

操作系统课程设计实验报告

黄毅成

2024.7.16

目录

1 Lab: Xv6 and Unix utilities	1
1.1 实验目的	1
1.2 实验步骤	1
1.2.1 运行 xv6	1
1.2.2 实现 sleep	2
1.2.3 实现 pingpong	2
1.2.4 实现 prime	3
1.2.5 实现 find	6
1.2.6 实现 xargs	8
1.3 评测结果	9
1.4 实验小结	10
2 Lab: System calls	10
2.1 实验目的	10
2.2 实验步骤	10
2.2.1 实现 trace	10
2.2.2 实现 sysinfo	13
2.3 评测结果	15
2.4 实验小结	15
3 Lab: Page tables	16
3.1 实验目的	16
3.2 实验步骤	16
3.2.1 实现系统调用加速	16
3.2.2 实现页表打印	17
3.2.3 实现被访问页面侦测	19
3.3 评测结果	22
3.4 实验小结	22
4 Lab: Traps	23
4.1 实验目的	23
4.2 实验步骤	23
4.2.1 了解 RISC-V 汇编	23
4.2.2 实现 Backtrace	26
4.2.3 实现 alarm	27
4.3 评测结果	30
4.4 实验小结	30

5	Lab: Copy-on-Write Fork for xv6	31
5.1	实验目的	31
5.2	实验步骤	31
5.3	评测结果	36
5.4	实验小结	36
6	Lab: Multithreading	37
6.1	实验目的	37
6.2	实验步骤	37
6.2.1	实现用户线程切换	37
6.2.2	使用 UNIX pthread 线程库实现一个线程安全的哈希表	40
6.2.3	实现 barrier	41
6.3	评测结果	42

1 Lab: Xv6 and Unix utilities

1.1 实验目的

本实验旨在熟悉 xv6 操作系统及其系统调用。通过本次实验，我将学习如何在 xv6 环境中设置并运行操作系统，理解并实现基本的 UNIX 实用程序。这包括启动 xv6、获取并管理源代码、编译和运行操作系统、以及实现和测试具体的系统调用和应用程序。本实验将帮助深入理解操作系统的运行机制 and 开发流程，为后续更复杂的操作系统编程打下坚实基础。

具体将实现以下程序：

- sleep 程序：编写一个 xv6 版本的 sleep 程序，使其根据用户指定的 ticks 暂停。
- pongpong 程序：编写一个使用管道和 fork 系统调用在父子进程间传递字节的程序。
- primes 程序：编写一个基于管道的并发素数筛选程序，将数值在进程间传递并筛选素数。
- find 程序：编写一个简化版的 find 程序，遍历目录树查找特定文件名。
- xargs 程序：编写一个简化版的 xargs 程序，从标准输入读取行并为每行运行一个命令。

在实现和测试这些系统调用的过程中，我们将深入理解操作系统的内部机制，并掌握如何在用户程序中调用这些功能。

1.2 实验步骤

1.2.1 运行 xv6

1. 利用以下指令获取 Lab xv6 源代码，查看 util 分支。

```
$ git clone git://g.csail.mit.edu/xv6-labs-2021
$ cd xv6-labs-2021
$ git checkout util
```

2. 输入 make qemu 命令构建并运行 xv6。

```
$ make qemu

...

xv6 kernel is booting

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

1.2.2 实现 sleep

1. 根据题目提示，可以实现如下代码：

sleep.c

```
#include "kernel/types.h" // 包含内核类型定义
#include "user/user.h" // 包含用户模式下的库函数

int main(int argc, char *argv[])
{
    // 检查命令行参数是否等于2，即程序名和等待时间参数
    if (argc != 2)
    {
        fprintf(2, "Error: Parameters Error\n"); // 打印错误信息
        exit(1); // 退出程序，返回状态码1
    }

    sleep(atoi(argv[1])); // 调用 sleep 函数，等待指定的秒数

    exit(0); // 程序正常退出，返回状态码0
}
```

2. 在 Makefile 文件中的 UPROGS 加上 \$U/_sleep\后利用 make qemu 命令编译启动 xv6 操作系统；
3. 在 xv6 命令行中运行 sleep，程序将暂停执行一段指定的时间：

```
$ sleep 10
```

1.2.3 实现 pingpong

1. 首先创建两条管道和用于储存数据的缓冲区。

初始化管道和缓冲区

```
int fd1[2];
int fd2[2];

pipe(fd1);
pipe(fd2);

char buffer[16];
```

2. 利用 `fork()` 创建子进程。

fork() 函数

```
if (fork() != 0)
{
    // 父线程执行的代码...
}
else
{
    // 子线程执行的代码...
}
```

3. 在父进程中发送 “ping” 并接收 “pong”。

父进程

```
// 父线程执行的代码
write(fd1[1], "ping", strlen("ping"));
read(fd2[0], buffer, 4);
printf("%d: received %s\n", getpid(), buffer);
```

4. 在子进程中发送 “pong” 并接收 “ping”。

子进程

```
// 子线程执行的代码
read(fd1[0], buffer, 4);
printf("%d: received %s\n", getpid(), buffer);
write(fd2[1], "pong", strlen("pong"));
```

5. 把程序写在 `pingpong.c` 中，用同样的方法在 `xv6` 中运行程序，可以观察到父子进程分别输出的结果。

```
$ pingpong
5: received ping
4: received pong
```

1.2.4 实现 prime

1. 首先了解程序的原理。原理示意图1-1。程序通过递归创建进程和使用管道通信的方式，逐步筛选出素数。每个进程从管道中读取数值，将第一个数识别为素数，并过滤掉其倍数，然后将剩余的数传递给下一个进程继续处理，从而并发地筛选出所有素数。

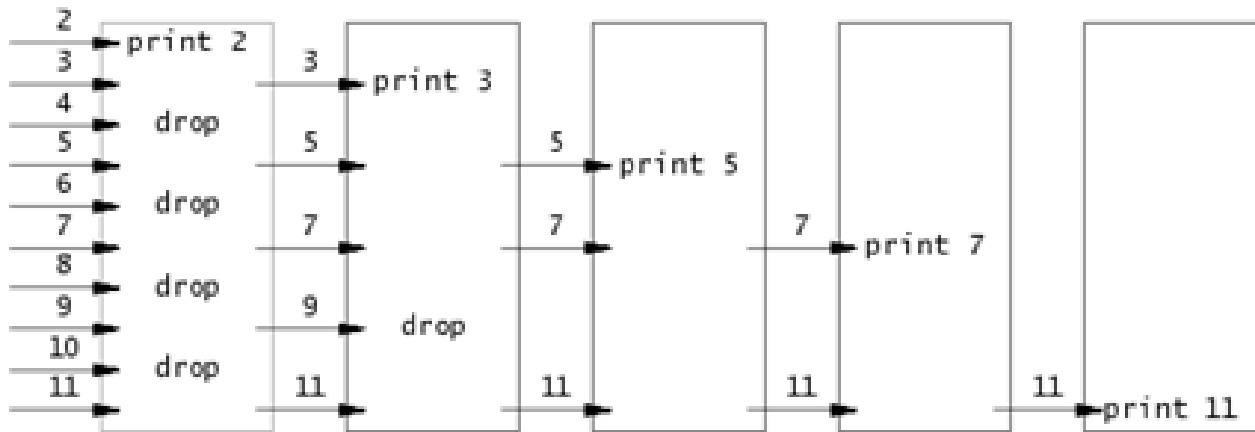


图 1-1: primes 程序原理

2. 把输出质数，创建子进程并向其传送数据的过程整合为一个递归函数。要注意文件描述符的回收，防止资源超限。

new_prime_proc 函数的实现

```
void new_prime_proc(int *old_pipe)
{
    close(old_pipe[1]); // 关闭旧管道的写端

    int first_num;
    // 从旧管道中读取第一个数，如果读取失败则退出
    if (!read(old_pipe[0], &first_num, sizeof(first_num)))
    {
        close(old_pipe[0]); // 关闭旧管道的读端
        exit(0); // 退出进程
    }

    fprintf(1, "prime %d\n", first_num); // 打印第一个数，即当前的素数

    int new_pipe[2];
    pipe(new_pipe); // 创建新管道

    int p_id = fork(); // 创建子进程
    if (p_id == 0)
        new_prime_proc(new_pipe); // 子进程递归调用处理新管道
    else
    {
        int t;
        // 在父进程中，从旧管道中读取数，如果不是first_num的倍数则写入新管道
```

```

while (read(old_pipe[0], &t, sizeof(t)))
{
    if (t % first_num != 0)
    {
        write(new_pipe[1], &t, sizeof(t));
    }
}

close(old_pipe[0]); // 关闭旧管道的读端
close(new_pipe[0]); // 关闭新管道的读端
close(new_pipe[1]); // 关闭新管道的写端

wait((int *)0); // 等待子进程结束
}
}

```

3. 在主函数中将 2 35 写入管道，调用递归函数。

main() 函数的实现

```

int main()
{
    int p[2];
    pipe(p); // 创建管道
    int i;
    // 向管道中写入从2到35的所有数
    for (i = 2; i <= 35; i++)
        write(p[1], &i, sizeof(i));
    new_prime_proc(p); // 调用处理函数处理管道中的数

    exit(0); // 退出程序
}

```

4. 把程序写在 primes.c 中，运行程序得到结果。

```

$ primes
prime 2
prime 3
...
prime 31

```

1.2.5 实现 find

1. 首先，仿照 ls.c 实现一个辅助函数 char*fntname(char *path)，把路径末尾的文件名提取出来，实现过程略。
2. 利用深度优先搜索的思想，实现一个 find 函数递归寻找文件。

find 函数的实现

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

    // 打开指定路径的文件或目录
    if ((fd = open(path, 0)) < 0)
    {
        fprintf(2, "find: cannot open %s\n", path);
        return;
    }

    // 获取文件或目录的状态信息
    if (fstat(fd, &st) < 0)
    {
        fprintf(2, "find1: cannot stat %s\n", path);
        close(fd);
        return;
    }

    // 根据文件或目录的类型进行处理
    switch (st.type)
    {
    case T_FILE:
        // 如果是文件，比较文件名是否与目标名称相同
        if (strcmp(target, path2name(path)) == 0)
        {
            printf("%s\n", path); // 输出文件路径
        }

        break;
```

```

case T_DIR:
    // 如果是目录，检查路径长度是否超出缓冲区范围
    if (strlen(path) + 1 + DIRSIZ + 1 > sizeof buf)
    {
        printf("find: path too long\n"); // 输出错误信息，路径过长
        break;
    }
    strcpy(buf, path); // 将路径复制到缓冲区
    p = buf + strlen(buf);
    *p++ = '/';

    // 遍历目录中的每个文件或子目录
    while (read(fd, &de, sizeof(de)) == sizeof(de))
    {
        if (de.inum == 0)
            continue; // 跳过空目录项

        memmove(p, de.name, DIRSIZ); // 将文件或目录名复制到缓冲区末尾
        p[DIRSIZ] = 0; // 添加字符串结束符

        // 获取文件或目录的状态信息
        if (stat(buf, &st) < 0)
        {
            // 输出错误信息，无法获取文件或目录的状态信息
            printf("find2: cannot stat %s\n", buf);
            continue;
        }

        // 排除当前目录和上级目录的特殊情况
        if (strlen(de.name) == 1 && de.name[0] == '.')
            continue;
        if (strlen(de.name) == 2
            && de.name[0] == '.'
            && de.name[1] == '.')
            continue;
        find(buf, target); // 递归调用查找函数，继续查找子目录
    }
    break;
}

```

```
    close(fd); // 关闭文件或目录
}
```

3. 在主函数中调用 find 函数。
 4. 将程序写在 find.c 中，运行程序得到结果。
-

```
$ find . b
./a/b
./b
```

1.2.6 实现 xargs

1. 首先初始化一些变量。

初始化

```
int i;
int arg_count = 0; // 参数数量计数器
char *args[MAXARG]; // 参数数组，最大长度为MAXARG
int initial_arg_count = arg_count; // 存储初始参数数量的位置

int line_index = 0; // 当前行的索引
char input_char; // 用于读取字符
char *current_line; // 指向当前处理的行
char line_buffer[512]; // 临时存储字符串的缓冲区
current_line = line_buffer;

// 将命令行参数复制到参数数组args中
for (i = 1; i < argc; ++i)
{
    args[arg_count++] = argv[i];
}
```

2. 循环处理读取的字符。

输入处理循环

```
while (read(0, &input_char, 1) > 0)
{
    // 对输入字符的操作...
}
```

3. 循环中的字符处理:

- 对于普通字符, 将其加入到当前行末尾;
- 遇到空格, 表示一个参数已经结束, 将参数添加到参数列表之中;

空格的处理

```
current_line[line_index] = '\0'; // 添加字符串结束符
line_index = 0; // 重置索引
args[arg_count++] = current_line; // 将当前单词添加到参数数组
char line_buffer[512]; // 重新分配缓冲区
current_line = line_buffer;
```

- 遇到回车, 表示一行结束, 将最后一个参数加入参数列表之中并执行命令。

回车的处理

```
// 处理换行符, 将当前行作为参数
current_line[line_index] = '\0'; // 添加字符串结束符
line_index = 0; // 重置索引

args[arg_count++] = current_line; // 将当前行添加到参数数组
args[arg_count] = 0; // 设置参数数组的结束标志

if (fork()) // 创建子进程
{
    wait(0); // 父进程等待子进程结束
    arg_count = initial_arg_count; // 重置参数数量
}
else
{
    exec(argv[1], args); // 子进程执行命令
}
```

4. 将程序写入 xargs.c 中, 运行程序得到结果。

```
$ echo hello too | xargs echo bye
bye hello too
```

1.3 评测结果

利用 grade-lab-util 脚本评测, 得到结果见图1-2

```

(base) aaa@aaa-virtual-machine:~/xv-6/xv6-labs-2021$ ./grade-lab-util
make: "kernel/kernel"已是最新。
== Test sleep, no arguments == sleep, no arguments: OK (1.2s)
== Test sleep, returns == sleep, returns: OK (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
== Test pingpong == pingpong: OK (1.0s)
== Test primes == primes: OK (1.0s)
== Test find, in current directory == find, in current directory: OK (1.0s)
== Test find, recursive == find, recursive: OK (1.4s)
== Test xargs == xargs: OK (1.0s)
== Test time ==
time: OK
Score: 100/100

```

图 1-2: 评测结果

1.4 实验小结

本实验中我初步了解了 xv6 这个操作系统的基本结构。了解了其利用 qemu 模拟器运行、编程的基本方法。本实验中需要实现的多为基本的系统功能，让我对 Unix 操作系统有了更深入的了解。

此外 primes 是我认为这里面最难的程序。首先，理解这个程序的实现原理就花费了我不少时间。在之后实现这个程序的过程之中，我在处理管道、父子进程的关系的时候遇到了一些困难。不过最后我还是成功实现了程序，这对我理解管道、进程并行有很大的帮助。

2 Lab: System calls

2.1 实验目的

本实验旨在进一步熟悉系统调用，重点掌握如何添加系统调用，理解系统调用的工作原理，内核态和用户态的联系。

本实验实现了两个系统调用：

- System call tracing: 实现对系统调用的跟踪；
- Sysinfo: 收集有关正在运行的系统的信息。

2.2 实验步骤

2.2.1 实现 trace

1. 首先，根据提示，在 Makefile 的 UPROGS 中添加 \$U/_trace\。
2. 在 user.h 文件的 system calls 部分添加 trace 函数。

对 user.h 的改动

```
// system calls
int fork(void);
...
// 添加 trace函数
int trace(int);
```

3. 在 usys.pl 末尾添加 entry("trace"); 以便在汇编语言中添加这个函数。
4. 进入 kernel 文件夹，在 syscall.h 中添加 SYS_trace 系统调用号的定义，值往下类推，即为 22。

对 syscall.h 的更改

```
// System call numbers
#define SYS_fork 1
...
// 为trace添加系统调用号
#define SYS_trace 22
```

5. 更改进程结构体。在 proc.h 中的进程结构体最后添加一个掩码 trace_mask，用于记录哪些进程要被追踪。

对进程结构体的更改

```
struct proc
{
    struct spinlock lock;
    ...
    // 添加掩码
    int trace_mask;
};
```

6. 在 sysproc.c 文件中，添加系统调用函数 sys_trace，在调用时设置掩码。

实现 sys_trace 系统调用

```
uint64 sys_trace(void)
{
    // 获取参数
    int mask;
    if (argint(0, &mask) < 0)
        return -1;
    // 获取当前进程
    struct proc *p = myproc();
```

```

// 写入掩码
p->trace_mask = mask;
return 0;
}

```

7. 在 syscall.c 中修改 syscall 函数，使其根据掩码判断是否输出当前调用信息。

对 syscall 函数的修改

```

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]();

        // 获得掩码
        int trace_mask = p->trace_mask;

        // 如果掩码对应，输出当前系统调用信息
        if ((trace_mask >> num) & 1)
        {
            printf("%d: syscall %s -> %d\n",
                p->pid,
                syscall_names[num - 1],
                p->trapframe->a0);
        }
    }
    else
    {
        ...
    }
}

```

8. 修改 proc.c 中的 freeproc 函数和 fork 函数，确保掩码在进程释放时重置掩码、在子进程创建时传递掩码。

```
static void freeproc(struct proc *p)
{
    if (p->trapframe)
        kfree((void *)p->trapframe);
    ...

    // 添加了这一行
    p->trace_mask = 0;
}

int fork(void)
{
    ...

    // 把子进程的掩码也设置为与父进程相同
    np->trace_mask = p->trace_mask;

    ...
}
```

2.2.2 实现 sysinfo

1. 首先，先仿照 trace 中的方法，在 Makefile、syscall.h、syscall.c、user.h、user.pl 中注册名为 sysinfo 的系统调用。
2. 在 kalloc.c 中仿照 kalloc 函数读取空闲内存链表的方式，实现一个计算空闲内存空间大小的函数 acquire_freemem。

acquire_freemem 的实现

```
uint64 acquire_freemem()
{
    struct run *r;
    uint64 cnt = 0;

    // 锁
    acquire(&kmem.lock);
    r = kmem.freelist;
    // 遍历链表
    while (r)
```



```

{
    r = r->next;
    cnt++;
}
// 释放锁
release(&kmem.lock);
// 返回空间大小（页数乘页尺寸）
return cnt * PGSIZE;
}

```

3. 在 `proc.c` 中，实现一个计算非使用状态进程个数的函数 `acquire_nproc`，利用记录进程的数组 `struct proc proc[NPROC]`。

acquire_nproc 的实现

```

uint64 acquire_nproc()
{
    struct proc *p;
    int cnt = 0;

    for (p = proc; p < &proc[NPROC]; ++p)
    {
        acquire(&p->lock);
        if (p->state != UNUSED)
            cnt++;
        release(&p->lock);
    }

    return cnt;
}

```

4. 最后，在 `sysproc.c` 中完成 `sys_sysinfo` 函数的实现。利用 `argaddr` 读取参数的地址，利用 `copyout` 对地址进行写入。

sys_sysinfo 的实现

```

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

```

```

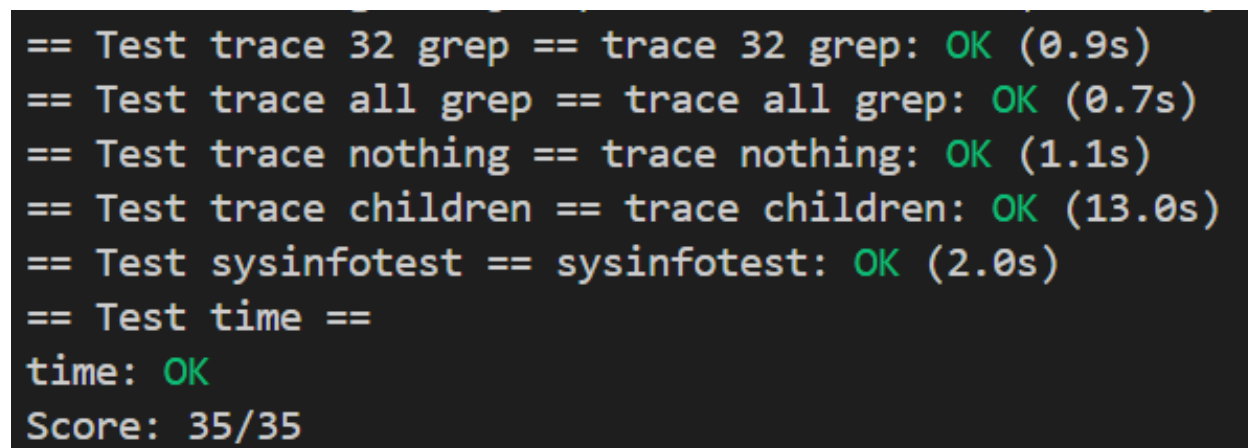
// 计算空闲进程数
info.nproc = acquire_nproc();
// 计算空闲内存数
info.freemem = acquire_freemem();

// 获取参数
if (argaddr(0, &addr) < 0)
    return -1;
// 写入结构体
if (copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
    return -1;
return 0;
}

```

2.3 评测结果

利用 grade-lab-syscall 脚本评测，得到评测结果如图2-3所示。



```

== Test trace 32 grep == trace 32 grep: OK (0.9s)
== Test trace all grep == trace all grep: OK (0.7s)
== Test trace nothing == trace nothing: OK (1.1s)
== Test trace children == trace children: OK (13.0s)
== Test sysinfotest == sysinfotest: OK (2.0s)
== Test time ==
time: OK
Score: 35/35

```

图 2-3: 评测结果

2.4 实验小结

在本实验中，我实现了两个系统调用。通过实验，我了解到了在内核中实现系统调用的一整套流程。

在实验中，为了实现我自己的系统调用，我阅读了许多 kernel 的代码，对内核有了更好的了解。通过仿照内核代码中一些函数的实现以及提示，我才得以完成我自己的系统调用。我在阅读内核代码的时候也遇到了一些困难，花费了不少时间来理解代码的含义。

这次实验大大加深了我对系统调用的了解，为之后的实验打下基础。

3 Lab: Page tables

3.1 实验目的

本实验中将涉及到操作系统中重要的**页表**概念。操作系统中的页表是一种数据结构，用于实现虚拟内存和物理内存之间的地址映射。每个进程都有自己的页表，将虚拟地址映射到物理地址，从而使得操作系统能够有效地管理内存，提供内存保护和共享内存功能。页表的概念对于操作系统课程至关重要，因为它涉及到内存管理、进程隔离和系统性能等核心问题。理解页表如何工作，有助于深入掌握操作系统的内存管理机制，了解虚拟内存的实现细节，并且能够解决与内存管理相关的各种问题。

本实验中将探索页表的具体操作并对其进行修改，以加快某些系统调用并检测哪些页面被访问。具体将实现以下内容：

- 利用页表加速系统调用；
- 打印页表；
- 检测被访问的页表。

3.2 实验步骤

3.2.1 实现系统调用加速

1. 首先，在 `proc.c` 文件中修改 `proc_pagetable` 函数，仿照前面的代码添加用户页表的映射，注意内存权限应为 `PTE_U`。

对 `proc_pagetable` 函数的修改

```
pagetable_t proc_pagetable(struct proc *p)
{
    pagetable_t pagetable;

    ...

    // 添加的部分
    if (mappages(pagetable, USYSCALL, PGSIZE,
                (uint64)(p->usyscall), PTE_R | PTE_U) < 0)
    {
        uvmunmap(pagetable, USYSCALL, 1, 0);
        uvmunmap(pagetable, TRAMPOLINE, 1, 0);
        uvmfree(pagetable, 0);
        return 0;
    }

    return pagetable;
}
```

2. 在 proc.c 文件中修改 allocproc 函数，仿照前面的代码，给进程分配用户页表。

对 allocproc 函数的修改

```
static struct proc * allocproc(void)
{
    ...

    // 添加的部分
    if ((p->usyscall = (struct usyscall *)kalloc()) == 0)
    {
        freeproc(p);
        release(&p->lock);
        return 0;
    }
    p->usyscall->pid = p->pid;

    ...

    return p;
}
```

3. 在 proc.c 文件中修改 freeproc 函数，仿照前面的代码，释放用户页表。

对 freeproc 函数的修改

```
freeproc(struct proc *p)
{
    ...
    // 添加的部分
    if (p->usyscall)
        kfree((void *)p->usyscall);
    p->usyscall = 0;
    ...
}
```

4. 如此设置，系统就能利用用户页表实现更快地进程号查询，从而加速系统调用。

3.2.2 实现页表打印

1. 首先，阅读 vm.c 文件中 freewalk 函数的代码，了解如何对页表项进行递归遍历和访问。

freewalk 函数的原型

```

// Recursively free page-table pages.
// All leaf mappings must already have been removed.
void freewalk(pagetable_t pagetable)
{
    // there are 2^9 = 512 PTEs in a page table.
    for (int i = 0; i < 512; i++)
    {
        pte_t pte = pagetable[i];
        if ((pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 0)
        {
            // this PTE points to a lower-level page table.
            uint64 child = PTE2PA(pte);
            freewalk((pagetable_t)child);
            pagetable[i] = 0;
        }
        else if (pte & PTE_V)
        {
            panic("freewalk: leaf");
        }
    }
    kfree((void *)pagetable);
}

```

2. 仿照 freewalk 函数，实现递归打印页表的 vmprint 函数

vmprint 函数的实现

```

void vmprint(pagetable_t pagetable, uint64 depth)
{
    // 如果递归深度大于2，则直接返回，RISC-V只有三级页表
    if (depth > 2)
        return;

    // 如果是最顶层页表，打印页表的地址
    if (depth == 0)
    {
        printf("page table %p\n", pagetable);
    }

    // 定义一个静态的前缀数组，用于打印格式化输出
    static char *prefix[] = {"..", ".. ..", ".. .. ."};
}

```

```

// 页表中有 $2^9 = 512$ 个页表项
for (int i = 0; i < 512; i++)
{
    pte_t pte = pagetable[i]; // 获取当前页表项
    if (pte & PTE_V) // 如果页表项有效
    {
        // 打印当前页表项的信息, 包括前缀、索引、页表项和物理地址
        printf("%s%d: pte %p pa %p\n", prefix[depth], i, pte, PTE2PA(pte));
        ;
        uint64 child = PTE2PA(pte); // 获取下一级页表的物理地址
        vmprint((pagetable_t)child, depth + 1); // 递归打印下一级页表
    }
}
}

```

3. 在 def.h 中声明 vmprint 函数以便调用。

4. 在 exec.c 文件中, 对 exec 函数进行修改, 调用 vmprint 函数, 使其在进程号为 1 是打印页表。

对 exec 函数的修改

```

int exec(char *path, char **argv)
{
    ...

    // 添加的语句
    if (p->pid == 1)
        vmprint(p->pagetable, 0);

    ...
}

```

3.2.3 实现被访问页面侦测

1. 首先, 在 riscv.h 文件中添加一个 PTE_A 常量, 查阅 RISC-V 手册可知其值应为 0b00100000, 即 2^6 。

PTE_A 的定义

```

#define PTE_A (1L << 6)

```

2. 在 vm.c 文件中实现 vm_pgaccess 函数，利用 PTE_A 标志判断给定虚拟地址的页表项是否被访问，若被访问返回 1 并清除 PTE_A 标志。在 def.h 中添加函数的声明。

vm_pgaccess 函数的实现

```
int vm_pgaccess(pagetable_t pagetable, uint64 va)
{
    pte_t *pte; // 页表项指针

    // 如果虚拟地址超出最大允许值，返回0
    if (va >= MAXVA)
        return 0;

    // 获取虚拟地址对应的页表项指针
    pte = walk(pagetable, va, 0);
    // 如果页表项不存在，返回0
    if (pte == 0)
        return 0;

    // 如果页表项的访问标志位被设置
    if ((*pte & PTE_A) != 0)
    {
        // 清除访问标志位
        *pte = *pte & (~PTE_A);
        // 返回1表示访问标志位被清除
        return 1;
    }

    // 如果访问标志位未设置，返回0
    return 0;
}
```

3. 在 sysproc.c 中补全 sys_pgaccess 系统调用，使之能够把页面访问的情况记录在掩码之中。

sys_pgaccess 函数的实现

```
int sys_pgaccess(void)
{
    uint64 addr; // 内存起始地址
    int len; // 内存区域的页数
    int bitmask; // 存储结果的用户地址
```

```

// 获取系统调用的第一个参数（内存起始地址），并检查是否成功
if (argaddr(0, &addr) < 0)
    return -1;

// 获取系统调用的第二个参数（内存区域的页数），并检查是否成功
if (argint(1, &len) < 0)
    return -1;

// 获取系统调用的第三个参数（存储结果的用户地址），并检查是否成功
if (argint(2, &bitmask) < 0)
    return -1;

// 检查页数是否在合理范围内（0到32页）
if (len > 32 || len < 0)
    return -1;

int res = 0; // 用于存储访问标志结果
int i;
struct proc *p = myproc(); // 获取当前进程结构体

// 遍历每一页，检查访问标志并存储结果
for (i = 0; i < len; i++)
{
    int va = addr + i * PGSIZE; // 计算每一页的虚拟地址
    int abit = vm_pgaccess(p->pagetable, va); // 检查页表项的访问标志

    res = res | abit << i; // 将访问标志结果合并到res中
}

// 将结果拷贝到用户空间，如果失败则返回-1
if (copyout(p->pagetable, bitmask, (char *)&res, sizeof(res)) < 0)
    return -1;

// 成功返回0
return 0;
}

```

3.3 评测结果

利用 grade-lab-pgtbl 脚本评测，得到评测结果如图3-4所示。

```
make: "kernel/kernel"已是最新。
== Test pgtbltest == (1.1s)
== Test   pgtbltest: ugetpid ==
   pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
   pgtbltest: pgaccess: OK
== Test pte printout == pte printout: OK (0.9s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test usertests == (243.4s)
== Test   usertests: all tests ==
   usertests: all tests: OK
== Test time ==
time: OK
Score: 46/46
```

图 3-4: 评测结果

3.4 实验小结

在本次实验中，我深入探索了操作系统中的页表概念，并通过一系列具体的实现来加深对该概念的理解。页表是操作系统中管理虚拟内存的重要数据结构，它通过将虚拟地址映射到物理地址，实现内存保护和内存共享功能。我通过本实验的操作，掌握了页表的操作方法和实现细节。

我在本实验上花费了不少时间，复习了操作系统课上讲过的页表概念，并研究内核代码。实验总体来说是有些难度。

在实验过程中，我不仅巩固了操作系统中页表的基本概念，还通过实际编程和调试，掌握了页表的具体实现方法和应用场景。通过这些实践操作，我更深入地理解了操作系统的内存管理机制，并积累了宝贵的编程经验和调试技巧。这些知识和技能对于我后续解决实际问题将大有裨益。

4 Lab: Traps

4.1 实验目的

这个实验将会探索系统调用是如何使用陷阱（trap）实现的。陷阱是一种特殊的处理机制，用于在计算机系统中捕获特定的事件或异常，并转移控制权给内核，从而确保系统能够安全地执行特权操作。理解陷阱机制对于掌握操作系统的核心功能和提升系统的安全性至关重要。实验内容具体包括：

- 了解 RISC-V 程序集，阅读 call.asm 中的函数 g、f 和 main 的代码，了解基础的汇编知识，并回答问题。
- 实现一个回溯（backtrace）功能，用于在操作系统内核发生错误时，输出调用堆栈上的函数调用列表。这有助于调试和定位错误发生的位置。
- 添加系统调用 sigalarm，周期性地为进程设置定时提醒。

4.2 实验步骤

4.2.1 了解 RISC-V 汇编

首先，利用 make fs.img 命令对 user 文件夹下的 call.c 文件进行编译，并生成 call.asm 汇编文件。阅读 call.asm 中的 g，f，和 main 函数，并对给出的问题做出回答。

call.asm

```
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
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: 7a050513  addi a0,a0,1952 # 7c8 <malloc+0xe8>
    30: 00000097  auipc ra,0x0
    34: 5f8080e7  jalr 1528(ra) # 628 <printf>
    exit(0);
    38: 4501  li a0,0
    3a: 00000097  auipc ra,0x0
    3e: 274080e7  jalr 628(ra) # 2ae <exit>

```

Q1: Which registers contain arguments to functions? For example, which register holds 13 in main's call to printf?

答: RISC-V 中一共有 a0 a7 一共 8 个寄存器。在 main 函数调用 printf 函数的过程中, 参数 13 被存储在了寄存器 a2 中。

Q2: Where is the call to function f in the assembly code for main? Where is the call to g?

答: 对于函数 f 调用是直接计算出了结果 12, 对于函数 g 的调用则是内联在了函数 f 中。

Q3: At what address is the function printf located?

答：查阅代码可知 printf 函数的地址是 0x628。

printf 函数

```
0000000000000628 <printf>:

void
printf(const char *fmt, ...)
{
    628: 711d  addi sp,sp,-96
    62a: ec06  sd ra,24(sp)
    62c: e822  sd s0,16(sp)
    62e: 1000  addi s0,sp,32
    630: e40c  sd a1,8(s0)
    632: e810  sd a2,16(s0)
    634: ec14  sd a3,24(s0)
    636: f018  sd a4,32(s0)
    638: f41c  sd a5,40(s0)
    63a: 03043823 sd a6,48(s0)
    63e: 03143c23 sd a7,56(s0)
    va_list ap;

    va_start(ap, fmt);
    642: 00840613 addi a2,s0,8
    646: fec43423 sd a2,-24(s0)
    vprintf(1, fmt, ap);
    64a: 85aa  mv a1,a0
    64c: 4505  li a0,1
    64e: 00000097 auipc ra,0x0
    652: dce080e7 jalr -562(ra) # 41c <vprintf>
}
    656: 60e2  ld ra,24(sp)
    658: 6442  ld s0,16(sp)
    65a: 6125  addi sp,sp,96
    65c: 8082  ret
```

Q4: What value is in the register ra just after the jalr to printf in main?

答：把运行到 jalr 处的 PC+4 存入 ra，也就是 0x38。

Q5: Run the following code. What is the output? The output depends on that fact that the RISC-V is little-endian. If the RISC-V were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

Code for Question

```
unsigned int i = 0x00646c72;
printf("H%x Wo%s", 57616, &i);
```

答：程序的输出是 He110 World。如果是大端序（big-endian），那么 i 应该设置为 0x726c6400 才能保证与小端序输出的内容相同，即“rld”。57616 是不需要改的，因为其十六进制无论大小端都为 0xe110。

Q6: In the following code, what is going to be printed after 'y='? Why does this happen?

Code for Question

```
printf("x=%d y=%d", 3);
```

答：它所输出的是寄存器 a2 的值，这是因为 printf 会从 a2 寄存器读取参数作为 y 的值。其具体值受先前调用的影响。

4.2.2 实现 Backtrace

1. 首先，在 defs.h 中添加 backtrace 函数的声明。

backtrace 函数的声明

```
void backtrace(void);
```

2. 根据题目要求，在 riscv.c 文件中添加一个 r_fp 函数，用于获取栈帧。

r_fp 函数的实现

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

3. 在 printf.c 文件中添加 backtrace 函数的实现。

backtrace 函数的实现

```
void backtrace(void)
```

```

{
    printf("backtrace:\n");

    // 获取当前栈帧指针
    uint64 fp = r_fp();

    // 计算当前栈帧指针所在页的起始地址
    uint64 base = PGROUNDUP(fp);

    // 当栈帧指针小于页的起始地址时，继续回溯
    while (fp < base)
    {
        // 打印前一个栈帧的返回地址
        printf("%p\n", *((uint64 *) (fp - 8)));

        // 更新栈帧指针为前一个栈帧的指针
        fp = *((uint64 *) (fp - 16));
    }
}

```

4.2.3 实现 alarm

1. 首先，修改 user.pl、user.h、syscall.h、syscall.c，添加两个系统调用 sigalarm 和 sigreturn，方法与前文相同。sigreturn 不接受任何参数，而 sigalarm 为 int sigalarm(int ticks, void (*handler)())。
2. 修改 proc.h 中的进程结构体，添加一些新成员，用于记录程序的中断信息。

对进程结构体的修改

```

struct proc
{
    ...

    int ticks; // 记录进程运行的tick
    int ticks_cnt; // tick计数
    uint64 handler; // 处理程序的入口地址
    struct trapframe tick_trapframe; // 记录进程运行上下文
    int handler_executing; // 记录当前处理程序是否允许的状态量
};

```

3. 修改 proc.c 中的 allocproc 函数，添加对结构体新成员的初始化。

```
static struct proc *
allocproc(void)
{
    ...

    p->ticks = 0;
    p->handler_executing = 0;

    return p;
}
```

4. 在 sysproc.c 中添加 sigalarm 和 sigreturn 函数的实现。实现中断程序的跳转和恢复。

sys_sigalarm 的实现

```
uint64 sys_sigalarm(void)
{
    int ticks; // 定义变量ticks, 用于存储定时器的滴答数
    uint64 handler; // 定义变量handler, 用于存储信号处理函数的地址

    // 获取第一个参数的值并赋给ticks
    argint(0, &ticks);
    // 获取第二个参数的值并赋给handler
    argaddr(1, &handler);

    // 获取当前进程的指针
    struct proc *p = myproc();
    // 设置当前进程的定时器滴答数
    p->ticks = ticks;
    // 设置当前进程的信号处理函数
    p->handler = handler;
    // 初始化滴答计数器为0
    p->ticks_cnt = 0;

    // 返回0, 表示成功
    return 0;
}
```

```
// 定义sys_sigreturn函数，返回类型为uint64
uint64 sys_sigreturn(void)
{
    // 获取当前进程的指针
    struct proc *p = myproc();

    // 将当前进程的trapframe恢复为tick_trapframe
    *(p->trapframe) = p->tick_trapframe;

    // 设置handler_executing为0，表示处理函数执行完毕
    p->handler_executing = 0;

    // 返回0，表示成功
    return 0;
}
```

5. 修改 trap.c 文件中的 usertrap 函数，定义操作系统处理中断的行为。

对 usertrap 函数的修改

```
void usertrap(void)
{
    ...

    // 如果这是一个时钟中断，则可能需要让出 CPU
    if (which_dev == 2)
    {
        // 检查进程是否设置了定时器
        if (p->ticks > 0)
        {
            p->ticks_cnt++; // 增加定时器计数

            // 如果信号处理程序未在执行中且计数器超过了设定的滴答数
            if (p->handler_executing == 0 && p->ticks_cnt > p->ticks)
            {
                p->ticks_cnt = 0; // 重置计数器
                p->tick_trapframe = *(p->trapframe); // 保存当前的 trapframe
                p->trapframe->epc = p->handler; // 设置程序计数器为信号处理程序
                                                的地址
                p->handler_executing = 1; // 标记信号处理程序正在执行
            }
        }
    }
}
```



```

    }
}

// 放弃 CPU，使其他进程能够运行
yield();
}

// 恢复到用户模式继续执行
usertrapret();
}

```

4.3 评测结果

利用 grade-lab-traps 脚本评测，得到评测结果如图4-5所示。fig:syscall 所示。

```

• (base) aaa@aaa-virtual-machine:~/xv-6/xv6-labs-2021$ ./grade-lab-traps
make: "kernel/kernel"已是最新。
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test == backtrace test: OK (0.9s)
== Test running alarmtest == (4.7s)
== Test alarmtest: test0 ==
alarmtest: test0: OK
== Test alarmtest: test1 ==
alarmtest: test1: OK
== Test alarmtest: test2 ==
alarmtest: test2: OK
== Test usertests == usertests: OK (228.1s)
== Test time ==
time: OK
Score: 85/85

```

图 4-5: 评测结果

4.4 实验小结

本次实验通过实现系统调用和陷阱机制，深入理解了操作系统的核心功能。

这些操作增强了我对计算机体系结构、异常处理机制和系统调用流程的理解。通过实际编写和测试系统调用，我更好地理解了用户态和内核态之间的交互机制，以及系统调用是如何在硬件和操作系统之间传递控制权的。实现陷阱机制和 backtrace 功能，使我熟悉了如何捕获和处理异常，以及如何在错误发生时进行有效的调试和错误定位。添加定时提醒功能（sigalarm）使我了解了如何在操作系统中实现高级特性，从而提高系统的可用性和可靠性。通过设置定时器，可以更好地理解系统中断。

总的来说，本次实验不仅增强了我对操作系统原理的理解，还提升了我在实际系统开发中的调试和错误处理能力。

5 Lab: Copy-on-Write Fork for xv6

5.1 实验目的

本实验旨在促进了解写时复制（Copy-On-Write, COW）机制，同时增进对 fork 函数分配内存机制的理解。在 xv6 操作系统内核中实现 COW 机制。通过实现 COW 机制，可以在 fork 系统调用时避免直接复制父进程的所有内存页，从而提高内存使用效率。当子进程尝试写入共享页面时，再复制相应的页面。

5.2 实验步骤

1. 首先，修改 riscv.h 文件，添加一个页表标志位 PTE_COW，用于记录页表是否处于 COW 状态。

对 riscv.h 文件的修改

```
#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // 1 -> user can access
#define PTE_COW (1L << 8) // 1 -> cow page
```

2. 在 kalloc.c 文件中添加一个数组 int reference[PHYSTOP / PGSIZE] 记录页表引用数，并设置一个锁确保对数组的安全操作。

添加记录引用数的数组和锁

```
int reference[PHYSTOP / PGSIZE];
struct spinlock refcountlock;
```

3. 修改 kalloc 函数和 kfree 函数，实现对 reference 数组的初始化和页面释放。

对 kalloc 函数的修改

```
void *kalloc(void)
{
    struct run *r;

    acquire(&kmem.lock);
    r = kmem.freelist;
```

```

// 如果有空闲页面，分配给请求者
if (r)
{
    kmem.freelist = r->next;
    // 添加部分
    acquire(&refcountlock);
    reference[((uint64)r) / PGSIZE] = 1;
    release(&refcountlock);
}
release(&kmem.lock);

if (r)
    memset((char *)r, 5, PGSIZE); // fill with junk
return (void *)r;
}

```

对 kfree 函数的修改

```

void kfree(void *pa)
{
    struct run *r;

    if (((uint64)pa % PGSIZE) != 0
        || (char *)pa < end
        || (uint64)pa >= PHYSTOP)
        panic("kfree");

    // 添加部分，减少引用计数
    acquire(&refcountlock);
    reference[((uint64)pa) / PGSIZE]--;
    release(&refcountlock);
    if (reference[((uint64)pa) / PGSIZE] > 0)
        return;

    // Fill with junk to catch dangling refs.
    memset(pa, 1, PGSIZE);

    r = (struct run *)pa;

    acquire(&kmem.lock);

```

```

    r->next = kmem.freelist;
    kmem.freelist = r;
    release(&kmem.lock);
}

```

4. 修改 trap.c 文件，添加一个 cowhandler 函数并修改 usertrap 函数，当发生缺页异常，为进程复制一个新物理页面。

cowhandler 函数的实现

```

int cowhandler(pagetable_t pagetable, uint64 va)
{
    char *mem;

    // 检查虚拟地址是否超过最大有效地址
    if (va >= MAXVA)
        return -1;
    // 获取页表条目 (PTE)
    pte_t *pte = walk(pagetable, va, 0);
    if (pte == 0)
        return -1;
    // 检查PTE是否有效、用户模式、COW标志
    if ((*pte & PTE_V) == 0 || (*pte & PTE_U) == 0 || (*pte & PTE_COW) == 0)
        return -1;
    // 分配一个新的物理页面
    if ((mem = kalloc()) == 0)
        return -1;
    // 获取原物理地址
    uint64 pa = PTE2PA(*pte);
    // 将原物理页面内容复制到新页面
    memmove((char *)mem, (char *)pa, PGSIZE);
    // 释放原物理页面
    kfree((void *)pa);
    // 获取PTE的标志位
    uint flags = PTE_FLAGS(*pte);
    // 更新PTE指向新物理页面，并设置为可写
    *pte = (PA2PTE(mem) | flags | PTE_W);
    // 清除COW标志
    *pte &= ~PTE_COW;

    return 0;
}

```

```
}
```

对 usertrap 函数的修改

```
void usertrap(void)
{
    ...

    // 检查陷阱原因
    if (r_scause() == 8)
    {
        ...
    }
    else if (r_scause() == 15)
    {
        // 处理Copy-On-Write (COW) 错误
        uint64 va = r_stval();
        if (va >= p->sz)
            p->killed = 1;
        int ret = cowhandler(p->pagetable, va);
        if (ret != 0)
            ->killed = 1;
    }
    else if ((which_dev = devintr()) != 0)
    {
        // 设备中断
        // ok
    }
    else
    {
        ...
    }

    ...

    usertrapret();
}
```

-
5. 在 vm.c 文件中修改 copyout 函数，添加 COW 页面的处理逻辑。

对 copyout 函数的修改

```

int copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
    uint64 n, va0, pa0;

    while (len > 0)
    {
        ...

        // 如果是Copy-On-Write页面, 处理COW逻辑
        if (checkcowpage(va0, pte, p))
        {
            char *mem;

            // 分配一个新的物理页面
            if ((mem = kalloc()) == 0)
            {
                p->killed = 1;
            }
            else
            {
                // 将原页面的数据复制到新页面
                memmove(mem, (char *)pa0, PGSIZE);

                // 获取页表条目的标志位
                uint flags = PTE_FLAGS(*pte);

                // 解除原页面映射
                uvmunmap(pagetable, va0, 1, 1);

                // 更新页表条目, 指向新页面并设置为可写
                *pte = (PA2PTE(mem) | flags | PTE_W);

                // 清除COW标志
                *pte &= ~PTE_COW;

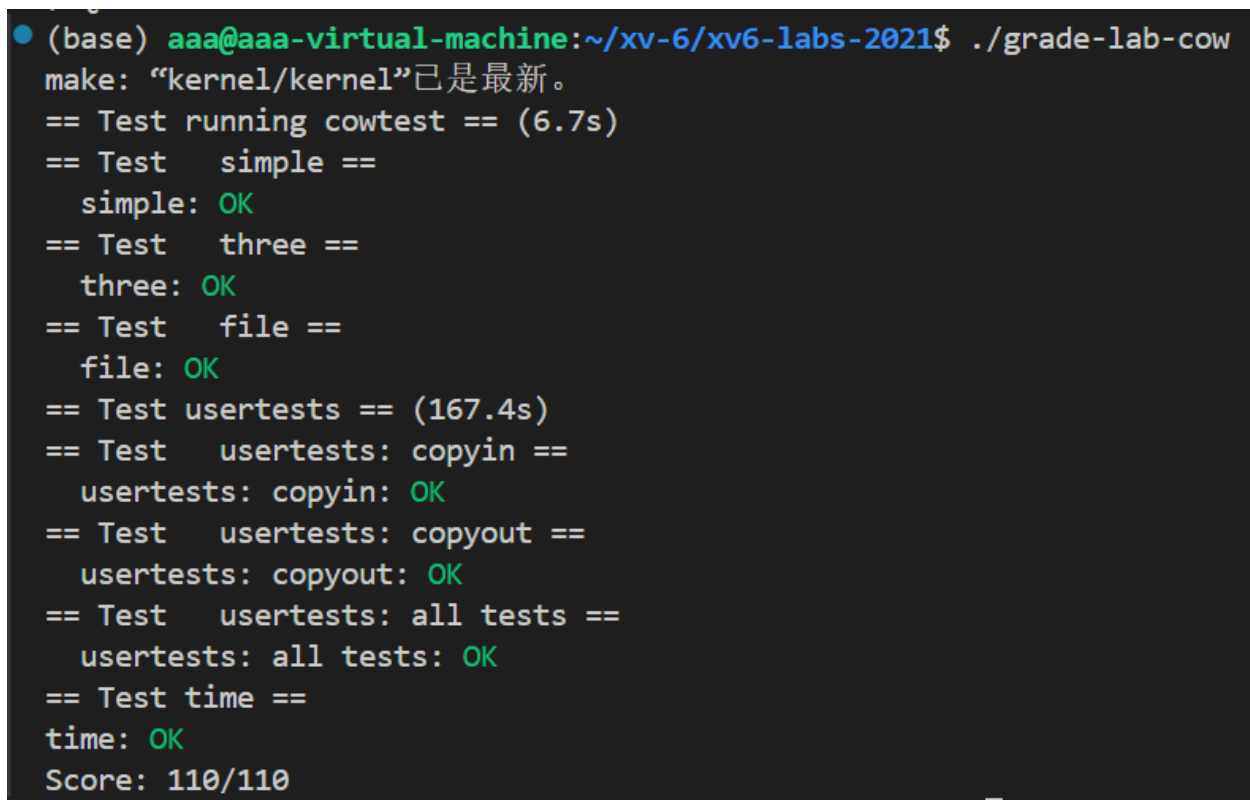
                // 更新物理地址
                pa0 = (uint64)mem;
            }
        }
    }
}

```

```
    ...  
}  
  
return 0;  
}
```

5.3 评测结果

利用 grade-lab-cow 脚本评测，得到结果如图5-6。



```
● (base) aaa@aaa-virtual-machine:~/xv-6/xv6-labs-2021$ ./grade-lab-cow  
make: "kernel/kernel"已是最新。  
== Test running cowtest == (6.7s)  
== Test    simple ==  
    simple: OK  
== Test    three ==  
    three: OK  
== Test    file ==  
    file: OK  
== Test usertests == (167.4s)  
== 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
```

图 5-6: 评测结果

5.4 实验小结

本次实验通过在 xv6 操作系统内核中实现写时复制（Copy-On-Write, COW）机制，加深了我对内存管理和进程创建的理解。在传统的 fork 系统调用中，父进程的所有内存页都会被直接复制给子进程，而 COW 机制通过延迟实际的内存复制操作，提高了内存使用效率。

通过实验，我不仅掌握了 COW 机制的基本原理，还增强了对操作系统内存管理和进程管理的理解，提升了在操作系统内核中进行实际编程和调试的能力。这些技能对于理解和开发高效的操作系统具有重要意义。

6 Lab: Multithreading

6.1 实验目的

本实验旨在进一步了解多线程编程，在操作系统中实现线程执行和切换等操作。实验将在用户级线程包中实现线程之间的切换，使用多个线程来加速程序，并实现 barrier。

6.2 实验步骤

6.2.1 实现用户线程切换

1. 修改 uthread.c 文件：

- 添加一个 context 结构体保存线程上下文的寄存器。

context 结构体

```
struct context
{
    uint64 ra;
    uint64 sp;

    uint64 s0;
    uint64 s1;
    uint64 s2;
    uint64 s3;
    uint64 s4;
    uint64 s5;
    uint64 s6;
    uint64 s7;
    uint64 s8;
    uint64 s9;
    uint64 s10;
    uint64 s11;
};
```

- 在 thread 结构体中添加一个保存上下文的 context 结构体成员。

对 thread 结构体的修改

```
struct thread
{
    char stack[STACK_SIZE];
    int state;
    struct context context; /* 线程上下文 */
};
```

```
};
```

- 修改 thread_schedule 函数实现线程切换。

对 thread_schedule 函数的修改

```
void thread_schedule(void)
{
    struct thread *t, *next_thread;

    ...

    if (current_thread != next_thread) // 如果需要切换线程
    {
        next_thread->state = RUNNING; // 设置下一个线程为运行状态
        t = current_thread; // 保存当前线程
        current_thread = next_thread; // 切换到下一个线程
        // 切换线程上下文
        thread_switch((uint64)&t->context, (uint64)&next_thread->context);
    }
    else
        next_thread = 0; // 如果没有切换线程, 则将next_thread置为0
}
```

- 修改 thread_create 函数初始化 context 结构体, 设置栈顶指针和函数指针。

对 thread_create 函数的修改

```
void thread_create(void (*func)())
{
    struct thread *t;

    for (t = all_thread; t < all_thread + MAX_THREAD; t++) // 查找空闲的线程结构
    {
        if (t->state == FREE) // 如果找到空闲线程
            break; // 跳出循环
    }
    t->state = RUNNABLE; // 设置线程状态为可运行

    //添加的部分
    t->context.sp = (uint64)&t->stack + STACK_SIZE; // 初始化线程栈顶指针
    t->context.ra = (uint64)func; // 设置返回地址为函数指针
}
```

}

2. 修改 `uthread_switch.S` 文件，实现 `thread_switch` 函数，保存当前线程的寄存器并恢复新线程的寄存器。

实现 `thread_switch` 汇编

```
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 */
```

6.2.2 使用 UNIX pthread 线程库实现一个线程安全的哈希表

这部分实现不需要使用 xv6 操作系统部分的代码。首先观察 ph.c 文件，编译运行，发现程序在单线程运行时，哈希表没有出现丢失 key 的情况，而在多线程运行时就会发生丢失 key 的情况。经过观察，发现 put 函数中的插入操作不是线程安全的，通过为这部分代码加入互斥锁便可解决问题。

1. 在全局变量中加入一个互斥锁。

声明互斥锁

```
struct entry *table[NBUCKET];
int keys[NKEYS];
int nthread = 1;
// 添加的互斥锁
pthread_mutex_t lock[NBUCKET];
```

2. 在 main 函数的开头初始化互斥锁。

初始化互斥锁

```
int main(int argc, char *argv[])
{
    ...

    for (int i = 0; i < NBUCKET; i++)
    {
        pthread_mutex_init(&lock[i], NULL);
    }

    ...
}
```

3. 在 put 函数中使用互斥锁，确保线程安全。

对 put 函数的修改

```
static void put(int key, int value)
{
    int i = key % NBUCKET;

    // is the key already present?
    struct entry *e = 0;
    for (e = table[i]; e != 0; e = e->next)
    {
        if (e->key == key)
```

```

        break;
    }
    if (e)
    {
        // update the existing key.
        e->value = value;
    }
    else
    {
        // 添加互斥锁
        pthread_mutex_lock(&lock[i]);
        insert(key, value, &table[i], table[i]);
        pthread_mutex_unlock(&lock[i]);
    }
}

```

6.2.3 实现 barrier

barrier 函数的作用是在并发编程中同步多个线程，使得它们在某个点上等待，直到所有线程都到达该点，然后再继续执行。这确保了所有线程在某个阶段结束前不会提前进入下一个阶段。

对 barrier 函数的实现

```

static void barrier()
{
    pthread_mutex_lock(&bstate.barrier_mutex); // 加锁保护屏障状态

    bstate.nthread++; // 已到达屏障的线程数加1

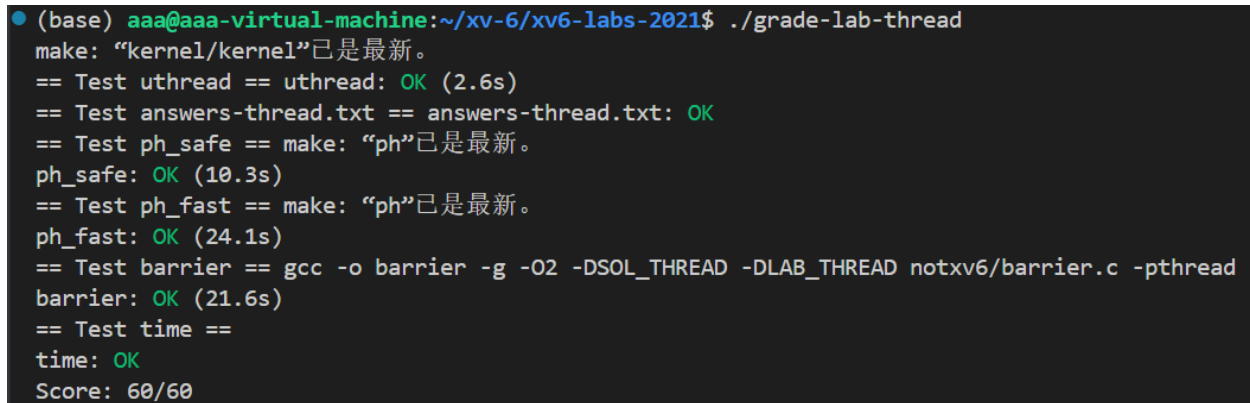
    if (bstate.nthread == nthread)
    { // 如果所有线程都到达屏障
        bstate.nthread = 0; // 重置已到达线程数
        bstate.round++; // 增加屏障轮次
        pthread_cond_broadcast(&bstate.barrier_cond); // 唤醒所有等待线程
    }
    else
    {
        // 等待其他线程到达屏障
        pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
    }
}

```

```
pthread_mutex_unlock(&bstate.barrier_mutex); // 解锁  
}
```

6.3 评测结果

利用 grade-lab-thread 脚本评测，得到结果如图6-7所示。



```
● (base) aaa@aaa-virtual-machine:~/xv-6/xv6-labs-2021$ ./grade-lab-thread  
make: "kernel/kernel"已是最新。  
== Test uthread == uthread: OK (2.6s)  
== Test answers-thread.txt == answers-thread.txt: OK  
== Test ph_safe == make: "ph"已是最新。  
ph_safe: OK (10.3s)  
== Test ph_fast == make: "ph"已是最新。  
ph_fast: OK (24.1s)  
== Test barrier == gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread  
barrier: OK (21.6s)  
== Test time ==  
time: OK  
Score: 60/60
```

图 6-7: 评测结果

6.4 实验小结

为什么使用单线程运行哈希表就不会丢失 key，而双线程就会丢失？

有两个线程时，缺失键的问题是因为存在竞态条件。多个线程同时访问和修改共享数据（哈希表）时，会导致不可预测的结果。例如，线程 1 和线程 2 同时检查同一个桶，并发现键不存在，随后都尝试插入该键，这会导致其中一个插入操作被覆盖，从而导致缺失键。使用互斥锁（mutex）来保护对哈希表的访问可以解决这个问题。

总结

本次实验深入探讨了多线程编程的实现与应用，涵盖了用户级线程切换、线程安全数据结构以及同步机制的实现。通过这些实现，我们不仅掌握了多线程编程的基本技术，还理解了线程同步和数据一致性的重要性。这些技能对于开发高效、可靠的并发程序具有重要意义。本实验为将来的复杂系统开发奠定了坚实基础。