apt/sources.list

$ sudo apt-get update

1. 获取源码

$ git clone git://g.csail.mit.edu/xv6-labs-2021

$ cd xv6-labs-2021

$ git checkout syscall

**1. 实验目的**

本实验旨在帮助了解系统调用跟踪的实现，以及如何修改 xv6 操作系统以添加新功能。我们需要添加一个有助于调试的新的trace系统调用。该功能包括创建一个名为 trace的系统调用，并将整数 "mask"作为参数。 "mask"的位数表示要跟踪哪些系统调用。通过实验，熟悉内核级编程，包括修改进程结构、处理系统调用和管理跟踪掩码。

此外，在本实验中将添加一个系统调用 sysinfo，用于收集运行系统的信息。系统调用需要一个参数：指向 struct sysinfo 的指针（参见 kernel/sysinfo.h）。内核应填写该结构体的字段：freemem 字段应设置为可用内存的字节数，nproc 字段应设置为状态不是 UNUSED 的进程数。

**2. 实验内容**

**System call tracing**

**1、函数声明**

在 Makefile 的 UPROGS 中添加 $U/\_trace

在 user/user.h中添加 trace 系统调用原型：

int trace(int);

在 user/usys.pl 脚本中添加 trace 对应的 entry

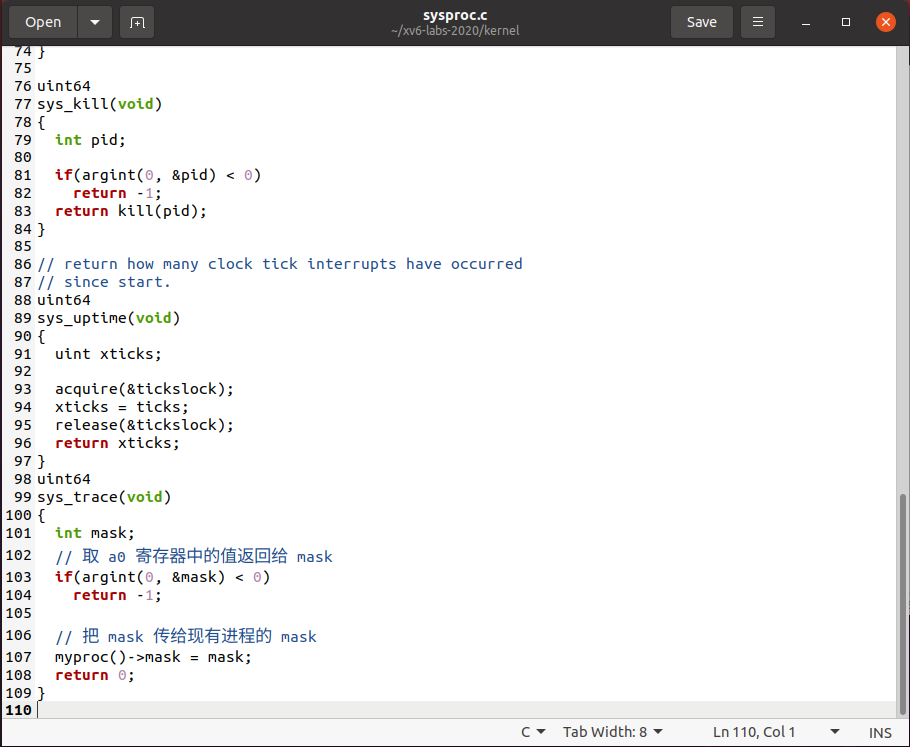
entry("trace");

在 kernel/syscall.h 中添加 trace 的系统调用号

#define SYS\_trace 22 /

**2、编写 trace 的系统调用函数**

在kernel/sysproc.c中添加一个sys\_trace()函数，通过在proc结构（参见kernel/proc.h）中的一个新变量tracemask来存储参数来实现新的系统调用。该函数使用 argint(0, &mask) 获取系统调用参数，并将其存储在当前进程的 proc 结构中的新变量 tracemask 中。

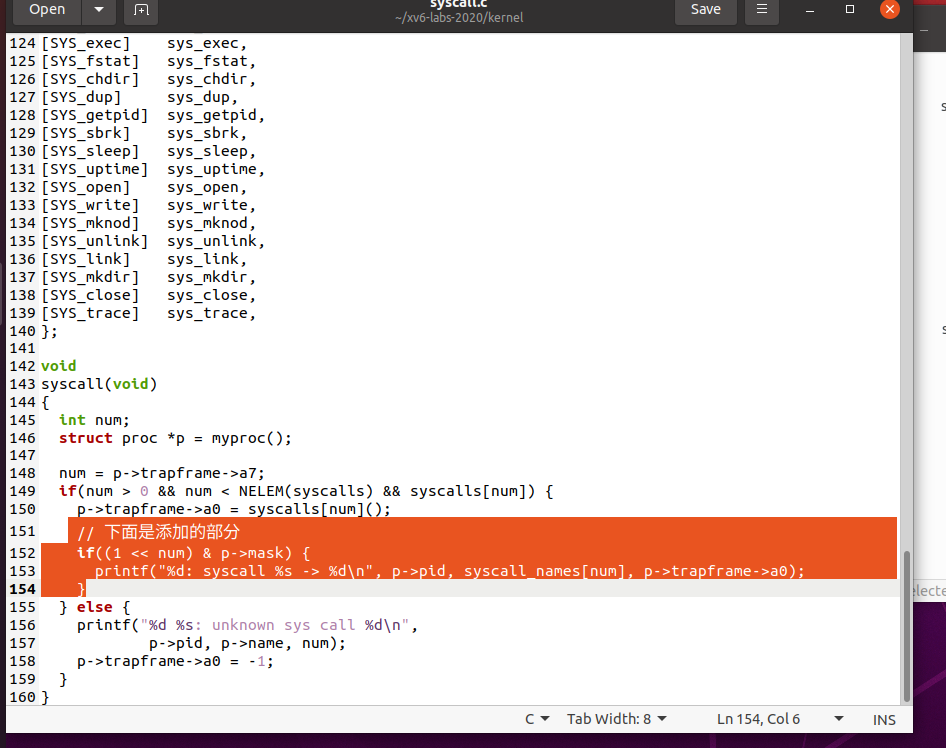


修改 kernel/proc.c 中的 fork() 函数，确保父进程的跟踪掩码能被复制到子进程中。

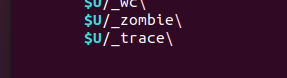


在 kernel/syscall.c 中修改 syscall() 函数，根据跟踪掩码打印相关的系统调用跟踪信息。

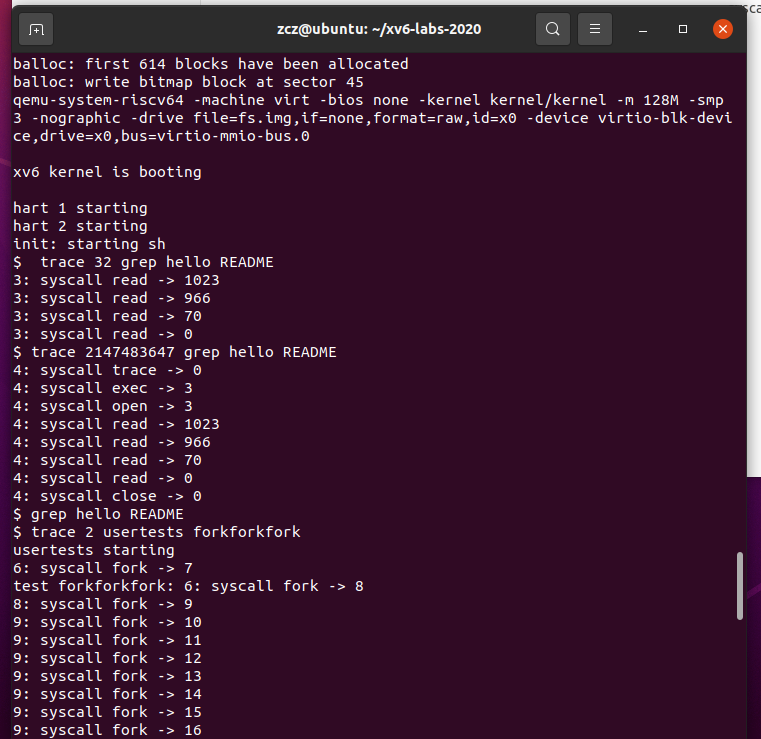
更新 syscalls 数组和 syscalls\_name 数组，添加对 sys\_trace() 函数的引用。



在 Makefile 的 UPROGS 部分添加 $U/\_trace，确保新的用户程序 trace 能被正确编译和链接。

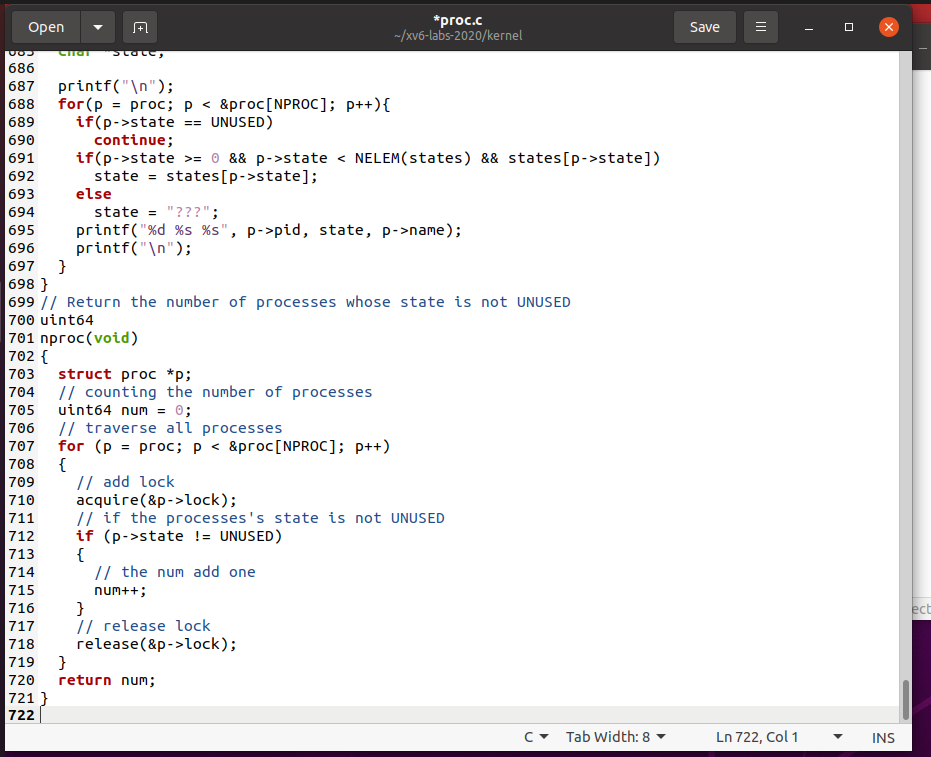


测试结果：

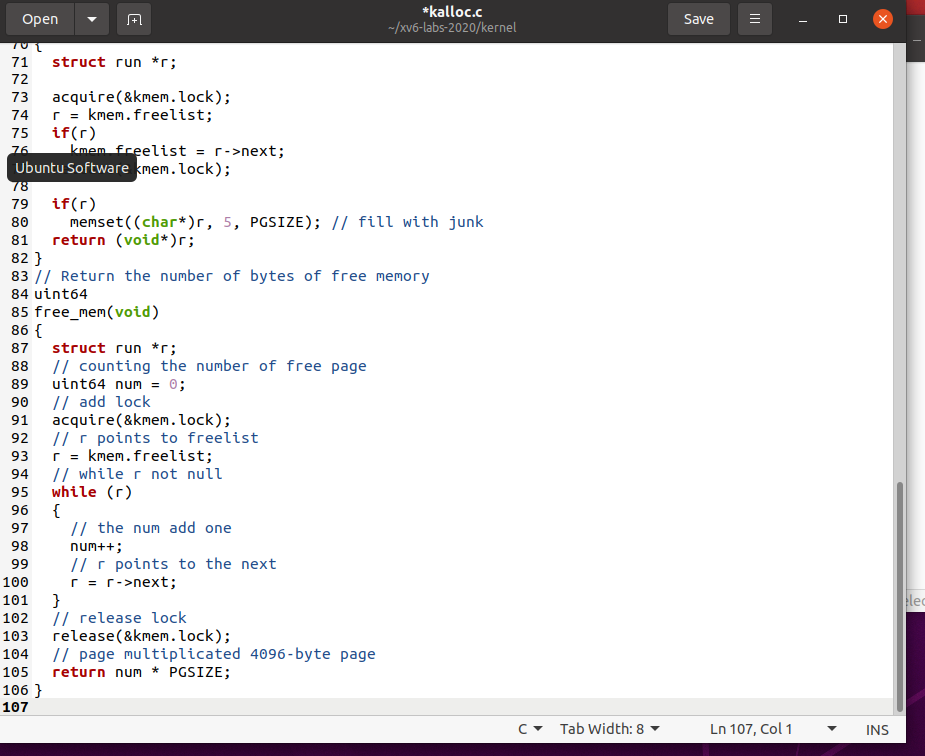
****

**Sysinfo**

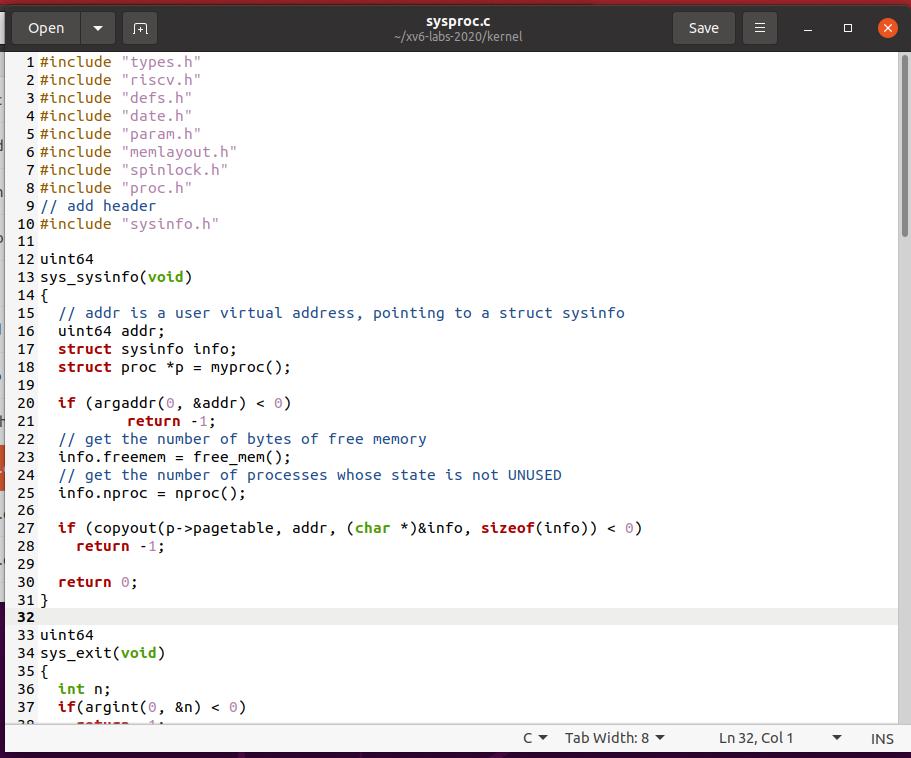
当然，通过 proc 结构体的定义，我们知道使用进程状态时必须加锁，我们在 **kernel/proc.c** 中新增函数 nproc 如下，通过该函数以获取可用进程数目：



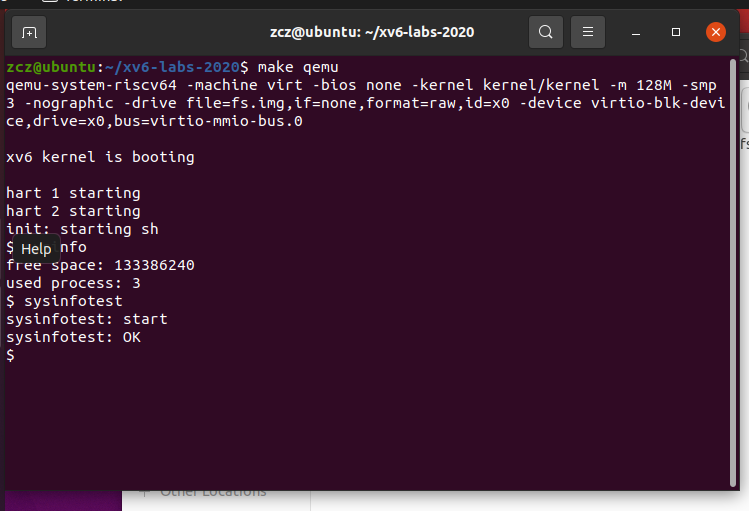
这里把从 end (内核后的第一个地址) 到 PHYSTOP (KERNBASE + 128\*1024\*1024) 之间的物理空间以 PGSIZE 为单位全部初始化为 1 ，然后每次初始化一个 PGSIZE 就把这个页挂在了 kmem.freelist 上，所以 kmem.freelist 永远指向最后一个可用页，那我们只要顺着这个链表往前走，直到 NULL 为止。所以我们就可以在 kernel/kalloc.c 中新增函数 free\_mem ，以获取空闲内存数量：

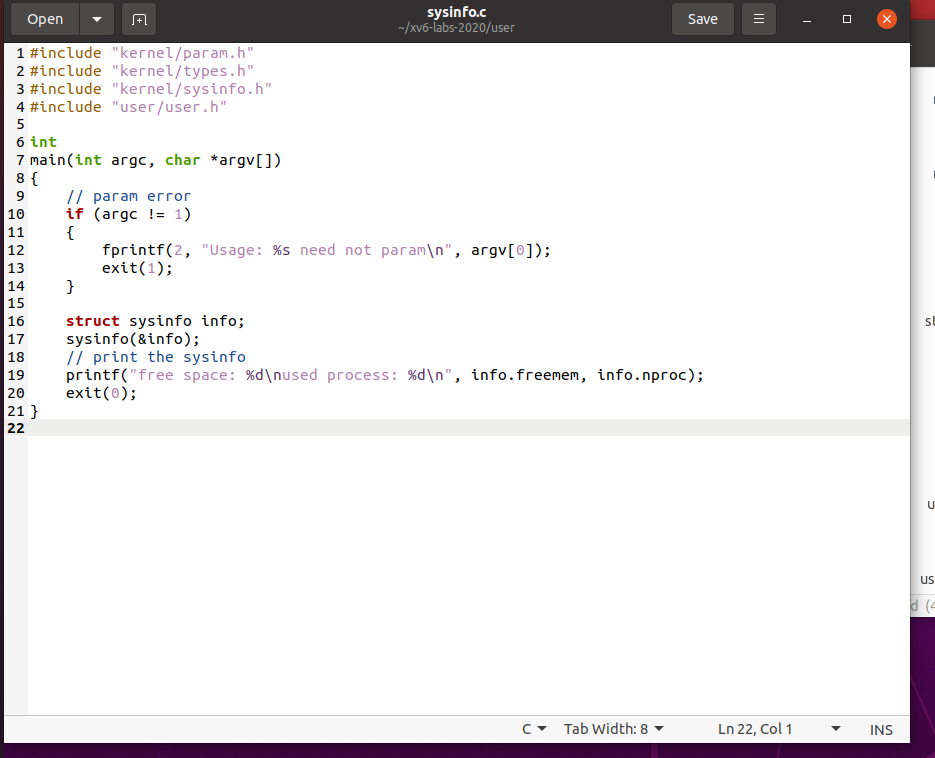


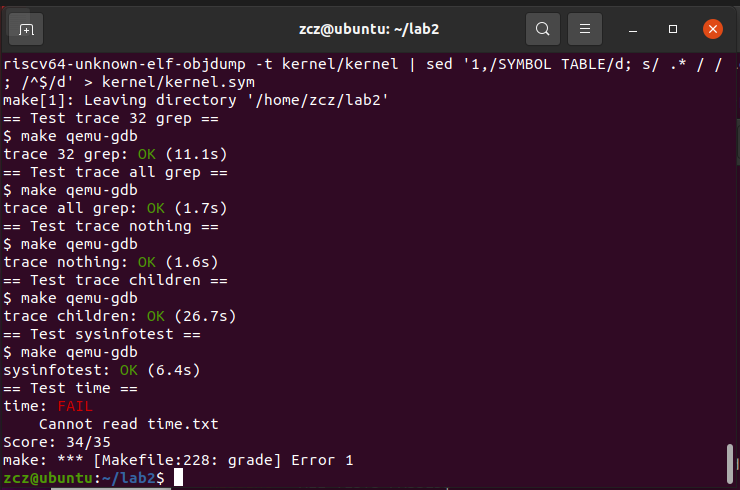
我们知道该函数其实就是把在内核地址 src 开始的 len 大小的数据拷贝到用户进程 pagetable 的虚地址 dstva 处，所以 sys\_sysinfo 函数实现里先用 argaddr 函数读进来我们要保存的在用户态的数据 sysinfo 的指针地址，然后再把从内核里得到的 sysinfo 开始的内容以 sizeof(info) 大小的的数据复制到这个指针上。模仿上面的例子，我们在 kernel/sysproc.c 文件中添加 sys\_sysinfo 函数的具体实现如下：



最后在 **user** 目录下添加一个 **sysinfo.c** 用户程序：







**3. 实验中遇到的问题和解决方法**

本次实验的核心在于收集系统运行的信息，比如收集可用内存的数量和进程数，因此首先遇到的困难在于要如何根据现有的源码提取出可供我们利用的参数，于是我参考了kalloc() 和 kfree() 等几个函数，可以看到内核通过 kmem.freelist 的一个链表维护未使用的内存，但是我不能遍历获取了可用的链表节点就认为任务完成了，我一开始忽视了链表的每个结点还对应了页表大小(PGSIZE)，PGSIZE的宏定义在源码中也有提示，所以我需要最后乘上这个页表大小才是真正获得了可用内存数。

**3. 实验心得**

在完成xv6操作系统的Lab 2，特别是关于系统调用跟踪（System Call Tracing）和系统信息（Sysinfo）的部分后，我获得了许多关于操作系统内核如何运作以及系统调用如何被实现和管理的深刻见解。

首先，系统调用跟踪的部分使我深入理解了系统调用的内部机制。通过实现一个新的系统调用trace，我学会了如何在内核中添加新的功能，并通过位掩码控制对特定系统调用的跟踪。这个过程不仅涉及到编码技巧，还需要对xv6内核的进程管理和系统调用处理流程有更深入的了解。实验中，我面临的最大挑战是确保跟踪功能正确无误地集成到现有的系统调用框架中。我需要确保新添加的代码不会影响到系统的其他部分，并且能够稳定运行。

对于sysinfo部分，通过添加一个能够返回系统运行信息的系统调用，我学习了如何从内核空间向用户空间传递信息。这不仅提高了我的系统编程能力，还加深了我对操作系统如何管理和监控其资源的理解。实现这一系统调用的过程中，我需要详细了解xv6内核如何管理进程、内存等资源，以便准确地收集和返回所需的信息。