实验报告

09023321 巩皓锴 2025年5月28日

一、实验内容

1. Reverse

编写一个名为 `reverse` 的简单程序。该程序应通过以下其中一种方式来运行：

```
prompt> ./reverse
prompt> ./reverse input.txt
prompt> ./reverse input.txt output.txt
```

详细题目略

2. Xv6-Syscall

向 xv6 添加一个系统调用。你的系统调用 `getreadcount()` 仅仅返回自内核启动以来，用户进程调用 `read()` 系统调用的总次数。

你新增的系统调用应该具有以下的返回值和参数：

```
int getreadcount(void)
```

你的系统调用返回一个计数器的值（你可以把它叫做 `readcount` 或其他名称），每当任何进程调用 `read()` 系统调用时，这个计数器就加一。

二、实验目的

1. **熟悉命令行程序的编写与文件操作**
   通过编写 `reverse` 程序，掌握 C 语言中文件读取、写入和标准输入输出的基本用法，理解命令行参数的处理方式，增强对 Linux 命令行环境的适应能力。
2. **掌握 xv6 操作系统内核的基本结构**
   理解 xv6 的系统调用机制，掌握如何添加和实现一个新的系统调用，了解内核与用户态之间的接口调用流程。
3. **加深对系统调用本质的理解**
   通过实现 `getreadcount()` 系统调用，认识系统调用的设计原理，理解内核态如何维护全局状态，并为用户态提供服务。
4. **提升动手能力和调试能力**
   通过对 xv6 源码的修改和测试，提升对复杂系统代码的阅读、修改和调试能力，为后续深入操作系统课程内容打下基础。

三、设计思路

Reverse

本程序 `reverse` 的目标是将输入中的文本按行为单位进行倒序输出。主要实现思路如下：

1. **参数处理**：判断命令行参数数量是否合法，打开输入输出文件，并检查输入输出是否是同一文件。
2. **读取输入**：使用 `getline()` 逐行读取输入内容，将每行保存在一个链表节点中，并插入到链表头部，实现“倒序”效果。
3. **写入输出**：从链表头到尾遍历，将每行写入输出文件或标准输出，实现倒序打印。
4. **清理资源**：关闭文件、释放内存、正常退出。

具体代码详见第四部分

2. Xv6-Syscall

需要添加的系统调用需要返回自内核启动以来，用户进程调用 `read()` 系统调用的总次数。主要实现思路如下：

1. **次数记录**：使用内核态全局变量存储 `read()` 系统的调用的次数。
2. **次数更新**：在系统调用 `read()` 的实现中实现对全局变量的自增操作。
3. **注册系统调用**：将新的系统调用写入注册文件，定义其调用号，并暴露到用户态。
4. **实现锁**：多进程或多线程调用 `read()` 时，可能不能正确的对全局变量进行自增，所以需要对其进行加锁保护。在声明全局变量的同时声明锁，在 xv6 启动时初始化锁。

具体代码详见第四部分

四、代码并附注释

Reverse

太长了，放到最后面的**附录 1 Reverse代码**

Xv6-Syscall

1. **次数记录**：

```
// file.c
+ int readcount = 0;
```

2. **次数更新**：

```
// sysfile.c
int
sys_read(void)
{
+ readcount++;
  ...
}
```

3. **注册系统调用**：

```
// syscall.h
#define SYS_getreadcount 22
// syscall.c
+ extern int sys_getreadcount(void);
+ [SYS_getreadcount] sys_getreadcount,
// sysfile.c
+ int sys_getreadcount(void)
+ {
+   return readcount;
+ }
// user.h
+ int getreadcount(void);
// usys.S
+ SYSCALL(getreadcount)
```

4. **实现锁**:

```
// file.c
+ struct spinlock readcount_lock;
void
fileinit(void)
{
  initlock(&ftable.lock, "ftable");
  + initlock(&readcount_lock, "readcount");
}
// sysfile.c
+ extern struct spinlock readcount_lock;
+ acquire(&readcount_lock);
  readcount++;
+ release(&readcount_lock);
```

五、程序运行结果

使用给出的测试脚本进行测试

```
❯ ./test-reverse.sh
test 1: passed
test 2: passed
test 3: passed
test 4: passed
test 5: passed
test 6: passed
test 7: passed

❯ ./test-getreadcount.sh
# 编译过程略
test 1: passed
test 2: passed
```

可以看到，测试全部通过，输出均符合预期。

六、实验流程记录

实验环境:
Windows Subsystem of Linux Ubuntu 24.04.2 LTS (Linux H-LAPTOP 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux)

软件准备

从课程群得到实验相关要求与测试脚本 `projects-students`，安装所需软件包

```
sudo apt update
sudo apt install -y build-essential qemu-system-x86
```

参考测试脚本中的提示，将 xv6 克隆到 `/path/to/Xv6-Syscall/src`

```
git clone git@github.com:harkerhand/xv6-public.git /path/to/Xv6-Syscall/src
```

在 `src` 目录中执行 `make qume` 即可进入 Xv6 环境，此时出现问题，我已有的环境中的 `gcc` 版本太新（14.2.0），其认为 xc6 中的部分代码危险，故拒绝编译，产生错误，查阅 Makefile，找到编译选项，删除 `-Werror` 新增 `-Wno-array-bounds`，即可正常编译。

进行实验一： Reverse

阅读题目要求，编写代码至 `path/to/Reverse/reverse.c`，编译

```
gcc path/to/Reverse/reverse.c -o path/to/Reverse/reverse
```

使用 `path/to/Reverse/test-reverse.sh` 进行测试（可能需要 `chmod` 给权限），参考测试结果修正代码的错误处理部分，直到测试通过。

进行实验二：Xv6-Syscall

查阅手册得到 xv6 系统的系统调用声明位置，其他系统调用辅证其位置正确，注册系统调用并实现，并暴露到用户态。

在 `/path/to/Xv6-Syscall` 目录中执行 `./test-getreadcount.sh` 进行测试，发现问题，需要安装 `expect` 软件包，遂安装之。

重新测试，又发现问题，测试点二（测试代码见**附录2 Syscall测试2**）输出，理论值为 `XV6_TEST_OUTPUT 200000`，实际出现 `XV6_TEST_OUTPUT 199995` 且末尾数字波动。思考问题，猜测为 `fork()` 后的两个进程对全局变量操作时候遇到的竞争问题，遂查阅手册，得到 xv6 中锁的实现，新增代码见前文锁的部分。

重新测试，测试通过。

所有操作记录均同步到该仓库： harkerhand/xv6-public: xv6 OS 。

七、实验体会

通过本次实验，我对C语言编程的基础操作有了更深的理解，尤其是文件的读取与写入、命令行参数的处理，以及标准输入输出的灵活运用。同时，完成 `reverse` 程序让我熟悉了如何设计并实现一个功能简单但细节丰富的命令行工具。

在xv6系统调用的部分，我实际接触了操作系统内核代码，理解了系统调用的整体流程，从内核态到用户态的交互机制。同时，通过添加计数器功能，深入体会了并发编程中共享资源的竞争问题，并学会了如何利用内核提供的锁机制来避免数据竞争，保证数据一致性。

实验过程中遇到的问题促使我学会了通过查阅文档、调试和分步测试来定位和解决问题，这提升了我的动手能力和解决问题的思维方式。特别是在调试系统调用和修改内核代码时，细致严谨的态度是必不可少的。

总之，本次实验不仅让我巩固了理论知识，也增强了实际操作能力，对后续学习操作系统和系统编程奠定了坚实基础。

八、附录

1 Reverse代码

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <errno.h>
// 定义链表节点结构，用于存储每一行文本
typedef struct LineNode {
    char *line;            // 存储一行文本内容
    struct LineNode *next; // 指向下一个节点
} LineNode;
// 从文件中读取所有行并反向构建链表
LineNode *read_lines(FILE *fp) {
    char *line = NULL;
    size_t len = 0;
    ssize_t read;
    LineNode *head = NULL; // 链表头节点
    // 逐行读取文件内容
    while ((read = getline(&line, &len, fp)) != -1) {
        // 为每一行文本分配新节点
        LineNode *new_node = malloc(sizeof(LineNode));
        if (!new_node) {
            fprintf(stderr, "malloc failed\n");
            free(line);
            return NULL;
        }
        // 复制行内容
        new_node->line = strdup(line);
        if (!new_node->line) {
            fprintf(stderr, "malloc failed\n");
            free(new_node);
            free(line);
            return NULL;
        }
        // 将新节点插入链表头部，实现反向存储
        new_node->next = head;
        head = new_node;
    }
    free(line);
```

```c
    return head;
}
// 将链表中的行按顺序写入文件
void write_reversed(FILE *fp, LineNode *head) {
    LineNode *cur = head;
    // 遍历链表，输出每一行
    while (cur) {
        fputs(cur->line, fp);
        cur = cur->next;
    }
}
// 释放链表占用的内存
void free_lines(LineNode *head) {
    LineNode *cur;
    while ((cur = head) != NULL) {
        head = head->next; // 保存下一个节点
        free(cur->line);   // 释放行内容
        free(cur);         // 释放节点
    }
}
int main(int argc, char *argv[]) {
    int fd_in = STDIN_FILENO;
    int fd_out = STDOUT_FILENO;
    FILE *in_fp = stdin, *out_fp = stdout;
    struct stat st1, st2;
    // 检查参数数量是否合法
    if (argc > 3) {
        fprintf(stderr, "usage: reverse <input> <output>\n");
        exit(EXIT_FAILURE);
    }
    // 处理输入文件
    if (argc >= 2) {
        // 打开输入文件
        fd_in = open(argv[1], O_RDONLY);
        if (fd_in < 0) {
            fprintf(stderr, "reverse: cannot open file '%s'\n", argv[1]);
            exit(EXIT_FAILURE);
        }
        // 转换为文件流
        in_fp = fdopen(fd_in, "r");
        if (!in_fp) {
            fprintf(stderr, "reverse: cannot open file '%s'\n", argv[1]);
            close(fd_in);
            exit(EXIT_FAILURE);
        }
    }
    // 处理输出文件
    if (argc == 3) {
        // 获取输入文件信息
        if (stat(argv[1], &st1) < 0) {
            fprintf(stderr, "reverse: cannot stat %s: %s\n", argv[1],
strerror(errno));
            if (fd_in != STDIN_FILENO) close(fd_in);
            exit(EXIT_FAILURE);
        }
        // 检查输入输出是否为同一个文件
```

```
        if (stat(argv[2], &st2) == 0) {
            if (st1.st_ino == st2.st_ino) {
                fprintf(stderr, "reverse: input and output file must differ\n");
                if (fd_in != STDIN_FILENO) close(fd_in);
                exit(EXIT_FAILURE);
            }
        }
        else if (errno != ENOENT) {
            // 如果stat出错且不是因为文件不存在，则报错
            fprintf(stderr, "reverse: cannot stat %s: %s\n", argv[2], strerror(errno));
            if (fd_in != STDIN_FILENO) close(fd_in);
            exit(EXIT_FAILURE);
        }
        // 打开输出文件
        fd_out = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
        if (fd_out < 0) {
            fprintf(stderr, "reverse: cannot open %s: %s\n", argv[2], strerror(errno));
            if (fd_in != STDIN_FILENO) close(fd_in);
            exit(EXIT_FAILURE);
        }
        // 转换为文件流
        out_fp = fdopen(fd_out, "w");
        if (!out_fp) {
            fprintf(stderr, "reverse: cannot open %s: %s\n", argv[2], strerror(errno));
            if (fd_in != STDIN_FILENO) close(fd_in);
            if (fd_out != STDOUT_FILENO) close(fd_out);
            exit(EXIT_FAILURE);
        }
    }
    // 读取所有行
    LineNode *lines = read_lines(in_fp);
    if (!lines && ferror(in_fp)) {
        fprintf(stderr, "malloc failed\n");
        if (fd_in != STDIN_FILENO) close(fd_in);
        if (fd_out != STDOUT_FILENO) close(fd_out);
        exit(EXIT_FAILURE);
    }
    // 按反序写出所有行
    write_reversed(out_fp, lines);
    free_lines(lines);
    // 关闭文件
    if (in_fp && in_fp != stdin) fclose(in_fp);
    if (out_fp && out_fp != stdout) fclose(out_fp);
    exit(EXIT_SUCCESS);
}
```

2 Syscall测试2

```c
#include "types.h"
#include "stat.h"
#include "user.h"
int main(int argc, char *argv[])
{
    int x1 = getreadcount();
    int rc = fork();
    int total = 0;
    int i;
    for (i = 0; i < 100000; i++) {
        char buf[100];
        (void)read(4, buf, 1);
    }
    // https://wiki.osdev.org/Shutdown
    // (void) shutdown();
    if (rc > 0) {
        (void)wait();
        int x2 = getreadcount();
        total += (x2 - x1);
        printf(1, "XV6_TEST_OUTPUT %d\n", total);
    }
    exit();
}
```