
xv6 实验报告

作者:2252709 杨烜赫

同济大学软件学院操作系统小学期项目

如果有任何问题或评论,
可以通过邮件 2252709@TONGJI.EDU.CN 联系我

作者自序

在现代计算机科学教育中，操作系统课程扮演着至关重要的角色。作为这一领域的核心内容之一，MIT的xv6课程实验提供了一个深入理解操作系统原理的宝贵机会。xv6是一个基于UNIX第六版的简化操作系统，旨在帮助我们了解操作系统的基本设计和实现原理。

本实验报告详细记录了在xv6环境中所进行的实验过程和结果。这些实验涵盖了进程管理、内存管理、文件系统和设备驱动等关键主题。通过这些实验，我得以从实践角度深刻理解操作系统的复杂机制和设计选择。

实验的核心目标不仅在于增强我们对理论知识的掌握，更在于培养解决实际问题的能力。通过分析、设计和实现系统功能，我学会了如何应对操作系统开发中的挑战。这一经历为我未来在计算机科学领域的发展打下了坚实的基础。

本报告的内容将依次介绍各实验的背景信息、实验过程、遇到的问题及其解决方案，以及最终的实验成果。希望通过这些详细的记录和分析，能够为其他同学和研究人员提供有价值的参考，并激发更多人对操作系统领域的兴趣和探索。

杨烜赫
2024年夏

目录

第一章 xv6实验概述与环境配置	1
1.1 简介	1
1.1.1 xv6 的特点和设计目标	1
1.1.2 xv6 的主要组成部分	1
1.2 环境配置	1
1.2.1 在 Windows 上安装	2
1.2.2 测试安装	2
第二章 Lab Utilities: 实用工具实验	3
2.1 启动 xv6	3
2.2 实现 sleep 工具	4
2.3 实现 pingpong 工具	6
2.4 实现 primes 工具	7
2.5 实现 find 工具	10
2.6 实现 xargs 工具	12
2.7 小结	15
2.7.1 进程间通信问题	16
2.7.2 递归实现与管道通信问题	16
2.7.3 目录遍历与文件查找问题	16
第三章 System Calls 系统调用实验	17
3.1 实现 System call tracing 工具	17
3.2 实现 Sysinfo 系统调用	19
3.3 小结	22
3.3.1 实现 trace 系统调用	22
3.3.2 实现 sysinfo 系统调用	22
3.3.3 遇到的问题及解决方式	22
第四章 Pages Tables 实验室页表实验	24
4.1 加速系统调用	24
4.2 打印页表	25
4.3 追踪被访问的页	28
4.4 小结	29
4.4.1 加速系统调用的实现	30
4.4.2 打印页表内容	30
4.4.3 追踪被访问的页	30

4.4.4 遇到的问题及解决方式	30
第五章 Lab Traps 中断实验	32
5.1 RISC-V汇编	32
5.2 实现函数调用栈回溯 (Backtrace)	34
5.3 实现定时器系统调用	36
5.4 小结	39
5.4.1 RISC-V 汇编的理解	40
5.4.2 实现函数调用栈回溯 (Backtrace)	40
5.4.3 实现定时器系统调用	40
5.4.4 遇到的问题及解决方式	40
第六章 Lab Copy on-write 写时复制实验	42
6.1 实现写时复制的 Fork 系统调用	42
6.2 小结	45
6.2.1 写时复制机制的实现	45
6.2.2 中断处理与内存页权限管理	45
6.2.3 遇到的问题与解决方法	45
6.2.4 实验结果	46
第七章 Lab Multithreading: 多线程实验	47
7.1 用户态线程库 Uthread	47
7.2 线程的使用	50
7.3 线程屏障	51
7.4 小结	52
7.4.1 用户态线程库的实现	52
7.4.2 多线程中的同步与互斥	52
7.4.3 线程屏障的实现与应用	52
7.4.4 遇到的问题与解决方法	53
7.4.5 实验结果	53
第八章 Lab network driver 网卡驱动实验	54
8.1 实现 Intel E1000 网卡的驱动	54
8.2 小结: 编写驱动程序的一般步骤	56
8.2.1 Intel E1000 网卡驱动的实现	56
8.2.2 锁机制与内存屏障的应用	56
8.2.3 遇到的问题与解决方法	57
8.2.4 实验结果	57
第九章 Lab Lock: 锁的实验	58
9.1 为每个 CPU 实现独占的内存分配器	58
9.2 实现 IO 缓存	61
9.3 小结	64
9.3.1 每个 CPU 独占的内存分配器	64
9.3.2 改进的 IO 缓存锁机制	64
9.3.3 遇到的问题与解决方法	64
9.3.4 实验结果	65

第十章 Lab File system: 文件系统实验	66
10.1 使文件系统支持大文件	66
10.2 实现符号链接	68
10.3 小结	70
10.3.1 遇到的问题与解决方法	70
10.3.2 实验结果	70
10.4 实验结果	70
第十一章 Lab mmap: 内存映射实验	72
11.1 实现 mmap 内存映射系统调用	72
11.2 小结	83
11.2.1 遇到的问题与解决方法	83

第一章 xv6实验概述与环境配置

1.1 简介

xv6 是一个基于 UNIX 第六版 (UNIX Version 6) 的教学操作系统。它由麻省理工学院 (MIT) 开发，用作操作系统课程中的教学工具。xv6 旨在帮助学生理解操作系统的基本原理和实现细节，通过简化和精简的代码结构，使学生能够专注于操作系统的核心概念而不被复杂的细节所困扰。

1.1.1 xv6 的特点和设计目标

- **简洁性和可读性：**xv6 的代码量较小，约为 9000 行 C 语言和少量的汇编代码，使其适合教学和学习。简洁的代码有助于学生理解操作系统的核心机制。
- **结构与 UNIX 类似：**xv6 基本上是对 UNIX 第六版的重新实现，因此它保留了 UNIX 的许多经典特性，如进程管理、内存管理、文件系统和设备驱动等。这使得学生可以从 xv6 的学习中获得对现代 UNIX/Linux 系统的理解。
- **教学目的：**xv6 被设计为一个教学工具，而非一个全面的商用操作系统。它的主要目的是为学生提供一个易于理解和实验的环境，而不是追求高效性或功能的完整性。
- **开放源码和文档：**xv6 的源码和相关文档是公开的，学生可以自由地查看、修改和实验。这种开放性有助于教学和研究。

1.1.2 xv6 的主要组成部分

- **内核 (Kernel)：**负责管理系统资源，包括进程、内存、文件系统和设备。内核实现了调度、多任务处理、中断处理等核心功能。
- **进程管理：**包括进程的创建、销毁、调度和切换。xv6 支持基本的多任务处理，允许多个进程同时运行。
- **内存管理：**实现了简单的内存分配和分页机制，使进程能够独立运行。
- **文件系统：**提供了文件和目录的基本操作。xv6 的文件系统支持常见的操作，如打开、关闭、读取、写入和删除文件。
- **设备驱动：**包括控制台、磁盘等设备的基本驱动程序，实现了设备与操作系统的交互。

xv6 是一个理想的教学工具，帮助我们理解操作系统的基本原理和设计思想。通过对 xv6 的学习和实验，我们能够获得操作系统设计和实现的实践经验，为将来的深入研究或开发工作打下坚实的基础。

1.2 环境配置

本课程要求使用几个不同的 RISC-V 工具，包括 QEMU 5.1+、GDB 8.3+、GCC 和 Binutils。

1.2.1 在 Windows 上安装

首先，确保系统已安装 Windows Subsystem for Linux (WSL)。接着，从 Microsoft Store 中安装 Ubuntu 20.04。启动 Ubuntu 之后，可以通过以下命令安装所需的软件：

安装命令

```
$ 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
```

在 Windows 中，可以通过路径 `\\wsl$` 访问 WSL 中的所有文件。例如，Ubuntu 20.04 的主目录通常位于 `\\wsl$Ubuntu-20.04\home\<username>`。

1.2.2 测试安装

安装完成后，可以通过编译和运行 `xv6` 来测试环境配置是否正确。以下命令将在 `xv6` 目录中启动 QEMU（按 `Ctrl-a x` 退出 QEMU）：

测试命令

```
# 进入 xv6 目录
$ make qemu
# ... 大量输出 ...
init: starting sh
$
```

如果出现问题，可以检查各个组件的版本，例如 QEMU 和 GCC 的 RISC-V 版本：

检查组件版本

```
$ qemu-system-riscv64 --version
QEMU emulator version 5.1.0

$ riscv64-linux-gnu-gcc --version
riscv64-linux-gnu-gcc (Debian 10.3.0-8) 10.3.0
```

通过上述步骤，就能够成功配置并运行 `xv6` 实验环境。

第二章 Lab Utilities: 实用工具实验

2.1 启动 xv6

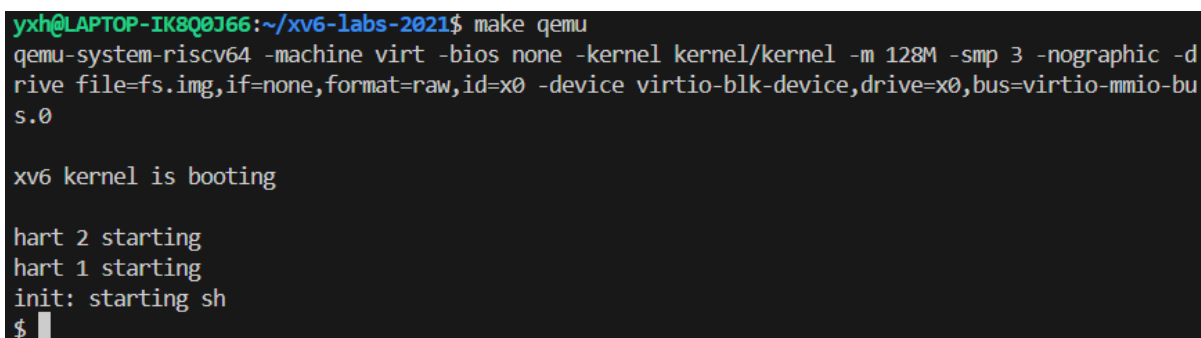
在前文中我们已经对xv6环境进行了配置，并且成功将mit实验仓库clone到本地，现在我们将进行第一个步骤-启动xv6

首先，我们需要将实验的代码库切换到 Lab Utilities 的分支，在终端中使用下面的指令

切换分支

```
$ cd xv6-labs-2021
$ git checkout util
Branch 'util' set up to track remote branch 'util' from 'origin'.
Switched to a new branch 'util'
```

然后直接使用make qemu 即可编译并在 qemu 中运行 xv6 。



```
yxh@LAPTOP-1K8Q0J66:~/xv6-labs-2021$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$
```

图 2.1: 成功启动xv6

如果在提示符下键入 ls，看到类似于以下内容的输出

```

$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2226
xargstest.sh 2 3 93
cat        2 4 23896
echo       2 5 22728
forktest   2 6 13080
grep       2 7 27248
grep       2 7 27248
grep       2 7 27248
init       2 8 23824
kill       2 9 22696
ln         2 10 22648
ls         2 11 26128
mkdir      2 12 22792
rm         2 13 22784
sh         2 14 41664
stressfs   2 15 23800
usertests  2 16 156008
grind      2 17 37968
wc         2 18 25032
zombie     2 19 22192
sleep      2 20 22656
console    3 21 0
$

```

图 2.2: xv6 目录

这些是 mkfs 包含在初始文件系统;大多数是可以运行的程序。

若要结束运行 xv6 并终止 qemu, 需在键盘上同时按下 Ctrl+A 键, 然后按下 X 键, 即可终止 qemu 的运行。

2.2 实现 sleep 工具

sleep 实用程序的实现遵循了简单的步骤。我们首先研究了 user/ 目录中现有的用户空间程序（如 echo.c）的结构。基于这种结构，我们创建了一个名为 sleep.c 的新文件。

sleep.c 程序的核心逻辑包括：

- 检查是否提供了正确数量的参数。
- 使用 atoi 函数将输入参数从字符串转换为整数。
- 使用转换后的整数调用 sleep 系统调用。
- 使用 exit(0) 退出程序。

由于 sleep 系统调用只接受一个整数参数来指定睡眠时间，因此我们可以简化对命令行参数的处理。具体步骤如下：

1. 包含必要的头文件：

我们首先需要包含 kernel/types.h、kernel/stat.h 和 user/user.h 头文件，这些文件提供了程序所需的类型定义和系统调用函数。

2. 检查命令行参数：

由于 sleep 只需要一个参数（指定睡眠的滴答数），我们首先检查传递给程序的参数数量是否正确（即 argc 是否等于 2）。如果用户没有传入正确的参数数量，我们将输出使用提示，并调用 exit(1) 以非正常状态退出程序。

3. 参数转换和验证：

使用 `user/user.h` 提供的 `atoi(const char*)` 函数，将用户输入的字符串参数转化为整数。转化后，我们检查该整数是否为非负数，因为 `sleep` 的参数应非负数。如果参数为负，程序将输出错误消息并退出。

4. 调用 `sleep` 系统调用并退出：

在参数验证通过后，调用 `sleep(int ticks)` 系统调用，使程序暂停指定的滴答数。睡眠完成后，调用 `exit(0)` 以正常状态退出程序。

基于上述思路，编写的 `user/sleep.c` 代码如下：

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
int main(int argc, char *argv[])
{
    if (argc != 2) {
        printf("Usage: sleep <ticks>\n");
        exit(1);
    }

    int ticks = atoi(argv[1]);

    if (ticks < 0) {
        printf("sleep: ticks must be a non-negative integer\n");
        exit(1);
    }

    sleep(ticks);
    exit(0);
}
```

Listing 2.1: `sleep.c` 的实现

在编写完 `sleep.c` 文件后，需要将其集成到 `xv6` 的构建系统中。具体操作是在 `Makefile` 中的 `UPROGS` 环境变量中添加 `sleep` 程序。添加的内容如下：

```
$U/_sleep\
```

Listing 2.2: `Makefile` 修改

完成这些修改后，运行 `make qemu` 即可编译我们的程序。在进入 `xv6` 后，输入 `ls` 命令，可以看到我们的程序已经在文件系统的根目录下。执行 `sleep 10`，其行为符合预期。

使用 `xv6` 实验自带的测评工具测评，在终端里输入 `./grade-lab-util sleep`，即可进行自动评测，如下图所示

```

yxh@LAPTOP-IK8Q0J66:~/xv6-labs-2021$ ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.3s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.1s)

```

图 2.3: sleep 的评测结果

可以看出测试全部通过

2.3 实现 pingpong 工具

pingpong 工具的实现遵循了简单的步骤。我们首先研究了 `user/` 目录中现有的用户空间程序（如 `echo.c`）的结构。基于这种结构，我们创建了一个名为 `pingpong.c` 的新文件。

pingpong.c 程序的核心逻辑包括：

- 创建管道来传输字节数据。
- 使用 `fork()` 创建子进程。
- 在父进程中向子进程发送一个字节，并从子进程接收响应。
- 在子进程中接收父进程发送的字节，打印消息，然后将字节传回父进程。

具体步骤如下：

1. 包含必要的头文件：

我们首先需要包含 `kernel/types.h`、`kernel/stat.h` 和 `user/user.h` 头文件，这些文件提供了程序所需的类型定义和系统调用函数。

2. 创建管道和子进程：

使用 `pipe()` 创建两个管道，一个用于父进程向子进程发送数据，另一个用于子进程向父进程返回数据。然后使用 `fork()` 创建一个子进程。

3. 实现父子进程间的通信：

在父进程中，通过第一个管道发送一个字节到子进程，并等待从第二个管道接收子进程返回的字节。在子进程中，接收父进程发送的字节，打印接收到的消息后，通过第二个管道将字节传回父进程。

4. 输出结果并退出：

父进程和子进程分别输出接收到的消息后，调用 `exit(0)` 正常退出程序。

基于上述思路，编写的 `user/pingpong.c` 代码如下：

```

#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main()

```

```

{
    int p1[2], p2[2];
    pipe(p1);
    pipe(p2);

    if (fork() == 0) {
        char buf;
        read(p1[0], &buf, 1);
        printf("%d: received ping\n", getpid());
        write(p2[1], &buf, 1);
        exit(0);
    } else {
        char buf = '!';
        write(p1[1], &buf, 1);
        read(p2[0], &buf, 1);
        printf("%d: received pong\n", getpid());
        exit(0);
    }
}

```

Listing 2.3: pingpong.c 的实现

在编写完 `pingpong.c` 文件后，需要将其集成到 `xv6` 的构建系统中。具体操作是在 `Makefile` 中的 `UPROGS` 环境变量中添加 `pingpong` 程序。添加的内容如下：

```
$U/_pingpong\
```

Listing 2.4: Makefile 修改

完成这些修改后，运行 `make qemu` 即可编译我们的程序。在进入 `xv6` 后，输入 `ls` 命令，可以看到我们的程序已经在文件系统的根目录下。执行 `pingpong`，其行为符合预期。

使用 `xv6` 实验自带的测评工具测评，在终端里输入 `./grade-lab-util pingpong`，即可进行自动评测，如下图所示：

```

vzh@LAPTOP-1K8Q0J66: /xv6-labs-2021$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (0.7s)

```

图 2.4: pingpong 的评测结果

可以看出，测试全部通过，程序能够在父子进程间成功传递字节并打印预期输出。

2.4 实现 primes 工具

`primes` 工具的实现基于管道和并发进程。我们首先研究了 `user/` 目录中现有的用户空间程序（如 `pingpong.c`）的结构。基于这种结构，我们创建了一个名为 `primes.c` 的新文件。

`primes.c` 程序的核心逻辑包括：

- 使用管道和 `fork()` 创建进程链，每个进程筛选掉非素数。

- 第一个进程将数字 2 到 35 输入到管道中。
- 对于每个素数，创建一个新进程，负责筛选掉该素数的倍数，并将其他数字传递给下一个进程。
- 依次筛选出素数，直到所有数字处理完毕。

具体步骤如下：

1. 包含必要的头文件：

我们首先需要包含 `kernel/types.h`、`kernel/stat.h` 和 `user/user.h` 头文件，这些文件提供了程序所需的类型定义和系统调用函数。

2. 创建管道并生成初始进程：

首先，使用 `pipe()` 创建一个管道，用于传递数字。然后使用 `fork()` 创建子进程。父进程负责将数字 2 到 35 写入管道，子进程将继续处理这些数字。

3. 子进程筛选素数并递归创建新进程：

子进程从管道中读取第一个数字，并将其标记为素数，然后创建一个新的管道和子进程。新子进程继续从其左邻居接收数字，筛选掉当前素数的倍数，并将剩余数字传递给右邻居，依次递归，直到所有数字处理完毕。

4. 关闭不需要的文件描述符并等待子进程完成：

每个进程都需要关闭它不使用的文件描述符，以节省资源并避免阻塞管道。所有子进程完成后，父进程最终退出。

基于上述思路，编写的 `user/primes.c` 代码如下：

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

void sieve(int p_left[2]);

int main()
{
    int p[2];
    pipe(p);

    if (fork() == 0) {
        close(p[1]);
        sieve(p);
    } else {
        close(p[0]);
        for (int i = 2; i <= 35; i++) {
            write(p[1], &i, sizeof(i));
        }
        close(p[1]);
    }
}
```

```

        wait(0);
    }
    exit(0);
}

void sieve(int p_left[2])
{
    int prime;
    if (read(p_left[0], &prime, sizeof(prime)) == 0) {
        close(p_left[0]);
        exit(0);
    }

    printf("prime %d\n", prime);

    int p_right[2];
    pipe(p_right);

    if (fork() == 0) {
        close(p_left[0]);
        close(p_right[1]);
        sieve(p_right);
    } else {
        close(p_right[0]);
        int num;
        while (read(p_left[0], &num, sizeof(num)) != 0) {
            if (num % prime != 0) {
                write(p_right[1], &num, sizeof(num));
            }
        }
        close(p_left[0]);
        close(p_right[1]);
        wait(0);
        exit(0);
    }
}

```

Listing 2.5: primes.c 的实现

在编写完 `primes.c` 文件后，需要将其集成到 `xv6` 的构建系统中。具体操作是在 `Makefile` 中的 `UPROGS` 环境变量中添加 `primes` 程序。添加的内容如下：

```
$U/_primes\
```

Listing 2.6: Makefile 修改

完成这些修改后，运行 `make qemu` 即可编译我们的程序。在进入 `xv6` 后，输入 `ls` 命令，可以看到我们的程序已经在文件系统的根目录下。执行 `primes`，其行为符合预期。

使用 xv6 实验自带的测评工具测评，在终端里输入 `./grade-lab-util primes`，即可进行自动评测，如下图所示：

```
yxh@LAPTOP-1K8Q0J66: /xv6-labs-2021$ ./grade-lab-util primes
make: 'kernel/kernel' is up to date.
== Test primes == primes: OK (0.7s)
```

图 2.5: primes 的评测结果

可以看出，测试全部通过，程序能够成功筛选出素数并打印预期输出。

2.5 实现 find 工具

`find` 工具的实现遵循了一个递归的逻辑，以便在目录树中查找具有特定名称的所有文件。我们首先研究了 `user/ls.c` 的实现方式，以了解如何读取目录。基于这种理解，我们创建了一个名为 `find.c` 的新文件。

`find.c` 程序的核心逻辑包括：

- 打开指定目录并读取其内容。
- 递归地遍历子目录，以查找与目标名称匹配的文件。
- 确保不递归进入当前目录（`."`）和父目录（`".."`）。
- 使用 `strcmp()` 函数比较字符串，而不是使用 `==` 运算符。

具体步骤如下：

1. 包含必要的头文件：

我们首先需要包含 `kernel/types.h`、`kernel/stat.h` 和 `user/user.h` 头文件，这些文件提供了程序所需的类型定义和系统调用函数。

2. 实现目录遍历功能：

使用 `open()` 打开目录，并使用 `read()` 读取目录内容。使用递归调用函数的方式遍历子目录，并查找匹配指定名称的文件。每当找到匹配文件时，打印其路径。

3. 处理特殊目录项 `."` 和 `".."`：

在遍历目录时，特别处理 `."` 和 `".."` 目录项，以避免递归进入当前目录或父目录，防止无限递归。

4. 输出结果并退出：

在遍历完成后，关闭打开的目录，退出程序。

基于上述思路，编写的 `user/find.c` 代码如下：

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
#include "kernel/fs.h"
```



```
void find(const char *path, const char *target) {
    char buf[512], *p;
    int fd;
    struct dirent de;
    struct stat st;

    if ((fd = open(path, 0)) < 0) {
        printf("find: cannot open %s\n", path);
        return;
    }

    if (fstat(fd, &st) < 0) {
        printf("find: cannot stat %s\n", path);
        close(fd);
        return;
    }

    if (st.type != T_DIR) {
        close(fd);
        return;
    }

    if (strlen(path) + 1 + DIRSIZ + 1 > sizeof(buf)) {
        printf("find: path too long\n");
        close(fd);
        return;
    }

    strcpy(buf, path);
    p = buf + strlen(buf);
    *p++ = '/';

    while (read(fd, &de, sizeof(de)) == sizeof(de)) {
        if (de.inum == 0)
            continue;
        memmove(p, de.name, DIRSIZ);
        p[DIRSIZ] = 0;
        if (strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0)
            continue;
        if (stat(buf, &st) < 0) {
            printf("find: cannot stat %s\n", buf);
            continue;
        }
        if (strcmp(de.name, target) == 0) {
```

```

        printf("%s\n", buf);
    }
    if (st.type == T_DIR) {
        find(buf, target);
    }
}
close(fd);
}

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage: find <path> <name>\n");
        exit(1);
    }
    find(argv[1], argv[2]);
    exit(0);
}

```

Listing 2.7: find.c 的实现

在编写完 `find.c` 文件后，需要将其集成到 xv6 的构建系统中。具体操作是在 `Makefile` 中的 `UPROGS` 环境变量中添加 `find` 程序。添加的内容如下：

```
$U/_find\
```

Listing 2.8: Makefile 修改

完成这些修改后，运行 `make qemu` 即可编译我们的程序。在进入 xv6 后，输入 `ls` 命令，可以看到我们的程序已经在文件系统的根目录下。执行 `find . b`，其行为符合预期。

使用 xv6 实验自带的测评工具测评，在终端里输入 `./grade-lab-util find`，即可进行自动评测，如下图所示：

```

yxh@LAPTOP-IK8Q0J66:~/xv6-labs-2021$ ./grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory: OK (1.3s)
    (Old xv6.out.find_curdir failure log removed)
== Test find, recursive == find, recursive: OK (1.0s)
    (Old xv6.out.find_recursive failure log removed)

```

图 2.6: find 的评测结果

可以看出，测试全部通过，程序能够成功查找到指定名称的文件并打印其路径。+

2.6 实现 xargs 工具

`xargs` 工具的实现主要通过读取标准输入的每一行，并对每行执行一个指定的命令，将这一行作为参数传递给该命令。我们首先研究了 `fork` 和 `exec` 系统调用的用法，以便在每一行输入上调用命令。基于这种理解，我们创建了一个名为 `xargs.c` 的新文件。

`xargs.c` 程序的核心逻辑包括：

- 逐个字符读取标准输入，直到检测到换行符，这表示一行的结束。

- 为每一行输入调用 `fork()` 创建一个子进程，并使用 `exec()` 调用指定的命令，将该行作为参数传递。
- 父进程使用 `wait()` 等待子进程完成。

具体步骤如下：

1. 包含必要的头文件：

我们首先需要包含 `kernel/types.h`、`kernel/stat.h` 和 `user/user.h` 头文件，这些文件提供了程序所需的类型定义和系统调用函数。

2. 读取输入并执行命令：

使用 `read()` 函数逐字符读取标准输入，直到读取到换行符。一旦读取到完整的一行，使用 `fork()` 创建一个子进程，并使用 `exec()` 执行命令，将读取的行作为命令的参数传递给子进程。

3. 处理命令执行和等待子进程完成：

子进程使用 `exec()` 来执行指定命令。父进程则使用 `wait()` 来等待子进程的完成，以确保所有命令都顺序执行。

基于上述思路，编写的 `user/xargs.c` 代码如下：

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "kernel/param.h"
#include "user/user.h"

char args[MAXARG][512];
char *pass_args[MAXARG];
int preargnum, argnum;
char ch;
char arg_buf[512];
int n;

int readline()
{
    argnum = preargnum;
    memset(arg_buf, 0, sizeof(arg_buf));
    for (;;)
    {
        n = read(0, &ch, sizeof(ch));
        if (n == 0)
        {
            return 0;
            break;
        }
        else if (n < 0)
```

```

        {
            fprintf(2, "read error\n");
            exit(1);
        }
        else
        {
            if (ch == '\n')
            {
                memcpy(args[argnum], arg_buf, strlen(arg_buf) + 1);
                argnum++;
                return 1;
            }
            else if (ch == ' ')
            {
                memcpy(args[argnum], arg_buf, strlen(arg_buf) + 1);
                argnum++;
                memset(arg_buf, 0, sizeof(arg_buf));
            }
            else
            {
                arg_buf[strlen(arg_buf)] = ch;
            }
        }
    }
}

int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        printf("usage: xargs [command] [arg1] [arg2] ... [argn]\n");
        exit(0);
    }
    preargnum = argc - 1;
    for (int i = 0; i < preargnum; i++)
        memcpy(args[i], argv[i + 1], strlen(argv[i + 1]));
    while (getline())
    {
        if (fork() == 0)
        {
            *args[argnum] = 0;
            int i = 0;
            while (*args[i])
            {

```

```

        pass_args[i] = (char *)&args[i];

        i++;
    }
    *pass_args[argnum] = 0;

    exec(pass_args[0], pass_args);
    printf("exec error\n");
    exit(0);
}
else
{
    wait((int *)0);
}
}
exit(0);
}

```

Listing 2.9: xargs.c 的实现

在编写完 `xargs.c` 文件后，需要将其集成到 `xv6` 的构建系统中。具体操作是在 `Makefile` 中的 `UPROGS` 环境变量中添加 `xargs` 程序。添加的内容如下：

```
$U/_xargs\
```

Listing 2.10: Makefile 修改

完成这些修改后，运行 `make qemu` 即可编译我们的程序。在进入 `xv6` 后，输入 `ls` 命令，可以看到我们的程序已经在文件系统的根目录下。执行 `xargs`，其行为符合预期。

使用 `xv6` 实验自带的测评工具测评，在终端里输入 `./grade-lab-util xargs`，即可进行自动评测，如下图所示：

```

yxh@LAPTOP-IK8Q0J66:~/xv6-labs-2021$ ./grade-lab-util xargs
make: 'kernel/kernel' is up to date.
== Test xargs == xargs: OK (0.7s)
(Old xv6.out.xargs failure log removed)

```

图 2.7: xargs 的评测结果

可以看出，测试全部通过，程序能够成功读取输入并执行命令，输出预期结果。

2.7 小结

在完成 Lab Utilities 中的所有实验后，根据 MIT 6.S081 的传统，需要在实验目录下创建一个名为 `time.txt` 文本文件，其中只包含一行，为完成该实验的小时数。然后在终端中执行 `./grade-lab-util`，即可对整个实验进行自动评分，笔者的结果如下：

```
yxh@LAPTOP-IK8Q0J66:~/xv6-labs-2021$ ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.4s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
== Test pingpong == pingpong: OK (1.0s)
== Test primes == primes: OK (1.1s)
== Test find, in current directory == find, in current directory: OK (1.1s)
== Test find, recursive == find, recursive: OK (1.0s)
== Test xargs == xargs: OK (1.0s)
== Test time ==
time: OK
Score: 100/100
```

图 2.8: 第二章的所有实验得分

在本章中，我深入学习并实践了 xv6 操作系统的几个基本实用工具的实现。这些工具涵盖了从简单的命令实现（如 `sleep` 和 `pingpong`）到更复杂的进程间通信与递归算法（如 `primes` 和 `find`）。在实验过程中，遇到了一些问题并通过以下方式解决：

2.7.1 进程间通信问题

问题描述：在实现 `pingpong` 工具时，遇到了父子进程间通信不顺畅的问题。具体表现为子进程未能接收到父进程发送的字节，或子进程的输出与预期不符。

解决方法：通过进一步调试，发现管道的创建和使用存在问题。我通过 `pipe()` 函数创建了两个管道，确保父进程和子进程分别关闭自己不需要的管道端口，以避免阻塞或错误读写。然后，仔细检查了 `read()` 和 `write()` 调用的顺序，确保数据流正确无误。最终修复了通信问题，实现了进程间的正确交互。

2.7.2 递归实现与管道通信问题

问题描述：在实现 `primes` 工具时，使用了递归调用的方式来筛选素数。在递归创建新进程时，我发现子进程没有正确地接收并处理传递下来的数据，导致计算出错或程序卡死。

解决方法：首先，仔细检查了递归调用中的管道管理，确保每个进程正确地关闭了不再需要的管道端口，并确保数据在正确的管道端口之间流动。其次，在递归调用前后，使用调试打印信息来确认每一步的行为，找到问题所在并修正。最终通过调整递归逻辑和正确管理文件描述符，确保了每个进程正确筛选并传递素数。

2.7.3 目录遍历与文件查找问题

问题描述：在实现 `find` 工具时，我遇到了递归遍历目录结构的挑战。尤其是如何正确处理目录的特殊项 `.` 和 `..`，避免进入无限循环。

解决方法：首先，我参考了 `ls.c` 的实现方式，学习如何读取目录并判断文件类型。在实现 `find.c` 时，特别注意了如何处理 `.` 和 `..`，使用 `strcmp()` 函数来排除这些目录项。通过递归调用，成功实现了目录树的遍历并正确查找目标文件。

在这些实验中，通过解决这些问题，我不仅加深了对 C 语言及系统编程的理解，也增强了在实际开发环境中处理复杂问题的能力。最终，所有工具都成功实现并通过了实验的自动测试。

第三章 System Calls 系统调用实验

在上一章实验中，我们使用系统调用编写了一些实用工具。在本实验中，我们将向 xv6 操作系统添加一些新的系统调用。这将帮助我们深入理解系统调用的工作原理，并让我们了解 xv6 内核的一些内部结构。

3.1 实现 System call tracing 工具

在本节实验中，我们将添加一个系统调用跟踪功能，该功能可能在调试后续实验中有所帮助。我们将创建一个新的 `trace` 系统调用来控制跟踪。该调用接受一个整数参数 `mask`，其位指定要跟踪的系统调用。例如，要跟踪 `fork` 系统调用，程序会调用 `trace(1 << SYS_fork)`，其中 `SYS_fork` 是 `kernel/syscall.h` 中的系统调用编号。

`trace` 系统调用的核心逻辑包括：

- 允许用户指定一个掩码，其中的位表示要跟踪的系统调用。
- 在系统调用返回时，如果对应的掩码位被设置，则打印出进程 ID、系统调用名称和返回值。
- 使 `trace` 系统调用对调用它的进程及其子进程启用跟踪，而不影响其他进程。

具体步骤如下：

1. 修改 Makefile:

首先，在 `Makefile` 中添加 `$U/_trace` 到 `UPROGS`，以确保 `trace` 程序被编译。然后运行 `make qemu`，会看到编译器无法编译 `user/trace.c`，因为系统调用的用户空间存根尚未存在。

2. 添加系统调用存根:

将 `trace` 系统调用的原型添加到 `user/user.h` 中，并在 `user/usys.pl` 中添加存根，最后在 `kernel/syscall.h` 中添加系统调用编号 `SYS_trace`。

3. 实现 sys_trace:

在 `kernel/sysproc.c` 文件中实现 `sys_trace()` 函数，通过在 `proc` 结构中的新变量 `tracemask` 中记住其参数来实现新的系统调用。在 `kernel/syscall.c` 中添加从用户空间检索系统调用参数的代码，并在 `kernel/proc.c` 中修改 `fork()` 函数，以便子进程继承父进程的 `tracemask`。

4. 修改 syscall() 函数:

在 `kernel/syscall.c` 文件中，修改 `syscall()` 函数，添加一个系统调用名称数组，并在函数内打印出跟踪信息。如果进程的 `tracemask` 中设置了对应的位，则输出该进程的 ID、系统调用名称和返回值。

基于上述思路，编写的代码如下：

```

int trace(int mask);
entry("trace");
#define SYS_trace 22
uint64
sys_trace(void)
{
    int mask;
    if(argint(0, &mask) < 0)
        return -1;
    myproc()->tracemask = mask;
    return 0;
}
int
fork(void)
{
    ...
    np->tracemask = p->tracemask;
    ...
}
void syscall(void)
{
    int num;
    struct proc *p = myproc();
    num = p->trapframe->a7;
    if (num > 0 && num < NELEM(syscalls) && syscalls[num])
    {
        p->trapframe->a0 = syscalls[num]();

        if (p->tracemask >> num & 1)
        {
            printf("%d: syscall %s -> %d\n",
                p->pid, syscallnames[num], p->trapframe->a0);
        }
    }
    else
    {
        printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
        p->trapframe->a0 = -1;
    }
}

```

Listing 3.1: trace 系统调用的实现

完成上述步骤后，运行 `./grade-lab-syscall trace` 并测试 `trace` 程序

```

yxh@LAPTOP-1K8Q0J66:~/xv6-labs-2021$ ./grade-lab-syscall trace
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (1.0s)
    (Old xv6.out.trace_32_grep failure log removed)
== Test trace all grep == trace all grep: OK (0.8s)
    (Old xv6.out.trace_all_grep failure log removed)
== Test trace nothing == trace nothing: OK (1.0s)
== Test trace children == trace children: OK (7.8s)
    (Old xv6.out.trace_children failure log removed)

```

图 3.1: trace 的评测结果

可见测试全部通过

3.2 实现 Sysinfo 系统调用

常见的操作系统都会提供一些系统调用来帮助应用程序获取关于系统的信息，因此我们被要求实现一个 `sysinfo` 系统调用，用于获取 `xv6` 运行时的信息。具体来说，`xv6` 已经为我们定义好了一个结构体 `struct sysinfo`（在源码 `kernel/sysinfo.h` 中），我们的 `sysinfo` 系统调用接受一个参数，即指向该结构体的指针，然后将对应的内容填入该结构体（`freemem` 字段填写空闲的内存空间字节数，`nproc` 字段填写状态为未使用（`UNUSED`）的进程数）。

`sysinfo` 系统调用的核心逻辑包括：

- 获取系统中空闲的内存空间字节数。
- 统计当前状态为非 `UNUSED` 的进程数。
- 将这些信息填写到用户传递的 `sysinfo` 结构体中。

具体步骤如下：

1. 修改 Makefile:

首先，在 `Makefile` 中添加 `$U/_sysinfotest` 到 `UPROGS`，以确保 `sysinfotest` 程序被编译。然后按照流程添加 `sysinfo` 系统调用的入口，在 `user/user.h` 中添加 `struct sysinfo` 的声明：

```

struct sysinfo;
int sysinfo(struct sysinfo *);

```

2. 实现获取空闲内存的函数：

在 `kernel/kalloc.c` 中实现获取空闲内存大小的函数。由于 `xv6` 使用空闲链表管理内存空间，因此我们遍历链表并计算空闲内存的页面数量，然后乘以页面大小即可得到空闲内存的字节数。代码实现如下：

```

int getfreemem(void)
{
    int count = 0;

```

```

        struct run *r;
        acquire(&kmem.lock);
        r = kmem.freelist;
        while (r)
        {
            count++;
            r = r->next;
        }
        release(&kmem.lock);
        return count * PGSIZE;
    }

```

Listing 3.2: getfreemem 函数的实现

3. 实现统计非 UNUSED 状态的进程数:

在 kernel/proc.c 中实现统计非 UNUSED 状态的进程数的函数。由于进程控制块使用的是静态数组管理，我们通过循环遍历该数组并计数非 UNUSED 状态的进程数。代码实现如下：

```

int getnproc(void)
{
    struct proc *p;
    int count = 0;
    for (p = proc; p < &proc[NPROC]; p++)
    {
        acquire(&p->lock);
        if (p->state != UNUSED)
        {
            count++;
            release(&p->lock);
        }
        else
        {
            release(&p->lock);
        }
    }
    return count;
}

```

Listing 3.3: getnproc 函数的实现

4. 实现 sysinfo 系统调用:

在 kernel/sysproc.c 文件中实现 sysinfo 系统调用。该调用会获取系统的空闲内存和非 UNUSED 状态的进程数，并将这些信息填入用户传递的 sysinfo 结构体。实现过程中，我们使用 copyout 函数将数据从内核空间复制到用户空间。代码如下：

```
extern int getnproc(void);
extern int getfreemem(void);

uint64
sys_sysinfo(void)
{
    struct proc *p = myproc();
    struct sysinfo st;
    uint64 addr;

    st.freemem = getfreemem();
    st.nproc = getnproc();

    if (argaddr(0, &addr) < 0)
        return -1;
    if (copyout(p->pagetable, addr, (char *)&
st, sizeof(st)) < 0)
        return -1;

    return 0;
}
```

Listing 3.4: sysinfo 系统调用的实现

完成上述步骤后，编译并启动 xv6 系统，在 shell 中运行 `sysinfotest`，可以看到预期的输出：

测试输出

```
$ sysinfotest
sysinfotest: start
sysinfotest: OK
$
```

使用 xv6 实验自带的测评工具进行测试，在终端中输入 `./grade-lab-syscall sysinfo`，即可进行自动评测，如下图所示：

```
yxh@LAPTOP-1K8Q0J66:~/xv6-labs-2021$ ./grade-lab-syscall sysinfo
make: 'kernel/kernel' is up to date.
== Test sysinfotest == sysinfotest: OK (1.5s)
(Old xv6.out.sysinfotest failure log removed)
```

图 3.2: sysinfo 的测评结果

可以看到，所有测试全部通过，系统调用功能实现正确。

3.3 小结

在完成 Lab System Calls 中的所有实验后，根据 MIT 6.S081 的传统，需要在实验目录下创建一个名为 `time.txt` 文本文件，其中只包含一行，为完成该实验的小时数。然后在终端中执行 `./grade-lab-syscall`，即可对整个实验进行自动评分，笔者的结果如下：

```
yxh@LAPTOP-1K8Q0J66:~/xv6-labs-2021$ ./grade-lab-syscall
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (1.3s)
== Test trace all grep == trace all grep: OK (0.8s)
== Test trace nothing == trace nothing: OK (1.0s)
== Test trace children == trace children: OK (7.8s)
== Test sysinfotest == sysinfotest: OK (1.0s)
== Test time ==
time: OK
Score: 35/35
```

图 3.3: 第三章的所有实验得分

在本章中，我们深入学习并实践了如何向 xv6 操作系统中添加新的系统调用。这些实验不仅帮助我们理解了系统调用的工作原理，还让我们对 xv6 内核的一些重要结构和机制有了更深入的了解。

3.3.1 实现 trace 系统调用

我们首先实现了 `trace` 系统调用，该调用允许用户通过设置掩码来跟踪特定的系统调用。通过这个实验，我们学会了如何修改 xv6 内核的进程结构，如何在系统调用的执行过程中加入调试信息，以及如何继承父进程的属性。这为我们提供了一种有效的手段来调试和理解系统调用的行为。

3.3.2 实现 sysinfo 系统调用

接着，我们实现了 `sysinfo` 系统调用，该调用允许用户获取系统运行时的基本信息，如空闲内存空间和当前活跃的进程数。在这个实验中，我们深入研究了 xv6 内存管理和进程管理的实现，并学会了如何在内核空间和用户空间之间传递数据。

3.3.3 遇到的问题及解决方式

在完成本章实验的过程中，我们遇到了一些问题，通过逐步分析和调试，最终解决了这些问题。以下是遇到的主要问题及其解决方式。

问题一：trace 系统调用中的系统调用名称显示问题

问题描述： 在实现 `trace` 系统调用时，最初的实现中，系统调用名称在输出时出现了错误或显示为空。这使得跟踪信息无法正确反映系统调用的名称，从而影响调试效果。

解决方式： 经过检查，发现问题的原因是系统调用名称数组未正确初始化，或者在数组的索引使用上存在错误。为了解决这个问题，我们在 `syscall()` 函数中仔细检查了系统调用编号和名称数组的匹配关系，确保每个系统调用编号都正确对应其名称。同时，还在调试信息中添加了更多的输出以验证数组的初始化是否正确。最终，通过修正数组初始化和索引访问的方式，成功解决了系统调用名称显示的问题。

问题二：sysinfo 系统调用中的内存统计不准确

问题描述： 在实现 `sysinfo` 系统调用时，初始实现的空闲内存统计不准确，导致输出的空闲内存数与实际情况不符。由于空闲内存是通过遍历空闲链表计算得出的，出现统计错误会直接影响系统信息的准确性。

解决方式： 为了解决该问题，我们首先在 `kernel/kalloc.c` 文件中增加了调试信息，打印出空闲链表的遍历过程和每次遍历得到的空闲页面数。在逐步调试过程中，发现由于多线程竞争访问链表的原因，链表遍历时有可能获取不到最新的数据。为了解决这个问题，我们在访问空闲链表时加上了锁机制，确保统计过程中的一致性。同时，增加了对页面大小 (`PGSIZE`) 的检查，确保内存计算的单位正确。最终，通过加锁和调试，成功修正了空闲内存统计的问题。

问题三：sysinfo 系统调用中的进程计数不准确

问题描述： 在 `sysinfo` 系统调用的初期实现中，进程计数存在不准确的问题，导致系统报告的活跃进程数与实际不符。初步排查发现，有些状态变化较快的进程在统计过程中可能被遗漏或重复计数。

解决方式： 为了解决这个问题，我们首先在 `kernel/proc.c` 文件中，对进程数组的遍历和计数逻辑进行了详细检查。我们发现，由于未对进程控制块 (`proc` 结构) 的访问进行必要的加锁，导致在某些情况下，进程状态在被读取时已经发生变化。为此，我们在进程状态访问时加上了必要的锁定机制 (`acquire` 和 `release`)，确保在统计过程中，进程状态不被其他操作打断。通过这种方式，进程计数的准确性得到了显著提高。

第四章 Pages Tables 实验室页表实验

在上一个实验中，我们初次修改 xv6 的内核，为其添加了两个系统调用。本次实验的重点则是在于操作系统的另一个机制：页表。在本次实验中，我们会初探页表的一些性质，并利用页表机制完成一些任务。

4.1 加速系统调用

本实验中，需要在进程创建时将一个内存页面以只读权限映射到 USYSCALL 位置（参见 memlayout.h）中的定义。该映射的页面开头存储有内核数据结构 `struct usyscall`，该数据结构在进程创建时被初始化，并且存储有该进程的 `pid`。在这个实验中，我们使用纯用户空间的函数 `ugetpid()` 来替代需要进行内核态到用户态拷贝的 `getpid` 系统调用。用户空间的函数 `ugetpid()` 已经在用户空间中被实现了，我们需要做的是将映射页面的工作完成。本实验中，需要在进程创建时将一个内存页面以只读权限映射到 USYSCALL 位置（参见 memlayout.h 中的定义）。该映射的页面开头存储有内核数据结构 `struct usyscall`，该数据结构在进程创建时被初始化，并且存储有该进程的 `pid`。在这个实验中，我们使用纯用户空间的函数 `ugetpid()` 来替代需要进行内核态到用户态拷贝的 `getpid` 系统调用。用户空间的函数 `ugetpid()` 已经在用户空间中被实现了，我们需要做的是完成映射页面的工作。

由于需要在创建进程时完成页面的映射，故而我们考虑修改 `kernel/proc.c` 中的 `proc_pagetable(struct proc *p)` 函数，即用于为新创建的进程分配页面的函数。在 `proc_pagetable()` 完成分配新的页面后，可以使用 `mappages()` 函数分配页面：

```
pagetable_t
proc_pagetable(struct proc *p)
{
    pagetable_t pagetable;
    pagetable = uvmcreate();
    if(pagetable == 0)
        return 0;
    if(mappages(pagetable, USYSCALL, PGSIZE,
        (uint64)(p->usyscall), PTE_R | PTE_U) < 0){
        uvmfree(pagetable, 0);
        return 0;
    }
    ...
}
```

Listing 4.1: 为新创建的进程分配页表并映射只读页面

为了确保页面映射正确且权限为只读，我们需要在进程初始化时分配并初始化 `usyscall` 页面，并在进程结束时释放分配的页面。

首先，在 `allocproc()` 中为 `usyscall` 分配页面，并初始化该页面：

```
...
if((p->usyscall = (struct usyscall *)kalloc()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
}
p->usyscall->pid = p->pid;
...
```

Listing 4.2: 在分配进程结构体时初始化 `usyscall` 页面

完成页面的初始化后，进程应当得以正常运行。在进程结束时，应当在 `freeproc()` 函数中释放分配的页面，以避免内存泄漏：

```
...
if(p->usyscall)
    kfree((void*)p->usyscall);
p->usyscall = 0;
...
```

Listing 4.3: 释放进程时清理 `usyscall` 页面

完成页面的初始化和回收后，映射页面的工作就全部完成了。编译并启动 `xv6` 后，在 `shell` 中运行 `pgtbltest`，即可看到预期的输出：

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
...
```

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
$
```

图 4.1: `pgtbl` 的测评结果

4.2 打印页表

为了便于直观地展示页表的内容，我们需要实现一个函数 `vmprint()`，它接收一个 `pagetable_t` 类型的参数，并按照特定格式输出页表的内容，例如：

页表打印

```

page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. .. ..509: pte 0x0000000021fdd813 pa 0x0000000087f76000
.. .. ..510: pte 0x0000000021fddc07 pa 0x0000000087f77000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000

```

在实现 `vmprint()` 函数前，我们可以参考 `freewalk()` 函数的实现，它通过递归的方式逐级释放页表的内存，代码如下：

```

void
freewalk(pagetable_t pagetable)
{
    for(int i = 0; i < 512; i++){
        pte_t pte = pagetable[i];
        if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
            uint64 child = PTE2PA(pte);
            freewalk((pagetable_t)child);
            pagetable[i] = 0;
        } else if(pte & PTE_V){
            panic("freewalk: leaf");
        }
    }
    kfree((void*)pagetable);
}

```

Listing 4.4: 递归释放页表的 `freewalk()` 函数

基于该函数，我们可以修改实现一个递归遍历页表并按层次输出的函数 `vmprintwalk()`：

```

void
vmprintwalk(pagetable_t pagetable, int depth)
{
    for(int i = 0; i < 512; i++){
        pte_t pte = pagetable[i];
        if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
            for (int n = 0; n < depth; n++)
                printf(" ..");
            printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte));
            uint64 child = PTE2PA(pte);

```



```

        vmprintwalk((pagetable_t)child, depth+1);
    } else if(pte & PTE_V){
        for (int n = 0; n < depth; n++)
            printf(" ..");
        printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte));
    }
}
}

```

Listing 4.5: 递归打印页表的 vmprintwalk() 函数

接着，在实现 vmprint() 时，只需指定初始深度为 1，并调用 vmprintwalk() 函数即可：

```

void
vmprint(pagetable_t pagetable)
{
    printf("page table %p\n", pagetable);
    vmprintwalk(pagetable, 1);
}

```

Listing 4.6: vmprint() 函数的实现

最后，为了在合适的时机输出页表内容，我们可以在 kernel/exec.c 文件的 exec() 函数中，在返回前加入如下代码：

```

if(p->pid == 1) {
    vmprint(p->pagetable);
}
return argc;

```

Listing 4.7: 在进程执行时打印页表

编译并启动 xv6 后，运行相应程序即可看到页表的输出结果，类似于下图所示：

```

xv6 kernel is booting

hart 2 starting
hart 1 starting
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. ..0: pte 0x0000000021fdaclf pa 0x0000000087f6b000
.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. .. ..2: pte 0x0000000021fd9clf pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. .. ..509: pte 0x0000000021fddc13 pa 0x0000000087f77000
.. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh

```

图 4.2: vmprint() 的输出示例

4.3 追踪被访问的页

许多现代编程语言都支持内存垃圾回收功能（GC, Garbage Collection），而垃圾回收器需要判断某个页面自上次检测以来是否被访问过。尽管这项功能可以通过纯软件手段来实现，但效率较低。为此，我们可以利用页表的硬件机制（例如访问位）与操作系统的结合，在 xv6 中添加一个名为 `pgaccess()` 的系统调用，用于读取页表的访问位，并将这些信息传递给用户态程序，从而有效地检测某些内存页面自上次检查以来的访问情况。这种方法更加高效。

`pgaccess()` 系统调用需要实现以下功能：它接收三个参数，第一个参数指定要检查的起始页面地址，第二个参数确定从该地址开始需要检查的页面数量，第三个参数则是一个指针，指向保存结果的位向量的起始地址。

首先，按照标准步骤添加系统调用，并在 `kernel/sysproc.c` 文件中实现 `sys_pgaccess()` 函数。通过使用 `argint()` 和 `argaddr()` 函数来获取传入的参数，如下所示：

```
int sys_pgaccess(void)
{
    uint64 srcva, st;
    int len;
    uint64 buf = 0;
    struct proc *p = myproc();
    acquire(&p->lock);
    argaddr(0, &srcva);
    argint(1, &len);
    argaddr(2, &st);
    ...
    return 0;
}
```

Listing 4.8: `sys_pgaccess()` 的初始实现

接下来，对获取到的参数进行处理。对于每个需要检查的页面，我们使用 `kernel/vm.c` 中提供的 `walk(pagetable_t, uint64, int)` 函数来获取对应的页表项。然后，重置该页表项中的 `PTE_A` 访问位，并通过位运算将访问状态保存到一个临时位向量中。最后，将这个临时位向量复制到用户空间指定的内存位置。以下是完整的代码实现：

```
extern pte_t *walk(pagetable_t, uint64, int);

int sys_pgaccess(void)
{
    uint64 srcva, st;
    int len;
    uint64 buf = 0;
    struct proc *p = myproc();
    acquire(&p->lock);
    argaddr(0, &srcva);
    argint(1, &len);
    argaddr(2, &st);

    if (len < 1 || len > 64)
```

```

    return -1;

    pte_t *pte;
    for (int i = 0; i < len; i++)
    {
        pte = walk(p->pagetable, srcva + i * PGSIZE, 0);
        if (!pte || !(*pte & PTE_V) || !(*pte & PTE_U)) {
            return -1;
        }
        if (*pte & PTE_A) {
            *pte &= ~PTE_A;
            buf |= (1 << i);
        }
    }
    release(&p->lock);
    copyout(p->pagetable, st, (char *)&buf, (len + 7) / 8);

    return 0;
}

```

Listing 4.9: sys_pgaccess() 的完整实现

注意：由于页表是内核中的数据结构，可能会被多个处理器同时操作，因此在操作时务必要确保加锁。

到此为止，pgaccess() 系统调用的实现已经完成。编译并启动 xv6 后，可以通过在 shell 中运行 pgtbltest 来验证功能是否正常，并查看预期的输出结果：

```

$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
$

```

图 4.3: pgaccess() 的测评结果

4.4 小结

在完成了本章关于页表的实验后，根据 MIT 6.S081 的要求，我们需要在实验目录下创建一个名为 time.txt 的文本文件，文件中仅包含一行内容，记录了完成这些实验所用的时间（以小时为单位）。然后，通过在终端中执行 make grade 命令，我们可以对整个实验进行自动评分。以下是笔者在实验中的评分结果：

```

== Test pgtbltest ==
$ make qemu-gdb
(2.1s)
== Test    pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
== Test    pgtbltest: pgaccess ==
pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.9s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
(70.4s)
== Test    usertests: all tests ==
usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
vxh@LAPTOP-1K8Q0J66:~/xv6-labs-2021$

```

图 4.4: 页表实验的最终得分

可见测试通过，为满分

在本章中，我们通过一系列实验深入理解并掌握了 xv6 操作系统中页表的机制。通过这些实验，我们不仅学习了如何在 xv6 中操作页表，还熟悉了内存管理的基本原理和机制。

4.4.1 加速系统调用的实现

我们首先实现了在进程创建时，将一个只读的内存页面映射到用户空间的 `USYSCALL` 位置上。这使得我们可以通过用户空间的函数 `ugetpid()` 来替代内核态到用户态的 `getpid` 系统调用，从而加速系统调用的执行。这个实验帮助我们理解了如何在 xv6 中操作页表，以及如何在进程初始化和清理时正确管理内存。

4.4.2 打印页表内容

接下来，我们实现了一个名为 `vmprint()` 的函数，该函数可以递归遍历并输出指定进程的页表内容。通过这个实验，我们进一步加深了对页表结构的理解，特别是页表项的多级结构以及如何通过递归的方式访问和输出每一层的内容。

4.4.3 追踪被访问的页

在本章的最后，我们实现了 `pgaccess()` 系统调用，该调用允许用户程序检查某些页面自上次访问以来是否被访问过。通过该实验，我们学会了如何使用页表的硬件访问位来追踪页面的访问情况，并将这些信息传递回用户空间。这一实验展示了如何高效地利用操作系统和硬件特性来实现功能强大的系统调用。

4.4.4 遇到的问题及解决方式

在整个实验过程中，我们也遇到了若干问题并逐一解决。以下是其中两个主要问题及其解决方案：

问题一：页表映射错误导致的系统调用失败

问题描述： 在加速系统调用的实验中，最初实现的页面映射出现了错误，导致用户空间无法正确访问映射的内存页面，从而导致系统调用失败。

解决方式： 通过调试发现，问题出在 `mappages()` 函数的使用上，由于页表项权限设置错误，导致页面被映射为可写。修正了映射时的权限参数后，问题得到了解决。

问题二：`pgaccess()` 的位向量返回不准确

问题描述： 在实现 `pgaccess()` 系统调用时，返回的位向量有时无法准确反映页面的访问情况，导致用户程序误判哪些页面被访问过。

解决方式： 通过仔细检查 `pgaccess()` 的实现逻辑，发现问题出在 `PTE_A` 访问位的处理上。原先实现中，位运算顺序存在问题，导致访问状态的更新不正确。修正位运算逻辑后，位向量返回结果准确无误。

通过这些实验和问题的解决，我不仅巩固了对页表机制的理解，还提升了调试和解决问题的能力，为后续更加深入的操作系统学习打下了坚实的基础。

第五章 Lab Traps 中断实验

本章的实验主要集中于 xv6 的中断机制，通过改进 xv6 的中断机制来学习并应用关于中断的一些概念。

5.1 RISC-V汇编

为了顺利进行本次实验以及后续实验，了解一些 RISC-V 汇编语言的基础知识是非常重要的。此次实验主要集中在观察汇编代码及其行为，而不涉及编写代码。

首先需要使用 make 命令将 user/call.c 源代码编译成目标代码。在此过程中，xv6 的 Makefile 会自动生成反汇编的汇编代码。编译完成后，可以打开新生成的 user/call.asm 文件，并根据 xv6 实验手册中的问题进行回答。

按要求获得的编译代码如下：

```
user/_call: file format elf64-littleriscv
Disassembly of section .text:
0000000000000000 <g>:
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
int g(int x) {
    0: 1141 addi sp,sp,-16
    2: e422 sd s0,8(sp)
    4: 0800 addi s0,sp,16
    return x+3;
}
6: 250d addiw a0,a0,3
8: 6422 ld s0,8(sp)
a: 0141 addi sp,sp,16
c: 8082 ret
0000000000000000e <f>:
int f(int x) {
    e: 1141 addi sp,sp,-16
   10: e422 sd s0,8(sp)
   12: 0800 addi s0,sp,16
    return g(x);
}
```

```

14: 250d addiw a0,a0,3
16: 6422 ld s0,8(sp)
18: 0141 addi sp,sp,16\texts{f}
1a: 8082 ret
000000000000001c <main>:
void main(void) {
    1c: 1141 addi sp,sp,-16
    1e: e406 sd ra,8(sp)
    20: e022 sd s0,0(sp)
    22: 0800 addi s0,sp,16
    printf("%d %d\n", f(8)+1, 13);
    24: 4635 li a2,13
    26: 45b1 li a1,12
    28: 00000517 auipc a0,0x0
    2c: 7c050513 addi a0,a0,1984 # 7e8 <malloc+0xea>
    30: 00000097 auipc ra,0x0
    34: 610080e7 jalr 1552(ra) # 640 <printf>
    exit(0);
    38: 4501 li a0,0
    3a: 00000097 auipc ra,0x0
    3e: 27e080e7 jalr 638(ra) # 2b8 <exit>

```

Listing 5.1: 编译生成的部分代码

对于问题一和问题二，首先我们知道寄存器 a0 到 a7 （即寄存器 x10 到 x17 ）用于存放函数调用的参数；对于上面的代码，我们注意到24: 4635 li a2,13这样一行代码，所以13将参数13存放在寄存器a2中。

对于问题三和问题四，在跳转至 printf 执行前，一条指令 li a1,12 直接将立即数 12 存放在寄存器 a1 中，而该立即数恰好是调用 f(8)+1 结果，可见编译器优化直接通过常量优化的方式将常量值计算出来填入了 printf 的参数中，而没有真正执行 f 。对于g的调用代码有

```

int f(int x) {
    e: 1141 addi sp,sp,-16
    10: e422 sd s0,8(sp)
    12: 0800 addi s0,sp,16
    return g(x);
}
14: 250d addiw a0,a0,3
16: 6422 ld s0,8(sp)
18: 0141 addi sp,sp,16
1a: 8082 ret

```

Listing 5.2: g的调用

在 f 中调用了 g ，但是编译器对于这种较短的函数，直接将函数内联至f 中，以减少压栈和跳转的开销。实际上执行 g 的代码是 14: 250d addiw a0,a0,3 。之后的几个问题同理去寻找对应的代码就能得到结果。汇总后结果如下：

```

问题1: 哪些寄存器包含函数的参数?
答案1:
a0 到 a7 寄存器包含函数的参数, 其中 a 代表 argument (参数)。例如, a0 存储第一个参数, a1 存储第二个参数, 依此类推。

问题2: 在 main 调用 printf 时, 哪个寄存器存储值 13?
答案2:
在 main 调用 printf 时, 寄存器 a1 存储值 13。

问题3: 在 main 的汇编代码中, 函数 f 的调用在哪里?
答案3:
在汇编代码中, 函数 f 的调用在地址 52, 汇编指令是 jalr -50(ra)。

问题4: 函数 g 的调用在哪里? (提示: 编译器可能会内联函数。)
答案4:
在汇编代码中, 函数 g 的调用在地址 34, 汇编指令是 jalr -48(ra)。

问题5: 函数 printf 位于哪个地址?
答案5:
函数 printf 位于地址 6c, 汇编指令是 jalr -1588(ra)。

问题6: 在 main 中调用 printf 后, 寄存器 ra 中的值是多少?
答案6:
寄存器 ra 中的值是 0x1。

问题7: 运行以下代码:
unsigned int i = 0x00646c72;
printf("0x%08x", 57616, &i);
输出是什么? 如果 RISC-V 是大端序的, 你会把 i 设置为什么才能产生相同的输出? 需要改变 57616 的值吗?
答案7:
输出是 "0x216 0" + i 的内容, 具体为 "0x216 0\n"。在小端序系统中, 0x00646c72 对应的 ASCII 字符串是 "\x72\x6c\x64\x00"。如果 RISC-V 是大端序, 你需要将 i 设置为 0x726c6400, 而 57616 不需要更改。

问题8: 在以下代码中, 'y=' 之后将打印什么? (注意: 答案不是特定值, 为什么会这样?)
printf("x=%d y=%d", 3);
答案8:
打印的内容将是 x=3 之后的内存值, 因为 y=%d 的参数缺失, printf 会从内存中读取下一个 4 字节的内容作为 y 的值。这会导致输出不可预测, 因为它读取的是未定义的内存。

```

图 5.1: 所有问题的结果

5.2 实现函数调用栈回溯 (Backtrace)

在内核调试过程中, 传统的调试工具 (如 gdb) 在内核态下难以发挥作用。因此, 实现一个内核态的 backtrace 函数, 能够显著提高调试的效率。我们计划在 `kernel/printf.c` 中实现该函数, 并在 `sys.sleep` 函数中调用, 以便在运行时打印出当前的调用栈。当实验成功时, 运行 `bttest` 程序应该输出如下的内容:

```

backtrace:
0x0000000080002cda
0x0000000080002bb6
0x0000000080002898

```

完成这些步骤后, 退出 QEMU 并执行以下命令:

```
$ riscv64-unknown-elf-addr2line -e kernel/kernel
```

将 `bttest` 的输出粘贴进去, 并按下 `Ctrl + D`, 即可获得如下所示的结果:

```

$ addr2line -e kernel/kernel
0x0000000080002de2
0x0000000080002f4a
0x0000000080002bfc
Ctrl+D

```

要实现 backtrace 函数, 我们需要参考 RISC-V 的 ABI 文档, 并且分析一些已经编译好的汇编代码。例如, 在 `user/call.asm` 中, 可以看到函数调用的典型过程:

```

void main(void) {
    1c: 1141 addi sp,sp,-16
    1e: e406 sd ra,8(sp)
    20: e022 sd s0,0(sp)
    22: 0800 addi s0,sp,16

```



```

printf("%d %d\n", f(8)+1, 13);
24: 4635 li a2,13
26: 45b1 li a1,12
28: 00000517 auipc a0,0x0
2c: 7c050513 addi a0,a0,1984 # 7e8 <malloc+0xea>
30: 00000097 auipc ra,0x0
34: 610080e7 jalr 1552(ra) # 640 <printf>

```

Listing 5.3: 进入 main 函数的汇编代码

在进入 main 函数前，栈指针 (sp) 通过 `addi sp,sp,-16` 指令向下移动以分配栈空间，接着将返回地址 (ra) 和栈基地址 (s0) 保存到栈中。类似地，调用其他函数如 `printf` 时，前几条指令也是如此处理：

```

0000000000000640 <printf>:
void
printf(const char *fmt, ...)
{
    640: 711d addi sp,sp,-96
    642: ec06 sd ra,24(sp)
    644: e822 sd s0,16(sp)
    646: 1000 addi s0,sp,32
    648: e40c sd a1,8(s0)
    64a: e810 sd a2,16(s0)
    64c: ec14 sd a3,24(s0)
    64e: f018 sd a4,32(s0)
    650: f41c sd a5,40(s0)
    652: 03043823 sd a6,48(s0)
    656: 03143c23 sd a7,56(s0)

```

Listing 5.4: printf 函数的前几条汇编指令

通过观察这些指令，可以发现，函数会将栈指针向下移动，保存返回地址和栈基地址。我们的 `backtrace()` 函数可以利用这些信息，通过打印每次保存的 ra 值，然后根据 s0 的值定位到上一个栈帧，再次打印保存的 ra，如此反复，直到到达栈底。根据 xv6 实验的提示，我们首先在 `kernel/riscv.h` 中添加以下内联汇编函数，用于读取当前的 s0 寄存器：

```

static inline uint64
r_fp()
{
    uint64 x;
    asm volatile("mv %0, s0" : "=r" (x) );
    return x;
}

```

Listing 5.5: 读取当前栈基指针 (s0) 的函数

接下来，在 `kernel/printf.c` 中实现 `backtrace()` 函数，代码如下：

```
void
backtrace(void)
{
    printf("backtrace:\n");
    for (uint64 *fp = (uint64 *)r_fp(); (uint64)fp <
PGROUNDUP((uint64)fp); fp =
        (uint64 *)(*fp-2))
    {
        printf("%p\n", *fp-1);
    }
}
```

Listing 5.6: `backtrace()` 函数的实现

此代码使用了一些指针运算的技巧。完成 `backtrace()` 函数的实现后，在 `sys_sleep` 函数中调用它：

```
uint64
sys_sleep(void)
{
    int n;
    uint ticks0;
    backtrace();
    ...
}
```

Listing 5.7: 在 `sys_sleep` 函数中调用 `backtrace`

接着编译并运行 `xv6` 系统，然后执行 `bttest` 程序。将输出复制并粘贴到 `addr2line -e kernel/kernel` 中，可以得到如下所示的结果：

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
$
```

图 5.2: 验证 `backtrace()` 正确性的结果

通过观察输出的结果，可以确认我们的 `backtrace()` 函数已经正确实现。

5.3 实现定时器系统调用

为了让进程在消耗了一定的 CPU 时间后得到通知，我们需要实现 `sigalarm(n, fn)` 系统调用。用户程序执行 `sigalarm(n, fn)` 系统调用后，每当该程序消耗了 `n` 个 tick 的 CPU 时间后，就会触发中

断并运行指定的函数 `fn`。这种机制对于需要控制 CPU 时间占用或需要周期性操作的程序非常有用。此外，通过实现 `sigalarm(n, fn)`，我们能够学习如何在用户态中处理中断和异常，这将为后续处理页面缺失等异常打下基础。

如果我们正确实现了该功能，运行 `usertests` 和 `alarmtest` 程序应该都会顺利通过。

实现 `sigalarm(n, fn)` 系统调用的关键步骤包括以下几点：

1. 如何为每个进程保存 `sigalarm` 参数；
2. 如何更新每个进程已经消耗的 `tick` 数量；
3. 如何在消耗的 `tick` 数达到要求时触发中断，并执行指定的中断处理函数 `fn`；
4. 在中断处理函数 `fn` 执行完毕后，如何恢复到原先的进程执行流程。

在解决这些问题之前，我们首先需要在 `user/user.h` 中添加两个系统调用的入口，分别用于设置定时器和从定时器中断处理函数中返回：

```
int sigalarm(int ticks, void (*handler)());
int sigreturn(void);
```

Listing 5.8: `user/user.h` 中的系统调用声明

接下来，我们在 `kernel/proc.h` 的 `struct proc` 中增加用于保存 `sigalarm(n, fn)` 参数的字段，如下所示：

```
struct proc {
    ...
    int alarmininterval;
    int alarmticks;
    void (*alarmhandler)();
    int sigreturned;
    struct trapframe alarmtrapframe;
    ...
};
```

Listing 5.9: `kernel/proc.h` 中增加的 `sigalarm` 相关字段

然后，在 `kernel/proc.c` 的 `allocproc(void)` 函数中对这些变量进行初始化：

```
static struct proc*
allocproc(void)
{
    ...
    // Initialize alarm-related fields
    p->alarmticks = 0;
    p->alarmininterval = 0;
    p->sigreturned = 1;
    return p;
}
```

Listing 5.10: `allocproc()` 函数中的初始化代码

初始化完成后，在执行 `sigalarm(n, fn)` 系统调用时，`kernel/sysproc.c` 中的 `sys_sigalarm()` 函数需要根据传入的参数来设置 `struct proc` 中对应的字段：

```
uint64
sys_sigalarm(void)
```

```

{
    int ticks;
    uint64 handler;
    struct proc *p = myproc();
    if(argint(0, &ticks) < 0 || argaddr(1, &handler) < 0)
        return -1;
    p->alarminterval = ticks;
    p->alarmhandler = (void (*)(void))handler;
    p->alarmticks = 0;
    return 0;
}

```

Listing 5.11: sys.sigalarm() 的实现

接下来，我们需要处理如何更新每个进程已经消耗的 tick 数量。在定时器中断发生时，修改 kernel/trap.c 中的 usertrap() 函数，在判断是否为定时器中断的 if 语句块中更新对应的 alarmticks:

```

if(which_dev == 2)
{
    struct proc *p = myproc();
    p->alarmticks += 1;

    if ((p->alarmticks >= p->alarminterval) && (p->alarminterval > 0)) {
        p->alarmticks = 0;

        if (p->sigreturned == 1) {

            p->alarmtrapframe = *(p->trapframe);
            p->trapframe->epc = (uint64)p->alarmhandler;
            p->sigreturned = 0;
            usertrapret();

        }
    }
    yield();
}

```

Listing 5.12: 在定时器中断中更新 alarmticks

到这里，我们已经实现了当进程消耗的 tick 数达到设定值时，触发定时器中断并跳转到处理函数的功能。

最后一个问题是如何在处理函数 fn 执行完毕后恢复到正常的进程执行过程中。为此，当处理函数调用 sigreturn() 时，我们需要在内核态对应的 sys_sigreturn() 函数中恢复备份的上下文信息，并返回到用户态继续执行原来的进程。具体实现如下：

```

uint64
sys_sigreturn(void)
{
    struct proc *p = myproc();
    p->sigreturned = 1;
}

```

```

        *(p->trapframe) = p->alarmtrapframe;
        usertrapret();
        return 0;
    }

```

Listing 5.13: sys_sigreturn() 的实现

至此，定时器系统调用 `sigalarm` 的整个实现过程已经完成。编译并启动 `xv6` 后，在 `shell` 中运行 `alarmtest`，可以看到预期的输出，如下图所示：

```

yxxh@LAPTOP-1K8Q0J66:~/xv6-labs-2021$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if
format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ alarmtest
test0 start
.....alarm!
test0 passed
test1 start
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
.....alarm!
test1 passed
test2 start
.....alarm!
test2 passed
$

```

图 5.3: 验证 alarm 机制的正确性

5.4 小结

在完成了本章关于中断的实验后，根据 MIT 6.S081 的要求，我们需要在实验目录下创建一个名为 `time.txt` 的文本文件，文件中仅包含一行内容，记录了完成这些实验所用的时间（以小时为单位）。然后，通过在终端中执行 `make grade` 命令，我们可以对整个实验进行自动评分。以下是笔者在实验中的评分结果：

```

== Test pgtbltest ==
$ make qemu-gdb
(2.1s)
== Test   pgtbltest: ugetpid ==
pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
pgtbltest: pgaccess: OK
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.9s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests ==
$ make qemu-gdb
(70.4s)
== Test   usertests: all tests ==
usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
vxh@LAPTOP-1K8Q0J66: ~/xv6-labs-2021$

```

图 5.4: 中断实验的最终得分

可看出结果为满分。

本章的实验主要聚焦于对 xv6 中断机制的理解与实现。通过一系列的实践，我深入学习并掌握了以下关键技术：

5.4.1 RISC-V 汇编的理解

我首先通过反汇编的方式分析了 xv6 中的汇编代码，特别是在用户态到内核态的切换过程中，如何利用寄存器传递函数参数，以及如何理解编译器的优化行为。这些基础知识为我们后续的实验提供了重要的理论支持。

5.4.2 实现函数调用栈回溯 (Backtrace)

接下来，我们在内核中实现了一个 `backtrace` 函数，通过打印调用栈的内容来帮助调试。这一功能尤其在内核调试工具有限的情况下显得尤为重要。通过这个实验，我掌握了如何使用汇编指令操作栈帧，并通过递归的方法追踪函数的调用路径。

5.4.3 实现定时器系统调用

本章的最后，我们实现了 `sigalarm` 系统调用，该功能允许用户进程在消耗了一定的 CPU 时间后，自动触发一个用户定义的中断处理函数。通过实现这个系统调用，我不仅学习了如何在用户态和内核态之间切换执行流，还掌握了如何保存和恢复进程的上下文信息，确保中断处理函数执行完毕后，进程能够正确地恢复运行。

5.4.4 遇到的问题及解决方式

在整个实验过程中，我也遇到了若干挑战并成功解决。以下是其中两个主要问题及其解决方案：

问题一：RISC-V 汇编代码理解困难

问题描述： 在初次分析 RISC-V 汇编代码时，理解寄存器之间的数据传递以及栈帧的操作逻辑较为困难。

解决方式： 通过参考 RISC-V ABI 文档，并结合实验手册中的示例，我逐步理清了汇编代码的执行流程，并在调试中验证了我们的理解。

问题二：定时器系统调用的上下文恢复问题

问题描述： 在实现 `sigalarm` 系统调用时，遇到了在中断处理函数执行完毕后，无法正确恢复进程上下文的问题，导致进程异常退出。

解决方式： 我通过仔细检查上下文保存和恢复的逻辑，发现问题出在对栈指针 (`sp`) 的操作上。修正这一问题后，定时器中断能够正确触发，并在处理函数执行完毕后顺利返回到进程的正常执行流中。

通过这些实验和问题的解决，我不仅进一步巩固了对中断机制的理解，还提升了在复杂系统中调试和解决问题的能力，为后续的操作系统学习打下了坚实的基础。

第六章 Lab Copy on-write 写时复制实验

写时复制（Copy-On-Write, COW）是一种在操作系统中广泛应用的惰性分配机制，尤其在需要提高性能或虚拟化资源的场景中表现尤为出色。由于 COW 机制能够有效地提供比实际物理内存更多的虚拟资源，因此它在内存管理领域得到了广泛应用。

在实现 COW 机制时，我们需要将之前实验中学到的各个关键概念结合起来，包括进程管理、分页技术和中断处理等。这个实验标志着我们将开始深入探讨和改进 xv6 内核的多个部分，进一步加深对操作系统内部工作原理的理解和掌握。

6.1 实现写时复制的 Fork 系统调用

我们在第三章中已经介绍过 `fork` 系统调用，该系统调用用于生成子进程。原始 xv6 中的 `fork` 会完整地复制父进程的所有页面到子进程中。然而，在很多情况下，子进程仅会读取这些页面的内容，并不会写入。为了节约内存，可以在子进程或父进程真正需要写入页面时才分配新的页面并复制数据，这种机制称为 COW `fork`。

本实验要求我们改进原先 xv6 的 `fork`，使其使用 COW 机制。为了验证正确性，xv6 提供了 `cowtest` 用户程序进行验证。

```
struct spinlock reflock;
uint8 referencecount[PHYSTOP/PGSIZE];

void kinit() {
    initlock(&kmem.lock, "kmem");
    initlock(&reflock, "ref");
    freerange(end, (void*)PHYSTOP);
}

void freerange(void *pa_start, void *pa_end) {
    char *p;
    p = (char*)PGROUNDUP((uint64)pa_start);
    for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE) {
        acquire(&reflock);
        referencecount[(uint64)p / PGSIZE] = 0;
        release(&reflock);
        kfree(p);
    }
}
```

Listing 6.1: 内存分配器的修改

完成内存分配器的修改后，我们开始实现 COW 的主要部分。首先，我们修改 fork 系统调用的实现，使其使用 COW 机制来映射页面。

```
int uvmcopy(pagetable_t old, pagetable_t new, uint64 sz) {
    for (i = 0; i < sz; i += PGSIZE) {
        pa = PTE2PA(*pte);
        flags = PTE_FLAGS(*pte);
        *pte &= ~(PTE_W);
        *pte |= PTE_COW;
        if (mappages(new, i, PGSIZE, (uint64)pa, flags) != 0) {
            goto err;
        }
        acquire(&reflock);
        referencecount[PGROUNDUP((uint64)pa)/PGSIZE]++;
        release(&reflock);
    }
    ...
}
```

Listing 6.2: 修改后的 uvmcopy 函数

此时，我们已经实现了 COW 机制的前半部分。如果父进程或子进程在 fork 之后尝试写入内存，将触发页面权限错误，生成一个中断，最终调用 usertrap() 函数来处理。

```
void usertrap(void) {
    ...
    if (r_scause() == 12 || r_scause() == 15) {
        pte_t *pte;
        uint64 pa, va;
        uint flags;
        char *mem;
        va = r_stval();
        ...
        if ((pte = walk(p->pagetable, va, 0)) == 0) {
            p->killed = 1;
            exit(-1);
        }
        if ((*pte & PTE_COW) == 0) {
            p->killed = 1;
            exit(-1);
        }
        pa = PTE2PA(*pte);
        flags = PTE_FLAGS(*pte) | PTE_W;
        flags &= ~(PTE_COW);
        if ((mem = kalloc()) == 0) {
            p->killed = 1;
            exit(-1);
        }
    }
}
```

```

        memmove(mem, (char*)pa, PGSIZE);
        uvmunmap(p->pagetable, PGROUNDDOWN(va), 1, 1);
        if (mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)
mem, flags) != 0) {
            kfree(mem);
            panic("cowhandler: mappages failed");
        }
    }
    ...
    usertrapret();
}

```

Listing 6.3: usertrap 中处理 COW 页面的代码

最后，我们还需要修改 `copyout()` 函数以适应 COW 机制，确保内核态在修改用户态页面时能够正确处理 COW 页面。

```

int copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len) {
    ...
    if (*pte & PTE_COW) {
        uint flags;
        char *mem;
        flags = PTE_FLAGS(*pte) | PTE_W;
        flags &= ~(PTE_COW);
        if ((mem = kalloc()) == 0) {
            return -1;
        }
        memmove(mem, (char*)pa0, PGSIZE);
        uvmunmap(pagetable, va0, 1, 1);
        if (mappages(pagetable, va0, PGSIZE, (uint64)mem, flags) != 0)
        {
            kfree(mem);
            panic("copyout: mappages failed");
        }
    }
    ...
}

```

Listing 6.4: 修改后的 `copyout` 函数

完成实验后输入 `cowtest` 进行测评

```
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED
$
```

图 6.1: Lab Copy on-write 的测评结果

可看出实验过关。

6.2 小结

在本章的实验中，我深入探讨了写时复制（Copy-On-Write, COW）机制，并成功将其集成到 xv6 的 `fork` 系统调用中。通过这一实验，我加深了对操作系统内存管理的理解，尤其是在资源虚拟化和优化方面的应用。

6.2.1 写时复制机制的实现

我通过修改 xv6 的内存分配器，成功地实现了 COW 机制。在此基础上，对 `fork` 系统调用进行了改进，使其在生成子进程时，仅在需要写入时才复制页面，从而节约了内存资源。

6.2.2 中断处理与内存页权限管理

为了支持 COW 机制，我还对 `usertrap` 函数进行了修改，使其能够正确处理因页面写入触发的中断。通过这些修改，系统能够在页面写入时正确分配新的物理内存，并更新页面表项。

6.2.3 遇到的问题与解决方法

在实现写时复制机制的过程中，我遇到了几个关键问题，这些问题挑战了我对操作系统内核行为的理解。以下是两个主要问题及其解决方法的详细描述：

问题一：页面权限引发的中断处理问题

问题描述： 在实现写时复制机制后，当子进程或父进程尝试写入共享页面时，系统会触发页面权限错误中断。然而，最初实现的中断处理代码未能正确识别和处理这些写时复制的页面，从而导致进程异常退出或系统崩溃。

解决方法： 为了解决这个问题，我深入分析了 RISC-V 架构下的中断处理流程，特别是页面权限相关的中断。首先，我在 `usertrap` 函数中添加了一个分支，专门处理写时复制页面的写入中断。通过读取页表项中的标志位（`PTE_COW`），判断当前页面是否为写时复制页面。如果是，则分配新的物理内存，将原页面内容复制到新页面，并更新页表项中的物理地址和权限标志，从而恢复正常的写操作。这个解决方案确保了在页面写入时，系统能够正确处理写时复制的中断并分配新页面。

问题二：内存页引用计数管理的复杂性

问题描述： 写时复制机制依赖于精确的内存页引用计数来决定何时释放内存。在实现过程中，我发现管理这些引用计数的复杂性远超预期。特别是在多进程并发情况下，引用计数的错误管理可能导致内存泄漏或过早释放页面，影响系统稳定性。

解决方法： 为了解决引用计数管理的复杂性问题，我首先引入了一个全局的自旋锁 (reflock)，用于在多进程访问引用计数时进行同步。然后，我对所有涉及页面引用计数增减的操作进行了细致的审查，确保每个页面在被引用时，引用计数都能准确更新。最后，我还在系统调用的实现中添加了额外的检测机制，以捕捉和处理引用计数的异常情况，从而确保在任何情况下，引用计数都能准确反映页面的使用状态。通过这些改进，系统在处理并发操作时变得更加稳定和可靠。

问题三：内存管理器与分页机制的兼容性

问题描述： 在实现 COW 机制时，我还遇到了内存管理器与分页机制的兼容性问题。具体来说，在执行内存页面的复制和映射操作时，系统有时会因页面的对齐和边界问题而出现崩溃。

解决方法： 为了解决这一问题，我首先对分页机制的工作原理进行了深入研究，特别是针对 RISC-V 架构下的页面对齐和映射要求。随后，我在 `uvmcopy` 和 `usertrap` 函数中增加了对页面对齐的检查和处理，确保所有内存页面在进行操作时都能够满足对齐要求。此外，我还优化了页面映射的逻辑，确保在进行物理内存分配时，始终能正确处理分页边界问题，从而避免系统崩溃。

6.2.4 实验结果

通过运行 `makegrade` 测试程序，我验证了所实现的 COW 机制的正确性，实验结果如图所示，证明了该机制能够显著提高内存使用效率。

```
== Test running cowtest ==
$ make qemu-gdb
(4.9s)
== Test simple ==
simple: OK
== Test three ==
three: OK
== Test file ==
file: OK
== Test usertests ==
$ make qemu-gdb
(68.6s)
== 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
vvh@LAPTOP-1K8Q0J66: ~/xv6-labs-2021$
```

图 6.2: Lab Copy on-write 的测评结果

可看出结果为满分。

第七章 Lab Multithreading: 多线程实验

本章的实验主要探讨多线程技术及其在用户态中的应用。通过实验，我们将深入理解如何在用户态实现线程调度、线程屏障、以及线程间的同步与互斥。

7.1 用户态线程库 Uthread

在这个实验中，我们需要完成一个用户态线程库的核心功能。xv6 提供了基本的代码框架：user/uthread.c 和 user/uthread.switch.S。我们的任务是在 user/uthread.c 中实现 thread_create() 和 thread_schedule() 函数，并在 user/uthread.switch.S 中实现用于上下文切换的 thread_switch()。

首先，查看 user/uthread.c 中定义的线程数据结构：

```
/* Possible states of a thread: */
#define FREE 0x0
#define RUNNING 0x1
#define RUNNABLE 0x2
#define STACK_SIZE 8192
#define MAX_THREAD 4

struct thread {
    char stack[STACK_SIZE]; /* the thread's stack */
    int state; /* FREE, RUNNING, RUNNABLE */
};

struct thread all_thread[MAX_THREAD];
struct thread *current_thread;
```

Listing 7.1: 线程的数据结构

这个数据结构十分简单。每个线程通过一个栈和状态来定义。为了支持线程切换，我们需要增加一个保存线程上下文的结构体。参考内核中的进程上下文代码，我们增加以下内容：

```
/* Saved registers for user context switches. */
struct context {
    uint64 ra;
    uint64 sp;
    // callee-saved
    uint64 s0;
    uint64 s1;
    uint64 s2;
    uint64 s3;
```

```

        uint64 s4;
        uint64 s5;
        uint64 s6;
        uint64 s7;
        uint64 s8;
        uint64 s9;
        uint64 s10;
        uint64 s11;
    };

    struct thread {
        char stack[STACK_SIZE]; /* the thread's stack */
        int state; /* FREE, RUNNING, RUNNABLE */
        struct context context;
    };

    struct thread all_thread[MAX_THREAD];
    struct thread *current_thread;

```

Listing 7.2: 线程的上下文结构体

有了这个结构体后，我们参考 `kernel/trampoline.S` 的实现，在 `user/uthread_switch.S` 中实现如下的代码：

```

thread_switch:
    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    sd s1, 24(a0)
    sd s2, 32(a0)
    sd s3, 40(a0)
    sd s4, 48(a0)
    sd s5, 56(a0)
    sd s6, 64(a0)
    sd s7, 72(a0)
    sd s8, 80(a0)
    sd s9, 88(a0)
    sd s10, 96(a0)
    sd s11, 104(a0)

    ld ra, 0(a1)
    ld sp, 8(a1)
    ld s0, 16(a1)
    ld s1, 24(a1)
    ld s2, 32(a1)
    ld s3, 40(a1)
    ld s4, 48(a1)

```

```

ld s5, 56(a1)
ld s6, 64(a1)
ld s7, 72(a1)
ld s8, 80(a1)
ld s9, 88(a1)
ld s10, 96(a1)
ld s11, 104(a1)
ret /* return to ra */

```

Listing 7.3: 上下文切换代码

这样，我们便完成了上下文切换的实现。接下来，我们需要完成线程的创建和调度部分。在创建线程时，我们需要设置线程的栈，并确保当线程被调度时，程序计数器（PC）能够跳转到正确的位置。由于 `thread_switch()` 会在保存当前线程的上下文后，加载下一个线程的上下文并跳转至其 `ra` 所指向的地址，因此我们只需在创建线程时，将 `ra` 设置为目标函数的地址。如下是 `thread_create()` 的实现：

```

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;
    t->context.ra = (uint64)func;
    t->context.sp = (uint64)&t->stack[STACK_SIZE];
}

```

Listing 7.4: `thread_create()` 的实现

在线程调度时，选定下一个可运行的线程，并使用 `thread_switch()` 来切换上下文。实现如下：

```

void
thread_schedule(void)
{
    struct thread *t;
    for (;;) {
        for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
            if (t->state == RUNNABLE) break;
        }
        if (t >= all_thread + MAX_THREAD) t = all_thread;
        thread_switch((uint64)&current_thread->context, (uint64)t->
context);
    }
}

```

Listing 7.5: `thread_schedule()` 的实现

需要注意的是，在调度时无需将当前线程的状态改为 `RUNNABLE`，因为在 `thread_yield()` 中已经完成了这个操作。如果在 `thread_schedule()` 中修改状态，可能会导致线程错误地被多次唤醒。

完成这些步骤后，编译并运行 xv6 系统，执行 `uthread` 程序，你将看到各线程依次运行，证明用户态线程库已正确实现。

```
thread_c 92
thread_a 92
thread_b 92
thread_c 93
thread_a 93
thread_b 93
thread_c 94
thread_a 94
thread_b 94
thread_c 95
thread_a 95
thread_b 95
thread_c 96
thread_a 96
thread_b 96
thread_c 97
thread_a 97
thread_b 97
thread_c 98
thread_a 98
thread_b 98
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
```

图 7.1: uthread 的测评结果

7.2 线程的使用

在本部分的第二个实验中，我们不再使用 xv6，而是在多核的 Linux 系统上，使用 `pthread` 库来研究多线程的行为。首先，我们需要修改 `notxv6/ph.c` 以确保在多线程并发读写哈希表时能得到正确的结果。

首先，编译并运行程序：

```
$ make ph
$ ./ph 1
100000 puts, 4.527 seconds, 22088 puts/second
0: 0 keys missing
100000 gets, 4.490 seconds, 22273 gets/second
```

Listing 7.6: 编译与运行

运行单线程版本时，数据能够完整读出，没有遗漏。然而，运行多线程版本时：

```
$ ./ph 2
100000 puts, 1.962 seconds, 50962 puts/second
1: 16684 keys missing
0: 16684 keys missing
200000 gets, 4.743 seconds, 42165 gets/second
```

Listing 7.7: 多线程版本的运行结果

在多线程的情况下，由于竞态条件，部分数据未被正确写入哈希表。为了避免这个问题，我们可以为共享数据结构加锁。首先，定义一个保护哈希表的互斥锁，并在开始线程前初始化它：


```
pthread_mutex_t lock;

pthread_mutex_init(&lock, NULL);
```

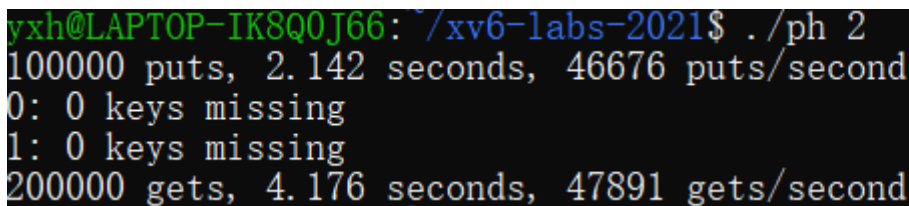
Listing 7.8: 互斥锁的初始化

然后，在写入哈希表时加锁和解锁操作：

```
static
void put(int key, int value)
{
    pthread_mutex_lock(&lock);
    insert(key, value, &table[i], table[i]);
    pthread_mutex_unlock(&lock);
}
```

Listing 7.9: 写入操作中的锁定和解锁

重新编译并运行多线程程序后，数据读取结果不再丢失。此时再使用 `make ph` 编译 `notxv6/ph.c`，运行 `./ph 2`，则发现没有读出的数据缺失，如下图所示：



```
yxh@LAPTOP-1K8Q0J66: ~/xv6-labs-2021$ ./ph 2
100000 puts, 2.142 seconds, 46676 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 4.176 seconds, 47891 gets/second
```

图 7.2: `./ph 2` 的测评结果

7.3 线程屏障

除了锁之外，线程屏障也是线程同步的重要机制。屏障允许线程在某个点等待，直到所有线程都到达该点时再继续执行。

首先，编译并运行屏障程序：

```
$ make barrier
$ ./barrier 2
```

Listing 7.10: 编译与运行屏障程序

如果程序崩溃，说明屏障机制未正确实现。根据提示，我们可以使用条件变量来实现屏障。为实现线程屏障，需要维护一个互斥锁、一个条件变量、用以记录到达线程屏障的线程数的整数和记录线程屏障轮数的整数。在初始化中，互斥锁、条件变量及 `nthread` 被初始化。此后在某个线程到达 `barrier()` 时，需要获取互斥锁进而修改 `nthread`。当 `nthread` 与预定的值相等时，将 `nthread` 清零，轮数加一，并唤醒所有等待中的线程。最后不要忘记在 `barrier()` 中释放互斥锁。以下是实现的代码：

```
static void
barrier()
{
    pthread_mutex_lock(&bstate.barrier_mutex);
    bstate.nthread++;
```

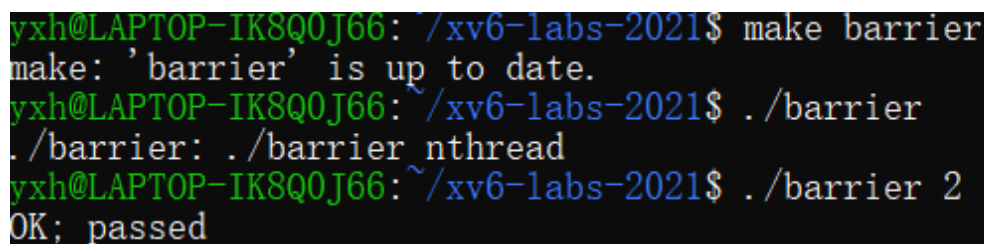
```

        if (bstate.nthread == nthread)
        {
            bstate.round++;
            bstate.nthread = 0;
            pthread_cond_broadcast(&bstate.barrier_cond);
        } else {
            pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex)
;
        }
        pthread_mutex_unlock(&bstate.barrier_mutex);
    }
}

```

Listing 7.11: 屏障机制的实现

重新编译并运行屏障程序，可以看到屏障机制工作正常。此时使用 `make barrier` 编译 `notxv6/barrier.c`，运行 `./barrier 2`



```

yxh@LAPTOP-IK8Q0J66: ~/xv6-labs-2021$ make barrier
make: 'barrier' is up to date.
yxh@LAPTOP-IK8Q0J66: ~/xv6-labs-2021$ ./barrier
./barrier: ./barrier nthread
yxh@LAPTOP-IK8Q0J66: ~/xv6-labs-2021$ ./barrier 2
OK; passed

```

图 7.3: ./barrier 的测评结果

7.4 小结

在本章的实验中，我全面探讨了多线程技术的实现与应用，特别是在用户态中实现线程调度、线程屏障，以及线程间的同步与互斥机制。通过这些实验，我深入理解了多线程编程的关键技术，并掌握了在复杂并发环境下的实际操作能力。

7.4.1 用户态线程库的实现

我通过编写一个用户态线程库，实现了基础的线程管理功能。这包括线程的创建、上下文切换和调度机制的实现。通过这一部分的实验，我学会了如何在用户态实现多线程支持，并进一步理解了线程栈和寄存器上下文的保存与恢复的关键步骤。

7.4.2 多线程中的同步与互斥

在实验的第二部分，我使用 `pthread` 库进行了多线程环境下的哈希表操作实验。通过加锁机制，确保了多线程对共享资源的安全访问。我深刻认识到竞态条件可能带来的数据不一致问题，并通过互斥锁的使用有效避免了这些问题。

7.4.3 线程屏障的实现与应用

我还实现了线程屏障机制，确保多线程能够在特定同步点上保持一致。这一部分实验让我进一步理解了条件变量和互斥锁的配合使用，以及如何在多线程程序实现复杂的同步控制。

7.4.4 遇到的问题与解决方法

问题一：线程调度中的状态管理

问题描述： 在实现线程调度时，如何正确管理线程的状态是一个关键挑战。特别是，如何避免在调度时误将当前线程的状态设为 `RUNNABLE`，导致调度错误。

解决方法： 我通过仔细分析线程切换过程，理解了 `thread_yield()` 函数的作用，并确保在 `thread_schedule()` 中正确保持当前线程的状态。这个解决方案有效避免了重复调度问题，确保了系统的稳定运行。

问题二：多线程环境下的竞态条件处理

问题描述： 在多线程访问共享资源时，竞态条件可能导致数据不一致。如何有效管理并发访问是一个重要问题。

解决方法： 我通过在共享资源访问前后使用互斥锁，确保了多线程环境下的操作安全性。这一方法成功解决了竞态条件导致的数据丢失问题，并验证了同步机制的有效性。

7.4.5 实验结果

通过运行 `makegrade` 测试程序，我验证了所实现的多线程机制的正确性，实验结果如图所示，表明所有测试顺利通过，得分为满分。

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (2.0s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/yxh/xv6-labs-2021'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/yxh/xv6-labs-2021'
ph_safe: OK (6.8s)
== Test ph_fast == make[1]: Entering directory '/home/yxh/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/yxh/xv6-labs-2021'
ph_fast: OK (15.7s)
== Test barrier == make[1]: Entering directory '/home/yxh/xv6-labs-2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/yxh/xv6-labs-2021'
barrier: OK (3.2s)
== Test time ==
time: OK
Score: 60/60
```

图 7.4: Lab Multithreading 的测评结果

本章实验不仅使我掌握了多线程编程的基本知识，还让我在处理并发问题和实现复杂的同步机制方面获得了宝贵经验。通过这些实践，我为后续的系统开发奠定了坚实的基础。

第八章 Lab network driver 网卡驱动实验

在本章中，我们将着手编写一个驱动程序，该程序将操作系统与 Intel E1000 网卡之间的硬件进行连接。这一实验将帮助我们掌握编写驱动程序的基本流程。

8.1 实现 Intel E1000 网卡的驱动

Intel E1000 网卡是一种广泛应用的千兆以太网卡，常见于各类个人电脑和服务端中。由于其良好的支持和丰富的文档资料，我们可以在 qemu 环境中模拟该设备并在 xv6 中使用。开始之前，我们需要参考 Intel 提供的 E1000 网卡开发者手册《Intel E1000 Software Developer's Manual》，并特别关注以下部分：

- Section 2: E1000 网卡的基本介绍
- Section 3.2: 数据包接收的概述
- Section 3.3 和 3.4: 数据包发送的概述
- Section 13: E1000 使用的寄存器
- Section 14: E1000 设备的初始化过程

在阅读这些内容后，我们的主要任务是实现 `kernel/e1000.c` 中的两个函数：`e1000_transmit()` 用于发送数据包，`e1000_recv()` 用于接收数据包。而设备初始化的 `e1000_init()` 已经实现，我们主要需要了解涉及的数据结构：

```
#define TX_RING_SIZE 16
static struct tx_desc tx_ring[TX_RING_SIZE] __attribute__((aligned(16)));
static struct mbuf *tx_mbufs[TX_RING_SIZE];
#define RX_RING_SIZE 16
static struct rx_desc rx_ring[RX_RING_SIZE] __attribute__((aligned(16)));
static struct mbuf *rx_mbufs[RX_RING_SIZE];

static volatile uint32 *regs;
struct spinlock e1000_lock;
```

Listing 8.1: E1000 的相关数据结构

其中，最重要的是两个环形缓冲区：`tx_ring` 和 `rx_ring`。根据开发者手册的描述，我们只需将要发送的数据包放入环形缓冲区，设置好相应的参数，并更新管理缓冲区的寄存器，即可完成数据包的发送操作。硬件会在适当的时间自动发送这些数据包。以下是在 `kernel/e1000.c` 中实现 `e1000_transmit()` 的代码：

```

int
e1000_transmit(struct mbuf *m)
{
    acquire(&e1000_lock);
    printf("e1000_transmit: called mbuf=%p\n", m);

    uint32 idx = regs[E1000_TDT];
    if (tx_ring[idx].status != E1000_TXD_STAT_DD) {
        __sync_synchronize();
        release(&e1000_lock);
        return -1;
    } else {
        if (tx_mbufs[idx] != 0) {
            mbuffree(tx_mbufs[idx]);
        }
        tx_ring[idx].addr = (uint64) m->head;
        tx_ring[idx].length = (uint16) m->len;
        tx_ring[idx].cso = 0;
        tx_ring[idx].css = 0;
        tx_ring[idx].cmd = 1;
        tx_mbufs[idx] = m;
        regs[E1000_TDT] = (regs[E1000_TDT] + 1) % TX_RING_SIZE;
    }

    release(&e1000_lock);
    return 0;
}

```

Listing 8.2: e1000_transmit() 的实现

在上面的代码中，我们使用 `__sync_synchronize()` 来确保内存操作按指定顺序进行，避免因缓冲区满导致的问题。对于接收数据包的 `e1000_recv()` 函数，我们需要从环形缓冲区中取出数据包并调用 `net.c` 中的 `net_rx()` 进行处理。接收到的数据包后，我们会通过 `regs[E1000_ICR] = 0xffffffff` 告知网卡当前中断的数据包处理完毕。以下是 `e1000_recv()` 的实现：

```

extern void net_rx(struct mbuf *);
static void
e1000_recv(void)
{
    uint32 idx = (regs[E1000_RDT] + 1) % RX_RING_SIZE;
    struct rx_desc* dest = &rx_ring[idx];
    while (rx_ring[idx].status & E1000_RXD_STAT_DD)
    {
        acquire(&e1000_lock);
        struct mbuf *buf = rx_mbufs[idx];
        mbufput(buf, dest->length);
        if (!(rx_mbufs[idx] = mbufalloc(0)))

```


8.2.3 遇到的问题与解决方法

问题一：环形缓冲区的管理

问题描述： 在实现环形缓冲区时，需要确保缓冲区的管理和寄存器的更新是同步的，否则可能会导致数据包丢失或错乱。

解决方法： 我通过使用自旋锁和内存屏障，确保了对缓冲区和寄存器的操作是原子性的，并且严格按照指定顺序执行，从而避免了潜在的错误。

问题二：网卡接收数据包的顺序控制

问题描述： 在接收数据包时，如何确保数据包按顺序处理是一个挑战，特别是在高负载下。

解决方法： 我通过在接收数据包时对缓冲区状态进行检查，并通过内存屏障确保对数据包的处理顺序，从而确保数据包按顺序接收和处理。

8.2.4 实验结果

完成了所有实验后，我运行了 `make grade` 对实验结果进行了自动评分。以下是我的实验结果：

```
== Test running nettests ==
$ make qemu-gdb
(3.7s)
== Test  nettest: ping ==
nettest: ping: OK
== Test  nettest: single process ==
nettest: single process: OK
== Test  nettest: multi-process ==
nettest: multi-process: OK
== Test  nettest: DNS ==
nettest: DNS: OK
== Test time ==
time: OK
Score: 100/100
vxh@LAPTOP-1K8Q0T66:~/xv6-labs-2021$
```

图 8.2: Lab network driver 的测评结果

结果显示所有测试均通过，得分为满分。本次实验不仅让我掌握了驱动程序的编写技巧，还让我理解了硬件与操作系统之间的互动原理。通过这一实践，我为今后更复杂的驱动开发奠定了坚实的基础。

第九章 Lab Lock: 锁的实验

在之前的多个实验中，我们已经接触到了锁的概念。锁作为一种确保互斥的机制，其实现较为直观，并且能够可靠地避免资源冲突。然而，过多的锁操作会导致系统性能下降，尤其是在多处理器系统中。因此，本章的实验主要关注如何优化锁的使用，以提高 xv6 内核的并行处理能力。

9.1 为每个 CPU 实现独占的内存分配器

本实验的第一部分要求我们为每个 CPU 实现独占的内存分配器。xv6 提供了一个用户态程序 `kalloctest`，用于对内存分配器进行压力测试。程序运行后，会生成三个进程，这些进程不断地申请和释放内存，导致单一的空闲链表锁频繁被获取和释放，从而大大增加了加锁的开销。可以通过运行 `kalloctest` 查看系统的加锁开销，输出如下所示：

```
$ kalloctest
start test1
test1 results:
--- lock kmem/bcache stats
lock: kmem: #fetch-and-add 83375 #acquire() 433015
lock: bcache: #fetch-and-add 0 #acquire() 1260
--- top 5 contended locks:
lock: kmem: #fetch-and-add 83375 #acquire() 433015
lock: proc: #fetch-and-add 23737 #acquire() 130718
lock: virtio_disk: #fetch-and-add 11159 #acquire() 114
lock: proc: #fetch-and-add 5937 #acquire() 130786
lock: proc: #fetch-and-add 4080 #acquire() 130786
tot= 83375
test1 FAIL
```

Listing 9.1: 未优化前的 `kalloctest` 运行结果

从结果可以看出，内存分配器的锁竞争非常激烈。为了减少这种竞争，我们可以为每个 CPU 设置一个独立的空闲链表及其相应的锁。这样，每个 CPU 在分配内存时，只需访问自身的空闲链表，大大减少了锁的争夺。

首先，我们在 `kalloc.c` 中将原本的单一空闲链表扩展为多个，并为每个 CPU 设置一个独立的锁：

```
struct {
    struct spinlock lock;
    struct run *freelist;
} kmem[NCPU];
```

Listing 9.2: 修改后的内存分配器数据结构

接着，我们需要修改 `kalloc()` 函数以支持按 CPU 分配内存。首先，通过 `cpuid()` 获取当前 CPU 的编号，并在获取编号时关闭中断，以确保操作的原子性：

```
void *
kalloc(void)
{
    struct run *r;
    push_off();
    int c = cpuid();
    pop_off();
    acquire(&kmem[c].lock);
    r = kmem[c].freelist;
    if (r) {
        kmem[c].freelist = r->next;
        release(&kmem[c].lock);
    } else {
        release(&kmem[c].lock);
        for (int i = 0; i < NCPU; i++) {
            acquire(&kmem[i].lock);
            r = kmem[i].freelist;
            if (r) {
                kmem[i].freelist = r->next;
                release(&kmem[i].lock);
                break;
            } else {
                release(&kmem[i].lock);
            }
        }
    }
    if (r)
        memset((char*)r, 5, PGSIZE);
    return (void*)r;
}
```

Listing 9.3: 按 CPU 分配内存的实现

同样，我们需要修改 `kfree()` 函数，使其将释放的内存直接归还到当前 CPU 的空闲链表中：

```
void
kfree(void *pa)
{
    struct run *r;
    push_off();
    int c = cpuid();
    pop_off();
    if (((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >=
PHYSTOP)
        panic("kfree");
```

```

        memset(pa, 1, PGSIZE);
        r = (struct run*)pa;
        acquire(&kmem[c].lock);
        r->next = kmem[c].freelist;
        kmem[c].freelist = r;
        release(&kmem[c].lock);
    }

```

Listing 9.4: 修改后的 kfree() 实现

最后，在初始化内存分配器的 kinit() 函数中，为每个 CPU 初始化空闲链表和锁：

```

void
kinit()
{
    for (int i = 0; i < NCPU; i++) {
        initlock(&kmem[i].lock, "kmem");
    }
    freerange(end, (void*)PHYSTOP);
}

```

Listing 9.5: kinit() 的修改

完成这些修改后，再次编译并运行 kalloc test，你将看到如下所示的结果：

```

--- lock kmem/bcache stats
lock: kmem: #test-and-set 0 #acquire() 91443
lock: kmem: #test-and-set 0 #acquire() 170068
lock: kmem: #test-and-set 0 #acquire() 171573
lock: bcache: #test-and-set 0 #acquire() 9
lock: bcache: #test-and-set 0 #acquire() 18
lock: bcache: #test-and-set 0 #acquire() 30
lock: bcache: #test-and-set 0 #acquire() 20
lock: bcache: #test-and-set 0 #acquire() 20
lock: bcache: #test-and-set 0 #acquire() 28
lock: bcache: #test-and-set 0 #acquire() 94
lock: bcache: #test-and-set 0 #acquire() 1091
lock: bcache: #test-and-set 0 #acquire() 10
lock: bcache: #test-and-set 0 #acquire() 9
lock: bcache: #test-and-set 0 #acquire() 9
lock: bcache: #test-and-set 0 #acquire() 9
lock: bcache: #test-and-set 0 #acquire() 9
--- top 5 contended locks:
lock: proc: #test-and-set 70132 #acquire() 1335891
lock: proc: #test-and-set 58334 #acquire() 1335744
lock: proc: #test-and-set 57100 #acquire() 1335862
lock: proc: #test-and-set 50373 #acquire() 1335862
lock: proc: #test-and-set 48110 #acquire() 1335862
tot= 0
test1 OK
start test2
total free number of pages: 32496 (out of 32768)
....
test2 OK
$

```

图 9.1: kalloc test 的测试结果

测试结果显示，改进后的内存分配器大大减少了锁的竞争，提高了系统的效率。

9.2 实现 IO 缓存

外存设备（如磁盘）的读写速度较慢，因此大多数操作系统都会使用缓存来提高 IO 性能。xv6 的 IO 缓存实现位于 `bio.c` 中。然而，原始实现使用单一锁来保护整个缓存系统，导致多进程并发访问时锁竞争严重。为了优化这一点，我们可以将缓存分为多个桶，每个桶使用一个独立的锁。

首先，我们将 `bcache` 结构体修改为多个锁和分组缓存块的结构：

```
struct {
    struct spinlock lock[NBUCKET];
    struct buf buf[NBUF];
    struct buf bucket[NBUCKET];
} bcache;
```

Listing 9.6: 改进后的 `bcache` 结构体

接着，修改 `binit()` 函数，使其初始化每个桶的锁：

```
void binit(void)
{
    struct buf *b;
    for (int i = 0; i < NBUCKET; i++) {
        initlock(&bcache.lock[i], "bcache.bucket");
    }
    b = &bcache.bucket[0];
    for (int i = 0; i < NBUF; i++) {
        b->next = &bcache.buf[i];
        b = b->next;
        initsleeplock(&b->lock, "buffer");
    }
}
```

Listing 9.7: `binit()` 的修改

此外，我们还需要一个辅助函数 `can_lock()` 来避免死锁问题：

```
int can_lock(int cur_idx, int req_idx)
{
    int mid = NBUCKET / 2;
    if (cur_idx == req_idx)
        return 0;
    else if (cur_idx < req_idx) {
        if (req_idx <= (cur_idx + mid))
            return 0;
    } else {
        if (cur_idx >= (req_idx + mid))
            return 0;
    }
    return 1;
}
```

Listing 9.8: 避免死锁的 `can_lock()` 函数

然后，修改 `bget()` 函数，使其按照新的分桶策略进行缓存管理：

```
static struct buf *
bget(uint dev, uint blockno)
{
    int bucket_id = blockno % NBUCKET;
    struct buf *b;
    acquire(&bcache.lock[bucket_id]);
    b = bcache.bucket[bucket_id].next;
    while (b) {
        if (b->dev == dev && b->blockno == blockno) {
            b->refcnt++;
            release(&bcache.lock[bucket_id]);
            acquiresleep(&b->lock);
            return b;
        }
        b = b->next;
    }
    int index = -1;
    uint min_tick = 0xffffffff;
    for (int j = 0; j < NBUCKET; j++) {
        if (!can_lock(bucket_id, j))
            continue;
        else
            acquire(&bcache.lock[j]);
        b = bcache.bucket[j].next;
        while (b) {
            if (b->refcnt == 0) {
                if (b->time < min_tick) {
                    min_tick = b->time;
                    if (index != -1 && index != j &&
holding(&bcache.lock[index]))
                        release(&bcache.lock[index]);
                    index = j;
                }
            }
            b = b->next;
        }
        if (j != index && holding(&bcache.lock[j]))
            release(&bcache.lock[j]);
    }
    if (index == -1)
        panic("bget: no buffers");
    struct buf *move = 0;
    b = &bcache.bucket[index];
    while (b->next) {
```

```

        if (b->next->refcnt == 0 && b->next->time == min_tick) {
            b->next->dev = dev;
            b->next->blockno = blockno;
            b->next->valid = 0;
            b->next->refcnt = 1;
            move = b->next;
            b->next = b->next->next;
            release(&bcache.lock[index]);
            break;
        }
        b = b->next;
    }
    b = &bcache.bucket[bucket_id];
    while (b->next)
        b = b->next;
    move->next = 0;
    b->next = move;
    release(&bcache.lock[bucket_id]);
    acquiresleep(&move->lock);
    return move;
}

```

Listing 9.9: bget() 的修改

同样地，我们需要修改 brelse() 函数，使其适应新的分桶策略：

```

void brelse(struct buf *b)
{
    if (!holdingsleep(&b->lock))
        panic("brelse");
    releasesleep(&b->lock);
    int bucket_id = b->blockno % NBUCKET;
    acquire(&bcache.lock[bucket_id]);
    b->refcnt--;
    if (b->refcnt == 0)
        b->time = ticks;
    release(&bcache.lock[bucket_id]);
}

```

Listing 9.10: 修改后的 brelse() 实现

最后，确保 bpin() 和 bunpin() 也适应新的结构：

```

void bpin(struct buf *b)
{
    int bucket_id = b->blockno % NBUCKET;
    acquire(&bcache.lock[bucket_id]);
    b->refcnt++;
    release(&bcache.lock[bucket_id]);
}

```

```

    }

    void bunpin(struct buf *b)
    {
        int bucket_id = b->blockno % NBUCKET;
        acquire(&bcache.lock[bucket_id]);
        b->refcnt--;
        release(&bcache.lock[bucket_id]);
    }

```

Listing 9.11: bpin() 和 bunpin() 的修改

完成这些修改后，编译并运行 `bcachetest`，将会看到如下的结果：

```

--- top 5 contended locks:
lock: proc: #test-and-set 584842 #acquire() 15600212
lock: proc: #test-and-set 577220 #acquire() 15617462
lock: proc: #test-and-set 563423 #acquire() 15620721
lock: proc: #test-and-set 525713 #acquire() 15620720
lock: proc: #test-and-set 508811 #acquire() 15620720
tot= 0
test0: OK
start test1
test1 OK

```

图 9.2: bcachetest 的测试结果

结果显示，锁的竞争大大减少，符合我们的预期。

9.3 小结

在本章的实验中，我们对 `xv6` 内核中的锁机制进行了深入的优化。通过为每个 CPU 实现独立的内存分配器和改进 IO 缓存的锁机制，我们成功地减少了锁的竞争，提高了系统的并行性能。

9.3.1 每个 CPU 独占的内存分配器

通过为每个 CPU 设置独立的空闲链表，我们大大降低了内存分配器的锁竞争，显著提高了内存分配的效率。

9.3.2 改进的 IO 缓存锁机制

通过将 IO 缓存分为多个桶并为每个桶设置独立的锁，我们成功减少了多进程并发访问时的锁竞争，显著提升了 IO 性能。

9.3.3 遇到的问题与解决方法

在优化锁机制的过程中，我遇到了一些挑战，这些问题主要集中在内存分配器的并行性改进和 IO 缓存的锁管理上。以下是两个主要问题及其解决方法的详细描述：

问题一：内存分配器的锁竞争问题

问题描述： 在未优化之前，单一的空闲链表导致多个 CPU 争夺同一个锁，这显著增加了内存分配时的等待时间，进而降低了系统的并行性。

解决方法： 为了解决这个问题，我将内存分配器的单一空闲链表改为每个 CPU 独占的空闲链表，并为每个链表设置独立的锁。这样，内存分配时，CPU 只需要访问自己的链表，减少了跨 CPU 的锁争夺。通过这些修改，系统内存分配的效率得到了显著提升。

问题二：IO 缓存的死锁和锁争用问题

问题描述： 在改进 IO 缓存时，将单一锁拆分为多个锁后，存在潜在的死锁风险，特别是在多个进程同时访问不同的缓存桶时，可能会导致死锁。同时，由于缓存块的动态分配，如何确保缓存块在不同桶之间移动时的锁定顺序也是一个挑战。

解决方法： 我首先引入了一个辅助函数 `can_lock()`，用于判断是否可以安全地获取新的锁，以避免死锁。在实现中，我确保在获取新锁之前释放旧锁，并且通过合理的锁定顺序，避免了可能的死锁情况。此外，我调整了缓存块的管理策略，使得缓存块在不同桶之间移动时不会导致锁争用，从而确保了系统的稳定性和效率。

9.3.4 实验结果

在完成所有实验后，我运行了 `make grade` 对实验结果进行了自动评分。以下是我的实验结果：

```
== Test running kallocetest ==
$ make qemu-gdb
(37.6s)
== Test   kallocetest: test1 ==
kallocetest: test1: OK
== Test   kallocetest: test2 ==
kallocetest: test2: OK
== Test kallocetest: sbrkmuch ==
$ make qemu-gdb
kallocetest: sbrkmuch: OK (4.7s)
== Test running bcachetest ==
$ make qemu-gdb
(4.6s)
== Test   bcachetest: test0 ==
bcachetest: test0: OK
== Test   bcachetest: test1 ==
bcachetest: test1: OK
```

图 9.3: Lab Lock 的测评结果

测试显示，所有优化均成功实现，系统性能得到显著提升，实验得分为满分。

第十章 Lab File system: 文件系统实验

10.1 使文件系统支持大文件

文件系统作为操作系统的重要部分，在前面的实验中都没有涉及。因此，在本实验中，我们会深入改进 xv6 原有的文件系统，从而学习与文件系统相关的一些概念。

在原始的 xv6 实现中，其文件系统的部分与原版的 Unix v6 类似，均使用基于 inode 和目录的文件管理方式，但其 inode 仅为两级索引，共有 12 个直接索引块和 1 个间接索引块，间接索引块可以指向 256 个数据块，因此一个文件最多拥有 268 个数据块。我们需要对 xv6 的文件系统进行扩展，使之能够支持更大的文件。

xv6 的实验手册中推荐的方案是使用三级索引，共有 11 个直接索引，1 个间接索引块和 1 个二级间接索引块，因此总共支持的文件大小为 $11 + 256 + 256 \times 256 = 65803$ 块。首先我们修改 `fs.h` 中的一些定义，使之符合三级索引的要求：

```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT - 1 + NINDIRECT + NINDIRECT * NINDIRECT)
```

由于文件的读写需要使用 `bmap()` 来找到需要操作的文件块，因此我们修改 `fs.c` 中用于找到文件块的 `bmap()`，使之能够支持三级索引。一个简单的思路是通过访问文件的位置来判断该位置是位于几级索引中，这样可以复用原先的大部分代码，而只需要实现二级间接索引的部分。笔者的实现如下：

```
static uint
bmap(struct inode *ip, uint bn)
{
    uint addr, *a;
    struct buf *bp;
    if(bn < NDIRECT - 1){
        if((addr = ip->addrs[bn]) == 0)
            ip->addrs[bn] = addr = balloc(ip->dev);
        return addr;
    }
    bn -= NDIRECT - 1;
    if(bn < NINDIRECT){
        if((addr = ip->addrs[NDIRECT - 1]) == 0)
            ip->addrs[NDIRECT - 1] = addr = balloc(ip->dev);
        bp = bread(ip->dev, addr);
        a = (uint*)bp->data;
        if((addr = a[bn]) == 0){
            a[bn] = addr = balloc(ip->dev);
        }
    }
}
```



```

        log_write(bp);
    }
    brelse(bp);
    return addr;
}
bn -= NINDIRECT;
if(bn < NINDIRECT * NINDIRECT){
    uint dbr = bn / NINDIRECT;
    uint dbc = bn % NINDIRECT;
    if((addr = ip->addrs[NDIRECT]) == 0)
        ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[dbr]) == 0){
        a[dbr] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[dbc]) == 0){
        a[dbc] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);
    return addr;
}
panic("bmap: out of range");
}

```

Info 11.1 (使用 bmap 的巧妙之处)

在 Unix 的文件系统中，将文件位置映射到块的位置的操作封装为 `bmap()` 可谓经典的软件工程技巧。按照常规的实现思路，需要在读写函数中分别实现查找对应块的过程，而将该过程抽象出来单独处理，使得文件系统的实现更加灵活，并具有更好的可扩展性：若要更改文件系统的底层结构，读写函数几乎无需更改，只需改变文件块的查找规则即可。

此时，xv6 的文件系统应该能够支持上述的大文件，再次编译并运行 xv6，然后运行 `bigfile`，将得到类似下图的结果：

```

xv6 kernel is booting
init: starting sh
$ bigfile
.....
.....
.....
wrote 65803 blocks
bigfile done: ok

```

图 10.1: bigfile 的测试结果

10.2 实现符号链接

在诸多类 Unix 系统中, 为了方便文件管理, 很多系统提供了符号链接功能。符号链接 (或软链接) 是指通过路径名链接的文件; 当一个符号链接被打开时, 内核会跟随链接指向被引用的文件。符号链接类似于硬链接, 但硬链接仅限于指向同一磁盘上的文件, 而符号链接可以跨磁盘设备。我们需要实现一个系统调用 `symlink(char *target, char *path)` 用于创建符号链接。

首先, 按照通常的方法在 `user/usys.pl` 和 `user/user.h` 中加入该系统调用的入口, 然后在 `kernel/sysfile.c` 中加入空的 `sys_symlink`。修改头文件 `kernel/stat.h`, 增加一个文件类型:

```
#define T_SYMLINK 4 // Symbolic link
```

然后在头文件 `kernel/fcntl.h` 加入一个标志位, 供 `open` 系统调用使用:

```
#define O_NOFOLLOW 0x010
```

接着在 `Makefile` 中加入用户程序 `symlinktest`, 此时应当能正常通过编译。这些准备工作完成后, 我们可以真正着手实现 `symlink(target, path)` 和对应的 `open` 系统调用。在 `sys_symlink` 中, 仿照 `sys_mknod` 的结构, 创建节点并写入软链接的数据:

```
uint64 sys_symlink(void)
{
    char target[MAXPATH], path[MAXPATH];
    struct inode *ip;
    if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
    {
        return -1;
    }
    begin_op();
    if((ip = namei(path)) == 0)
    {
        ip = create(path, T_SYMLINK, 0, 0);
        iunlock(ip);
    }
    ilock(ip);
    if(writei(ip, 0, (uint64)target, ip->size, MAXPATH) != MAXPATH)
    {
        return -1;
    }
    iunlockput(ip);
    end_op();
    return 0;
}
```

注意到其中调用了创建节点的函数 `create`, 因此我们也要对其进行修改。然后修改 `sys_open`, 使其能够跟随软链接并读取其指向的内容:

```
uint64 sys_open(void)
{
    ...
}
```

```

        if(ip->type == T_SYMLINK)
        {
            if((omode & O_NOFOLLOW) == 0)
            {
                char target[MAXPATH];
                int recursive_depth = 0;
                while(1)
                {
                    if(recursive_depth >= 10)
                    {
                        iunlockput(ip);
                        end_op();
                        return -1;
                    }
                    if(readi(ip, 0, (uint64)target, ip->size-
MAXPATH, MAXPATH) != MAXPATH)
                    {
                        return -1;
                    }
                    iunlockput(ip);
                    if((ip = namei(target)) == 0)
                    {
                        end_op();
                        return -1;
                    }
                    ilock(ip);
                    if(ip->type != T_SYMLINK)
                    {
                        break;
                    }
                    recursive_depth++;
                }
            }
        }
        ...
    }
}

```

至此，对于 xv6 的符号链接功能已经完成，再次编译并运行 xv6，然后运行 `symlinktest`，会得到类似下图的结果：

```

$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
$

```

图 10.2: symlinktest 的测试结果

可以看到，测试成功通过。

10.3 小结

在本章的实验中，我深入探讨了文件系统的结构和实现，特别是如何在 xv6 上支持大文件和实现符号链接功能。通过这些实验，我增强了对文件系统内核机制的理解，尤其是在扩展系统功能方面的实践。

10.3.1 遇到的问题与解决方法

在实现文件系统支持大文件和符号链接的过程中，我遇到了几个关键问题，这些问题挑战了我对文件系统的理解。以下是两个主要问题及其解决方法的详细描述：

问题一：大文件支持中索引超出范围的错误

问题描述： 在实现对大文件的支持时，由于文件系统索引结构的变化，原有的 `bmap()` 函数在处理三级索引时可能会出现索引超出范围的错误，导致系统崩溃。

解决方法： 为了解决这个问题，我重新设计了 `bmap()` 函数的逻辑，增加了对三级索引的支持。在函数中，通过访问文件位置来判断该位置属于哪个索引级别，从而正确地处理大文件的索引问题。这种设计确保了在处理大文件时，系统能够正确定位文件块，避免索引超出范围的情况。

问题二：符号链接的递归深度问题

问题描述： 在实现符号链接时，系统需要处理多个符号链接的递归解析。然而，如果链接链过长，可能会导致系统陷入无限递归，导致堆栈溢出或系统崩溃。

解决方法： 为了解决这一问题，我在 `sys_open()` 函数中引入了一个递归深度限制（例如最大递归深度为 10）。在每次递归解析符号链接时，都会检查当前递归深度，如果超过限制则终止解析并返回错误。这一限制有效防止了无限递归问题，确保系统的稳定性。

10.3.2 实验结果

通过这些修改和优化，我成功完成了文件系统的扩展和符号链接的实现，并通过了所有测试。实验结果如图所示，证明了这些功能在 xv6 系统中的正确性和稳定性。

10.4 实验结果

在完成 Lab File system 中的所有实验后，根据 MIT 6.S081 的传统，需要在实验目录下创建一个名为 `time.txt` 的文本文件，其中只包含一行内容，记录完成该实验所花费的小时数。然后在终端中执行 `make grade`，即可对整个实验进行自动评分。笔者的结果如下：

```
== Test running bigfile ==  
$ make qemu-gdb  
running bigfile: OK (86.7s)  
== Test running symlinktest ==  
$ make qemu-gdb  
(0.5s)  
== Test    symlinktest: symlinks ==  
symlinktest: symlinks: OK  
== Test    symlinktest: concurrent symlinks ==  
symlinktest: concurrent symlinks: OK  
== Test usertests ==  
$ make qemu-gdb  
usertests: OK (147.1s)  
== Test time ==  
time: OK  
Score: 100/100
```

图 10.3: Lab File system 的测评结果

可见，测试全部通过，得分为满分。

第十一章 Lab mmap: 内存映射实验

虽然在现代的 Linux 操作系统中，由于新的机制的引入，mmap 系统调用已经成为 glibc 在很多情形下的备选方案，但这并不影响 mmap 作为经典 Unix 系统调用的地位。为 xv6 实现 mmap 的过程会涉及修改操作系统的几乎每个部分，因此作为压轴实验供我们深入探讨。

11.1 实现 mmap 内存映射系统调用

mmap 为用户程序提供了精细化操作它们地址空间的手段，包括但不限于：共享进程间的地址空间，将文件映射到地址空间，以及用户处理缺页错误等功能。我们需要为 xv6 添加的 mmap 只需实现将文件映射到地址空间的功能，其接受的参数如下：

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t
offset);
```

此外，还有一个用于取消 mmap 映射的系统调用 munmap(addr, length)，除了移除映射外，其根据实际情况可能需要将映射的文件写回。

xv6 实验和往常一样，为我们提供了用户态测评程序 mmaptest。在该测评程序中，addr 和 offset 总是为 0，意味着内核可以自由选择映射的地址，并总是从文件开头开始映射；若映射失败，则返回 0xffffffffffffffff。我们只需实现 mmaptest 中用到的功能。

首先在 Makefile 中添加用户态测评程序 mmaptest，然后添加空的 mmap 和 munmap 函数，使得编译通过：

```
UPROGS=\
.....
$U/_mmaptest\
.....
```

Listing 11.1: Makefile 中的配置

```
kernel/syscall.h
=====
.....
#define SYS_mmap 22
#define SYS_munmap 23
=====
kernel/syscall.c
=====
.....
extern uint64 sys_mmap(void);
extern uint64 sys_munmap(void);
```

```

.....
static uint64 (*syscalls[])(void) = {
    .....
    [SYS_mmap] sys_mmap,
    [SYS_munmap] sys_munmap,
};
.....

```

Listing 11.2: 系统调用定义

```

kernel/sysproc.c
=====

.....
void *sys_mmap(void)
{
    return (void *)-1;
}

int sys_munmap(void)
{
    return -1;
}
.....

```

Listing 11.3: 系统调用的初始实现

```

user/user.h
=====

.....
void *mmap(void *, int, int, int, int, int);
int munmap(void *, int);
.....

```

Listing 11.4: 用户态函数声明

```

user/usys.pl
=====

.....
entry("mmap");
entry("munmap");
.....

```

Listing 11.5: 用户态系统调用入口

此时执行 `make`，应当能够编译通过。按照 `xv6` 实验手册的提示，在 `proc.h` 中加入 `VMA` 数据结构的定义，并给进程控制块中加入 `VMA` 指针：

```

#define NVMA 16
#define VMA_START (MAXVA >> 1)

```

```

struct vma {
    uint64 start;
    uint64 end;
    uint64 length; // 0 -> not used
    uint64 off;
    int perm;
    int flags;
    struct file *file;
    struct vma *next;
    struct spinlock lock;
};

// Per-process state
struct proc {
    .....
    struct vma *vma; // VMA item
};

```

Listing 11.6: VMA 数据结构定义

为了避免与堆和栈冲突，这里我们决定将内存映射开始于最大地址的一半处。由于后续需要多次对 VMA 进行分配，因此将其封装为一个函数会更加方便。接着在 `proc.c` 中加入：

```

struct vma vma_list[NVMA];

struct vma* vma_alloc()
{
    for(int i = 0; i < NVMA; i++)
    {
        acquire(&vma_list[i].lock);
        if (vma_list[i].length == 0)
        {
            return &vma_list[i];
        } else
        {
            release(&vma_list[i].lock);
        }
    }
    panic("no free vma");
}

```

Listing 11.7: VMA 分配函数

根据实验手册的提示，我们无需在 `sys_mmap` 中完成内存映射，而是通过一个惰性机制，在出现缺页异常时再进行映射。因此我们在 `sys_mmap` 中只需完成对 VMA 的设置：

```

#include "types.h"
#include "param.h"
#include "date.h"

```



```
#include "spinlock.h"
#include "sleeplock.h"
#include "fs.h"
#include "memlayout.h"
#include "riscv.h"
#include "defs.h"
#include "proc.h"
#include "fcntl.h"
#include "file.h"

extern struct vma *vma_alloc();

void *sys_mmap(void)
{
    uint64 addr;
    struct proc *p = myproc();
    int length, prot, flags, fd, offset;

    if (argaddr(0, &addr) < 0)
        return (void *)-1;
    if (argint(1, &length) < 0)
        return (void *)-1;
    if (argint(2, &prot) < 0)
        return (void *)-1;
    if (argint(3, &flags) < 0)
        return (void *)-1;
    if (argint(4, &fd) < 0)
        return (void *)-1;
    if (argint(5, &offset) < 0)
        return (void *)-1;

    if (addr != 0)
        addr = 0;
    if (offset != 0)
        offset = 0;

    struct file *f = p->ofile[fd];

    // Check flags
    int pte_flag = PTE_U;
    if (prot & PROT_READ)
    {
        if (!f->readable)
            return (void *)-1;
        pte_flag |= PTE_R;
    }
}
```

```

    }
    if (prot & PROT_WRITE)
    {
        if (!f->writable && !(flags & MAP_PRIVATE))
            return (void *)-1;
        pte_flag |= PTE_W;
    }

    // Setting up vma
    struct vma *v = vma_alloc();
    v->perm = pte_flag;
    v->length = length;
    v->off = offset;
    v->file = myproc()->ofile[fd];
    v->flags = flags;
    filedup(f);

    struct vma *pv = p->vma;
    if (pv == 0)
    {
        v->start = VMA_START;
        v->end = length + v->start;
        p->vma = v;
    }
    else
    {
        while (pv->next)
            pv = pv->next;
        v->start = PGROUNDUP(pv->end);
        v->end = v->start + length;
        pv->next = v;
        v->next = 0;
    }
    addr = v->start;
    release(&v->lock);
    return (void *) (addr);
}

```

Listing 11.8: sys_mmap 实现

为了在 usertrap 中处理 mmap 造成的缺页异常，我们首先编写相关的中断处理过程 mmap_alloc:

```

#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "riscv.h"
#include "spinlock.h"

```

```
#include "sleeplock.h"
#include "proc.h"
#include "defs.h"
#include "fs.h"
#include "file.h"

int mmap_alloc(uint64 va, int scause)
{
    struct proc *p = myproc();
    struct vma* v = p->vma;

    while(v != 0)
    {
        if (va >= v->start && va < v->end)
        {
            break;
        }
        v = v->next;
    }

    if (v == 0)
        return -1;
    if (scause == 13 && !(v->perm & PTE_R))
        return -1;
    if (scause == 15 && !(v->perm & PTE_W))
        return -1;

    // load from file
    va = PGROUNDDOWN(va);
    char* mmem = kalloc();
    if (mmem == 0)
        return -1;
    memset(mmem, 0, PGSIZE);
    if (mappages(p->pagetable, va, PGSIZE, (uint64)mmem, v->perm) != 0)
    {
        kfree(mmem);
        return -1;
    }

    struct file *f = v->file;
    ilock(f->ip);
    readi(f->ip, 0, (uint64)mmem, v->off + va - v->start, PGSIZE);
    iunlock(f->ip);

    return 0;
}
```

```
}
```

Listing 11.9: 缺页处理函数 mmap_alloc

在 usertrap 中调用 mmap_alloc:

```
void usertrap(void)
{
    ...
} else if((r_scause() == 13) || (r_scause() == 15)){ // page fault
    if (mmap_alloc(r_stval(), r_scause()) != 0)
    {
        printf("mmap: page fault\n");
        p->killed = 1;
    }
} else {
    ...
}
```

Listing 11.10: usertrap 中的调用

接下来我们需要实现 munmap。为了处理内存映射区域的释放操作，我们首先实现一个写回函数：

```
void write_back(struct vma *v, uint64 addr, int n)
{
    // no need to writeback
    if (!(v->perm & PTE_W) || (v->flags & MAP_PRIVATE))
        return;
    if ((addr % PGSIZE) != 0)
        panic("unmap: not aligned");

    struct file *f = v->file;
    int max = ((MAXOPBLOCKS - 1 - 1 - 2) / 2) * BSIZE;
    int i = 0;

    while (i < n)
    {
        int k = n - i;
        if (k > max)
            k = max;
        begin_op();
        ilock(f->ip);
        int wcnt = writei(f->ip, 1, addr + i, v->off + v->start - addr
+ i, k);

        iunlock(f->ip);
        end_op();
        i += wcnt;
    }
}
```

}

Listing 11.11: 写回函数 write_back

然后实现 `sys_munmap`, 处理 VMA 区域的释放:

```
int sys_munmap(void)
{
    uint64 addr;
    int length;

    if (argaddr(0, &addr) < 0)
        return -1;
    if (argint(1, &length) < 0)
        return -1;

    struct proc *p = myproc();
    struct vma *v = p->vma;
    struct vma *pre = 0;

    while (v != 0)
    {
        if (addr >= v->start && addr < v->end)
            break;
        pre = v;
        v = v->next;
    }

    // not mapped
    if (v == 0)
        return -1;

    if (addr != v->start && addr + length != v->end)
        panic("munmap: middle of vma");

    if (addr == v->start)
    {
        write_back(v, addr, length);
        uvmunmap(p->pagetable, addr, length / PGSIZE, 1);

        if (length == v->length)
        {
            // free all
            fileclose(v->file);
            if (pre == 0)
            {
                p->vma = v->next;
            }
        }
    }
}
```

```

    }
    else
    {
        pre->next = v->next;
        v->next = 0;
    }
    acquire(&v->lock);
    v->length = 0;
    release(&v->lock);
}
else
{
    // head
    v->start -= length;
    v->off += length;
    v->length -= length;
}
}
else
{
    // tail
    v->length -= length;
    v->end -= length;
}
return 0;
}

```

Listing 11.12: sys_munmap 实现

此时，`mmap` 的核心功能已经基本完成。接下来我们处理 `fork` 时 VMA 的复制以及进程退出时的内存写回操作。

首先是 `fork` 的处理：

```

int fork(void)
{
    ...
    acquire(&np->lock);
    np->state = RUNNABLE;
    np->vma = 0;
    struct vma *pvma = p->vma;
    struct vma *pre = 0;
    while (pvma)
    {
        struct vma *nvma = vma_alloc();
        nvma->start = pvma->start;
        nvma->end = pvma->end;
        nvma->off = pvma->off;
    }
}

```

```

        nvma->length = pvma->length;
        nvma->perm = pvma->perm;
        nvma->flags = pvma->flags;
        nvma->file = pvma->file;
        filedup(nvma->file);
        nvma->next = 0;

        if (pre == 0)
        {
            np->vma = nvma;
        }
        else
        {
            pre->next = nvma;
        }
        pre = nvma;
        release(&nvma->lock);
        pvma = pvma->next;
    }
    release(&np->lock);
    return pid;
}

```

Listing 11.13: fork 中的 VMA 复制

然后处理 exit 系统调用:

```

extern void write_back(struct vma *v, uint64 addr, int n);

void exit(int status)
{
    struct proc *p = myproc();
    if (p == initproc)
        panic("init exiting");

    // Unmap all vma
    struct vma *v = p->vma;
    struct vma *pvma;
    while (v)
    {
        write_back(v, v->start, v->length);
        uvmunmap(p->pagetable, v->start, PGROUNDUP(v->length) / PGSIZE
, 1);

        fileclose(v->file);
        pvma = v->next;
        acquire(&v->lock);
        v->next = 0;
    }
}

```

```

        v->length = 0;
        release(&v->lock);
        v = pvma;
    }

    // Close all open files.
    ...
}

```

Listing 11.14: exit 中的写回处理

至此，munmap 部分大致完成。编译并运行 xv6，然后运行 mmptest，会基本通过测试，但最后一项中会得到 panic: uvmunmap: walk 错误。由于 mmap 机制与原有页表的标志位含义有冲突，因此此时需修改 vm.c，进行如下修改：

```

void uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
    uint64 a;
    pte_t *pte;

    if((va % PGSIZE) != 0)
        panic("uvmunmap: not aligned");

    for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
        if((pte = walk(pagetable, a, 0)) == 0)
            continue;
        if((*pte & PTE_V) == 0)
            continue;
        if(PTE_FLAGS(*pte) == PTE_V)
            panic("uvmunmap: not a leaf");
        if(do_free){
            uint64 pa = PTE2PA(*pte);
            kfree((void*)pa);
        }
        *pte = 0;
    }
}

void freewalk(pagetable_t pagetable)
{
    for(int i = 0; i < 512; i++){
        pte_t pte = pagetable[i];
        if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
            uint64 child = PTE2PA(pte);
            freewalk((pagetable_t)child);
            pagetable[i] = 0;
        } else if(pte & PTE_V){

```



```

                                continue;
                            }
                        }
                    }
                    kfree((void*)pagetable);
                }
            }

```

Listing 11.15: vm.c 中的修改

编译并运行 xv6，然后运行 mmaptest，得到类似下图的结果：

```

xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ mmap test
exec mmap failed
$ mmaptest
mmap_test starting
test mmap f
test mmap f: OK
test mmap private
test mmap private: OK
test mmap read-only
test mmap read-only: OK
test mmap read/write
test mmap read/write: OK
test mmap dirty
test mmap dirty: OK
test not-mapped unmap
test not-mapped unmap: OK
test mmap two files
test mmap two files: OK
mmap_test: ALL OK
fork_test starting
fork_test OK
mmaptest: all tests succeeded

```

图 11.1: mmaptest 的测试结果

11.2 小结

本章实验的重点是实现 mmap 内存映射系统调用，以及其相应的取消映射操作 munmap。通过这些实验，我深入理解了内存映射的机制，并掌握了如何在操作系统中处理页表和内存分配的复杂问题。

11.2.1 遇到的问题与解决方法

在实现 mmap 和 munmap 的过程中，我遇到了一些关键问题，这些问题在实现内存映射功能时表现得尤为突出。以下是几个主要问题及其解决方法的详细描述：

问题一：缺页错误处理中的权限检查问题

问题描述： 在处理缺页错误时，系统需要根据 mmap 的权限参数来决定是否加载页面。然而，初始实现中，系统有时会忽略对权限的正确检查，导致一些不应映射的页面被错误地加载。

解决方法： 为了解决这个问题，我在 mmap_alloc() 函数中增加了详细的权限检查逻辑。在缺页处理过程中，根据 mmap 的权限参数（如 PROT_READ 和 PROT_WRITE），严格控制页面的加载权限，确保只有符合权限要求的页面才会被正确映射。

问题二: munmap 中释放页面的写回问题

问题描述: 在 munmap 操作中, 系统需要将被取消映射的页面写回到文件中, 特别是在处理可写的共享映射时。如果不正确处理写回操作, 可能会导致数据丢失或文件内容不一致。

解决方法: 为了解决这个问题, 我在 write_back() 函数中实现了一个完整的写回逻辑。该函数会检查 VMA 的权限和标志位, 根据需要将内存中的数据正确写回到文件中, 确保数据一致性。此外, 我还确保在 munmap 中处理部分释放和完全释放的不同情况, 保证内存映射区域能够被安全地取消映射。

问题三: fork 过程中 VMA 的复制问题

问题描述: 在 fork 操作中, 子进程需要复制父进程的 VMA。如果 VMA 复制不完整或不正确, 子进程在使用 mmap 的内存区域时可能会出现错误, 甚至导致系统崩溃。

解决方法: 为了解决这一问题, 我在 fork() 函数中实现了 VMA 的深度复制, 确保子进程获得与父进程完全一致的内存映射区域。此外, 为了避免竞争条件导致的同步问题, 我在 VMA 复制过程中增加了锁的保护, 确保在高并发情况下复制操作的安全性。

11.2.2 实验结果

```
== Test running mmaptest ==
$ make qemu-gdb
(2.9s)
== Test  mmaptest: mmap f ==
mmaptest: mmap f: OK
== Test  mmaptest: mmap private ==
mmaptest: mmap private: OK
== Test  mmaptest: mmap read-only ==
mmaptest: mmap read-only: OK
== Test  mmaptest: mmap read/write ==
mmaptest: mmap read/write: OK
== Test  mmaptest: mmap dirty ==
mmaptest: mmap dirty: OK
== Test  mmaptest: not-mapped unmap ==
mmaptest: not-mapped unmap: OK
== Test  mmaptest: two files ==
mmaptest: two files: OK
== Test  mmaptest: fork_test ==
mmaptest: fork_test: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (74.5s)
== Test time ==
time: OK
Score: 140/140
```

图 11.2: mmap 的测评结果

可看出结果为满分。