Xv6实验报告

丁桢垚

2025

目录

[**环境搭建** 5](#_Toc206756950)

[**搭建步骤** 5](#_Toc206756951)

[安装虚拟机 5](#_Toc206756952)

[配置环境 5](#_Toc206756953)

[**Lab: Xv6 and Unix utilities** 5](#_Toc206756954)

[**Boot xv6 (easy)** 5](#_Toc206756955)

[1）实验目的 5](#_Toc206756956)

[2）实验内容 6](#_Toc206756957)

[3）实验中遇到的问题及解决方法 7](#_Toc206756958)

[4）实验心得 7](#_Toc206756959)

[**sleep (**easy**)** 7](#_Toc206756960)

[1）实验目的 7](#_Toc206756961)

[2）实验步骤 7](#_Toc206756962)

[3）实验中遇到的问题及解决方法 9](#_Toc206756963)

[4）实验心得 9](#_Toc206756964)

[**pingpong (**easy**)** 9](#_Toc206756965)

[1）实验目的 9](#_Toc206756966)

[2）实验步骤 9](#_Toc206756967)

[3）实验中遇到的问题及解决方法 11](#_Toc206756968)

[4）实验心得 11](#_Toc206756969)

[**primes (**moderate**)/(**hard**)** 11](#_Toc206756970)

[1）实验目的 11](#_Toc206756971)

[2）实验步骤 12](#_Toc206756972)

[3）实验中遇到的问题及解决方法 14](#_Toc206756973)

[4）实验心得 14](#_Toc206756974)

[**find (**moderate**)** 14](#_Toc206756975)

[1）实验目的 14](#_Toc206756976)

[2）实验步骤 15](#_Toc206756977)

[3）实验中遇到的问题及解决方法 17](#_Toc206756978)

[4）实验心得 17](#_Toc206756979)

[**xargs (**moderate**)** 17](#_Toc206756980)

[1）实验目的 17](#_Toc206756981)

[2）实验步骤 17](#_Toc206756982)

[3）实验中遇到的问题及解决方法 20](#_Toc206756983)

[4）实验心得 20](#_Toc206756984)

[**Lab: system calls** 21](#_Toc206756985)

[**Using gdb (easy)** 21](#_Toc206756986)

[1）实验目的 21](#_Toc206756987)

[2）实验步骤 21](#_Toc206756988)

[3）实验中遇到的问题及解决方法 22](#_Toc206756989)

[4）实验心得 22](#_Toc206756990)

[**System call tracing (moderate)** 22](#_Toc206756991)

[1）实验目的 22](#_Toc206756992)

[2）实验步骤 23](#_Toc206756993)

[3）实验中遇到的问题及解决方法 25](#_Toc206756994)

[4）实验心得 25](#_Toc206756995)

[**Sysinfo (moderate)** 26](#_Toc206756996)

[1）实验目的 26](#_Toc206756997)

[2）实验步骤 26](#_Toc206756998)

[3）实验中遇到的问题及解决方法 27](#_Toc206756999)

[4）实验心得 27](#_Toc206757000)

[**Lab: page tables** 28](#_Toc206757001)

[**Speed up system calls (easy)** 28](#_Toc206757002)

[1）实验目的 28](#_Toc206757003)

[2）实验步骤、 28](#_Toc206757004)

[3）实验中遇到的问题及解决方法 29](#_Toc206757005)

[4）实验心得 29](#_Toc206757006)

[**Print a page table (easy)** 30](#_Toc206757007)

[1）实验目的 30](#_Toc206757008)

[2）实验步骤 30](#_Toc206757009)

[3）实验中遇到的问题及解决方法 31](#_Toc206757010)

[4）实验心得 31](#_Toc206757011)

[**Detect which pages have been accessed (hard)** 32](#_Toc206757012)

[1）实验目的 32](#_Toc206757013)

[2）实验步骤 32](#_Toc206757014)

[3）实验中遇到的问题及解决方法 34](#_Toc206757015)

[4）实验心得 34](#_Toc206757016)

[**Lab: traps** 34](#_Toc206757017)

[**RISC-V assembly (easy)** 34](#_Toc206757018)

[1）实验目的 34](#_Toc206757019)

[2）实验步骤 34](#_Toc206757020)

[3）实验中遇到的问题及解决方法 35](#_Toc206757021)

[4）实验心得 36](#_Toc206757022)

[**Backtrace (moderate)** 36](#_Toc206757023)

[1）实验目的 36](#_Toc206757024)

[2）实验步骤 36](#_Toc206757025)

[3）实验中遇到的问题及解决方法 38](#_Toc206757026)

[4）实验心得 38](#_Toc206757027)

[**Alarm (**hard**)** 38](#_Toc206757028)

[1）实验目的 38](#_Toc206757029)

[2）实验步骤 38](#_Toc206757030)

[3）实验中遇到的问题及解决方法 42](#_Toc206757031)

[4）实验心得 42](#_Toc206757032)

[**Lab: Copy-on-Write Fork for xv6** 42](#_Toc206757033)

[**Implement copy-on-write fork(hard)** 42](#_Toc206757034)

[1）实验目的 42](#_Toc206757035)

[2）实验步骤 43](#_Toc206757036)

[3）实验中遇到的问题及解决方法 47](#_Toc206757037)

[4）实验心得 48](#_Toc206757038)

[**Lab: Multithreading** 48](#_Toc206757039)

[**Uthread: switching between threads (moderate)** 48](#_Toc206757040)

[1）实验目的 48](#_Toc206757041)

[2）实验步骤 48](#_Toc206757042)

[3）实验中遇到的问题及解决方法 51](#_Toc206757043)

[4）实验心得 51](#_Toc206757044)

[**Using threads (moderate)** 52](#_Toc206757045)

[1）实验目的 52](#_Toc206757046)

[2）实验步骤 52](#_Toc206757047)

[3）实验中遇到的问题及解决方法 54](#_Toc206757048)

[4）实验心得 54](#_Toc206757049)

[**Barrier(moderate)** 55](#_Toc206757050)

[1）实验目的 55](#_Toc206757051)

[2）实验步骤 55](#_Toc206757052)

[3）实验中遇到的问题及解决方法 56](#_Toc206757053)

[4）实验心得 57](#_Toc206757054)

[**Lab: networking** 57](#_Toc206757055)

[**Your Job (hard)** 57](#_Toc206757056)

[1）实验目的 57](#_Toc206757057)

[2）实验步骤 57](#_Toc206757058)

[3）实验中遇到的问题及解决方法 61](#_Toc206757059)

[4）实验心得 62](#_Toc206757060)

[**Lab: locks** 62](#_Toc206757061)

[**Memory allocator (moderate)** 62](#_Toc206757062)

[1）实验目的 62](#_Toc206757063)

[2）实验步骤 62](#_Toc206757064)

[3）实验中遇到的问题及解决方法 65](#_Toc206757065)

[4）实验心得 65](#_Toc206757066)

[**Buffer cache (hard)** 66](#_Toc206757067)

[1）实验目的 66](#_Toc206757068)

[2）实验步骤 66](#_Toc206757069)

[3）实验中遇到的问题及解决方法 70](#_Toc206757070)

[4）实验心得 70](#_Toc206757071)

[**Lab: file system** 70](#_Toc206757072)

[**Large files (moderate)** 70](#_Toc206757073)

[1）实验目的 70](#_Toc206757074)

[2）实验步骤 71](#_Toc206757075)

[3）实验中遇到的问题及解决方法 77](#_Toc206757076)

[4）实验心得 77](#_Toc206757077)

[**Symbolic links (**moderate**)** 77](#_Toc206757078)

[1）实验目的 77](#_Toc206757079)

[2）实验步骤 77](#_Toc206757080)

[3）实验中遇到的问题及解决方法 81](#_Toc206757081)

[4）实验心得 81](#_Toc206757082)

[**Lab: mmap** 81](#_Toc206757083)

[**mmap (hard)** 81](#_Toc206757084)

[1）实验目的 81](#_Toc206757085)

[2）实验步骤 82](#_Toc206757086)

[3）实验中遇到的问题及解决方法 90](#_Toc206757087)

[4）实验心得 91](#_Toc206757088)

[**附录** 91](#_Toc206757089)

[git仓库地址 91](#_Toc206757090)

# **环境搭建**

## **搭建步骤**

### 安装虚拟机

1. 下载虚拟化平台，如vmware或virtualbox
2. 下载带有ubuntu操作系统的.iso文件，导入到虚拟机平台中注册并创建一个虚拟机。

### 配置环境

1. 直接运行以下命令安装所需工具：

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

1. 克隆xv6实验仓库，运行以下git命令：

|  |
| --- |
| $ git clone git://g.csail.mit.edu/xv6-labs-2022 |

1. 在 xv6 目录中运行以下命令，若能编译并运行 xv6（按 Ctrl-a x 退出 qemu），则安装成功

|  |
| --- |
| $ make qemu |

# **Lab: Xv6 and Unix utilities**

## **Boot xv6 (**[**easy**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 掌握 xv6 操作系统实验环境的搭建方法，包括在 Athena 机器或个人计算机上的环境配置。
2. 学会使用 Git 版本控制系统获取 xv6 实验源代码，理解分支管理（如util分支）的基本操作。
3. 熟悉 xv6 操作系统的编译与运行流程，能够通过make qemu命令成功启动 xv6 系统。
4. 了解 xv6 初始文件系统的结构，能够在系统中执行简单命令（如ls）并观察结果。
5. 掌握 qemu 模拟器的基本操作，包括查看进程信息（Ctrl-p）和退出模拟器（Ctrl-a x）。

### 2）实验内容

1. **获取 xv6 源代码**
2. 使用 Git 克隆实验仓库：

|  |
| --- |
| git clone git://g.csail.mit.edu/xv6-labs-2022 |

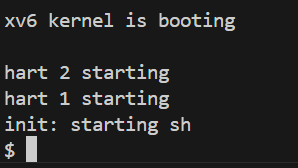
1. 进入仓库目录：

|  |
| --- |
| cd xv6-labs-2022 |

1. 验证当前分支：通过git status确认处于util分支，输出应显示 “On branch util” 且工作区干净。
2. **熟悉 Git 基本操作**

* 查看仓库历史：使用git log查看源代码的提交记录，了解仓库的版本演变。
* 提交本地修改：完成练习后，通过git commit -am '提交说明'记录当前进度。
* 查看变更内容：使用git diff查看与上次提交的差异，或git diff origin/util对比与初始util分支的差异。

1. **编译并运行 xv6**
2. 编译并启动 xv6：执行make qemu命令，系统会自动完成编译（生成内核、用户程序等）、创建文件系统镜像（fs.img）并通过 qemu 模拟器启动 xv6。  
   编译过程中会显示一系列编译命令（如riscv64-unknown-elf-gcc编译内核文件、mkfs/mkfs创建文件系统），最终 qemu 启动后会显示内核启动信息：



1. 在 xv6 中执行命令：在$提示符后输入ls，查看初始文件系统中的文件列表，包括README、cat、sh等文件和程序。
2. 查看进程信息：在 xv6 命令行中按下Ctrl-p，内核会打印当前进程信息（初始状态下通常有init和sh两个进程）。
3. 退出 qemu 模拟器：按下Ctrl-a后再按下x，即可退出模拟器。

### 3）实验中遇到的问题及解决方法

1. **问题**：git clone时出现网络连接失败，无法获取仓库代码。  
   **解决方法**：检查网络连接状态，确认 Athena 机器或个人计算机可访问外部网络；若网络正常，尝试更换 Git 仓库地址（如使用 HTTPS 协议的地址https://g.csail.mit.edu/xv6-labs-2022）。
2. **问题**：执行make qemu后，xv6 启动卡在 “xv6 kernel is booting” 阶段，无后续输出。  
   **解决方法**：可能是 qemu 模拟器配置或版本问题。检查是否安装了正确的 RISC-V 版本 qemu（如qemu-system-riscv64），并确保编译过程无错误（重新执行make clean && make qemu清理并重新编译）。

### 4）实验心得

本次实验是 xv6 操作系统学习的入门实践，通过实际操作掌握了实验环境搭建、源代码获取、系统编译与运行等基础流程。Git 版本控制系统的使用让我意识到其在代码管理中的重要性，通过commit和diff命令可以清晰地跟踪自己的修改，为后续实验的迭代开发提供了便利。

在启动 xv6 并执行ls命令时，我直观地看到了操作系统文件系统的初始结构，结合输出的文件信息（如权限、大小等），对 xv6 的用户态程序组织有了初步认识。而Ctrl-p查看进程的功能则让我联想到操作系统内核对进程管理的底层实现，为后续学习进程调度、内存管理等核心知识埋下了伏笔。

## **sleep (**[easy](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解 xv6 操作系统中用户程序与内核交互的基本机制，特别是系统调用的使用流程。
2. 掌握 xv6 命令行参数的解析方法，学会处理用户输入的合法性校验。
3. 实现 UNIX 风格的sleep程序，使其能够根据用户指定的tick数暂停执行。
4. 熟悉 xv6 程序的编译流程，包括在Makefile中添加新程序的方法。
5. 学会使用实验提供的测试工具验证程序正确性，通过make grade或指定测试命令进行检验

### 2）实验步骤

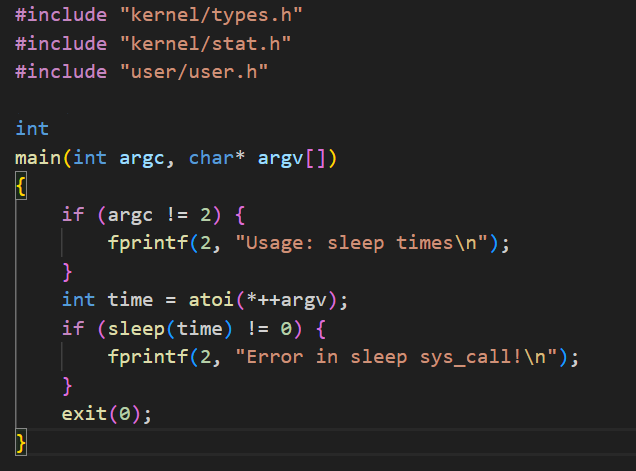
**1. 需求分析与前期准备**

* **功能需求**：实现sleep程序，接收一个整数参数（表示tick数），使程序暂停对应时长后退出。
* **前置知识**：阅读 xv6 书籍第 1 章了解系统基本架构，参考user/目录下已有程序（如echo.c、grep.c）的结构，学习命令行参数处理方式。
* **关键接口**：使用atoi函数将字符串参数转换为整数，调用 xv6 提供的sleep系统调用实现暂停功能，通过exit系统调用结束程序。

**2. 代码实现步骤**

1. **创建sleep.c文件**：在user目录下新建sleep.c文件，用于编写程序逻辑。
2. **参数校验**：在main函数中检查命令行参数数量（argc）是否为 2，若不符合则通过fprintf(2, ...)向标准错误输出提示信息（2表示标准错误文件描述符）。
3. **参数转换**：使用atoi函数将第二个参数（argv[1]）从字符串转换为整数，作为sleep的时长参数。
4. **调用系统调用**：调用sleep系统调用，传入转换后的整数参数；若系统调用返回非 0 值，输出错误信息。
5. **程序退出**：无论sleep调用成功与否，最终通过exit(0)结束程序。

实现代码如下：

**3. 编译配置与测试**

1. **修改Makefile**：在UPROGS变量中添加sleep程序，确保编译时能生成对应的用户程序。UPROGS中需包含\_sleep\条目。
2. **编译与运行 xv6**：执行make qemu命令，重新编译 xv6 内核及用户程序，启动模拟器。
3. **测试基本功能**：在 xv6 shell 中输入sleep 10，观察程序是否暂停一段时间后返回命令提示符。
4. **验证错误处理**：测试无参数或参数错误的情况，确认程序能输出正确的错误提示（如Usage: sleep times）。
5. **通过自动化测试**：执行make grade或./grade-lab-util sleep，验证程序是否通过所有测试用例。

### 3）实验中遇到的问题及解决方法

1. **问题**：忘记在Makefile中添加sleep程序，导致编译后无法在 shell 中找到sleep命令。  
   **解决方法**：检查Makefile的UPROGS部分，添加\_sleep\条目，确保程序被正确编译并加入文件系统。
2. **问题**：参数解析时误将argv[0]作为输入参数，导致转换的数值错误。  
   **解决方法**：明确argv数组的含义（argv[0]为程序名，argv[1]为第一个参数），通过\*++argv或argv[1]获取正确参数。
3. **问题**：未处理sleep系统调用的返回值，无法判断系统调用是否成功。  
   **解决方法**：根据 xv6 内核实现（sys\_sleep返回 0 表示成功），添加返回值判断逻辑，输出错误信息便于调试。
4. **问题**：程序未调用exit(0)，导致执行完成后无响应。  
   **解决方法**：明确用户程序需通过exit系统调用主动退出，在sleep完成后添加exit(0)确保程序正常结束。

### 4）实验心得

本次实验通过实现sleep程序，深入理解了 xv6 中用户程序的开发流程。命令行参数的校验逻辑让我意识到用户输入合法性的重要性，这在实际开发中是保证程序健壮性的基础。

## **pingpong (**[easy](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解 UNIX 管道（pipe）的工作原理，掌握管道在进程间通信中的基本应用。
2. 熟悉进程创建（fork）机制，理解父进程与子进程的关系及执行流程。
3. 掌握read、write系统调用在管道操作中的使用方法，实现进程间的字节传递。
4. 学会使用getpid系统调用获取进程 ID，完成程序输出格式化要求。
5. 掌握 xv6 环境下多进程程序的编译配置（修改Makefile）和测试方法。

### 2）实验步骤

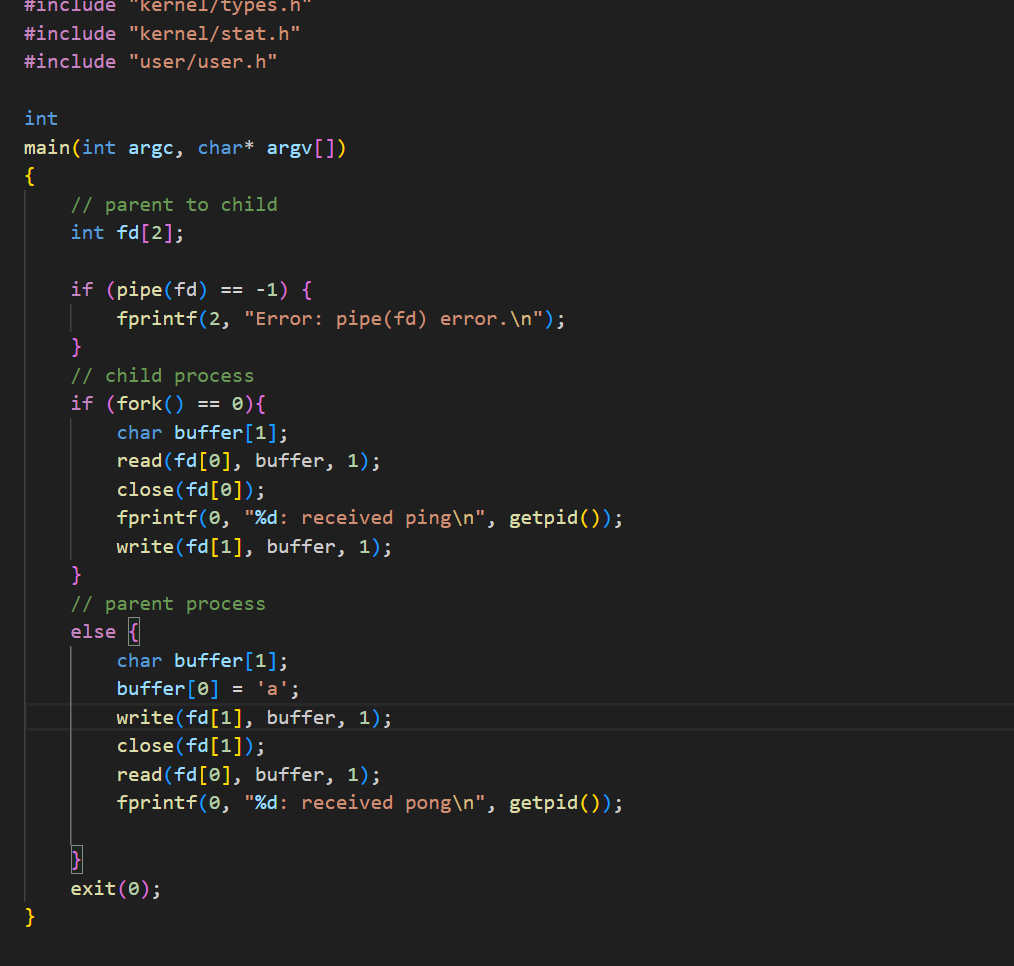
**1. 需求分析与核心思路**

* **功能需求**：实现一个 “乒乓” 程序，通过两个方向的管道通信，父进程向子进程发送一个字节，子进程接收后打印信息并回传字节，父进程接收后打印信息并退出。
* **核心机制**：
  + 使用pipe创建管道（包含读端和写端文件描述符），用于进程间数据传递。
  + 使用fork创建子进程，实现父子进程的并发执行。
  + 父进程先向管道写端发送字节，子进程从读端接收后回传，父进程再接收回传数据。

**2. 代码实现步骤**

1. **创建管道**：定义数组fd[2]存储管道的读端（fd[0]）和写端（fd[1]），调用pipe(fd)创建管道，若失败则输出错误信息。
2. **创建子进程**：通过fork()创建子进程，根据返回值区分父子进程逻辑（返回 0 为子进程，非 0 为父进程）。
3. **子进程逻辑**：
   * 从管道读端（fd[0]）读取 1 个字节到缓冲区。
   * 关闭读端（完成读取后释放资源）。
   * 调用getpid()获取自身 PID，打印 “<pid>: received ping”。
   * 向管道写端（fd[1]）回传读取到的字节。
4. **父进程逻辑**：
   * 向管道写端（fd[1]）写入一个字节（如'a'）。
   * 关闭写端（完成写入后释放资源）。
   * 从管道读端（fd[0]）读取子进程回传的字节。
   * 调用getpid()获取自身 PID，打印 “<pid>: received pong”。
5. **程序退出**：父子进程均通过exit(0)正常退出。

实现代码如下：

**3. 编译配置与测试**

1. **修改Makefile**：在UPROGS变量中添加\_pingpong\条目，确保程序被编译并加入 xv6 文件系统。
2. **编译与启动**：执行make qemu重新编译系统并启动 xv6 模拟器。
3. **功能测试**：在 xv6 shell 中输入pingpong，观察输出是否符合预期（子进程打印 “received ping”，父进程打印 “received pong”）。
4. **验证正确性**：程序应成功实现字节的双向传递，输出格式与示例一致（如4: received ping和3: received pong）。

### 3）实验中遇到的问题及解决方法

1. **问题**：管道创建失败，pipe(fd)返回 - 1，输出错误信息。  
   **解决方法**：检查管道创建的返回值判断是否成功，确保代码中正确处理错误情况；xv6 对管道数量有一定限制，重启模拟器后重试问题解决。
2. **问题**：子进程或父进程阻塞在read调用，程序无输出。  
   **解决方法**：管道是单向通信的，需确保读写端使用正确（读端fd[0]、写端fd[1]）；检查是否忘记关闭不需要的文件描述符，未关闭写端可能导致读端无法检测到数据结束，关闭后阻塞问题解决。
3. **问题**：输出的 PID 顺序混乱或信息缺失。  
   **解决方法**：确认父子进程的逻辑分离正确（fork()返回 0 为子进程）；确保fprintf的文件描述符正确（0表示标准输出），避免因输出目标错误导致信息不显示。
4. **问题**：编译时提示pingpong命令未找到。  
   **解决方法**：检查Makefile的UPROGS是否正确添加\_pingpong\，重新执行make qemu确保编译生效。

### 4）实验心得

本次实验通过实现管道通信的 “乒乓” 程序，深入理解了进程间通信的基本原理。管道作为 UNIX 经典的 IPC 机制，其 “半双工” 特性（单向数据流）要求开发者严格区分读端和写端的使用，这让我意识到资源管理（如关闭无用文件描述符）在系统编程中的重要性。

## **primes (**[moderate](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)/(**[hard](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解并发程序设计思想，掌握基于管道（pipe）和进程创建（fork）的进程间协作机制。
2. 实现 Doug McIlroy 提出的管道式素数筛算法，通过多个进程协同筛选出 2 到 35 之间的素数。
3. 掌握进程间通过管道传递二进制数据（4 字节整数）的方法，理解管道的读写特性（读端在写端关闭时返回 0）。
4. 学会合理管理文件描述符，避免因资源泄露导致 xv6 系统资源耗尽。
5. 理解进程同步机制，确保主进程在所有子进程、孙进程等完成工作后再退出。
6. 掌握在 xv6 中添加新程序到 Makefile 并完成编译、测试的流程。

### 2）实验步骤

**1. 需求分析与算法原理**

* **功能需求**：实现并发素数筛，通过多个进程协作，从 2 到 35 的整数中筛选出素数并打印。每个进程负责一个素数，过滤掉能被该素数整除的数，将剩余数传递给下一个进程。
* **算法原理**：
  + 第一个进程（主进程）向管道写入 2 到 35 的整数。
  + 每个后续进程从左侧管道读取数据，第一个数为素数并打印，然后创建新管道和子进程，将不能被该素数整除的数写入新管道，由子进程继续筛选。
  + 进程间通过管道单向传递 4 字节整数，通过 fork 实现并发，形成筛选流水线。

**2. 代码实现步骤**

1. **主进程初始化**：
   * main 函数中创建初始管道 p，通过 fork 创建第一个子进程（筛选起点）。
   * 主进程关闭管道读端，向写端写入 2 到 35 的整数，完成后关闭写端，调用 wait(0) 等待所有子进程结束后退出。
2. **筛选进程逻辑（new\_proc 函数）**：
   * 接收一个管道 p 作为参数，关闭管道写端，从读端读取第一个整数（该数为素数）并打印。
   * 继续从管道读取下一个整数，若读取成功则创建新管道 newfd 和子进程：
     + 父进程（当前筛选进程）关闭新管道读端，将不能被当前素数整除的数写入新管道写端，全部写入后关闭原管道读端和新管道写端，等待子进程结束。
     + 子进程调用 new\_proc(newfd) 递归处理新管道中的数据，继续筛选。
   * 若读取失败（管道写端关闭），则进程退出，结束当前筛选环节。

实现代码如下：

void new\_proc(int p[2]) {

    int prime;

    int n;

    // close the write side of p

    close(p[1]);

    if (read(p[0], &prime, 4) != 4) {

        fprintf(2, "Error in read.\n");

        exit(1);

    }

    printf("prime %d\n", prime);

    if (read(p[0], &n, 4) == 4){

        int newfd[2];

        pipe(newfd);

        // father

        if (fork() != 0) {

            close(newfd[0]);

            if (n % prime) write(newfd[1], &n, 4);

            while (read(p[0], &n, 4) == 4) {

                if (n % prime) write(newfd[1], &n, 4);

            }

            close(p[0]);

            close(newfd[1]);

            wait(0);

        }

        // child

        else {

            new\_proc(newfd);

        }

    }

}

int

main(int argc, char\* argv[])

{

    int p[2];

    pipe(p);

    // 子进程

    if (fork() == 0) {

        new\_proc(p);

    }

    // 父进程

    else {

        close(p[0]);

        for (int i = 2; i <= 35; ++i) {

            if (write(p[1], &i, 4) != 4) {

                fprintf(2, "failed write %d into the pipe\n", i);

                exit(1);

            }

        }

        close(p[1]);

        wait(0);

        exit(0);

    }

    return 0;

}

**3. 编译配置与测试**

1. **修改 Makefile**：在 UPROGS 变量中添加 \_primes\ 条目，确保程序被编译并加入 xv6 文件系统。
2. **编译与启动**：执行 make qemu 重新编译系统并启动 xv6 模拟器。
3. **功能测试**：在 xv6 shell 中输入 primes，观察输出是否为 2 到 35 之间的所有素数（2、3、5、7、11、13、17、19、23、29、31），且顺序正确。
4. **验证资源管理**：确保程序不会因文件描述符未关闭而提前退出，主进程能等待所有子进程完成后再退出。

### 3）实验中遇到的问题及解决方法

1. **问题**：程序运行中 xv6 提示 “too many open files”，资源耗尽。  
   **解决方法**：未及时关闭不需要的管道端导致文件描述符泄露。在每个进程中，明确关闭不需要的读端或写端（如子进程创建后关闭原管道写端，父进程关闭新管道读端），避免资源占用超限。
2. **问题**：主进程提前退出，导致部分素数未打印。  
   **解决方法**：主进程未正确等待所有子进程结束。在主进程写入数据后，需调用 wait(0) 等待子进程（及所有孙进程）完成；每个筛选进程在写入数据后也需 wait(0) 等待自己的子进程，确保整个流水线完成后再退出。
3. **问题**：读取或写入数据时出现乱码或错误，素数筛选结果不正确。  
   **解决方法**：管道中传递的是 4 字节整数，但 read/write 调用时未指定正确的字节数。确保每次读写均为 4 字节（sizeof(int)），避免因数据截断或多读取导致的错误。
4. **问题**：递归创建子进程时逻辑错误，导致筛选中断。  
   **解决方法**：子进程应递归处理新管道，而父进程需继续处理原管道剩余数据。通过 fork() 返回值严格区分父子进程逻辑，确保父进程完成所有数据筛选后再关闭管道并等待子进程。

### 4）实验心得

本次实验通过实现管道式素数筛，深入理解了并发程序中进程协作的精髓。多个进程通过管道形成 “流水线”，每个进程专注于筛选特定素数的倍数，这种分工模式体现了 UNIX 管道 “连接小程序完成复杂任务” 的设计哲学。

## **find (**[moderate](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解 UNIX 文件系统的目录结构，掌握目录遍历的基本方法。
2. 实现一个简化版的 find 程序，能够在指定目录树中查找具有特定名称的文件。
3. 学会使用递归算法遍历子目录，处理目录中的 . 和 .. 特殊条目。
4. 掌握文件元数据获取（stat 系统调用）和目录项读取（read 系统调用）的方法。
5. 熟悉 C 字符串处理函数（如 strcmp、strcpy 等），理解字符串比较与内存操作的细节。
6. 掌握在 xv6 中添加新程序到 Makefile 并完成编译、测试的流程。

### 2）实验步骤

**1. 需求分析与核心思路**

* **功能需求**：实现 find 程序，接收两个参数（起始目录路径和目标文件名），递归遍历该目录树，打印所有与目标文件名匹配的文件路径。
* **核心机制**：
  + 使用 open 打开目录，read 读取目录项（struct dirent）。
  + 通过 stat 系统调用获取文件元数据，判断文件类型（目录或普通文件）。
  + 对目录项递归处理：遇到子目录则递归调用 find 函数，遇到普通文件则比较文件名。
  + 跳过 .（当前目录）和 ..（父目录），避免循环遍历。

**2. 关键代码实现**

void

find(char\* path, char\* name)

{

    char buf[512], \*p;

    int fd;

    // dir descriptor

    struct dirent de;

    // file descriptor

    struct stat st;

    if ((fd = open(path, 0)) < 0) {

        fprintf(2, "find: cannot open %s\n", path);

        return;

    }

    if (fstat(fd, &st) == -1) {

        fprintf(2, "find: cannot fstat %s\n", path);

        close(fd);

        return;

    }

    // printf("switch to '%s'\n", path);

    switch (st.type) {

        case T\_DEVICE:

        case T\_FILE:

            fprintf(2, "find: %s not a path value.\n", path);

            close(fd);

            // printf("==='%s' is a File\n", path);

            break;

        case T\_DIR:

            // printf("==='%s' is a dir\n", path);

            if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){

                printf("ls: path too long\n");

                break;

            }

            // create full path

            strcpy(buf, path);

            p = buf + strlen(buf);

            \*p++ = '/';

            // read dir infomation for file and dirs

            while (read(fd, &de, sizeof(de)) == sizeof de) {

                if (de.inum == 0)

                    continue;

                if (strcmp(".", rtrim(de.name)) == 0 || strcmp("..", rtrim(de.name)) == 0)

                    continue;

                // copy file name to full path

                memmove(p, de.name, DIRSIZ);

                // create a string with zero ending.

                p[DIRSIZ] = '\0';

                // stat each of files

                if (stat(buf, &st) == -1) {

                    fprintf(2, "find: cannot stat '%s'\n", buf);

                    continue;

                }

                // printf("===file:'%s'\n", buf);

                if (st.type == T\_DEVICE || st.type == T\_FILE) {

                    if (strcmp(name, rtrim(de.name)) == 0) {

                        printf("%s\n", buf);

                        // for (int i = 0; buf[i] != '\0'; ++i) {

                        //     printf("'%d'\n", buf[i]);

                        // }

                    }

                }

                else if (st.type == T\_DIR) {

                    find(buf, name);

                }

            }

    }

}

### 3）实验中遇到的问题及解决方法

1. **问题**：递归遍历陷入无限循环（不断访问 . 或 ..）。  
   **解决方法**：在读取目录项后，显式跳过名称为 . 和 .. 的条目，避免递归调用自身或父目录。
2. **问题**：路径拼接时缓冲区溢出。  
   **解决方法**：在拼接路径前检查长度（strlen(path) + 1 + DIRSIZ + 1）是否超过缓冲区大小（sizeof(buf)），避免内存越界。
3. **问题**：无法打开目录或获取文件状态，提示错误。  
   **解决方法**：添加错误处理逻辑（如 open 和 stat 的返回值检查），确保程序在遇到无效路径时能友好提示并继续执行。

### 4）实验心得

本次实验通过实现 find 程序，深入理解了 UNIX 风格文件系统的层次结构和目录遍历机制。

字符串处理是本次实验的关键细节。xv6 目录项的固定长度设计（DIRSIZ）导致名称后可能带有冗余空格，这要求我们在比较前进行清洗，这种 “底层细节处理” 让我意识到系统编程中对数据结构精确理解的重要性。

## **xargs (**[moderate](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解 UNIX xargs 程序的核心功能：从标准输入读取行数据，将每行作为额外参数拼接至指定命令后并执行。
2. 掌握进程创建（fork）与程序执行（exec）的协作机制，实现父进程等待子进程完成（wait）的同步逻辑。
3. 学会从标准输入逐字符读取行数据，处理输入边界（如行长限制）。
4. 掌握命令参数数组的构建方法，理解 exec 系统调用对参数格式的要求（以 NULL 结尾的字符串数组）。
5. 熟悉 xv6 中 MAXARG 等系统限制，确保参数数组不越界。
6. 完成程序编译配置（修改 Makefile）并通过测试脚本验证功能正确性。

### 2）实验步骤

**1. 需求分析与核心思路**

* **功能需求**：实现 xargs 程序，接收命令作为参数，从标准输入读取每行数据，将每行作为额外参数追加到命令后，通过 fork+exec 执行命令，父进程等待子进程完成。
* **核心机制**：
  + 从标准输入逐字符读取，直到遇到换行符（\n），将读取内容作为一行数据。
  + 构建参数数组：初始参数为 xargs 接收的命令及参数，追加输入行作为新参数，以 NULL 结尾。
  + 对每行输入，通过 fork 创建子进程，调用 exec 执行拼接后的命令，父进程通过 wait 等待子进程结束。

**2. 关键代码实现**

#define DEBUG 0

// 宏定义

#define debug(codes) if(DEBUG) {codes}

void xargs\_exec(char\* program, char\*\* paraments);

void

xargs(char\*\* first\_arg, int size, char\* program\_name)

{

    char buf[1024];

    debug(

        for (int i = 0; i < size; ++i) {

            printf("first\_arg[%d] = %s\n", i, first\_arg[i]);

        }

    )

    char \*arg[MAXARG];

    int m = 0;

    while (read(0, buf+m, 1) == 1) {

        if (m >= 1024) {

            fprintf(2, "xargs: arguments too long.\n");

            exit(1);

        }

        if (buf[m] == '\n') {

            buf[m] = '\0';

            debug(printf("this line is %s\n", buf);)

            memmove(arg, first\_arg, sizeof(\*first\_arg)\*size);

            // set a arg index

            int argIndex = size;

            if (argIndex == 0) {

                arg[argIndex] = program\_name;

                argIndex++;

            }

            arg[argIndex] = malloc(sizeof(char)\*(m+1));

            memmove(arg[argIndex], buf, m+1);

            debug(

                for (int j = 0; j <= argIndex; ++j)

                    printf("arg[%d] = \*%s\*\n", j, arg[j]);

            )

            arg[argIndex+1] = 0;

            xargs\_exec(program\_name, arg);

            free(arg[argIndex]);

            m = 0;

        } else {

            m++;

        }

    }

}

void

xargs\_exec(char\* program, char\*\* paraments)

{

    if (fork() > 0) {

        wait(0);

    } else {

        debug(

            printf("child process\n");

            printf("    program = %s\n", program);

            for (int i = 0; paraments[i] != 0; ++i) {

                printf("    paraments[%d] = %s\n", i, paraments[i]);

            }

        )

        if (exec(program, paraments) == -1) {

            fprintf(2, "xargs: Error exec %s\n", program);

        }

        debug(printf("child exit");)

    }

}

int

main(int argc, char\* argv[])

{

    debug(printf("main func\n");)

    char \*name = "echo";

    if (argc >= 2) {

        name = argv[1];

        debug(

            printf("argc >= 2\n");

            printf("argv[1] = %s\n", argv[1]);

        )

    }

    else {

        debug(printf("argc == 1\n");)

    }

    xargs(argv + 1, argc - 1, name);

    exit(0);

}

**3. 编译配置与测试**

1. **修改 Makefile**：在 UPROGS 变量中添加 \_xargs\ 条目，确保程序被编译并加入文件系统。
2. **功能测试**：
   * 基础测试：echo hello too | xargs echo bye 应输出 bye hello too。
   * 多行输入测试：(echo 1 ; echo 2) | xargs -n 1 echo 应分别输出 1 和 2。
   * 结合 find 测试：find . b | xargs grep hello 应对找到的文件执行 grep hello。
3. **验证测试脚本**：运行 sh < xargstest.sh，确认输出符合预期（包含多个 hello 及 $ 提示符）。

### 3）实验中遇到的问题及解决方法

1. **问题**：输入行过长导致缓冲区溢出。  
   **解决方法**：在读取输入时添加长度检查（m >= 1024），超过限制时输出错误并退出，避免内存越界。
2. **问题**：参数数组未以 NULL 结尾，导致 exec 执行异常。  
   **解决方法**：明确设置 arg[argIndex + 1] = NULL，确保 exec 能正确识别参数结束位置。
3. **问题**：动态分配的参数内存未释放，导致内存泄漏。  
   **解决方法**：使用 malloc 分配输入行参数后，在子进程执行完成后通过 free 释放内存，避免资源浪费。

### 4）实验心得

本次实验通过实现 xargs 程序，深入理解了 UNIX 中 “管道 + 命令协作” 的经典模式。xargs 作为连接标准输入与命令参数的桥梁，其核心逻辑虽简单，但涉及输入处理、参数构建、进程管理等多个系统编程关键环节。

在输入处理中，逐字符读取并处理换行符的逻辑让我体会到 “流处理” 的细节 —— 操作系统将输入抽象为字节流，程序需自行界定行边界。参数数组的构建则需严格遵循 exec 的要求，尤其是 NULL 结尾的约定，这体现了系统调用对接口规范性的严格要求。

# **Lab: system calls**

## **Using gdb (**[**easy**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 掌握使用 GDB 调试 xv6 内核的基本方法，包括设置断点、单步执行、查看堆栈回溯和源代码布局等操作。
2. 理解 xv6 系统调用的执行流程，通过调试跟踪系统调用入口函数 syscall 的调用关系。
3. 熟悉进程控制块（struct proc）的结构，能够通过 GDB 查看进程的关键属性（如 trapframe 中的寄存器值）。
4. 理解 RISC-V 架构下 CPU 特权模式的切换机制，通过寄存器（如 sstatus）分析 CPU 的运行模式历史。
5. 掌握内核崩溃（panic）的调试方法，能够通过崩溃信息（sepc、scause）定位错误指令和原因。
6. 学会通过内核汇编文件（kernel.asm）和 GDB 断点分析内存访问错误等内核异常。

### 2）实验步骤

**1. GDB 基础调试环境搭建**

1. 启动带 GDB 支持的 xv6 模拟器：在终端执行 make qemu-gdb，xv6 会暂停等待 GDB 连接。
2. 启动 GDB 并连接内核：在另一个终端进入 xv6 目录，执行 riscv64-unknown-elf-gdb kernel/kernel，在 GDB 交互界面中输入 target remote localhost:26000 连接模拟器。

**2. 系统调用流程调试**

1. **设置断点与查看堆栈**：在 GDB 中执行 b syscall 设置断点，输入 c 让内核继续运行。当触发断点后，执行 layout src 查看源代码布局，执行 backtrace 查看堆栈回溯，分析 syscall 函数的调用者。
2. **查看进程控制块（struct proc）**：单步执行（n）过 struct proc \*p = myproc(); 语句后，执行 p /x \*p 以十六进制格式查看当前进程的 proc 结构，重点观察 trapframe 中的寄存器值。
3. **分析 CPU 特权模式**：执行 p /x $sstatus 查看 sstatus 寄存器值，通过分析寄存器中的特权模式标志位，确定 CPU 进入内核模式前的运行模式。

**3. 内核崩溃调试模拟**

1. **制造内核崩溃**：修改 kernel/syscall.c 中的 syscall 函数，将 num = p->trapframe->a7; 替换为 num = \*(int \*)0;（访问空地址），重新编译并执行 make qemu，观察内核 panic 信息（包含 scause、sepc、stval 等）。
2. **定位崩溃指令**：在 kernel/kernel.asm 中搜索 panic 信息中的 sepc 值，找到对应的汇编指令，分析该指令的功能及涉及的寄存器。
3. **通过 GDB 验证崩溃原因**：在 GDB 中设置断点 b \*sepc值（如 b \*0x8000215a），执行 layout asm 查看汇编指令，结合 scause 寄存器值（如页错误）分析内存访问错误的原因。

**4. 关键问题记录与分析**

1. 通过堆栈回溯确定 syscall 函数的调用者。

答：函数 usertrap() 调用了 syscall() 函数

1. 记录 p->trapframe->a7 的值并解释其含义。

答：得到 a7 的值为 7 。根据参考教材 xv6 book 第二章和 user/initcode.S 中的代码可知，这个 a7 寄存器中保存了将要执行的系统调用号。这里的系统调用号为 7，在 kernel/syscall.h 中可以找到，这个系统调用为 SYS\_exec

1. 根据 sstatus 寄存器值判断 CPU 之前的运行模式。

答：用户模式

1. 定位内核崩溃时的汇编指令，分析涉及的寄存器及崩溃原因。

答：内核 panic 在 lw a3,0(zero)。num 代表 a3 寄存器。内核因为加载了一个未使用的地址 0 处的内存数据而崩溃（Load page fault）。地址 0 并不映射到内核空间中（从 0x80000000 开始）。scause 中的异常代码证实了上述观点。

1. 确定崩溃时运行的进程名称及 PID。

答：这个二进制的名字为 initcode ，其 process id 为 1

### 3）实验中遇到的问题及解决方法

**问题**：执行 p /x \*p 时提示 “No symbol "p" in current context”。  
**解决方法**：单步执行（n）需跨过 struct proc \*p = myproc(); 语句，确保变量 p 已被初始化后再查看其值。

### 4）实验心得

本次实验通过 GDB 深入内核调试，让我对 xv6 的系统调用流程和 RISC-V 架构有了更直观的认识。GDB 的断点、堆栈回溯和寄存器查看功能，为分析内核运行状态提供了关键工具 —— 例如通过 backtrace 清晰看到 syscall 由陷阱处理函数调用，印证了 “用户态系统调用通过陷阱进入内核” 的理论。

## **System call tracing (**[**moderate**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解 xv6 系统调用的底层实现机制，掌握添加新系统调用的完整流程（包括用户态接口与内核态实现）。
2. 实现 trace 系统调用，使其能够通过 “掩码（mask）” 指定需要跟踪的系统调用类型，掩码的每一位对应一种系统调用。
3. 确保跟踪功能在进程及其后续创建的子进程中生效，且不影响其他无关进程。
4. 修改内核，使其在被跟踪的系统调用返回时打印进程 ID、系统调用名称及返回值。
5. 验证跟踪功能的正确性，确保其符合实验示例中的输出格式和行为逻辑。

### 2）实验步骤

**1. 准备工作：添加 trace 系统调用的用户态支持**

1. **修改 Makefile**：在 UPROGS 变量中添加 $U/\_trace，确保 trace 程序被编译并加入文件系统。
2. **添加用户态接口**：
   * 在 user/user.h 中添加 trace 系统调用的函数原型：int trace(int);。
   * 在 user/usys.pl 中添加 stub 生成指令：entry("trace");，确保编译时生成对应的系统调用汇编 stub（user/usys.S）。
   * 在 kernel/syscall.h 中添加系统调用编号：#define SYS\_trace 22（编号需与现有系统调用不冲突）。

**2. 内核态实现 trace 系统调用**

1. **扩展进程控制块（struct proc）**：在 kernel/proc.h 的 struct proc 中添加新成员 int trace\_mask;，用于存储当前进程的跟踪掩码（记录需要跟踪的系统调用）。
2. **实现 sys\_trace 内核函数**：在 kernel/sysproc.c 中添加系统调用处理函数，从用户态获取掩码参数并存储到当前进程的 trace\_mask 中：

uint64

sys\_trace(void)

{

  int mask;

  argint(0, &mask);

  struct proc \*p = myproc();

  p->mask = mask;

  return 0;

}

**确保子进程继承跟踪掩码**：修改 kernel/proc.c 中的 fork 函数，在复制父进程属性时将 trace\_mask 传递给子进程：

  np->mask = p->mask;

**3.修改内核系统调用处理逻辑以打印跟踪信息**

1. **添加系统调用名称数组**：在 kernel/syscall.c 中定义系统调用名称数组，与 syscall.h 中的编号对应，用于跟踪时打印名称：

static char\* syscalls\_name[] = {

[SYS\_fork]    "syscall fork",

[SYS\_exit]    "syscall exit",

[SYS\_wait]    "syscall wait",

[SYS\_pipe]    "syscall pipe",

[SYS\_read]    "syscall read",

[SYS\_kill]    "syscall kill",

[SYS\_exec]    "syscall exec",

[SYS\_fstat]   "syscall fstat",

[SYS\_chdir]   "syscall chdir",

[SYS\_dup]     "syscall dup",

[SYS\_getpid]  "syscall getpid",

[SYS\_sbrk]    "syscall sbrk",

[SYS\_sleep]   "syscall sleep",

[SYS\_uptime]  "syscall uptime",

[SYS\_open]    "syscall open",

[SYS\_write]   "syscall write",

[SYS\_mknod]   "syscall mknod",

[SYS\_unlink]  "syscall unlink",

[SYS\_link]    "syscall link",

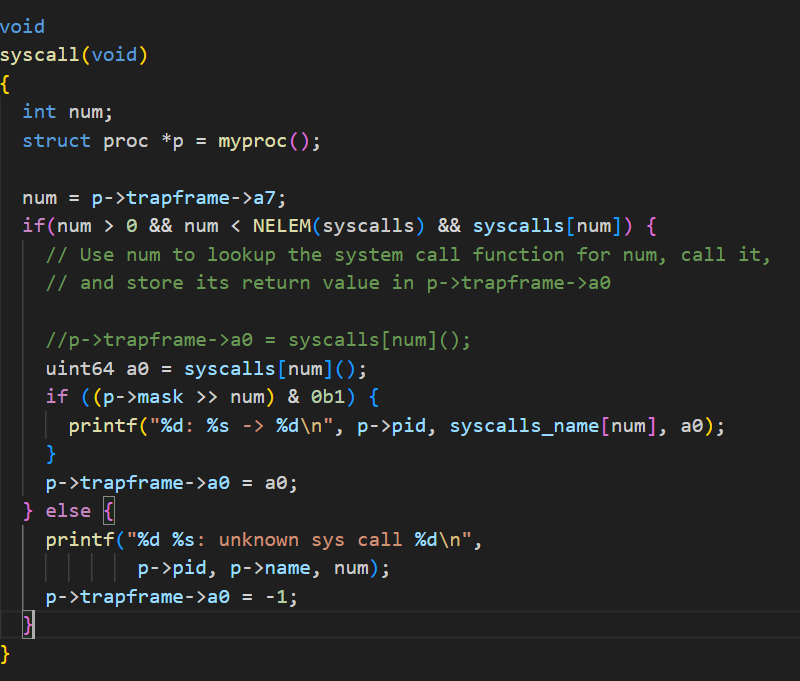
[SYS\_mkdir]   "syscall mkdir",

[SYS\_close]   "syscall close",

[SYS\_trace]   "syscall trace",

[SYS\_sysinfo] "syscall sysinfo",

};

**修改 syscall 函数打印跟踪信息**：在 kernel/syscall.c 的 syscall 函数中，当系统调用执行完成后，检查当前进程的 mask，若对应系统调用的位被设置，则打印跟踪信息：

### 3）实验中遇到的问题及解决方法

1. **问题**：trace 程序编译失败，提示 “未定义的 reference to 'trace'”。  
   **解决方法**：遗漏了用户态接口配置。检查 user/user.h 中是否添加 trace 原型，user/usys.pl 中是否添加 entry("trace")，重新编译确保生成 usys.S 中的 stub。
2. **问题**：跟踪信息未打印，即使设置了掩码。

**解决方法**：

* + 检查 sys\_trace 函数是否正确将掩码存入 proc->trace\_mask（可能未正确调用 argint 获取用户参数）。
  + 确认 syscall 函数中掩码检查逻辑正确：(p->trace\_mask & (1 << num)) 中的位运算是否正确，系统调用编号 num 是否与数组索引一致。

1. **问题**：子进程未继承跟踪掩码，跟踪仅在父进程生效。  
   **解决方法**：在 fork 函数中遗漏了 trace\_mask 的复制逻辑，需在 kernel/proc.c 的 fork 中添加 np->trace\_mask = p->trace\_mask;。

### 4）实验心得

本次实验深入理解了 xv6 系统调用的完整生命周期，从用户态的 ecall 指令触发，到内核态的系统调用分发、执行与返回。添加新系统调用的过程（用户态 stub、编号定义、内核函数实现）让我掌握了操作系统扩展功能的基本流程。

## **Sysinfo (**[**moderate**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 掌握在 xv6 中添加新系统调用的完整流程，包括用户态接口定义、内核态实现及数据交互机制。
2. 实现 sysinfo 系统调用，使其能够收集系统运行状态信息，具体包括：
   * freemem：系统中空闲内存的字节数。
   * nproc：状态非 UNUSED 的进程数量。
3. 理解内核态与用户态之间的数据传递机制，掌握使用 copyout 函数将内核数据复制到用户空间的方法。
4. 学会遍历内核数据结构（如内存分配器的空闲链表、进程列表）以获取系统信息。
5. 通过测试程序 sysinfotest 验证 sysinfo 系统调用的正确性，确保输出 “sysinfotest: OK”。

### 2）实验步骤

**.1. 准备工作：添加 sysinfo 系统调用的用户态支持**

1. **修改 Makefile**：在 UPROGS 变量中添加 $U/\_sysinfotest，确保测试程序 sysinfotest 被编译并加入文件系统。
2. **定义用户态接口**：
   * 在 user/user.h 中预声明 struct sysinfo 并添加系统调用原型
   * 在 user/usys.pl 中添加系统调用 stub 生成指令：entry("sysinfo");，确保编译时生成对应的汇编 stub（user/usys.S）。
   * 在 kernel/syscall.h 中添加系统调用编号：#define SYS\_sysinfo 23（需与现有编号不冲突）。

**2. 内核态实现 sysinfo 系统调用**

1. **定义 struct sysinfo 结构体**：在 kernel/sysinfo.h 中定义用于传递系统信息的结构体（需与用户态定义一致）

**添加辅助函数收集系统信息**：

* **收集空闲内存**：在 kernel/kalloc.c 中添加函数 get\_freemem()，遍历内存分配器的空闲链表（kmem.freelist），计算空闲内存总字节数（每个空闲块大小为 PGSIZE，即 4096 字节）
* **收集进程数量**：在 kernel/proc.c 中添加函数 get\_nproc()，遍历进程列表（proc 数组），统计状态非 UNUSED 的进程数量

**实现 sys\_sysinfo 内核函数**：在 kernel/sysproc.c 中添加系统调用处理函数，获取用户态传递的结构体指针，收集系统信息并通过 copyout 复制到用户空间：

uint64

sys\_sysinfo(void)

{

  uint64 si\_addr;

  argaddr(0, &si\_addr);

  int nproc;

  int freemem;

  nproc = proc\_not\_unsed\_num();

  freemem = free\_mem\_num();

  struct sysinfo sysinfo;

  sysinfo.freemem = freemem;

  sysinfo.nproc = nproc;

  struct proc \*p = myproc();

  if (copyout(p->pagetable, si\_addr, (char \*)&sysinfo, sizeof(sysinfo)) < 0){

     return -1;

  }

  return 0;

}

### 3）实验中遇到的问题及解决方法

1. **问题**：用户态程序编译失败，提示 “struct sysinfo 未定义”。  
   **解决方法**：在 user/user.h 中遗漏了 struct sysinfo 的预声明或定义，需确保用户态与内核态对该结构体的定义一致（字段顺序、类型相同）。
2. **问题**：copyout 函数返回错误，导致系统调用失败。  
   **解决方法**：
   * 检查用户态传递的指针是否有效（可能用户程序传递了无效地址）。
   * 确认 copyout 的参数顺序正确：copyout(pagetable, 用户地址, 内核数据地址, 长度)，避免地址或长度传递错误。

### 4）实验心得

本次实验通过实现 sysinfo 系统调用，深入理解了 xv6 中系统调用的完整生命周期，从用户态的函数调用到内核态的信息收集与数据返回。核心难点在于内核态与用户态的数据交互，copyout 函数的使用让我意识到操作系统对内存安全的严格控制 —— 内核必须通过特定接口（而非直接访问）向用户态传递数据，以防止用户程序篡改内核内存。

# **Lab: page tables**

## **Speed up system calls (**[**easy**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解用户态与内核态通过共享内存区域加速系统调用的原理，掌握 “零内核交叉” 系统调用的实现机制。
2. 学会在 xv6 中为进程页表添加特定虚拟地址映射，掌握 mappages 等页表操作函数的使用。
3. 实现 getpid() 系统调用的优化：通过在用户态与内核态之间共享只读页存储进程 PID，避免系统调用的内核陷入开销。
4. 掌握共享页的生命周期管理，包括在进程创建时分配初始化、进程退出时释放的完整流程。
5. 理解页表权限控制（如只读权限）在保护共享数据中的作用，确保用户态无法篡改内核维护的共享信息。

### 2）实验步骤、

**1. 理解共享页加速原理**

通过在用户虚拟地址空间中映射一个内核管理的只读页（虚拟地址为 USYSCALL），内核在该页中存储进程的固定或低频变化信息（如 PID）。用户态程序可直接读取该页数据，无需执行 ecall 陷入内核，从而加速系统调用。

**2. 核心实现步骤**

**（1）定义共享数据结构与虚拟地址**

* memlayout.h 中已定义虚拟地址 USYSCALL（如 0x3fffff000）和结构体 struct usyscall

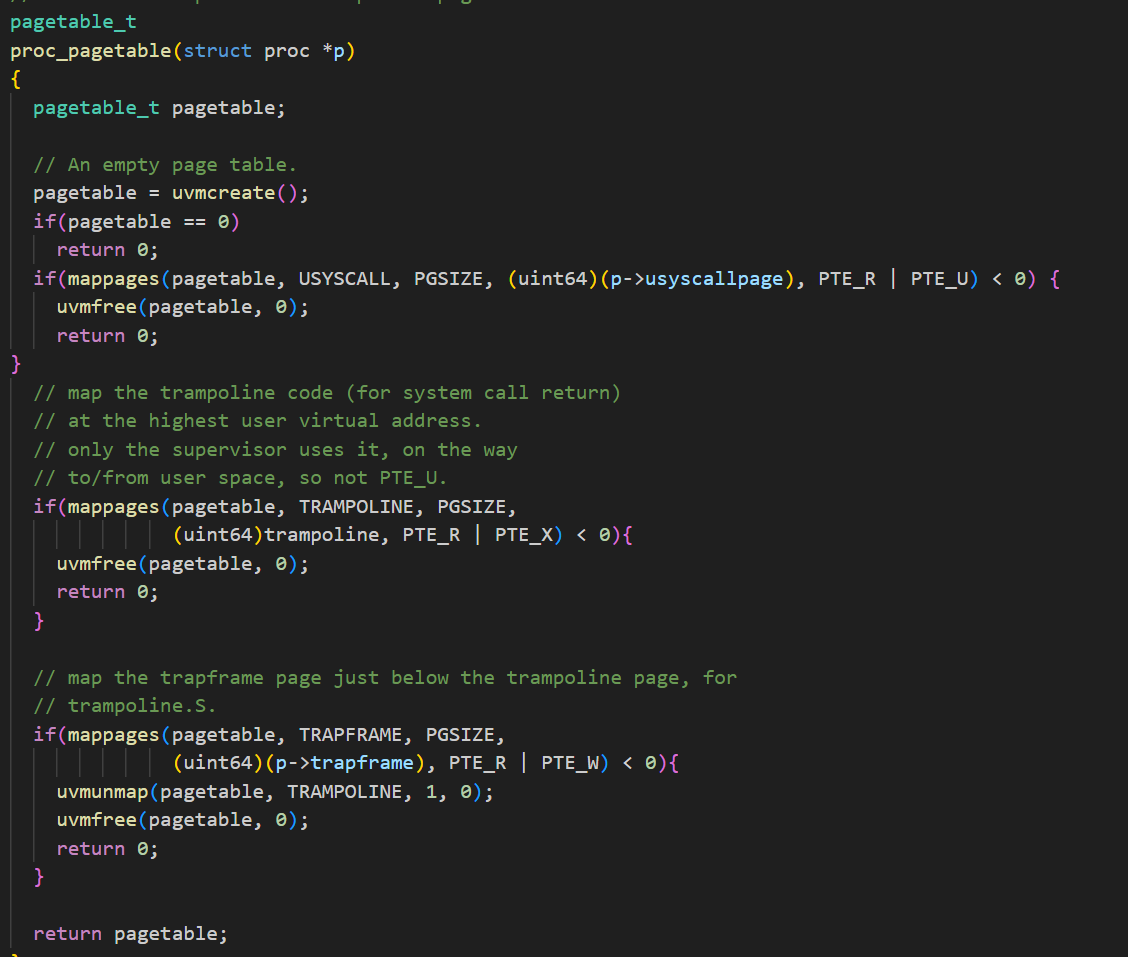
**（2）在进程页表中添加共享页映射**

在 kernel/proc.c 的 proc\_pagetable 函数中，为进程页表添加 USYSCALL 到物理页的映射，权限设为用户态可读（PTE\_R | PTE\_U）

**（3）进程创建时初始化共享页**

在 kernel/proc.c 的 allocproc 函数中，为新进程分配共享页物理内存，初始化 struct usyscall 并存储 PID

**（4）进程退出时释放共享页**

在 kernel/proc.c 的 freeproc 函数中，释放共享页的物理内存，避免内存泄漏

### 3）实验中遇到的问题及解决方法

1. **问题**：用户态程序无法读取 USYSCALL 地址，提示页错误。  
   **解决方法**：页表映射权限错误，未设置 PTE\_U 标志（用户态可访问），修改映射权限为 PTE\_R | PTE\_U 后解决。
2. **问题**：共享页初始化后 PID 始终为 0，测试失败。  
   **解决方法**：在 allocproc 中初始化 uc->pid 时，误将 p->pid 写成固定值 0，修正为 uc->pid = p->pid 后正确获取 PID。

### 4）实验心得

本次实验通过共享内存优化系统调用，让我深刻理解了 “减少内核交叉” 对性能的提升作用。传统系统调用需经过 “用户态陷入内核态 - 执行系统调用 - 返回用户态” 的流程，而共享只读页使高频调用（如 getpid）可直接在用户态完成，显著降低开销。

## **Print a page table (**[**easy**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 深入理解 RISC-V 架构下三级页表的结构与映射机制，掌握页表项（PTE）的组成及关键标志位（如有效位 PTE\_V、权限位 PTE\_R/W/X 等）的含义。
2. 实现 vmprint() 函数，能够递归遍历页表结构，按层级打印有效页表项的索引、内容及物理地址，直观展示页表的层级关系。
3. 通过在 exec.c 中插入页表打印调用，观察第一个进程（init 进程）的页表布局，验证页表打印功能的正确性。
4. 结合实验输出与 xv6 页表原理（如图 3-4），分析页表映射的实际含义，理解用户态与内核态内存的映射关系及权限控制。

### 2）实验步骤

**1. 实现 vmprint() 函数**

**（1）添加函数原型与调用**

* 在 kernel/defs.h 中添加 vmprint() 函数原型，确保可在 exec.c 中调用
* 在 kernel/exec.c 中 exec 函数返回前插入打印逻辑，仅打印 PID 为 1 的进程（init 进程）的页表

**（2）实现页表递归打印逻辑**

* 在 kernel/vm.c 中实现 vmprint() 函数，通过递归遍历三级页表，打印有效页表项的信息：

void

vmprint(pagetable\_t pagetable)

{

  if (printdeep == 0)

    printf("page table %p\n", (uint64)pagetable);

  for (int i = 0; i < 512; i++) {

    pte\_t pte = pagetable[i];

    if (pte & PTE\_V) {

      for (int j = 0; j <= printdeep; j++) {

        printf("..");

      }

      printf("%d: pte %p pa %p\n", i, (uint64)pte, (uint64)PTE2PA(pte));

    }

    // pintes to lower-level page table

    if((pte & PTE\_V) && (pte & (PTE\_R|PTE\_W|PTE\_X)) == 0){

      printdeep++;

      uint64 child\_pa = PTE2PA(pte);

      vmprint((pagetable\_t)child\_pa);

      printdeep--;

    }

  }

}

**2. 实验输出分析与问题回答**

启动 xv6 后，init 进程的页表打印输出类似示例格式，结合输出与图 3-4（xv6 三级页表结构示意图）分析如下：

**（1）输出与图 3-4 的对应关系**

图 3-4 展示了 RISC-V 三级页表的层级映射：虚拟地址被分为 3 个 9 位的索引（VPN[2]、VPN[1]、VPN[0]），分别对应顶级页表、二级页表、三级页表的索引，最终通过三级页表项映射到物理页。实验输出中，缩进层级对应页表级别（顶级无缩进前缀，二级前缀为 “..”，三级前缀为 “.. ..”），索引值即 VPN[2]、VPN[1]、VPN[0] 的值。

**（2）页 0 包含的内容**

输出中三级页表索引 0 的 PTE 对应虚拟地址的低地址区域。在 init 进程中，这部分通常映射用户态程序的代码段或数据段（如 init 程序的指令和初始化数据），是进程执行的核心代码区域。

**（3）页 2 包含的内容**

三级页表索引 2 的 PTE 通常对应用户态堆或数据段的扩展区域。init 进程作为第一个用户进程，其堆区域用于动态内存分配，因此页 2 可能存储进程运行中动态分配的数据。

**（4）用户态对页 1 映射内存的访问权限**

需查看页 1 对应 PTE 的权限位（PTE\_R、PTE\_W、PTE\_X、PTE\_U）。若 PTE 包含 PTE\_U（用户态可访问）且 PTE\_R 或 PTE\_W 被设置，则用户态进程可读写该内存；若缺少 PTE\_U，则用户态无法访问（仅内核态可访问）。实验输出中，若页 1 的 PTE 含 PTE\_U | PTE\_R，则用户态可读取；若含 PTE\_W，则可写入。

**（5）倒数第三页包含的内容**

输出中倒数第三页通常对应内核态映射区域（如 USYSCALL 共享页或内核数据结构）。在 init 进程中，这部分可能是内核与用户态共享的只读数据页（如 struct usyscall 所在页），或内核为用户进程预映射的关键内存区域（如 trap 向量表）。

### 3）实验中遇到的问题及解决方法

1. **问题**：递归打印时层级缩进错误，无法区分页表级别。

**解决方法**：使用静态变量 printdeep 记录当前递归深度（页表级别），每进入下一级页表 printdeep++，回溯时 printdeep--，确保缩进与层级一致。

1. **问题**：打印了无效页表项（PTE\_V=0），输出冗余。

**解决方法**：添加 if (pte & PTE\_V) 判断，仅处理有效页表项，过滤无效项。

### 4）实验心得

本次实验通过实现页表打印函数，直观展示了 RISC-V 三级页表的层级结构，加深了对虚拟地址到物理地址映射过程的理解。递归遍历页表的逻辑让我体会到 “分而治之” 思想在处理复杂数据结构中的应用 —— 将三级页表的打印拆解为单级页表的遍历与子页表的递归处理，简化了逻辑。

## **Detect which pages have been accessed (**[**hard**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解 RISC-V 页表项（PTE）中访问位（PTE\_A）的作用，掌握硬件如何通过该位标记页面的访问状态（读或写）。
2. 实现 pgaccess 系统调用，能够检测指定范围内用户虚拟页的访问情况，将结果以位掩码形式返回给用户态。
3. 掌握页表遍历技术，通过 walk 函数定位目标虚拟页对应的页表项，检查并清除 PTE\_A 位，确保后续访问可被重新检测。
4. 熟悉内核与用户态的数据交互机制，使用 copyout 函数将内核中的位掩码结果复制到用户空间缓冲区。
5. 验证系统调用的正确性，确保 pgtbltest 中的 pgaccess 测试用例通过。

### 2）实验步骤

**1. 定义访问位宏（PTE\_A）**

在 kernel/riscv.h 中定义 RISC-V 页表项的访问位（PTE\_A），根据 RISC-V 特权架构手册，PTE\_A 位于第 6 位：

#define PTE\_A (1L << 6)

**2. 实现 pgaccess 系统调用**

在 kernel/sysproc.c 中实现 sys\_pgaccess 函数，核心逻辑包括解析用户参数、遍历页表检查访问位、生成位掩码并返回结果：

**（1）解析系统调用参数**

通过 argaddr 和 argint 提取用户传入的三个参数：

* 起始虚拟地址（va）：待检查的第一个页面的虚拟地址。
* 页面数量（pagenum）：需要检查的页面总数。
* 结果缓冲区地址（abitsaddr）：用户态存储位掩码结果的地址。

**（2）遍历页表检查访问位**

对每个待检查的页面（从 va 开始，每次递增 PGSIZE），通过 walk 函数获取对应的页表项（PTE），检查 PTE\_A 位是否置位：

* 若 PTE\_A 置位，在结果位掩码中设置对应位（第 i 位对应第 i 个页面）。
* 检查后清除 PTE\_A 位，确保后续访问可被重新检测。

uint64 maskbits = 0;

struct proc \*proc = myproc();

for (int i = 0; i < pagenum; i++) {

uint64 curr\_va = va + i \* PGSIZE;

// 查找该虚拟地址对应的页表项

pte\_t \*pte = walk(proc->pagetable, curr\_va, 0);

if (pte == 0) {

panic("pgaccess: page not exist");

}

if ((\*pte) & PTE\_A) {

maskbits |= (1L << i);

}

\*pte &= ~PTE\_A;

}

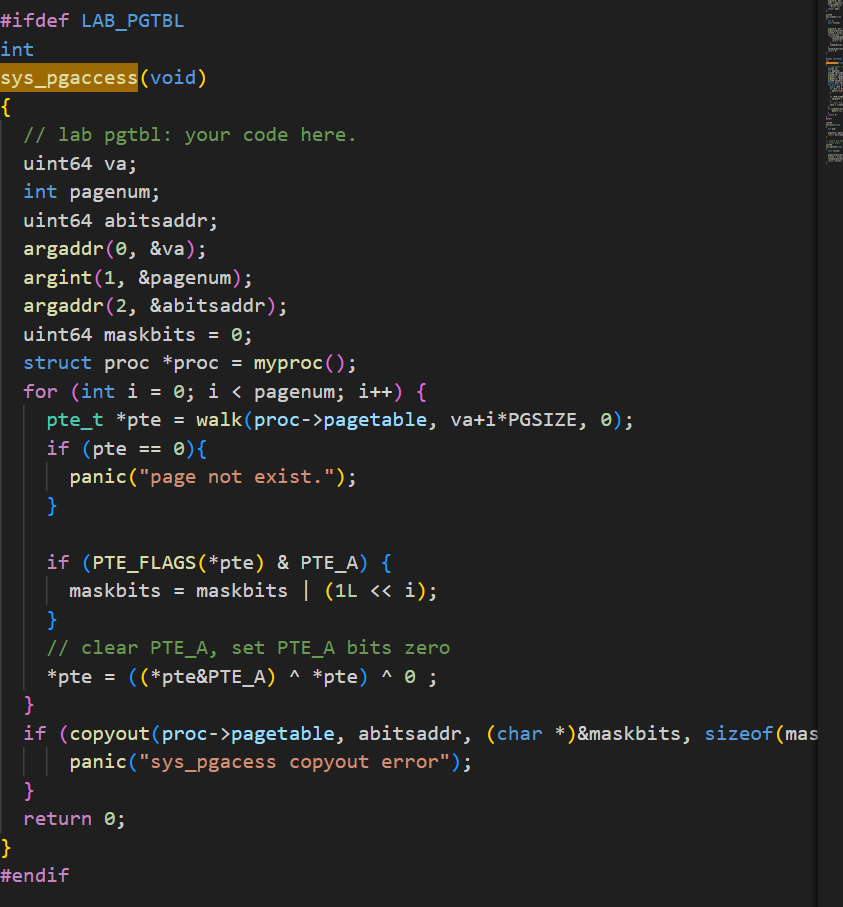
**（3）将结果复制到用户空间**

通过 copyout 函数将内核中的位掩码（maskbits）复制到用户态提供的缓冲区（abitsaddr）：

if (copyout(proc->pagetable, abitsaddr, (char \*)&maskbits, sizeof(maskbits)) < 0) {

panic("pgaccess: copyout failed"); // 复制失败处理

}



**3. 测试与验证**

编译内核并运行 pgtbltest，若 pgaccess 测试用例通过，说明系统调用实现正确。测试逻辑通过访问指定页面后调用 pgaccess，验证位掩码是否正确标记被访问的页面。

### 3）实验中遇到的问题及解决方法

1. **问题**：PTE\_A 定义错误，导致无法正确检测访问位。  
   **解决方法**：查阅 RISC-V 特权架构手册，确认 PTE\_A 位于第 6 位（1L << 6），修正宏定义。
2. **问题**：walk 函数返回空指针（pte == 0），触发 panic。  
   **解决方法**：检查用户传入的起始虚拟地址（va）是否对齐（必须是 PGSIZE 倍数），确保待检查页面在用户地址空间内，避免访问无效地址。

### 4）实验心得

本次实验通过实现 pgaccess 系统调用，深入理解了 RISC-V 页表中访问位（PTE\_A）的硬件机制 —— 每当页面被访问（读或写），硬件会自动置位 PTE\_A，这为跟踪页面访问提供了关键依据。清除 PTE\_A 位的操作则确保了系统调用能准确反映 “自上次检测以来的访问情况”，这对垃圾回收等需要动态跟踪内存使用的场景至关重要。

# **Lab: traps**

## **RISC-V assembly (**[**easy**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解 RISC-V 架构下函数调用的寄存器使用规范，掌握函数参数传递的寄存器分配机制。
2. 学会分析汇编代码中函数调用的实现方式，识别函数调用指令（如 jal、jalr）及内联函数的特征。
3. 掌握从汇编代码中获取函数地址、返回地址等关键信息的方法。
4. 理解小端序（Little-endian）与大端序（Big-endian）对数据存储的影响，能够根据字节序转换数据格式。
5. 分析函数参数数量不匹配时的输出结果，理解栈参数传递的不确定性。

### 2）实验步骤

**1. 分析 call.asm 汇编代码**

通过 make fs.img 编译 user/call.c 生成 user/call.asm，分析其中 g、f、main 函数的汇编实现，重点关注函数调用、寄存器使用及参数传递方式。

**2. 核心问题解答**

**（1）函数参数寄存器**

在 RISC-V 架构中，函数参数通过寄存器 a0-a7 传递，其中第 1 个参数存于 a0，第 2 个参数存于 a1，以此类推。例如，main 中调用 printf 时，参数 13 作为第 2 个参数存储在 a1 寄存器中。

**（2）main 中对 f 和 g 的调用位置**

在 call.asm 中，f 和 g 可能被编译器内联（Inline）优化，因此未生成独立的 jal 调用指令。内联函数会将代码直接嵌入调用处，而非通过函数调用指令跳转，因此汇编中看不到对 f 和 g 的显式调用。

**（3）printf 函数的地址**

在 call.asm 中，printf 的地址可通过符号表或调用指令获取。例如，若存在指令 jal ra, printf，则 printf 的地址为该指令的目标地址（可从汇编注释或符号表中读取，如 0x0000000000001234）。

**（4）jalr 到 printf 后 ra 寄存器的值**

ra（返回地址寄存器）存储函数调用后的下一条指令地址。在 main 中调用 printf 时，jalr 指令执行后，ra 的值为 main 函数中 printf 调用指令的下一条指令地址（例如，若调用指令位于 0x0000000000001122，则 ra 为 0x0000000000001126）。

**3. 字节序与输出分析**

**（1）代码输出**

给定代码：

unsigned int i = 0x00646c72;

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

* i 的十六进制值 0x00646c72 按小端序存储，字节顺序为 0x72、0x6c、0x64、0x00，对应 ASCII 字符 'r'、'l'、'd'、'\0'。
* 57616 转换为十六进制为 0xe110，因此 "H%x" 输出 He110。
* 字符串参数 &i 指向 'rld'，因此 "Wo%s" 输出 World。
* 最终输出为 He110 World。

**（2）大端序下的调整**

若 RISC-V 为大端序，i 的字节顺序需为 0x00、0x64、0x6c、0x72 才能存储 'rld'，因此 i 应设为 0x726c6400。57616（0xe110）在大端序中存储顺序不影响其十六进制打印结果，无需修改。

**4. 参数数量不匹配的输出分析**

代码 printf("x=%d y=%d", 3); 中，格式字符串需要 2 个参数，但仅提供 1 个。此时，printf 会从栈或寄存器中读取第 2 个参数，但其值为不确定的垃圾值（可能是栈中的残留数据），因此 y= 后会打印随机值。

### 3）实验中遇到的问题及解决方法

1. **问题**：无法在汇编中找到 f 和 g 的调用指令。  
   **解决方法**：意识到编译器可能对短函数进行内联优化，内联函数无独立调用指令，需通过代码嵌入位置分析。
2. **问题**：混淆 ra 寄存器的作用，无法确定返回地址。  
   **解决方法**：查阅 RISC-V 手册，明确 ra 存储函数调用后的下一条指令地址，通过汇编指令位置计算该值。

### 4）实验心得

本次实验通过分析 RISC-V 汇编代码，深入理解了函数调用的底层机制。RISC-V 的寄存器参数传递规范（a0-a7）和返回地址寄存器（ra）的使用，让我清晰看到了高级语言函数调用在硬件层面的实现。内联函数的优化方式则体现了编译器对代码效率的提升，尽管增加了汇编分析的难度，但也展示了软硬件协同设计的重要性。

## **Backtrace (**[**moderate**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解 RISC-V 架构下栈帧的结构与布局，掌握函数调用时栈帧的创建与链接机制。
2. 实现 backtrace() 函数，能够通过遍历栈帧获取当前函数调用链的返回地址，生成函数调用回溯信息。
3. 学会通过读取栈帧指针（s0 寄存器）和解析栈帧内容，递归向上遍历整个调用栈。
4. 将回溯功能集成到内核中，在 sys\_sleep 系统调用和 panic 函数中调用，以便调试时查看函数调用路径。
5. 验证回溯功能的正确性，通过 bttest 测试程序和 addr2line 工具解析返回地址对应的源代码位置。

### 2）实验步骤

**1. 实现栈帧指针读取函数**

在 kernel/riscv.h 中添加读取栈帧指针（s0 寄存器）的内联汇编函数，用于获取当前栈帧的起始地址：

static inline uint64

r\_fp()

{

uint64 x;

asm volatile("mv %0, s0" : "=r" (x) );

return x;

}

该函数通过内联汇编直接读取 s0 寄存器，s0 在 RISC-V 中用于存储当前栈帧指针。

**2. 实现 backtrace() 函数**

在 kernel/printf.c 中实现回溯函数，通过栈帧指针遍历调用栈，打印每个栈帧中保存的返回地址：

void

backtrace(void)

{

uint64 fp = r\_fp(); // 获取当前栈帧指针

uint64 stack\_page\_start = PGROUNDDOWN(fp);

printf("backtrace:\n");

while (fp != stack\_page\_start) {

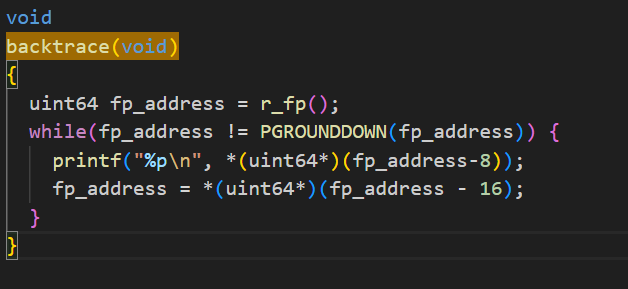
uint64 ret\_addr = \*(uint64\*)(fp - 8);

printf("%p\n", ret\_addr);

fp = \*(uint64\*)(fp - 16);

}

}



**栈帧结构解析**：

* 每个栈帧由栈帧指针 fp（s0）标识，fp 指向栈帧内的固定位置。
* fp-8：存储当前函数的返回地址（调用者的下一条执行指令）。
* fp-16：存储上一级栈帧的 fp（调用者的栈帧指针），形成链表结构。
* 内核栈为单页内存（页对齐），因此当 fp 到达页起始地址（PGROUNDDOWN(fp)）时，遍历终止。

**3. 集成回溯功能到内核**

1. **添加函数原型**：在 kernel/defs.h 中声明 backtrace()，确保其他文件可调用：
2. **在 sys\_sleep 中调用**：修改 kernel/sysproc.c 的 sys\_sleep 函数，添加 backtrace() 调用，使得 bttest 测试时触发回溯：

int

sys\_sleep(void)

{

int n;

uint ticks0;

backtrace(); // 调用回溯函数

argint(0, &n);

if(n < 0)

n = 0;

// ... 原有逻辑 ...

}

1. **在 panic 中调用**：修改 kernel/printf.c 的 panic 函数，添加 backtrace()，使得内核崩溃时自动打印调用栈：

### 3）实验中遇到的问题及解决方法

1. **问题**：无法正确读取栈帧指针，r\_fp() 函数返回值错误。  
   **解决方法**：最初误将函数命名为 r\_sp() 并读取 sp 寄存器（栈指针），导致获取的不是栈帧指针。修正函数名为 r\_fp() 并读取 s0 寄存器，正确获取栈帧指针。
2. **问题**：回溯遍历无限循环，未正确终止。  
   **解决方法**：未正确计算栈页起始地址，终止条件错误。通过 PGROUNDDOWN(fp) 获取栈所在页的起始地址，当 fp 等于该地址时终止循环，避免越界访问。

### 4）实验心得

本次实验通过实现回溯功能，深入理解了 RISC-V 架构下函数调用的栈帧机制。栈帧通过 s0 寄存器形成链表结构，每个栈帧保存返回地址和上一级栈帧指针，这种设计使得遍历调用链成为可能。

## **Alarm (**[hard](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解 xv6 中定时器中断的处理机制，掌握用户态与内核态在中断场景下的协作方式。
2. 实现 sigalarm(interval, handler) 系统调用，使进程在消耗指定 CPU 时钟 tick 后自动执行用户定义的 handler 函数。
3. 实现 sigreturn() 系统调用，确保 handler 执行完成后，进程能从被中断的位置正确恢复执行，且寄存器状态保持一致。
4. 解决 handler 重入问题（避免在 handler 未返回时再次触发），确保周期性 alarm 功能正确。
5. 通过 alarmtest 测试程序和 usertests -q 验证实现的正确性，确保无副作用影响其他内核功能。

### 2）实验步骤

**1. 准备工作：添加系统调用框架**

**（1）修改 Makefile 编译测试程序**

在 Makefile 的 UPROGS 中添加 $U/\_alarmtest，确保 alarmtest.c 被编译为用户程序：

**（2）定义系统调用接口与编号**

* 在 user/user.h 中添加 sigalarm 和 sigreturn 原型：
* 在 user/usys.pl 中添加系统调用 stub 生成指令：
* 在 kernel/syscall.h 中定义系统调用编号：
* 在 kernel/syscall.c 的 syscalls 数组中注册处理函数：

c

**2. 扩展进程结构存储 alarm 信息**

在 kernel/proc.h 的 struct proc 中添加字段，记录 alarm 配置、状态和被中断的上下文：

struct proc {

struct spinlock lock;

// Alarm 相关字段

uint64 handler\_va; // 用户 handler 函数虚拟地址

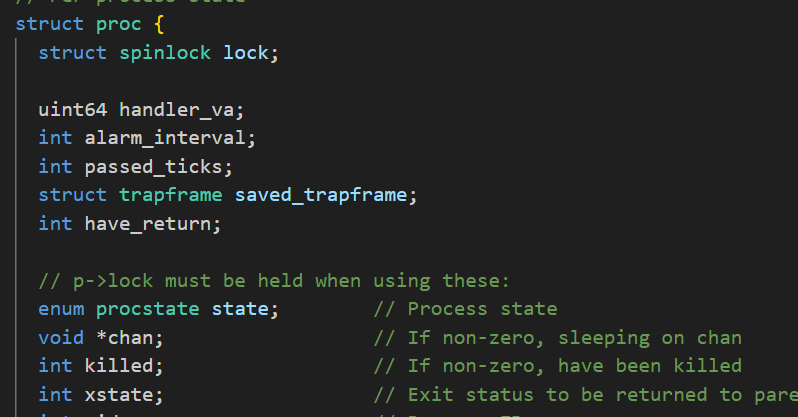
int alarm\_interval; // 触发间隔（ticks）

int passed\_ticks; // 已流逝的 ticks

struct trapframe saved\_trapframe; // 保存被中断时的陷阱帧

int have\_return; // 标记 handler 是否已返回（防止重入）

... 其他原有字段 ...

}; 

**3. 实现 sys\_sigalarm 系统调用**

在 kernel/sysproc.c 中实现 sys\_sigalarm，用于设置 alarm 间隔和 handler：

uint64 sys\_sigalarm(void) {

int ticks;

uint64 handler\_va;

// 从用户态获取参数：ticks 间隔和 handler 地址

argint(0, &ticks);

argaddr(1, &handler\_va);

struct proc \*p = myproc();

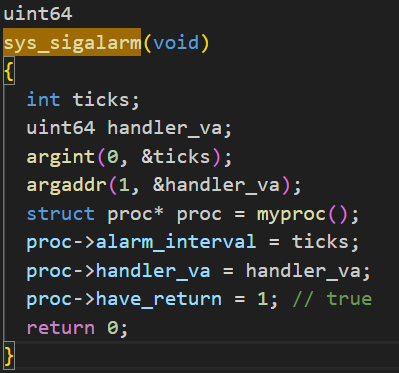
p->alarm\_interval = ticks;

p->handler\_va = handler\_va;

p->have\_return = 1;

p->passed\_ticks = 0;

return 0;

} 

**4. 实现定时器中断处理逻辑**

在 kernel/trap.c 的 usertrap 函数中，处理定时器中断（which\_dev == 2），累计 ticks 并触发 handler：

void usertrap(void) {

... 原有逻辑 ...

// 处理定时器中断

if (which\_dev == 2) {

struct proc \*p = myproc();

// 若已设置 alarm 且 handler 已返回（可触发）

if (p->alarm\_interval > 0 && p->have\_return) {

p->passed\_ticks++; // 累计 ticks

// 当累计 ticks 达到间隔时，触发 handler

if (p->passed\_ticks >= p->alarm\_interval) {

// 保存当前陷阱帧（用于后续恢复）

p->saved\_trapframe = \*p->trapframe;

// 修改 epc 为 handler 地址，用户态将执行 handler

p->trapframe->epc = p->handler\_va;

// 重置计数，标记 handler 未返回（防止重入）

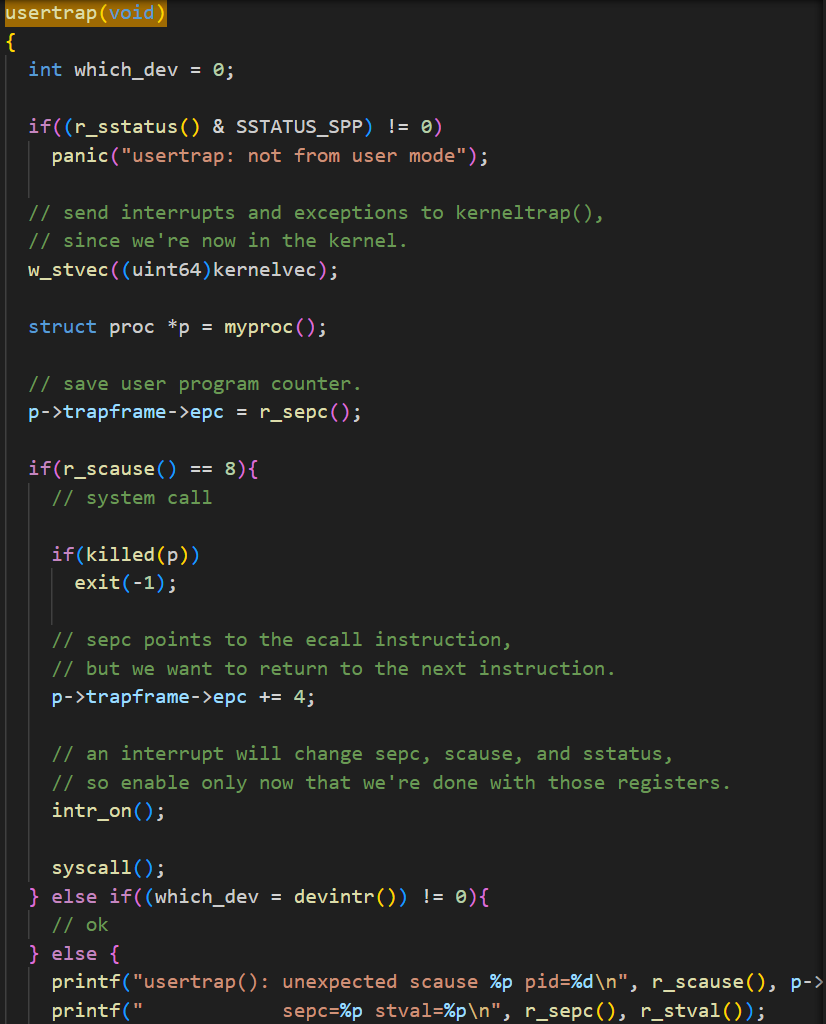
p->passed\_ticks = 0;

p->have\_return = 0;

}

}

yield(); // 定时器中断后让出 CPU

} 

usertrapret(); // 返回到用户态

}

**5. 实现 sys\_sigreturn 系统调用**

在 kernel/sysproc.c 中实现 sys\_sigreturn，用于从 handler 恢复到被中断的状态：

uint64 sys\_sigreturn(void) {

struct proc \*p = myproc();

\*p->trapframe = p->saved\_trapframe;

// 标记 handler 已返回，允许下次触发

p->have\_return = 1;

return p->trapframe->a0; // 恢复原系统调用的返回值

}

### 3）实验中遇到的问题及解决方法

1. **问题**：handler 执行后进程崩溃，无法恢复原执行流程。  
   **解决方法**：未正确保存和恢复陷阱帧（trapframe）。在触发 handler 前，通过 p->saved\_trapframe = \*p->trapframe 保存所有寄存器状态；sigreturn 时恢复该陷阱帧，确保进程从被中断的指令继续执行。
2. **问题**：handler 被重复触发（重入），导致测试 2 失败。  
   **解决方法**：添加 have\_return 标记，当 handler 开始执行时设为 0，sigreturn 后设为 1。定时器中断处理时仅当 have\_return == 1 才允许触发新的 handler，防止重入。

### 4）实验心得

本次实验深入理解了 xv6 的中断处理与进程状态管理机制。sigalarm 和 sigreturn 的协作设计展示了用户态与内核态如何通过系统调用和陷阱帧实现 “用户级中断”：内核负责计时和触发，用户态负责执行自定义逻辑，最终通过保存 / 恢复上下文确保无缝衔接。

# **Lab: Copy-on-Write Fork for xv6**

## **Implement copy-on-write fork(**[**hard**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解写时复制（Copy-on-Write, COW）机制的原理，通过延迟物理内存页的复制，提高 fork 系统调用的效率并节省内存。
2. 修改 uvmcopy 函数，使父子进程共享物理内存页而非立即复制，同时清除页表项（PTE）的写权限并标记为 COW 页。
3. 实现缺页中断处理逻辑，当进程写入 COW 页时，分配新物理页、复制内容并更新页表项，恢复写权限。
4. 维护物理页的引用计数，确保物理页在最后一个引用它的进程释放后才被真正回收。
5. 适配 copyout 函数，使其正确处理 COW 页，避免内核写入共享页时的冲突。
6. 通过 cowtest 和 usertests -q 验证实现的正确性，确保所有测试用例通过。

### 2）实验步骤

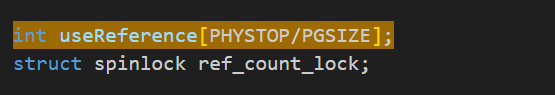
**1. 维护物理页引用计数**

在 kernel/kalloc.c 中添加引用计数机制，跟踪每个物理页被多少进程引用：

* 定义引用计数数组 useReference（按物理页地址索引）和保护锁 ref\_count\_lock：

int useReference[PHYSTOP/PGSIZE]; // 每个物理页的引用计数

struct spinlock ref\_count\_lock; // 保护引用计数的锁



* 修改 kalloc 初始化新分配页的引用计数为 1：

if(r) {

kmem.freelist = r->next;

acquire(&ref\_count\_lock);

useReference[(uint64)r / PGSIZE] = 1; // 新页初始引用计数为1

release(&ref\_count\_lock);

}

* 修改 kfree 仅在引用计数为 0 时释放页：

acquire(&ref\_count\_lock);

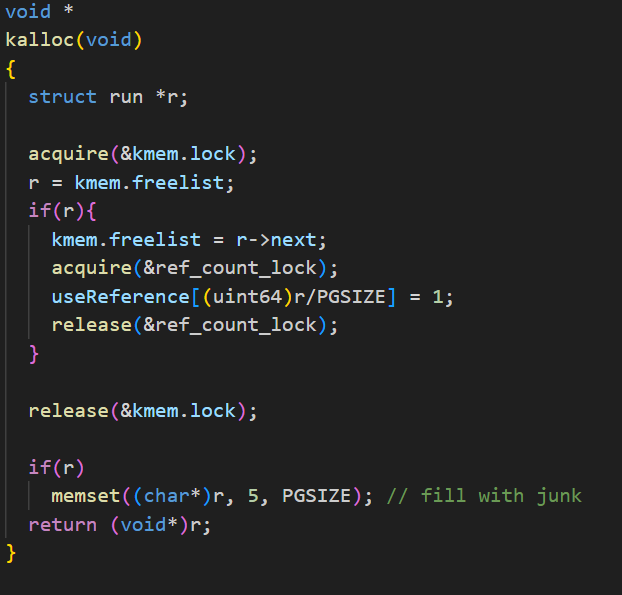
useReference[(uint64)pa/PGSIZE] -= 1; // 减少引用计数

int temp = useReference[(uint64)pa/PGSIZE];

release(&ref\_count\_lock);

if (temp > 0) // 仍有引用，不释放

return;



**2. 修改 uvmcopy 实现 COW 共享**

在 kernel/vm.c 中修改 uvmcopy 函数，使父子进程共享物理页而非复制：

* 对父进程中可写的页（PTE\_W 置位），清除其写权限并标记为 COW 页（使用 PTE 的 RSW 保留位）：

if (\*pte & PTE\_W) {

\*pte &= ~PTE\_W; // 清除写权限

\*pte |= PTE\_RSW; // 标记为COW页

}

* 增加共享页的引用计数，将父进程的物理页直接映射到子进程页表：

pa = PTE2PA(\*pte);

acquire(&ref\_count\_lock);

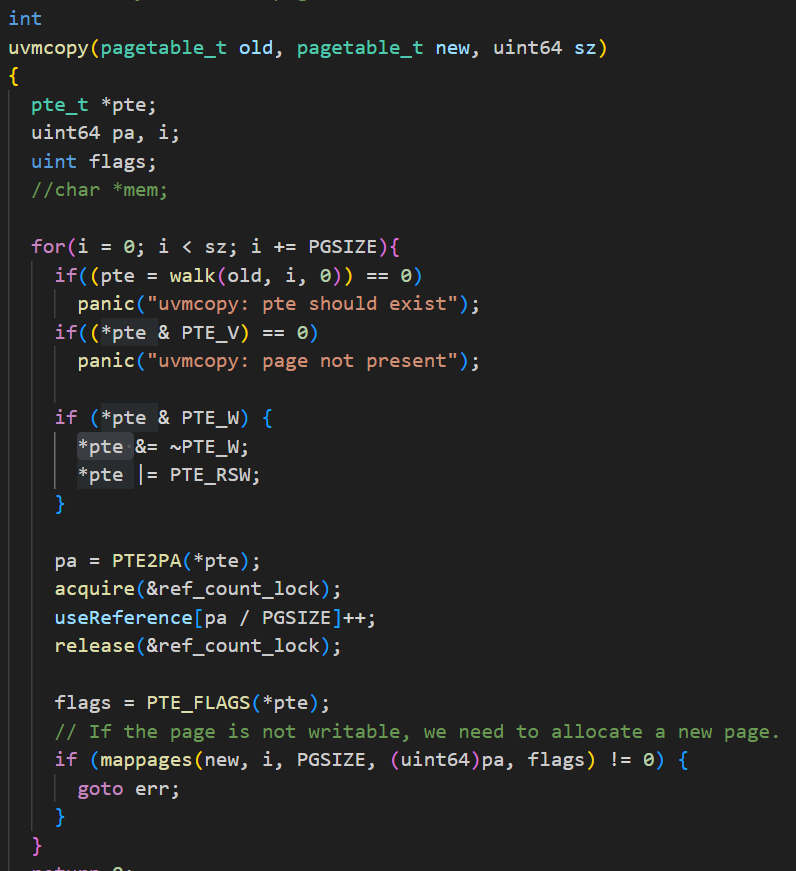
useReference[pa/PGSIZE] += 1;

release(&ref\_count\_lock);

if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0) {

goto err;

}



**3. 实现 COW 缺页中断处理**

在 kernel/trap.c 中添加缺页中断处理逻辑，当写入 COW 页时复制页面：

* 定义 cowhandler 函数，处理写时缺页：

int cowhandler(pagetable\_t pagetable, uint64 va) {

if (va >= MAXVA || (pte = walk(pagetable, va, 0)) == 0 ||

(\*pte & (PTE\_RSW | PTE\_U | PTE\_V)) != (PTE\_RSW | PTE\_U | PTE\_V)) {

return -1;

}

if ((mem = kalloc()) == 0) return -1; // 内存不足

uint64 pa = PTE2PA(\*pte);

memmove(mem, (char\*)pa, PGSIZE);

kfree((void\*)pa);

uint flags = PTE\_FLAGS(\*pte);

\*pte = (PA2PTE(mem) | flags | PTE\_W) & ~PTE\_RSW;

return 0;

}

* 在 usertrap 中处理写缺页中断（r\_scause() == 15）：

else if (r\_scause() == 15) { // 写缺页中断

uint64 va = r\_stval(); // 故障虚拟地址

if (va >= p->sz || cowhandler(p->pagetable, va) != 0) {

p->killed = 1; // 处理失败，终止进程

}

}

**4. 适配 copyout 处理 COW 页**

在 kernel/vm.c 中修改 copyout 函数，确保内核写入用户 COW 页时触发复制：

* 检查目标页是否为 COW 页，若是则分配新页并更新页表：

pte\_t \*pte = walk(pagetable, va0, 0);

if (checkcowpage(va0, pte, p)) { // 检查COW标记

if ((mem = kalloc()) == 0) {

p->killed = 1;

return -1;

}

memmove(mem, (char\*)pa0, PGSIZE); // 复制内容

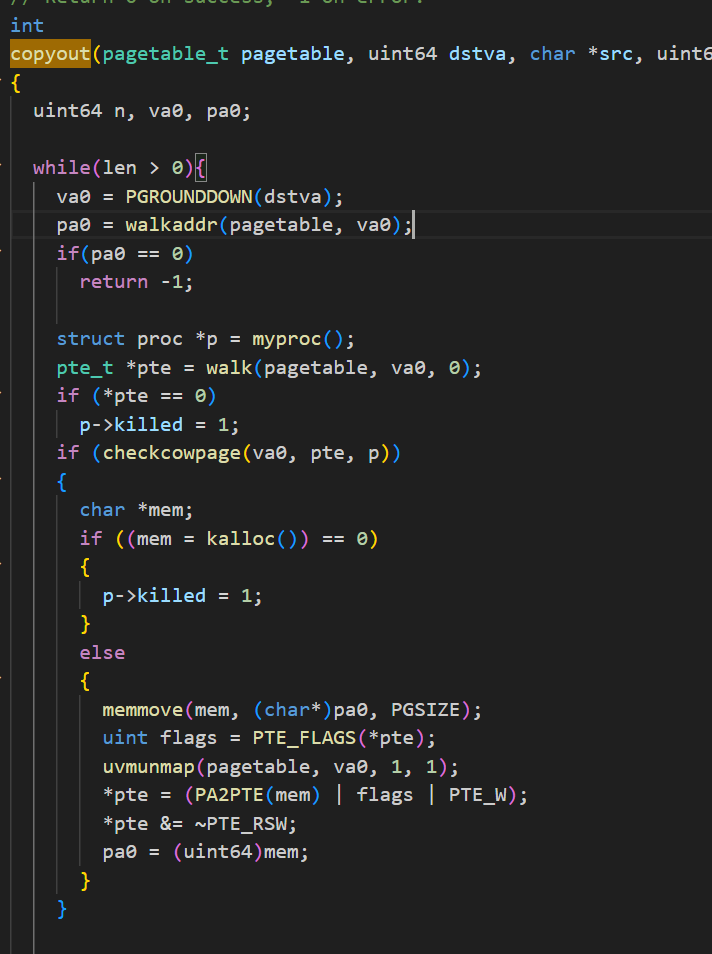
uvmunmap(pagetable, va0, 1, 1); // 减少原页引用

// 更新页表项为新页，恢复写权限

\*pte = (PA2PTE(mem) | flags | PTE\_W) & ~PTE\_RSW;

pa0 = (uint64)mem; // 指向新页

}



### 3）实验中遇到的问题及解决方法

1. **问题**：引用计数同步错误，导致物理页被过早释放或内存泄漏。  
   **解决方法**：使用 ref\_count\_lock 保护引用计数的增减操作，确保多进程并发修改时的原子性。
2. **问题**：COW 页标记错误，导致非 COW 页（如只读文本段）被误处理。  
   **解决方法**：仅对父进程中原本可写（PTE\_W 置位）的页标记 COW（PTE\_RSW），只读页保持共享且不标记，写入时直接终止进程。

### 4）实验心得

本次实验深入理解了写时复制（COW）机制在提高 fork 效率中的核心作用。通过延迟物理页的复制，COW 显著减少了 fork 时的内存开销，尤其对内存密集型进程效果明显。实现过程中，引用计数的设计是关键 —— 它准确跟踪物理页的共享状态，确保页在最后一个引用释放后才被回收，避免了内存泄漏和悬垂指针。

# **Lab: Multithreading**

## **Uthread: switching between threads (**[**moderate**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解用户级线程的上下文切换机制，掌握线程创建与调度的核心原理。
2. 设计并实现线程上下文结构，保存和恢复线程切换时的寄存器状态。
3. 完成 thread\_create 函数，为新线程初始化栈和上下文，确保线程首次运行时能正确执行目标函数。
4. 实现 thread\_switch 汇编函数，完成线程间的上下文切换，包括保存当前线程状态和恢复目标线程状态。
5. 修改 thread\_schedule 函数，在调度时调用 thread\_switch 实现线程切换，确保多线程按预期交替执行。
6. 验证实现的正确性，确保 uthread 程序能输出三个线程交替执行的结果，通过 make grade 测试。

### 2）实验步骤

**1. 定义线程上下文结构**

在 user/uthread.c 中定义 struct context，用于保存线程切换时需要保存的寄存器（主要是被调用者保存寄存器）：

struct context {

uint64 ra; // 返回地址寄存器

uint64 sp; // 栈指针寄存器

uint64 s0;

uint64 s1;

uint64 s2;

uint64 s3;

uint64 s4;

uint64 s5;

uint64 s6;

uint64 s7;

uint64 s8;

uint64 s9;

uint64 s10;

uint64 s11;

};

**选择被调用者保存寄存器的原因**：根据 RISC-V 调用约定，被调用者（如 thread\_switch）需要保存这些寄存器，确保切换后恢复线程时，寄存器状态与切换前一致。

**2. 扩展线程结构**

在 struct thread 中添加 context 字段，用于存储线程的上下文信息：

struct thread {

char stack[STACK\_SIZE]; /\* 线程栈 \*/

int state; /\* 状态：FREE, RUNNING, RUNNABLE \*/

struct context context; /\* 线程上下文（寄存器状态） \*/

};

**3. 实现 thread\_create 函数**

修改 thread\_create 函数，初始化新线程的上下文，使其首次运行时执行目标函数：

//uthread.c

void thread\_create(void (\*func)()) {

struct thread \*t;

for (t = all\_thread; t < all\_thread + MAX\_THREAD; t++) {

if (t->state == FREE) break;

}

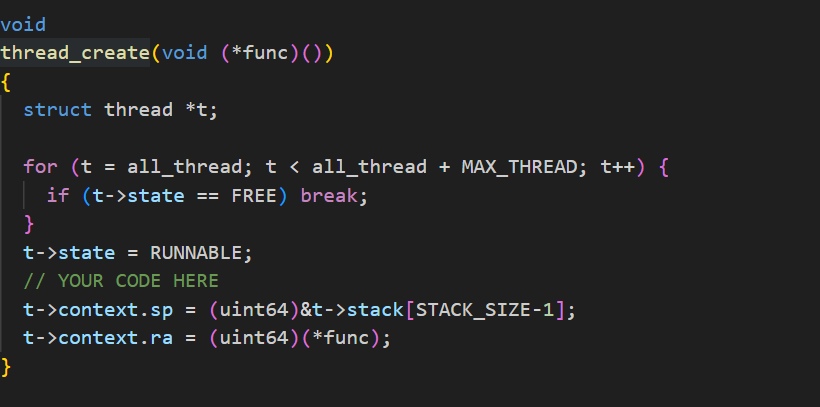
t->state = RUNNABLE;

// 初始化栈指针：指向栈顶（栈从高地址向低地址增长）

t->context.sp = (uint64)&t->stack[STACK\_SIZE - 1];

// 初始化返回地址：指向目标函数（线程首次运行时从该函数开始）

t->context.ra = (uint64)func;

} 

* 栈指针（sp）设置为线程栈的最高地址（stack[STACK\_SIZE-1]），符合栈的增长方向。
* 返回地址（ra）设置为目标函数（func）的地址，确保线程首次调度时从该函数开始执行。

**4. 实现 thread\_switch 汇编函数**

在 user/uthread\_switch.S 中实现上下文切换逻辑，保存当前线程的寄存器状态并恢复目标线程的状态：

* 函数参数 a0 为当前线程的 context 指针，a1 为目标线程的 context 指针。
* 通过 sd（存储双字）和 ld（加载双字）指令完成寄存器的保存和恢复。

**5. 修改 thread\_schedule 函数**

在调度逻辑中调用 thread\_switch 完成线程切换：

//uthread.c

void thread\_schedule(void) {

struct thread \*t, \*next\_thread;

// 查找下一个可运行的线程

next\_thread = 0;

for (t = all\_thread; t < all\_thread + MAX\_THREAD; t++) {

if (t->state == RUNNABLE) {

next\_thread = t;

break;

}

}

if (next\_thread == 0) {

printf("thread\_schedule: no runnable threads\n");

exit(1);

}

// 切换到下一个线程

struct thread \*old\_thread = current\_thread;

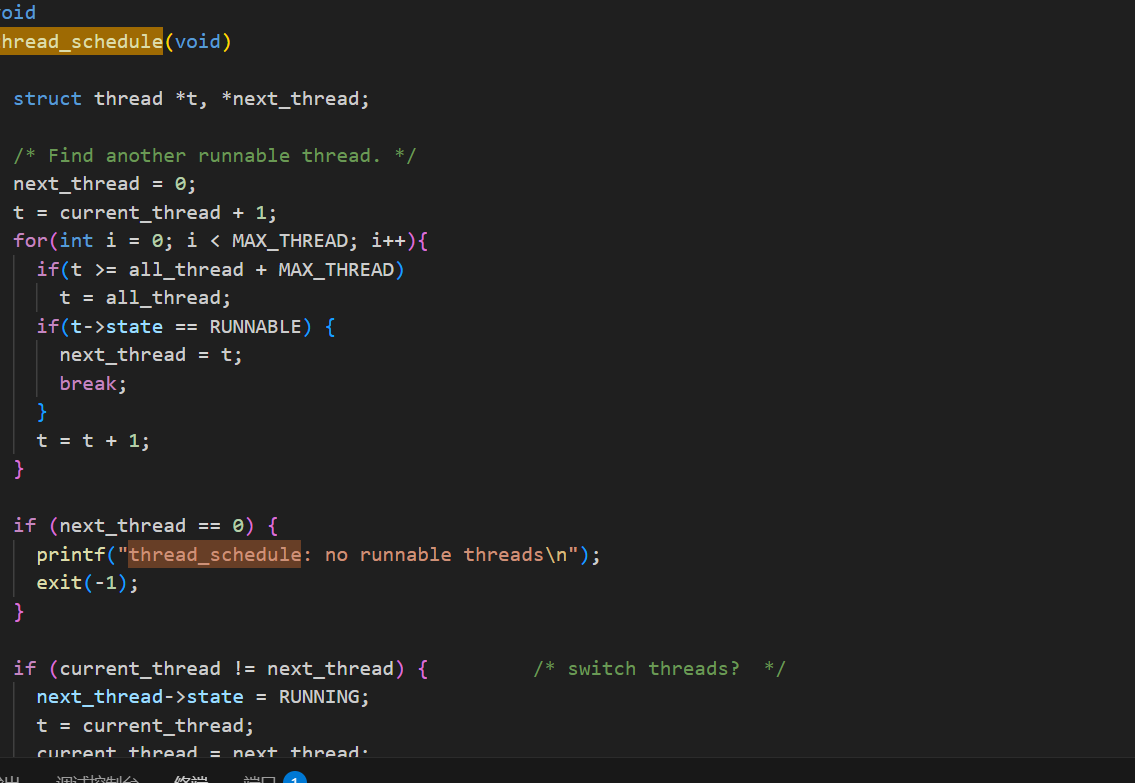
current\_thread = next\_thread;

current\_thread->state = RUNNING;

// 调用线程切换函数：保存旧线程上下文，恢复新线程上下文

thread\_switch(&old\_thread->context, &current\_thread->context);

}



### 3）实验中遇到的问题及解决方法

1. **问题**：新线程创建后首次运行时崩溃，未执行目标函数。  
   **解决方法**：thread\_create 中未正确初始化 ra 寄存器。将 t->context.ra 设置为目标函数地址，确保线程首次调度时从该函数开始执行。
2. **问题**：栈指针设置错误，导致线程运行时栈溢出或数据损坏。  
   **解决方法**：栈在 RISC-V 中从高地址向低地址增长，需将 sp 初始化为 &t->stack[STACK\_SIZE-1]（栈顶），而非栈底。

### 4）实验心得

本次实验深入理解了用户级线程的上下文切换机制，核心在于保存和恢复线程的寄存器状态。通过定义 struct context 存储关键寄存器，thread\_switch 汇编函数实现状态的切换，使得线程能够在暂停后准确恢复执行。线程创建时的栈和上下文初始化是关键步骤。栈指针的正确设置确保了线程有足够的栈空间执行函数，而返回地址的初始化则决定了线程首次运行的入口。这让我意识到，用户级线程的本质是通过控制栈和寄存器状态，欺骗 CPU 执行不同的代码段。

## **Using threads (**[**moderate**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解多线程环境下共享数据结构的并发访问问题，识别哈希表操作中的竞态条件。
2. 掌握线程同步机制（互斥锁）的使用，通过添加适当的锁确保哈希表操作的线程安全性。
3. 优化锁的粒度，从全局锁改进为细粒度的桶级锁，在保证正确性的同时提高并行性能。
4. 验证多线程程序的正确性（确保无丢失键）和性能（多线程相比单线程的加速比≥1.25）。

### 2）实验步骤

**1. 问题分析：多线程下的键丢失原因**

原始哈希表在多线程环境下（如ph 2）出现键丢失，根本原因是**竞态条件**：当两个线程同时操作哈希表的同一链表时，可能导致节点被覆盖。例如：

* 线程 1 计算哈希值后，准备插入新节点到链表头部（table[i]）。
* 线程 2 同时计算相同哈希值，插入新节点并更新table[i]。
* 线程 1 继续执行时，仍使用旧的table[i]值作为新节点的next，导致线程 2 插入的节点被覆盖，最终丢失。

**2. 初步解决：全局锁保证正确性**

为解决竞态条件，首先在put操作中添加全局锁，确保同一时间只有一个线程修改哈希表：

//ph.c

pthread\_mutex\_t global\_lock; // 全局锁

void put(int key, int value) {

int i = key % NBUCKET;

pthread\_mutex\_lock(&global\_lock); // 加锁

struct entry \*e = table[i];

while (e) {

if (e->key == key) {

e->value = value;

pthread\_mutex\_unlock(&global\_lock); // 解锁

return;

}

e = e->next;

}

insert(key, value, &table[i], table[i]); // 插入新节点

pthread\_mutex\_unlock(&global\_lock); // 解锁

}

* **效果**：键丢失问题解决，但多线程性能不佳（并行性被完全抑制）。

**3. 性能优化：桶级锁提高并行度**

全局锁虽保证正确性，但限制了所有线程的并行操作。由于哈希表的不同桶（table[i]）相互独立，可使用**桶级锁**（每个桶一个锁），允许不同桶上的操作并行执行：

**（1）定义桶级锁数组**

pthread\_mutex\_t lock[NBUCKET]; // 每个桶对应一个锁

**（2）初始化锁**

在main函数中初始化所有桶的锁：

**（3）修改put操作，仅锁定对应桶**

void put(int key, int value) {

int i = key % NBUCKET;

struct entry \*e = table[i];

while (e) {

if (e->key == key) {

e->value = value;

return;

}

e = e->next;

}

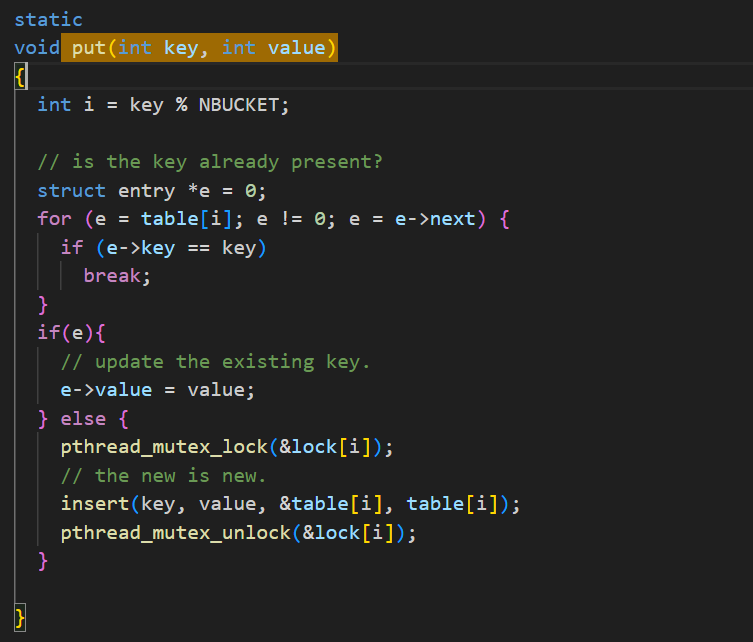
// 仅锁定当前桶，其他桶可并行操作

pthread\_mutex\_lock(&lock[i]);

insert(key, value, &table[i], table[i]); // 插入新节点（临界区）

pthread\_mutex\_unlock(&lock[i]);

}



### 3）实验中遇到的问题及解决方法

1. **问题**：全局锁导致双线程性能未提升（甚至下降）。

**解决方法**：分析哈希表结构，发现不同桶的操作无冲突，改用桶级锁，允许不同桶上的并行操作，提升性能。

1. **问题**：锁初始化遗漏，导致运行时崩溃。

**解决方法**：在main函数中通过循环初始化所有桶的锁（pthread\_mutex\_init），确保每个锁可用。

### 4）实验心得

本次实验深入理解了多线程并发访问共享数据结构的核心问题 —— 竞态条件，以及如何通过锁机制保证线程安全。全局锁虽简单，但完全扼杀并行性，不适用于高性能场景。而细粒度锁（如桶级锁）通过降低锁的范围，在保证正确性的同时保留了并行能力，显著提升了多线程性能。

## **Barrier(**[**moderate**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

本次实验旨在实现一个线程同步屏障，确保所有参与线程都到达屏障点后，才能继续执行后续操作。具体要求如下：

1. 使用 pthread 条件变量实现线程间的同步
2. 解决多个线程在屏障处的同步问题，避免线程提前离开屏障
3. 正确处理多轮屏障调用，确保每轮屏障都能正确同步所有线程
4. 处理线程快速循环可能导致的竞争条件

### 2）实验步骤

1. **环境准备**：在真实计算机环境中（非 xv6 或 qemu 模拟器），使用提供的 barrier.c 文件作为基础
2. **代码实现**：修改 barrier () 函数，实现正确的屏障功能

static void

barrier()

{

pthread\_mutex\_lock(&bstate.barrier\_mutex); // 锁定互斥锁

// 增加已到达本轮屏障的线程计数

bstate.nthread++;

// 如果不是所有线程都到达屏障，则等待

if (bstate.nthread < nthread) {

pthread\_cond\_wait(&bstate.barrier\_cond, &bstate.barrier\_mutex);

}

// 如果所有线程都到达屏障，增加轮次并重置计数器

if (bstate.nthread == nthread) {

bstate.round++;

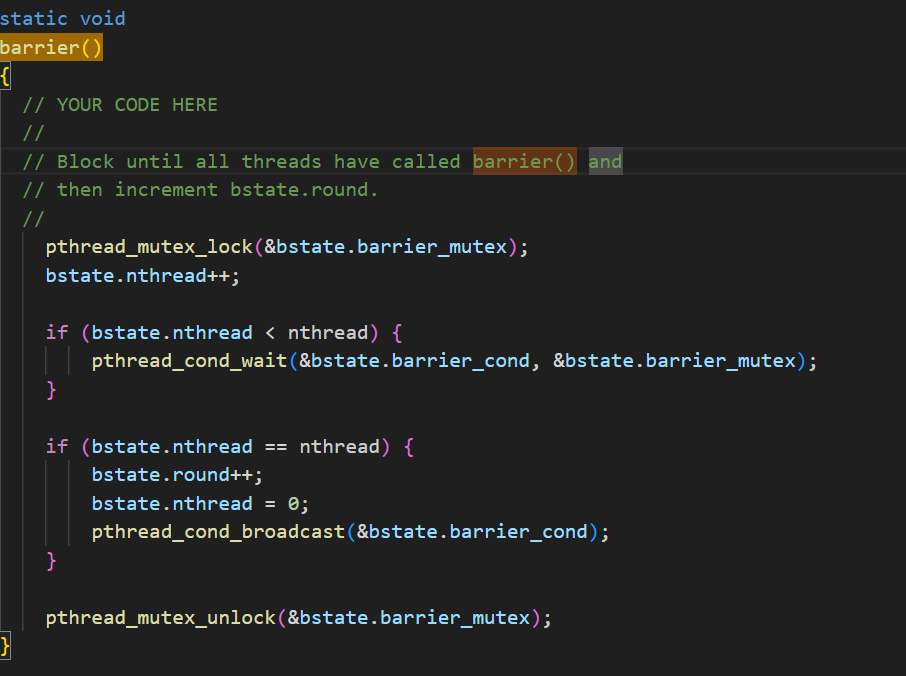
bstate.nthread = 0;

pthread\_cond\_broadcast(&bstate.barrier\_cond); // 通知所有等待线程

}

pthread\_mutex\_unlock(&bstate.barrier\_mutex); // 解锁互斥锁

}

a

1. **编译测试**：

使用make barrier命令编译程序

分别使用 1 个、2 个和多个线程进行测试：./barrier 1、./barrier 2、./barrier 4

验证程序是否能正常运行，无断言错误发生

1. **通过测试**：确保程序能够通过make grade的 barrier 测试

### 3）实验中遇到的问题及解决方法

**问题 ：断言错误**

初始代码运行时出现断言错误Assertion 'i == t' failed，这是因为一个线程在其他线程到达屏障前就离开了屏障。

**解决方法**：使用条件变量确保所有线程都到达屏障后才继续执行，通过pthread\_cond\_wait让先到达的线程等待，直到最后一个线程到达后使用pthread\_cond\_broadcast唤醒所有等待线程。

**问题 ：竞争条件**

线程快速循环时，可能在其他线程还未离开屏障前就开始下一轮，导致计数器混乱。

**解决方法**：通过互斥锁确保对共享变量bstate.nthread和bstate.round的操作是原子的，避免多个线程同时修改这些变量。

### 4）实验心得

通过本次屏障实现实验，我深入理解了条件变量的工作机制：pthread\_cond\_wait会原子地释放互斥锁并进入等待状态，被唤醒时又会重新获取互斥锁，这一特性是实现复杂同步的关键。线程同步的复杂性：即使是看似简单的屏障功能，也需要仔细处理多线程间的交互，特别是在多轮同步的情况下，需要正确管理状态变量的重置。这次实验让我对多线程编程中的同步机制有了更深入的理解，也认识到在并发编程中细致处理每一个同步细节的重要性。

# **Lab: networking**

## **Your Job (**[**hard**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解 E1000 网卡的工作原理，掌握 DMA（直接内存访问）机制在网络数据传输中的应用。
2. 实现 E1000 网卡驱动的发送函数（e1000\_transmit）和接收函数（e1000\_recv），完成数据包的收发功能。
3. 掌握描述符环（Descriptor Ring）机制，正确处理发送环（TX Ring）和接收环（RX Ring）的循环使用。
4. 解决多线程 / 中断环境下的并发问题，通过锁机制确保驱动操作的线程安全性。
5. 验证驱动正确性，确保通过make grade的所有测试，支持 xv6 操作系统完成网络通信（包括 ARP、UDP、DNS 等协议交互）。

### 2）实验步骤

**实验背景**

E1000 网卡通过描述符环实现与内存的数据交互：发送环（TX Ring）用于存放待发送数据包的描述符，接收环（RX Ring）用于存放接收数据包的描述符。驱动需通过操作这些描述符和网卡控制寄存器，完成数据包的收发管理。

**核心实现步骤**

**1. 发送函数（e1000\_transmit）实现**

发送函数负责将网络栈传递的数据包（mbuf）放入 TX Ring，通知网卡发送，并管理资源释放。

//e1000.c

int e1000\_transmit(struct mbuf \*m) {

acquire(&e1000\_lock); // 加锁确保并发安全

// 1. 获取当前TX Ring的下一个可用描述符索引（通过TDT寄存器）

uint32 tail = regs[E1000\_TDT];

struct tx\_desc\* desc = tx\_ring + tail;

// 2. 检查当前描述符是否已完成发送（DD位是否置位）

if ((desc->status & E1000\_TXD\_STAT\_DD) == 0) {

release(&e1000\_lock);

return -1; // 描述符未释放，发送失败

}

// 3. 释放上一次发送的mbuf（若存在）

if (tx\_mbufs[tail])

mbuffree(tx\_mbufs[tail]);

// 4. 填充描述符：设置数据包地址、长度和命令

desc->addr = (uint64)m->head; // 数据包在内存中的地址

desc->length = m->len; // 数据包长度

desc->cmd = E1000\_TXD\_CMD\_EOP | E1000\_TXD\_CMD\_RS; // 标记帧结束并请求发送

// 5. 保存mbuf指针，待发送完成后释放

tx\_mbufs[tail] = m;

// 6. 更新TDT寄存器，通知网卡新描述符可用

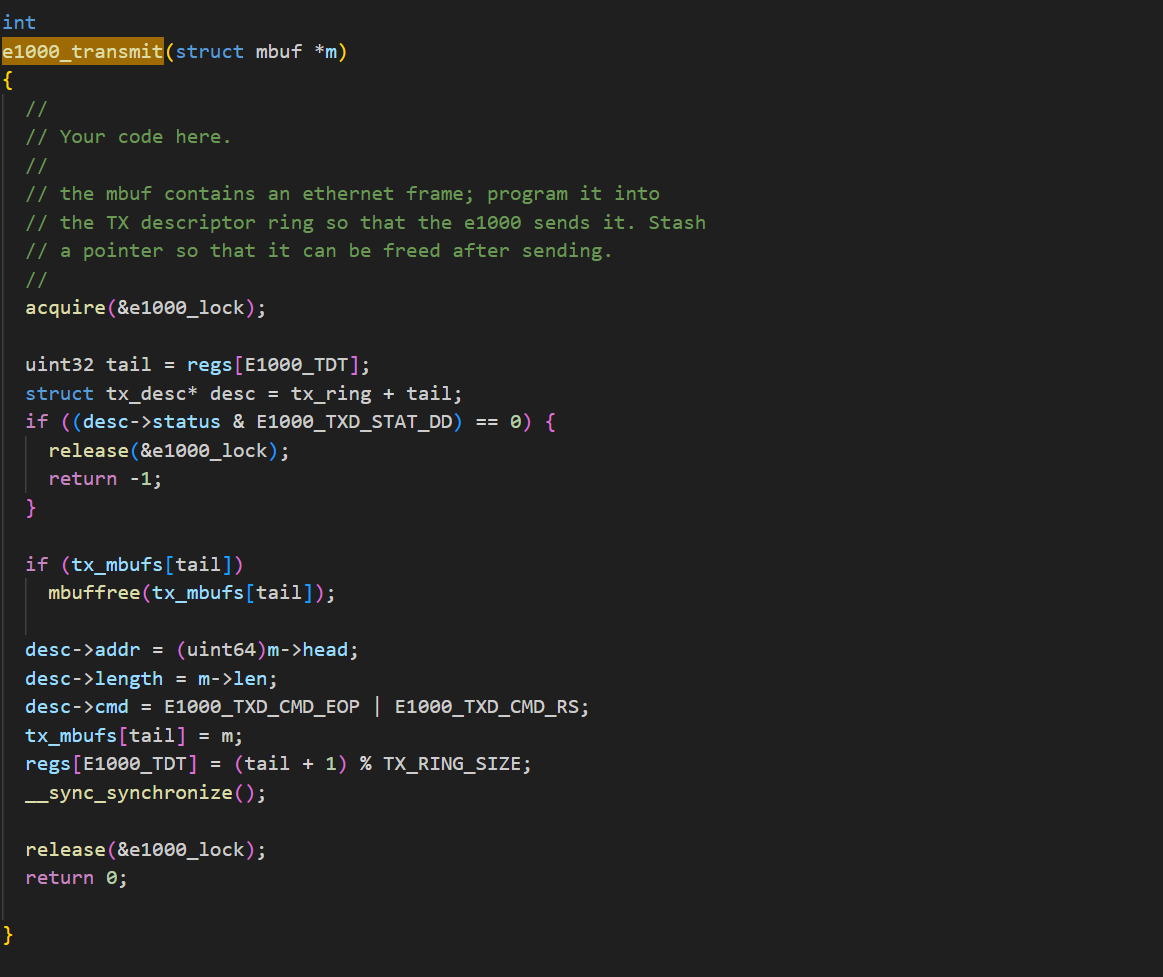
regs[E1000\_TDT] = (tail + 1) % TX\_RING\_SIZE;

\_\_sync\_synchronize(); // 内存屏障，确保硬件可见更新

release(&e1000\_lock); // 解锁

return 0; // 发送成功

}



**2. 接收函数（e1000\_recv）实现**

接收函数负责从 RX Ring 中取出已接收的数据包，提交给网络栈，并为下一次接收准备新的缓冲区。

static void e1000\_recv(void) {

acquire(&e1000\_lock); // 加锁确保并发安全

while (1) {

// 1. 计算下一个待检查的描述符索引（RDT + 1）

uint32 tail = (regs[E1000\_RDT] + 1) % RX\_RING\_SIZE;

struct rx\_desc\* desc = rx\_ring + tail;

// 2. 检查是否有新数据包（DD位是否置位）

if ((desc->status & E1000\_RXD\_STAT\_DD) == 0)

break; // 无新数据包，退出循环

// 3. 处理接收的数据包：设置长度并提交给网络栈

struct mbuf\* m = rx\_mbufs[tail];

mbufput(m, desc->length); // 更新mbuf长度

// 4. 释放锁后提交数据包（避免网络栈操作持有锁）

release(&e1000\_lock);

net\_rx(m); // 提交给网络栈

acquire(&e1000\_lock);

// 5. 分配新mbuf作为下一次接收的缓冲区

m = mbufalloc(0);

rx\_mbufs[tail] = m;

desc->addr = (uint64)m->head; // 设置新缓冲区地址

desc->status = 0; // 清除状态位

// 6. 更新RDT寄存器，通知网卡描述符已处理

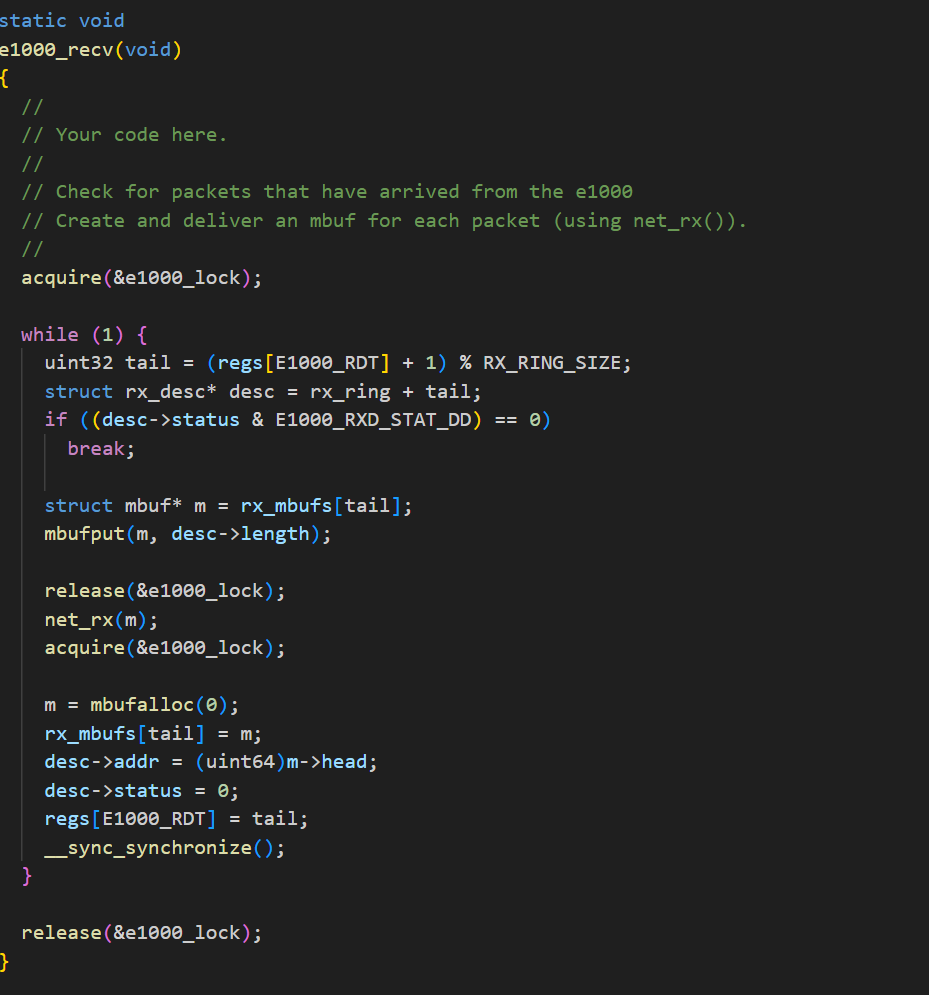
regs[E1000\_RDT] = tail;

\_\_sync\_synchronize(); // 内存屏障，确保硬件可见更新

}

release(&e1000\_lock); // 解锁

}



**关键机制说明**

* **描述符环管理**：TX/RX Ring 是循环数组，通过TDT（发送尾指针）和RDT（接收尾指针）寄存器跟踪网卡与驱动的操作位置。
* **状态同步**：通过描述符的DD（Descriptor Done）位判断网卡是否完成操作（发送完成 / 接收完成）。
* **并发控制**：使用互斥锁（e1000\_lock）避免多线程 / 中断同时操作描述符环和寄存器，确保数据一致性。
* **内存屏障**：\_\_sync\_synchronize()确保 CPU 对寄存器和描述符的修改被网卡硬件正确感知。

### 3）实验中遇到的问题及解决方法

**问题 1：发送数据包时断言失败或内存泄漏**

**现象**：多次发送后出现mbuf未释放，导致内存耗尽或断言错误。

**原因**：未及时释放已完成发送的mbuf，或在发送前未检查描述符是否可用。

**解决方法**：在填充新描述符前，检查DD位确认前一次发送完成，若完成则通过mbuffree释放旧mbuf，避免内存泄漏。

**问题 2：接收数据包后无法再次接收**

**现象**：只能接收 1-2 个数据包，后续无数据到达。

**原因**：未及时为接收描述符分配新的mbuf，或未更新RDT寄存器通知网卡。

**解决方法**：每次处理完接收数据包后，立即用mbufalloc分配新缓冲区，更新描述符地址，并将RDT寄存器更新为当前索引，让网卡知道新缓冲区可用。

### 4）实验心得

通过本次实验，我深入理解了网络设备驱动的工作原理，体会到软硬件协同设计的复杂性，也掌握了并发控制和资源管理在底层驱动中的实践方法。认识了网卡驱动的核心原理：E1000 网卡通过描述符环实现无 CPU 干预的数据传输（DMA），驱动的核心是管理这些描述符的生命周期，包括填充发送数据、回收发送资源、处理接收数据、准备新接收缓冲区。

# **Lab: locks**

## **Memory allocator (**[**moderate**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解 xv6 内存分配器中锁竞争的成因，分析单一全局锁对多核心环境下内存分配性能的影响。
2. 实现基于 per-CPU 空闲列表（per-CPU freelist）的内存分配器，通过细化锁粒度减少锁竞争。
3. 设计并实现内存窃取（stealing）机制，解决当某 CPU 空闲列表为空时的内存分配问题。
4. 验证优化后的内存分配器在减少锁竞争方面的效果，确保通过kalloctest和usertests的所有测试。
5. 掌握多核心环境下并发资源管理的设计思想，理解锁粒度与系统性能的平衡关系。

### 2）实验步骤

**1. 原始内存分配器的问题分析**

xv6 原始内存分配器采用单一全局空闲列表（freelist），并通过一个全局锁（kmem.lock）保护对空闲列表的访问。在多核心环境下，当多个 CPU 同时进行内存分配（kalloc）或释放（kfree）时，会频繁竞争该全局锁，导致大量的锁等待（体现在#test-and-set计数较高），严重影响系统性能。kalloctest的测试结果显示，原始分配器的kmem锁存在显著竞争（#test-and-set数值较大）。

**2. 优化方案设计：per-CPU 空闲列表**

为解决单一全局锁的竞争问题，采用 per-CPU 空闲列表设计，核心思路如下：

* 为每个 CPU 维护独立的空闲列表（freelist）和对应的锁，减少跨核心的锁竞争。
* 内存释放时，优先将内存块归还给当前 CPU 的空闲列表，提高本地内存利用率。
* 内存分配时，优先从当前 CPU 的空闲列表获取内存块；若本地列表为空，则从其他 CPU 的空闲列表中 “窃取” 内存块。

**3. 核心实现步骤**

**（1）数据结构设计**

修改内存分配器的核心数据结构，从单一全局空闲列表改为 per-CPU 数组结构：

struct {

struct spinlock lock; // 每个CPU独立的锁

struct run \*freelist; // 每个CPU独立的空闲列表

} kmem[NCPU]; // NCPU为核心数量

**（2）初始化多核心锁**

实现initmemlock函数，为每个 CPU 的空闲列表初始化锁，确保锁名称以 “kmem” 开头：

static void initmemlock() {

for (int i = 0; i < NCPU; ++i) {

initlock(&kmem[i].lock, "kmem"); // 锁名称符合要求

}

}

void kinit() {

initmemlock(); // 初始化所有CPU的锁

freerange(end, (void\*)PHYSTOP); // 初始内存全部分配给运行kinit的CPU

}

**（3）内存释放优化（kfree）**

修改kfree函数，将释放的内存块归还给当前 CPU 的空闲列表，减少跨核心锁竞争：

//kalloc.c

void kfree(void \*pa) {

struct run \*r;

// 验证内存块合法性（省略）

r = (struct run\*)pa;

push\_off(); // 关闭中断，确保cpuid获取安全

int id = cpuid(); // 获取当前CPU编号

acquire(&kmem[id].lock); // 加当前CPU的锁

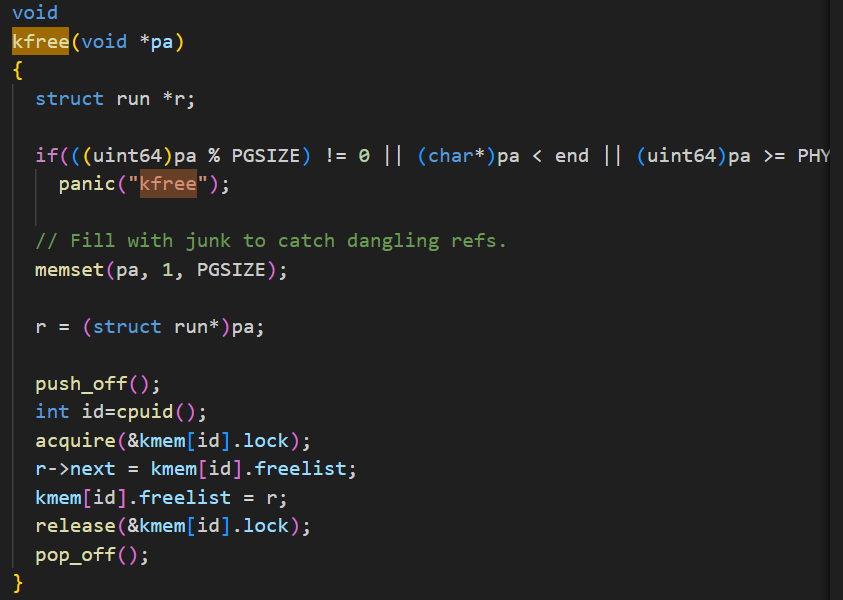
r->next = kmem[id].freelist; // 插入当前CPU的空闲列表

kmem[id].freelist = r;

release(&kmem[id].lock); // 释放当前CPU的锁

pop\_off(); // 恢复中断

}



**（4）内存分配优化（kalloc）**

修改kalloc函数，优先从当前 CPU 的空闲列表分配内存，若为空则从其他 CPU 窃取：

void \*kalloc(void) {

struct run \*r;

push\_off(); // 关闭中断

int id = cpuid(); // 获取当前CPU编号

// 1. 尝试从当前CPU的空闲列表分配

acquire(&kmem[id].lock);

r = kmem[id].freelist;

if (r) {

kmem[id].freelist = r->next; // 取下第一个空闲块

}

release(&kmem[id].lock);

// 2. 若当前CPU无空闲块，从其他CPU窃取

if (!r) {

for (int i = 0; i < NCPU; ++i) {

if (i == id) continue; // 跳过当前CPU

acquire(&kmem[i].lock);

r = kmem[i].freelist;

if (r) {

kmem[i].freelist = r->next; // 窃取一个空闲块

}

release(&kmem[i].lock);

if (r) break; // 成功窃取后退出循环

}

}

pop\_off(); // 恢复中断

if (r) memset((char\*)r, 5, PGSIZE); // 填充垃圾数据，便于调试

return (void\*)r;

}

**4. 关键机制说明**

* **per-CPU 隔离**：每个 CPU 操作本地空闲列表，通过独立锁实现并发隔离，减少锁竞争。
* **内存窃取**：当本地空闲列表为空时，遍历其他 CPU 的空闲列表获取内存，确保系统整体内存利用率。
* **中断控制**：使用push\_off()和pop\_off()关闭 / 开启中断，确保cpuid()获取的当前 CPU 编号安全有效，避免中断导致的 CPU 切换问题。
* **锁命名规范**：所有锁名称以 “kmem” 开头，符合实验要求，便于锁竞争统计。

### 3）实验中遇到的问题及解决方法

**问题 1：CPU 编号获取不安全**

**现象**：多核心环境下，cpuid()返回的 CPU 编号可能因中断切换而不准确，导致内存块被释放到错误的空闲列表。

**原因**：未关闭中断时，获取 CPU 编号后可能因中断处理切换到其他 CPU，导致内存操作与 CPU 编号不匹配。

**解决方法**：在调用cpuid()前后使用push\_off()和pop\_off()关闭中断，确保 CPU 编号获取和内存操作在同一 CPU 上完成。

**问题 2：窃取机制导致的锁竞争**

**现象**：当多个 CPU 同时需要窃取内存时，可能对同一目标 CPU 的锁产生新的竞争。

**原因**：窃取机制遍历其他 CPU 的空闲列表时，若多个 CPU 同时尝试获取同一 CPU 的锁，会导致新的锁冲突。

**解决方法**：优化窃取顺序（如按固定顺序遍历 CPU），减少并发窃取同一 CPU 的概率；同时保持窃取逻辑简洁，缩短持有其他 CPU 锁的时间，降低冲突窗口。

### 4）实验心得

通过本次实验，我深入理解了多核心环境下内存分配器的设计挑战，掌握了通过锁粒度优化和局部性利用提升并发性能的方法，也认识到并发编程中细节控制对系统正确性的重要影响。认识了并发编程的细节处理：多核心环境下的并发控制需要关注细节，如 CPU 编号的安全获取、锁的正确释放顺序、临界区的最小化等。任何疏忽都可能导致难以调试的竞争条件或内存错误，而push\_off()/pop\_off()等中断控制机制是保证并发正确性的重要工具。

## **Buffer cache (**[**hard**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解 xv6 缓冲区缓存（bcache）中全局锁导致的锁竞争问题，分析多进程并发访问磁盘块时的性能瓶颈。
2. 设计并实现基于哈希桶（hash bucket）的细粒度锁机制，替代原始的全局锁，减少锁竞争。
3. 优化缓冲区缓存的查找、分配和释放逻辑，确保并发环境下不同磁盘块的操作可并行执行。
4. 验证优化后的缓冲区缓存在减少锁竞争方面的效果，确保bcachetest测试通过，且usertests -q无错误。
5. 掌握共享资源并发访问的优化策略，理解锁粒度与系统性能的平衡关系。

### 2）实验步骤

**1. 原始缓冲区缓存的问题分析**

xv6 原始缓冲区缓存通过全局锁（bcache.lock）保护整个缓存列表，所有对缓存块的查找、分配、释放操作都需要获取该全局锁。在多进程环境下（如bcachetest创建多个进程密集读写文件），大量进程会频繁竞争该全局锁，导致严重的锁等待（体现为bcache锁的#test-and-set计数较高），显著降低文件系统性能。

**2. 优化方案设计：哈希桶与桶级锁**

为解决全局锁竞争问题，采用 “哈希桶 + 桶级锁” 的设计思路，核心方案如下：

* 将缓存块按磁盘块号（blockno）通过哈希函数映射到多个独立的哈希桶（bucket）。
* 每个哈希桶维护自己的缓存块链表和独立的锁，不同桶的操作可并行执行。
* 查找缓存块时，仅需锁定目标哈希桶，而非整个缓存，大幅减少锁竞争范围。
* 简化缓存替换策略（移除 LRU 机制），当缓存未命中时，直接重用未被引用（refcnt=0）的缓存块，降低并发控制复杂度。

**3. 核心实现步骤**

**（1）数据结构设计**

引入哈希桶结构，替代原始的全局单链表，每个桶独立管理缓存块：

// 哈希桶结构：每个桶包含独立的锁和缓存块链表

struct bucket{

struct spinlock lock; // 桶级锁，名称以"bcache"开头

struct buf head; // 桶内缓存块链表头

};

// 缓冲区缓存整体结构

struct {

struct buf buf[NBUF]; // 所有缓存块数组

struct bucket bucket[NBUCKET]; // 哈希桶数组（NBUCKET=19，质数减少冲突）

} bcache;

**（2）哈希函数设计**

使用简单哈希函数将磁盘块号映射到哈希桶，减少不同块号的哈希冲突：

// 哈希函数：将blockno映射到[0, NBUCKET-1]

static uint hash\_v(uint key) {

return key % NBUCKET; // NBUCKET=19（质数），降低冲突概率

}

**（3）初始化优化**

初始化所有哈希桶的锁和链表，并初始化缓存块的睡眠锁：

运行

void binit(void) {

// 初始化所有缓存块的睡眠锁

for (int i = 0; i < NBUF; ++i) {

initsleeplock(&bcache.buf[i].lock, "buffer");

}

// 初始化所有哈希桶（锁和链表）

for (int i = 0; i < NBUCKET; ++i) {

initlock(&bcache.bucket[i].lock, "bcache.bucket"); // 锁名称符合要求

bcache.bucket[i].head.prev = &bcache.bucket[i].head;

bcache.bucket[i].head.next = &bcache.bucket[i].head;

}

}

**（4）缓存块查找与分配（bget函数优化）**

修改bget函数，通过哈希桶定位缓存块，仅锁定目标桶，减少锁竞争：

//bio.c

static struct buf\* bget(uint dev, uint blockno) {

uint v = hash\_v(blockno); // 计算哈希值，定位目标桶

struct bucket\* bucket = &bcache.bucket[v];

// 1. 尝试在目标桶中查找缓存块

acquire(&bucket->lock); // 仅锁定目标桶

for (struct buf \*buf = bucket->head.next; buf != &bucket->head; buf = buf->next) {

if (buf->dev == dev && buf->blockno == blockno) {

buf->refcnt++; // 增加引用计数

release(&bucket->lock);

acquiresleep(&buf->lock); // 获取缓存块的睡眠锁

return buf;

}

}

// 2. 未命中，分配新缓存块（重用未使用的块）

for (int i = 0; i < NBUF; ++i) {

// 原子操作检查并标记缓存块为"使用中"，避免并发分配冲突

if (!bcache.buf[i].used && !\_\_atomic\_test\_and\_set(&bcache.buf[i].used, \_\_ATOMIC\_ACQUIRE)) {

struct buf \*buf = &bcache.buf[i];

// 初始化缓存块信息

buf->dev = dev;

buf->blockno = blockno;

buf->valid = 0;

buf->refcnt = 1;

// 将缓存块插入目标桶的链表

buf->next = bucket->head.next;

buf->prev = &bucket->head;

bucket->head.next->prev = buf;

bucket->head.next = buf;

release(&bucket->lock);

acquiresleep(&buf->lock);

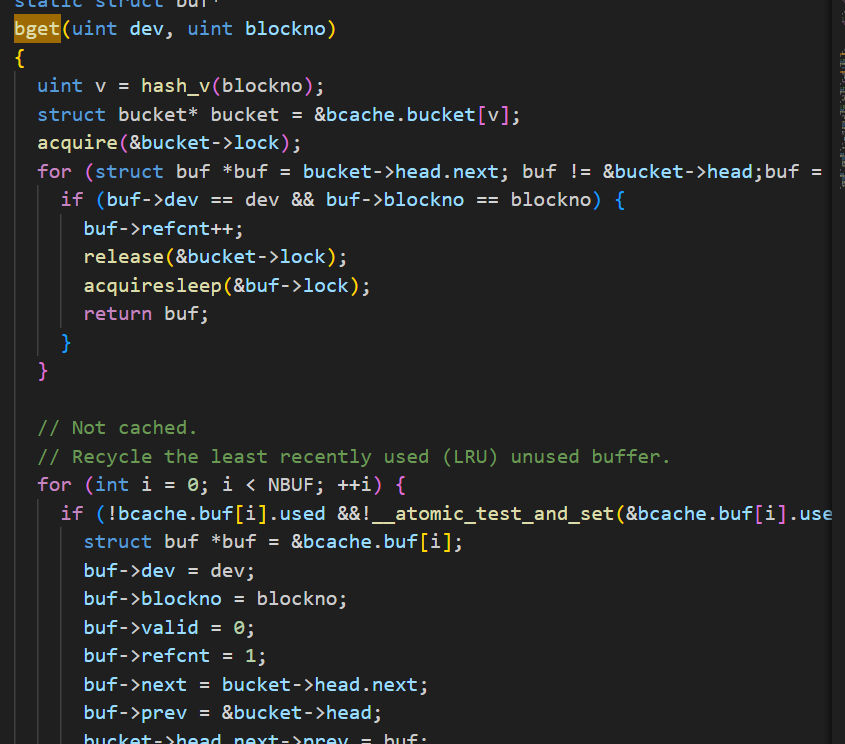
return buf;

}

}

panic("bget: no buffers");

}



**（5）缓存块释放（brelse函数优化）**

修改brelse函数，仅操作目标哈希桶的锁，实现缓存块的安全释放：

void brelse(struct buf \*b) {

if (!holdingsleep(&b->lock))

panic("brelse");

releasesleep(&b->lock); // 释放缓存块的睡眠锁

uint v = hash\_v(b->blockno); // 定位所属哈希桶

struct bucket\* bucket = &bcache.bucket[v];

acquire(&bucket->lock); // 仅锁定目标桶

b->refcnt--;

if (b->refcnt == 0) { // 若引用计数为0，标记为未使用，可重用

// 从桶链表中移除（简化LRU，直接回收）

b->next->prev = b->prev;

b->prev->next = b->next;

\_\_atomic\_clear(&b->used, \_\_ATOMIC\_RELEASE); // 原子标记为未使用

}

release(&bucket->lock); // 释放桶锁

}

**（6）缓存块固定与解锁（bpin/bunpin优化）**

修改缓存块固定和解锁函数，仅操作目标桶的锁，避免全局锁竞争：

void bpin(struct buf \*b) {

uint v = hash\_v(b->blockno);

struct bucket\* bucket = &bcache.bucket[v];

acquire(&bucket->lock);

b->refcnt++; // 增加引用计数，防止被回收

release(&bucket->lock);

}

void bunpin(struct buf \*b) {

uint v = hash\_v(b->blockno);

struct bucket\* bucket = &bcache.bucket[v];

acquire(&bucket->lock);

b->refcnt--; // 减少引用计数

release(&bucket->lock);

}

**4. 关键机制说明**

* **哈希桶隔离**：通过哈希函数将不同块号的缓存块分散到 19 个桶（质数减少冲突），每个桶独立加锁，不同桶的操作完全并行。
* **细粒度锁控制**：所有缓存操作仅锁定目标桶，而非全局缓存，大幅降低锁竞争概率。
* **原子状态标记**：使用\_\_atomic\_test\_and\_set和\_\_atomic\_clear原子操作标记缓存块的used状态，避免并发分配冲突。
* **简化 LRU 策略**：移除原始 LRU 链表，当缓存块引用计数为 0 时直接回收，减少并发环境下的链表维护开销。
* **锁命名规范**：所有桶锁名称以 “bcache.bucket” 开头，符合实验要求，便于锁竞争统计。

### 3）实验中遇到的问题及解决方法

**问题 1：哈希冲突导致的局部锁竞争**

**现象**：部分哈希桶的锁竞争仍然较高，bcachetest中某些桶的#test-and-set计数不为零。

**原因**：哈希函数设计不合理，导致多个热门块号映射到同一桶，引发局部竞争。

**解决方法**：选择质数（19）作为哈希桶数量（NBUCKET=19），通过模运算（key % NBUCKET）分散块号，减少冲突概率。

**问题 2：并发分配缓存块导致重复使用**

**现象**：多进程并发申请缓存块时，出现同一缓存块被多个进程同时分配的错误。

**原因**：检查缓存块 “未使用” 状态和标记为 “使用中” 的操作非原子，存在竞态条件。

**解决方法**：使用\_\_atomic\_test\_and\_set原子操作，确保 “检查 - 标记” 过程的原子性，避免并发分配冲突。

### 4）实验心得

通过本次实验，我深入理解了缓冲区缓存的工作原理，掌握了基于哈希的细粒度锁设计方法，认识到哈希桶的数量和哈希函数直接影响冲突率。选择质数（19）作为桶数量可有效减少冲突，而简单的模运算哈希函数在保证分散性的同时降低了计算开销。合理的哈希策略能平衡各桶的负载，避免局部竞争成为新瓶颈。

# **Lab: file system**

## **Large files (**[**moderate**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解 xv6 文件系统中文件大小的限制原理，分析原始设计中 inode 结构对文件大小的制约。
2. 修改 xv6 文件系统，通过引入双重间接块（doubly-indirect block）扩展文件最大容量。
3. 调整 inode 结构中的块地址存储方式，在不改变 inode 大小的前提下，将直接块数量从 12 个减少到 11 个，以容纳双重间接块地址。
4. 实现对双重间接块的地址映射（bmap）和释放（itrunc）功能，确保文件系统能正确管理大型文件的块分配与回收。
5. 验证修改后的文件系统能支持最大 65803 个块的文件，并通过所有用户态测试（usertests -q）。

### 2）实验步骤

**1. 原始文件系统的限制分析**

xv6 原始文件系统中，每个 inode 包含 12 个直接块地址和 1 个单重间接块地址，导致文件最大容量为：

* 直接块：12 个
* 单重间接块：256 个（每个间接块可存储 256 个地址，BSIZE=1024，每个地址 4 字节）
* 总计：12 + 256 = 268 个块

为突破此限制，本实验引入双重间接块，新的容量计算为：

* 直接块：11 个（减少 1 个以容纳双重间接块）
* 单重间接块：256 个
* 双重间接块：256 × 256 = 65536 个
* 总计：11 + 256 + 65536 = 65803 个块

**2. 核心实现步骤**

**（1）修改宏定义与数据结构**

调整 inode 中块地址的存储布局，增加双重间接块支持：

**（2）修改块地址映射函数（bmap）**

扩展bmap函数，使其能处理双重间接块的地址映射，实现逻辑块号到物理块号的转换：

// kernel/fs.c

static uint bmap(struct inode \*ip, uint bn) {

struct buf \*bp;

uint addr, \*a;

// 1. 处理直接块（0 ~ NDIRECT-1）

if (bn < NDIRECT) {

if ((addr = ip->addrs[bn]) == 0) {

addr = balloc(ip->dev); // 分配新块

if (addr == 0)

return 0;

ip->addrs[bn] = addr;

}

return addr;

}

// 2. 处理单重间接块（NDIRECT ~ NDIRECT+NINDIRECT-1）

bn -= NDIRECT;

if (bn < NINDIRECT) {

// 查找单重间接块，若不存在则分配

if ((addr = ip->addrs[NDIRECT]) == 0) {

addr = balloc(ip->dev);

if (addr == 0)

return 0;

ip->addrs[NDIRECT] = addr;

}

bp = bread(ip->dev, addr); // 读取单重间接块

a = (uint\*)bp->data;

if ((addr = a[bn]) == 0) {

addr = balloc(ip->dev); // 分配数据块

if (addr == 0) {

brelse(bp);

return 0;

}

a[bn] = addr;

log\_write(bp); // 写回间接块

}

brelse(bp);

return addr;

}

// 3. 处理双重间接块（剩余块号）

bn -= NINDIRECT;

if (bn < NDBL\_INDIRECT) {

// 查找双重间接块，若不存在则分配

if ((addr = ip->addrs[NDIRECT + 1]) == 0) {

addr = balloc(ip->dev);

if (addr == 0)

return 0;

ip->addrs[NDIRECT + 1] = addr;

}

bp = bread(ip->dev, addr); // 读取双重间接块

a = (uint\*)bp->data;

// 计算单重间接块索引和数据块索引

uint idx = bn / NINDIRECT; // 单重间接块在双重间接块中的索引

uint off = bn % NINDIRECT; // 数据块在单重间接块中的索引

// 查找单重间接块，若不存在则分配

if ((addr = a[idx]) == 0) {

addr = balloc(ip->dev);

if (addr == 0) {

brelse(bp);

return 0;

}

a[idx] = addr;

log\_write(bp); // 写回双重间接块

}

brelse(bp);

// 读取单重间接块，分配数据块

bp = bread(ip->dev, addr);

a = (uint\*)bp->data;

if ((addr = a[off]) == 0) {

addr = balloc(ip->dev);

if (addr == 0) {

brelse(bp);

return 0;

}

a[off] = addr;

log\_write(bp); // 写回单重间接块

}

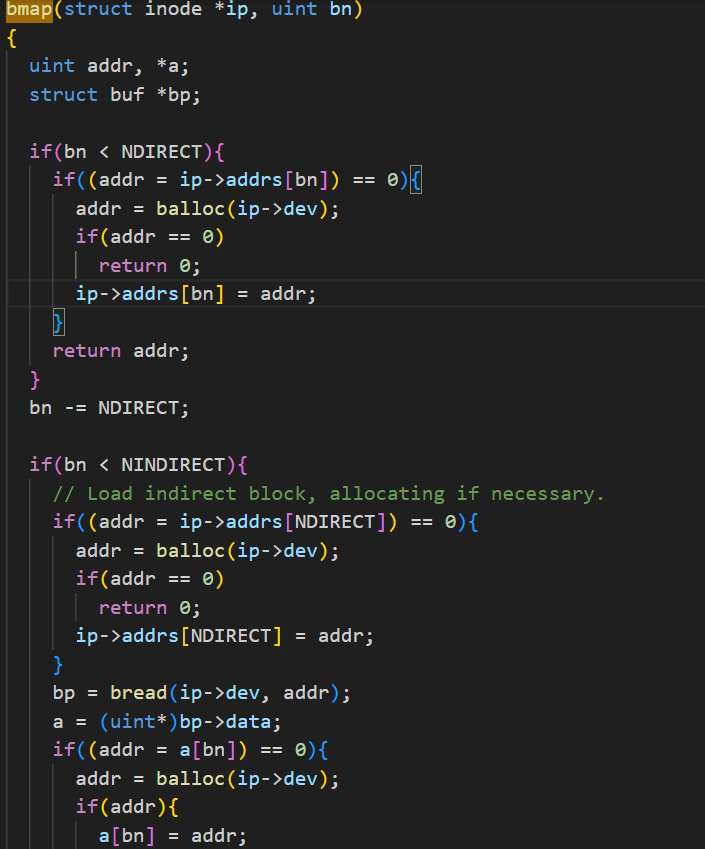
brelse(bp);

return addr;

}

panic("bmap: out of range"); // 超出最大块号

}



**（3）修改块释放函数（itrunc）**

扩展itrunc函数，使其能释放双重间接块所引用的所有块：

// kernel/fs.c

void itrunc(struct inode \*ip) {

int i, j;

struct buf \*bp, \*bp2;

uint \*a, \*b;

// 释放直接块

for (i = 0; i < NDIRECT; i++) {

if (ip->addrs[i]) {

bfree(ip->dev, ip->addrs[i]);

ip->addrs[i] = 0;

}

}

// 释放单重间接块

if (ip->addrs[NDIRECT]) {

bp = bread(ip->dev, ip->addrs[NDIRECT]);

a = (uint\*)bp->data;

for (i = 0; i < NINDIRECT; i++) {

if (a[i])

bfree(ip->dev, a[i]);

}

brelse(bp);

bfree(ip->dev, ip->addrs[NDIRECT]);

ip->addrs[NDIRECT] = 0;

}

// 释放双重间接块

if (ip->addrs[NDIRECT + 1]) {

bp = bread(ip->dev, ip->addrs[NDIRECT + 1]); // 读取双重间接块

a = (uint\*)bp->data;

for (i = 0; i < NINDIRECT; i++) {

if (a[i]) {

bp2 = bread(ip->dev, a[i]); // 读取单重间接块

b = (uint\*)bp2->data;

for (j = 0; j < NINDIRECT; j++) { // 释放数据块

if (b[j])

bfree(ip->dev, b[j]);

}

brelse(bp2);

bfree(ip->dev, a[i]); // 释放单重间接块

a[i] = 0;

}

}

brelse(bp);

bfree(ip->dev, ip->addrs[NDIRECT + 1]); // 释放双重间接块

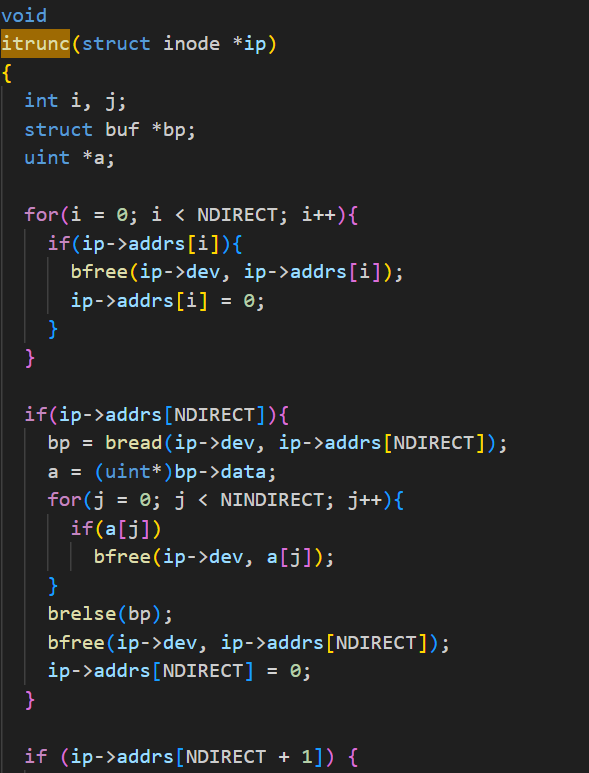
ip->addrs[NDIRECT + 1] = 0;

}

ip->size = 0;

iupdate(ip); // 更新inode到磁盘

}



**3. 关键机制说明**

* **地址映射层级**：通过三级地址映射（直接块→单重间接块→双重间接块）扩展文件容量，每级间接块可索引 256 个下一级块地址。
* **按需分配**：间接块（单重 / 双重）仅在需要时分配，避免浪费磁盘空间。例如，小文件仍可仅使用直接块，无需分配间接块。
* **递归释放**：itrunc函数通过递归方式释放所有层级的块，从数据块到单重间接块，再到双重间接块，确保无内存泄漏。
* **兼容性**：修改后的数据结构保持 inode 大小不变，确保与文件系统其他部分兼容。

### 3）实验中遇到的问题及解决方法

**问题 1：块号范围计算错误**

**现象**：当文件大小超过单重间接块范围时，读写操作出现错误或恐慌。

**原因**：逻辑块号（bn）在直接块、单重间接块、双重间接块之间的划分错误，导致索引越界。

**解决方法**：重新梳理块号范围：

**问题 2：间接块未正确写回磁盘**

**现象**：分配新的间接块后，系统崩溃或重启后，间接块地址丢失，导致文件数据损坏。

**原因**：分配间接块后，未通过log\_write将间接块的修改写入日志，导致数据未持久化。

**解决方法**：在分配新的间接块（单重或双重）并更新其地址数组后，立即调用log\_write(bp)将修改写回磁盘，确保数据持久化。

### 4）实验心得

通过本次实验，不仅掌握了文件系统中多级寻址的实现方法，更深刻理解了操作系统在资源管理中面临的权衡与设计决策，为后续更复杂的存储系统设计奠定了基础。

## **Symbolic links (**[moderate](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解符号链接（软链接）与硬链接的区别，掌握符号链接的工作原理。
2. 为 xv6 操作系统添加symlink系统调用，支持创建符号链接。
3. 修改open系统调用，使其能够正确跟随符号链接，直至找到目标文件。
4. 实现循环链接检测机制，防止符号链接形成环时导致的无限递归。
5. 添加O\_NOFOLLOW标志，允许进程打开符号链接本身而不跟随其指向。
6. 确保其他系统调用（如link、unlink）对符号链接本身操作，而非其指向的目标文件。
7. 通过symlinktest和usertests验证实现的正确性。

### 2）实验步骤

**1. 符号链接与硬链接的区别**

硬链接是通过 inode 引用文件，只能指向同一文件系统中的文件，删除原始文件后硬链接仍然有效。而符号链接是通过路径名引用文件，可跨文件系统，本质上是一个包含目标路径的特殊文件，删除原始文件后符号链接会失效。

**2. 核心实现步骤**

**（1）添加必要的定义**

* **文件类型**：在stat.h中添加T\_SYMLINK类型，表示符号链接
* **打开标志**：在fcntl.h中添加O\_NOFOLLOW标志，用于open系统调用
* **系统调用号**：在syscall.h中添加SYS\_symlink系统调用号：

**（2）实现symlink系统调用**

symlink系统调用创建一个新的符号链接文件，存储目标路径。实现如下：

//sysfile.c

uint64 sys\_symlink(void) {

char target[MAXPATH], path[MAXPATH];

if (argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)

return -1;

begin\_op(); // 开始文件系统操作

// 创建符号链接文件（类型为T\_SYMLINK）

struct inode\* inode = create(path, T\_SYMLINK, 0, 0);

if (inode == 0) {

end\_op();

return -1;

}

// 写入目标路径长度和路径内容到符号链接文件

int len = strlen(target);

if (writei(inode, 0, (uint64)&len, 0, sizeof(len)) < 0) {

iunlockput(inode);

end\_op();

return -1;

}

if (writei(inode, 0, (uint64)target, sizeof(len), len + 1) < 0) {

iunlockput(inode);

end\_op();

return -1;

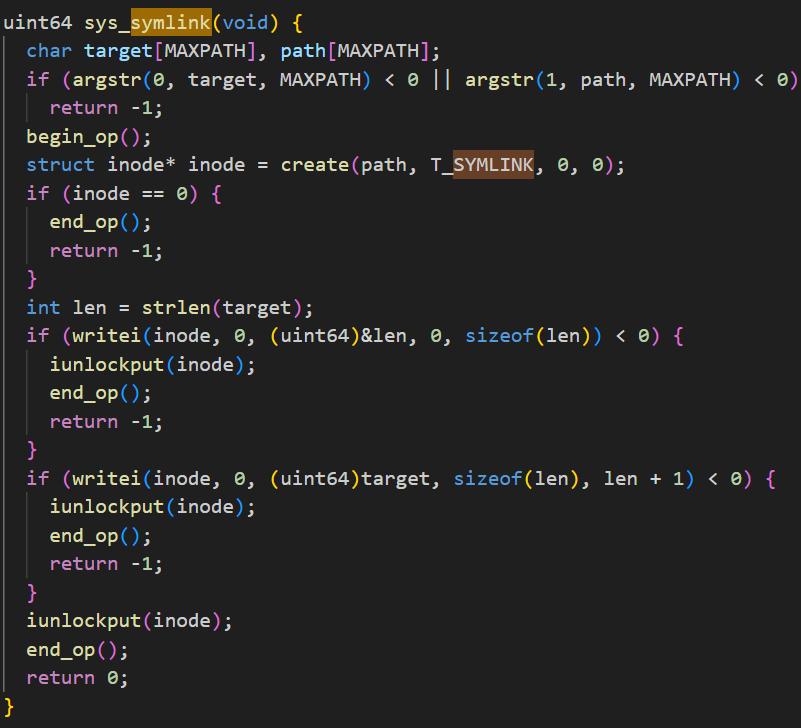
}

iunlockput(inode);

end\_op(); // 结束文件系统操作

return 0;

}



**关键逻辑**：

* 调用create创建类型为T\_SYMLINK的特殊文件。
* 将目标路径的长度和内容写入该文件的数据块中。

**（3）修改open系统调用以处理符号链接**

修改sys\_open函数，使其能跟随符号链接（除非指定O\_NOFOLLOW）：

if (ip->type == T\_SYMLINK && !(omode & O\_NOFOLLOW)) {

int cnt = 0;

// 循环跟随符号链接，直到找到非链接文件

while (ip->type == T\_SYMLINK) {

if (++cnt > THRESHOLD) {

iunlockput(ip);

end\_op();

return -1;

}

// 读取符号链接中存储的目标路径

int len = 0;

if (readi(ip, 0, (uint64)&len, 0, sizeof(len)) != sizeof(len)) {

iunlockput(ip);

end\_op();

return -1;

}

char path[MAXPATH];

if (readi(ip, 0, (uint64)path, sizeof(len), len + 1) != len + 1) {

iunlockput(ip);

end\_op();

return -1;

}

// 释放当前inode，解析新路径

iunlockput(ip);

if ((ip = namei(path)) == 0) { // 解析目标路径

end\_op();

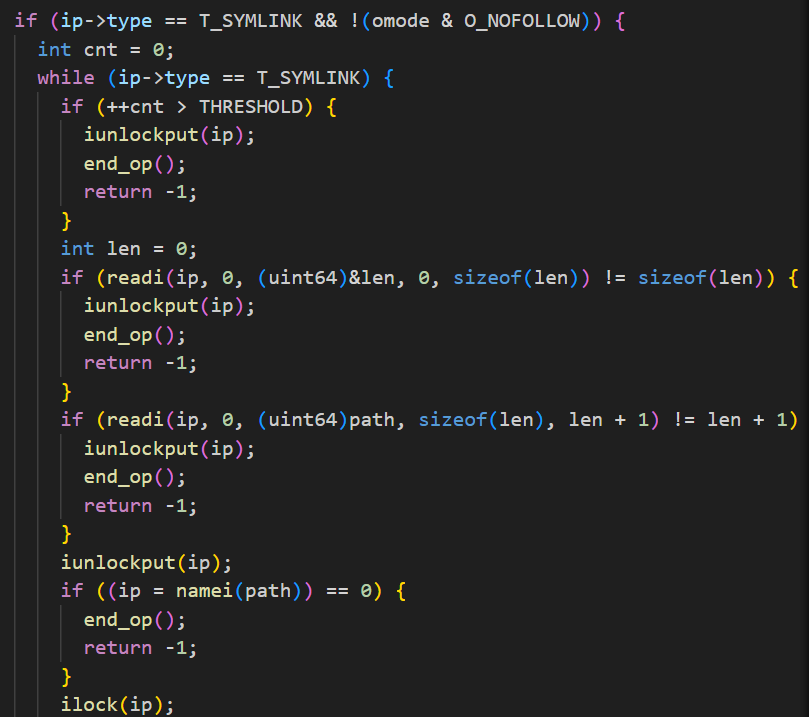
return -1;

}

ilock(ip);

}

}



**关键逻辑**：

* 若打开的是符号链接且未指定O\_NOFOLLOW，则跟随链接。
* 读取符号链接中存储的目标路径，重新解析路径。

**3. 关键机制说明**

* **符号链接存储**：符号链接本身是一个特殊文件（类型T\_SYMLINK），其数据块中存储目标路径的长度和内容。
* **路径跟随**：open系统调用通过递归解析符号链接的目标路径，直至找到非链接文件。
* **循环检测**：通过限制最大跟随深度（10 层），避免符号链接环导致的无限递归。
* **操作隔离**：link、unlink等系统调用直接操作符号链接本身，不跟随指向；只有open在未指定O\_NOFOLLOW时才跟随链接。

### 3）实验中遇到的问题及解决方法

**问题 1：目标路径读取不完整**

**现象**：跟随符号链接时，目标路径被截断或读取错误。

**原因**：写入 / 读取符号链接的路径时，未正确处理路径长度和终止符。

**解决方法**：先写入路径长度（len = strlen(target)），再写入路径内容（包括\0终止符），确保读取时能准确获取完整路径

**问题 2：循环链接导致无限递归**

**现象**：当符号链接形成环（如a->b且b->a），open会陷入无限递归，导致系统崩溃。

**原因**：未检测循环链接，递归跟随无终止条件。

**解决方法**：添加深度计数器，当跟随次数超过阈值（10）时返回错误：

### 4）实验心得

通过本次实验，不仅掌握了符号链接的实现方法，认识了符号链接是一种特殊文件，其内容为目标文件的路径名，而非数据或 inode 引用。这与硬链接有本质区别，硬链接直接指向 inode，而符号链接指向路径，因此具有跨文件系统的能力。更深入理解了 xv6 文件系统的路径解析机制和系统调用设计，为后续更复杂的文件系统扩展奠定了基础。

# **Lab: mmap**

## **mmap (**[**hard**](https://pdos.csail.mit.edu/6.828/2022/labs/guidance.html)**)**

### 1）实验目的

1. 理解内存映射（mmap）系统调用的工作原理，掌握文件与进程地址空间的映射机制。
2. 实现mmap系统调用，支持将文件映射到进程虚拟地址空间，支持MAP\_SHARED（共享映射）和MAP\_PRIVATE（私有映射）两种模式。
3. 实现munmap系统调用，支持解除内存映射并将MAP\_SHARED模式下的修改写回文件。
4. 采用懒加载（lazy allocation）策略，在页故障时动态分配物理内存并读取文件内容，提高大文件映射效率。
5. 维护虚拟内存区域（VMA）结构，管理进程的内存映射信息，确保进程生命周期（创建、fork、退出）中映射的正确性。
6. 验证实现的正确性，确保mmaptest和usertests -q所有测试通过。

### 2）实验步骤

**1. 核心概念与设计思路**

内存映射通过将文件内容直接映射到进程虚拟地址空间，允许进程以访问内存的方式操作文件。本实验采用懒加载策略：mmap仅创建映射关系（不立即分配内存或读取文件），首次访问映射区域时触发页故障，此时再分配物理内存、读取文件内容并建立映射。

关键数据结构为虚拟内存区域（VMA），记录映射的地址范围、权限、文件指针等信息。每个进程维护一个 VMA 数组，管理其所有内存映射。

**2. 核心实现步骤**

**（1）数据结构定义**

* **VMA 结构**：记录内存映射的关键信息，定义于proc.h：

struct vma {

int used; // 标记VMA是否被使用

void\* addr; // 映射起始地址

uint64 length; // 映射长度

int prot; // 权限（PROT\_READ/PROT\_WRITE等）

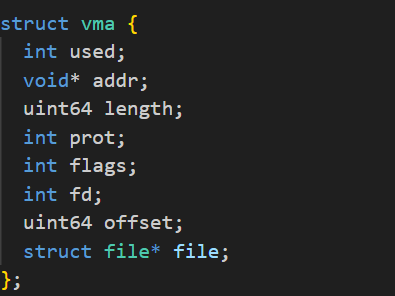
int flags; // 映射标志（MAP\_SHARED/MAP\_PRIVATE）

int fd; // 关联文件描述符

uint64 offset; // 文件偏移量（本实验固定为0）

struct file\* file; // 关联文件指针（引用计数管理）

};



* **进程结构扩展**：在struct proc中添加 VMA 数组和映射地址顶部指针：

**（2）mmap系统调用实现**

mmap负责在进程地址空间中分配虚拟地址范围，在sysfile.c创建 VMA 记录映射关系：

uint64 sys\_mmap(void) {

void \*addr; // 用户建议地址（本实验忽略，内核分配）

uint64 length; // 映射长度

int prot; // 权限（PROT\_READ/PROT\_WRITE）

int flags; // 映射标志（MAP\_SHARED/MAP\_PRIVATE）

int fd; // 文件描述符

struct file \*file; // 关联文件

uint64 offset; // 文件偏移量（本实验固定为0）

// 解析系统调用参数

argaddr(0, (void \*)&addr); // 忽略用户指定地址

argint64(1, &length);

argint(2, &prot);

argint(3, &flags);

argfd(4, &fd, &file);

argint64(5, &offset);

// 权限检查：读权限需文件可读，共享写需文件可写

if (file->readable == 0 && (prot & PROT\_READ))

return -1;

if (file->writable == 0 && (prot & PROT\_WRITE) && (flags & MAP\_SHARED))

return -1;

struct proc \*p = myproc();

// 查找空闲VMA槽位

for (int i = 0; i < NVMA; ++i) {

if (p->vmas[i].used) continue;

// 从高地址向下分配映射区域（避免与堆冲突）

addr = (void \*)PGROUNDDOWN(p->mmap\_top - length);

// 初始化VMA

p->vmas[i].used = 1;

p->vmas[i].addr = addr;

p->vmas[i].length = length;

p->vmas[i].prot = prot;

p->vmas[i].flags = flags;

p->vmas[i].file = file;

p->vmas[i].offset = offset;

// 更新映射顶部地址

p->mmap\_top = (uint64)addr;

// 增加文件引用计数（确保文件不被提前关闭）

filedup(file);

return (uint64)addr; // 返回分配的虚拟地址

}

return -1; // VMA数组已满

}



**关键逻辑**：

* 从进程高地址（mmap\_top）向下分配连续虚拟地址，避免与堆（从低地址向上增长）冲突。
* 通过filedup增加文件引用计数，确保映射期间文件结构不被释放。
* 仅记录映射关系，不立即分配物理内存或读取文件（懒加载）。

**（3）页故障处理（懒加载实现）**

当进程访问mmap映射的虚拟地址但未建立物理映射时，触发页故障。在trap.c中处理页故障，动态分配内存并读取文件：

else if (r\_scause() == 13 || r\_scause() == 15) {

uint64 addr = r\_stval(); // 故障地址

if (addr >= MAXVA) { // 地址越界

setkilled(p);

goto out;

}

// 查找故障地址所属的VMA

struct vma\* vma = 0;

for (int i = 0; i < NVMA; ++i) {

if (p->vmas[i].used == 0) continue;

uint64 begin = (uint64)p->vmas[i].addr;

uint64 end = begin + p->vmas[i].length;

if (addr >= begin && addr < end) {

vma = &p->vmas[i];

break;

}

}

if (vma == 0) { // 地址不在任何VMA中

setkilled(p);

goto out;

}

// 权限检查

if (r\_scause() == 13 && vma->file->readable == 0) // 读故障但文件不可读

setkilled(p);

if (r\_scause() == 15 && vma->file->writable == 0) // 写故障但文件不可写

setkilled(p);

// 分配物理内存并清零

void\* pa = kalloc();

if (pa == 0) {

setkilled(p);

goto out;

}

memset(pa, 0, PGSIZE);

// 计算页对齐的故障地址

addr = PGROUNDDOWN(addr);

// 设置页表权限（用户可访问+读写执行权限）

int perm = PTE\_U;

if (vma->prot & PROT\_READ) perm |= PTE\_R;

if (vma->prot & PROT\_WRITE) perm |= PTE\_W;

if (vma->prot & PROT\_EXEC) perm |= PTE\_X;

// 建立虚拟地址到物理地址的映射

if (mappages(p->pagetable, addr, PGSIZE, (uint64)pa, perm) != 0) {

kfree(pa);

setkilled(p);

goto out;

}

// 从文件读取数据到物理页

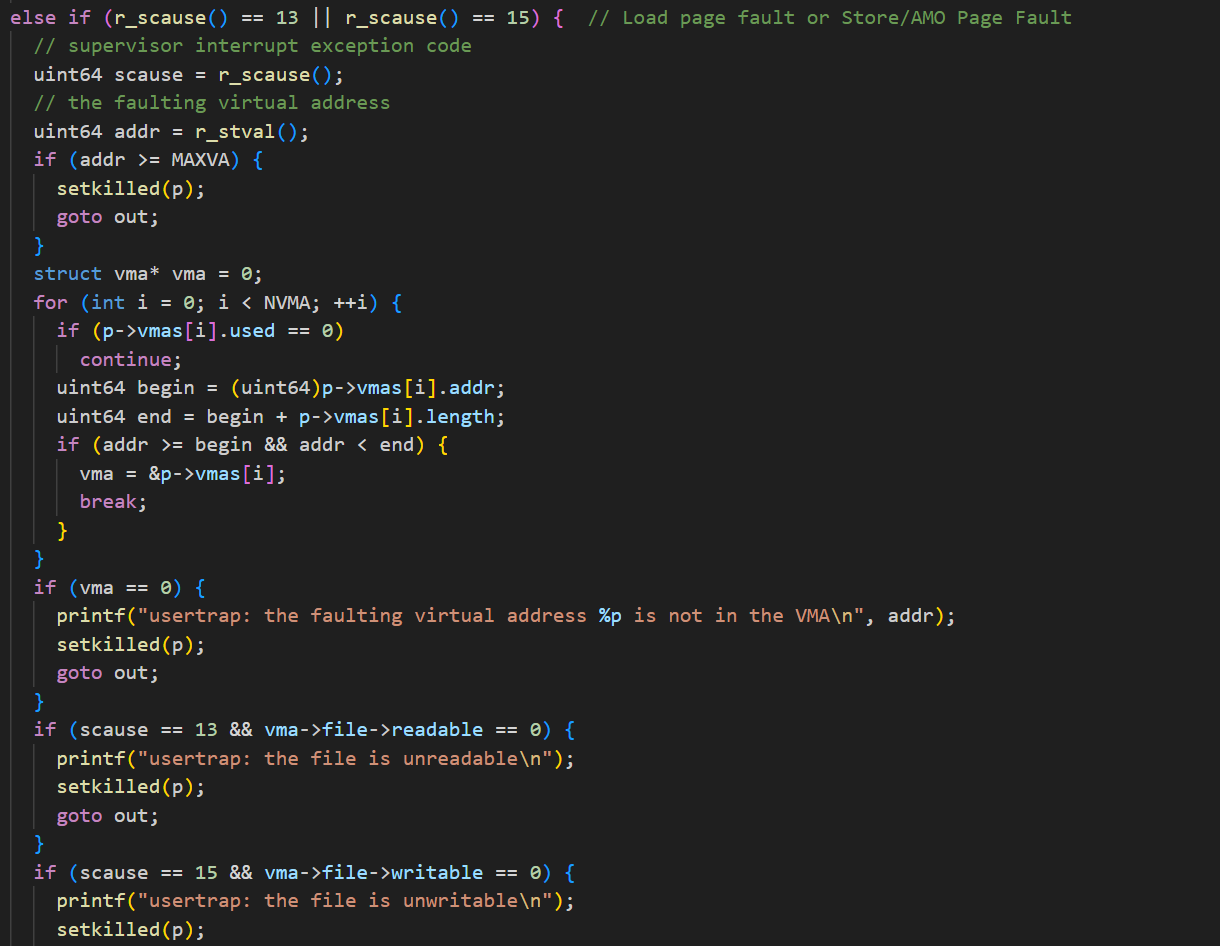
ilock(vma->file->ip); // 锁定inode

// 读取文件偏移 = 故障地址 - 映射起始地址 + 文件初始偏移

readi(vma->file->ip, 1, addr, addr - (uint64)vma->addr + vma->offset, PGSIZE);

iunlock(vma->file->ip); // 解锁inode

}

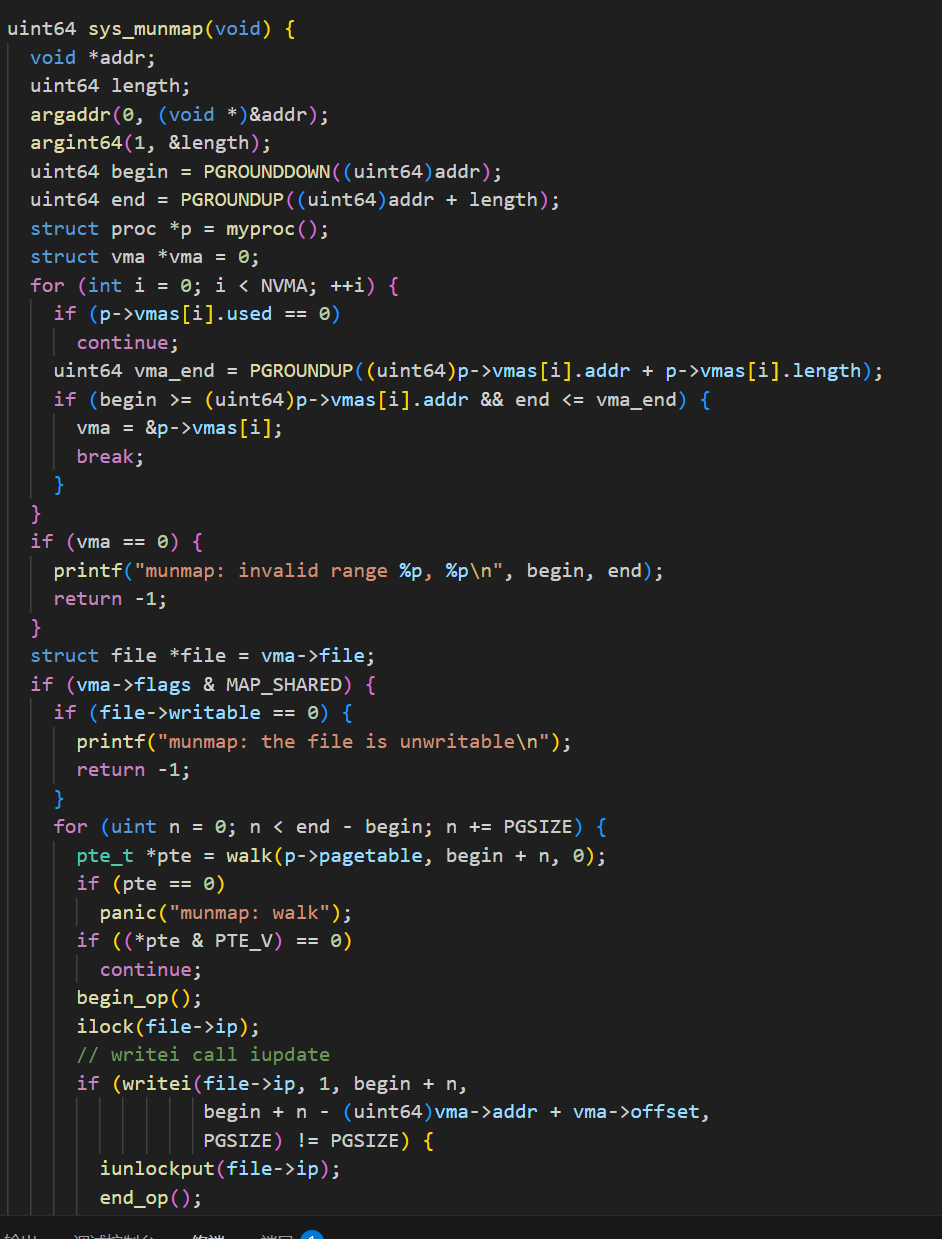


**关键逻辑**：

* 故障地址合法性检查：确保地址在进程 VMA 范围内。
* 动态分配物理内存，根据 VMA 权限设置页表项（PTE）。
* 从文件读取对应偏移的数据到物理页，完成映射建立。

**（4）munmap系统调用实现**

sysfile.c



uint64 sys\_munmap(void) {

void \*addr; // 解除映射的起始地址

uint64 length; // 解除映射的长度

argaddr(0, (void \*)&addr);

argint64(1, &length);

uint64 begin = PGROUNDDOWN((uint64)addr);

uint64 end = PGROUNDUP((uint64)addr + length);

// 查找地址所属的VMA

struct proc \*p = myproc();

struct vma \*vma = 0;

for (int i = 0; i < NVMA; ++i) {

if (p->vmas[i].used == 0) continue;

uint64 vma\_end = PGROUNDUP((uint64)p->vmas[i].addr + p->vmas[i].length);

if (begin >= (uint64)p->vmas[i].addr && end <= vma\_end) {

vma = &p->vmas[i];

break;

}

}

if (vma == 0) return -1; // 无效地址范围

// 若为共享映射，将修改写回文件

if (vma->flags & MAP\_SHARED) {

if (vma->file->writable == 0) return -1; // 文件不可写

begin\_op(); // 开始文件操作

ilock(vma->file->ip); // 锁定inode

// 遍历解除映射的页面，写回修改

for (uint n = 0; n < end - begin; n += PGSIZE) {

pte\_t \*pte = walk(p->pagetable, begin + n, 0);

if (pte && (\*pte & PTE\_V)) { // 仅处理已映射的页

// 写回文件：偏移 = 虚拟地址 - 映射起始地址 + 文件初始偏移

writei(vma->file->ip, 1, begin + n,

begin + n - (uint64)vma->addr + vma->offset, PGSIZE);

}

}

iunlockput(vma->file->ip); // 解锁并减少inode引用

end\_op(); // 结束文件操作

}

// 解除虚拟地址映射

for (uint64 i = begin; i < end; i += PGSIZE) {

pte\_t \*pte = walk(p->pagetable, i, 0);

if (pte && (\*pte & PTE\_V)) {

uvmunmap(p->pagetable, i, 1, 1); // 解除映射并释放物理内存

}

}

// 更新VMA（支持部分解除映射）

uint64 vma\_end = PGROUNDUP((uint64)vma->addr + vma->length);

if (begin == (uint64)vma->addr && end == vma\_end) {

// 完全解除映射：标记VMA为未使用，减少文件引用

vma->used = 0;

filered(vma->file); // 减少文件引用计数

} else if (begin == (uint64)vma->addr) {

// 解除头部：调整VMA起始地址和长度

vma->addr = (void \*)end;

vma->length -= end - begin;

vma->offset += end - begin;

} else if (end == vma\_end) {

// 解除尾部：调整VMA长度

vma->length = begin - (uint64)vma->addr;

} else {

panic("munmap: 不支持中间区域解除"); // 本实验不支持中间区域解除

}

return 0;

}

**关键逻辑**：

* 写回MAP\_SHARED修改：遍历解除映射的页面，通过writei将数据写回文件对应偏移。
* 解除映射：通过uvmunmap清除页表项并释放物理内存。
* VMA 更新：支持完全解除或部分解除（头部 / 尾部），维护 VMA 的正确性。

**（5）进程生命周期中的 VMA 管理**

* **Fork 时 VMA 复制**：在fork函数中复制父进程的 VMA 数组，并增加文件引用计数
* **进程退出时 VMA 清理**：在exit函数中解除所有映射，释放资源

**3. 关键机制总结**

* **懒加载**：mmap仅创建映射关系，首次访问时通过页故障分配内存和读取文件，提高大文件映射效率。
* **VMA 管理**：通过 VMA 数组跟踪进程所有映射，记录地址范围、权限、文件等关键信息，是实现映射的核心数据结构。
* **权限控制**：严格检查文件读写权限与映射权限的匹配，防止非法访问。
* **共享映射写回**：munmap时将MAP\_SHARED模式下的修改通过writei写回文件，确保数据持久化。
* **进程一致性**：在fork和exit中正确处理 VMA 复制和清理，维护进程生命周期中映射的正确性。

### 3）实验中遇到的问题及解决方法

**问题 1：VMA 地址分配冲突**

**现象**：多次mmap后，新分配的映射地址与已有映射重叠，导致访问错误。

**原因**：mmap从mmap\_top向下分配地址时，未正确计算页对齐的地址范围。

**解决方法**：通过PGROUNDDOWN确保分配的起始地址是页对齐的，且新映射长度不会超出剩余地址空间

**问题 2：页故障地址判断错误**

**现象**：访问映射区域时触发页故障，但无法正确匹配到对应的 VMA，导致进程被杀死。

**原因**：VMA 地址范围检查逻辑错误，未考虑映射长度的边界。

**解决方法**：修正 VMA 地址范围判断，确保故障地址在[vma->addr, vma->addr + vma->length)区间内

### 4）实验心得

通过本次实验，我认识了mmap 的懒加载策略显著提高了大文件映射的效率。通过推迟物理内存分配和文件读取到首次访问，避免了不必要的资源消耗，使大文件甚至超物理内存大小的文件映射成为可能，深入理解了内存映射的底层实现机制，掌握了虚拟内存管理、页故障处理、共享资源管理等核心技术，为理解更复杂的内存管理系统奠定了基础。

# **附录**

## git仓库地址

https://github.com/dingzhenyao/xv6labs2022