

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	2
1.1	实验目的	2
1.2	实验步骤	2
1.2.1	运行 xv6	2
1.2.2	实现 sleep	3
1.2.3	实现 pingpong	3
1.2.4	实现 prime	4
1.2.5	实现 find	7
1.2.6	实现 xargs	9
1.3	评测结果	10
1.4	实验小结	11
2	Lab: System calls	11
2.1	实验目的	11
2.2	实验步骤	11
2.2.1	实现 trace	11
2.2.2	实现 sysinfo	14
2.3	评测结果	16

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和开发流程，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。

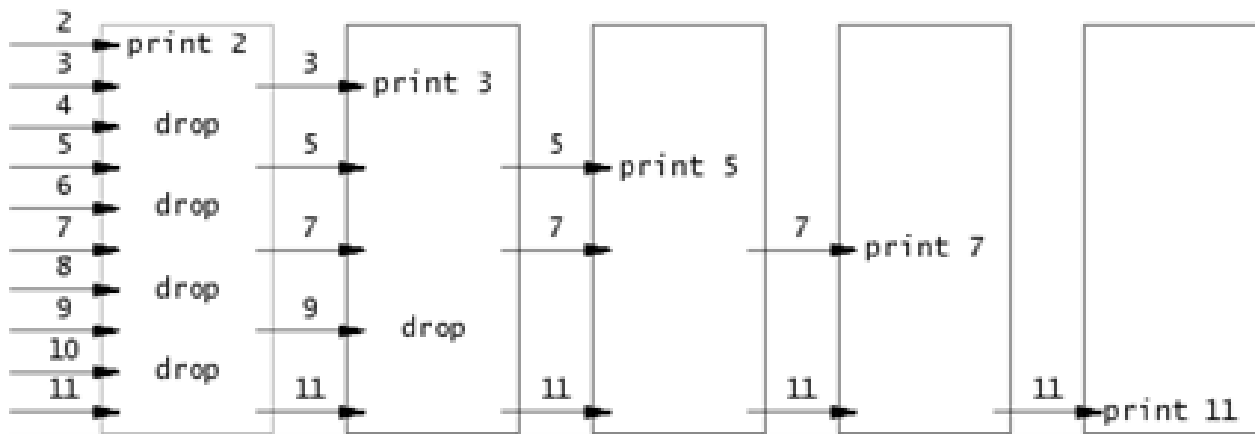


图 1-1: primes 程序原理

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_freeman。

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();
}

```

```

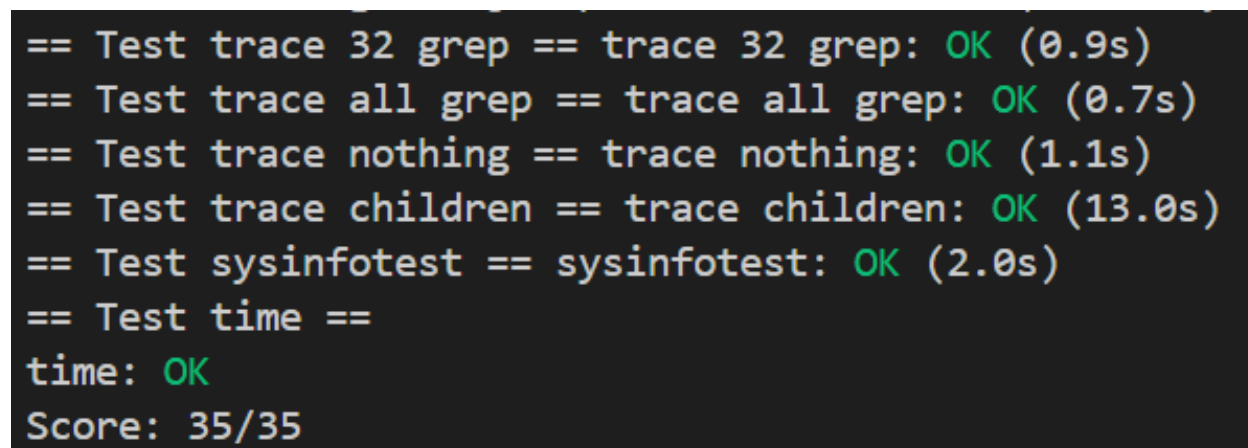
// 计算空闲进程数
info.nproc = acquire_nproc();
// 计算空闲内存数
info.freemem = acquire_freemem();

// 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。这次试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。