Lab - Xv6 and Unix utilities 目录

1. Boot xv6 (easy)

1.1 实验目的

- 学习如何设置实验环境并编译运行xv6内核
- 了解如何用QEMU进行内核的模拟
- 了解git的基本用法

1.2 实验步骤

1. 安装qemu、gcc、GDB、git等依赖项

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

2. 克隆xv6实验的Git仓库到本地，并进入该目录

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

3. 检查状态

```
$ git status
```

输出应为：

```
On branch util
Your branch is up to date with 'origin/util'.

nothing to commit, working tree clean
```

此外，可以通过查看日志来发现一些与旧版本的差异：

```
$ git log
```

4. 在GitHub上创建名为`xv6-labs-TJ-SSE-OS`的项目。考虑到这一实验仅作为子项目，在本地将其放入更高一级的文件夹下，并删除其目录下的`.git`文件夹（否则会作为子模块导致上传失效）。输入以下命令进行初始化：

```
$ git init
$ git add .
$ git commit -m 'First Upload Test'
$ git branch -M main
```

值得注意的是，接下来需要先把SSH公钥添加到GitHub帐户，才能继续进行：

```
$ git remote add origin git@github.com:Aloz14/xv6-labs-TJ-SSE-OS.git
$ git push -u origin main
```

而后便发现本地仓库内容已经同步到GitHub上了。

5. 使用QEMU在项目目录下构建并运行 xv6：

```
$ make qemu
```

运行 make qemu 命令后，将进行编译和链接操作，最终启动 qemu 模拟器。

运行成功应当看到类似如下的显示：

```
riscv64-unknown-elf-gcc    -c -o kernel/entry.o kernel/entry.S
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -
DSOL_UTIL -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-
relax -I. -fno-stack-protector -fno-pie -no-pie   -c -o kernel/start.o
kernel/start.c
...
riscv64-unknown-elf-ld -z max-page-size=4096 -N -e main -Ttext 0 -o
user/_zombie user/zombie.o user/ulib.o user/usys.o user/printf.o
user/umalloc.o
riscv64-unknown-elf-objdump -S user/_zombie > user/zombie.asm
riscv64-unknown-elf-objdump -t user/_zombie | sed '1,/SYMBOL TABLE/d; s/ .*
/ /; /^$/d' > user/zombie.sym
mkfs/mkfs fs.img README  user/xargstest.sh user/_cat user/_echo
user/_forktest user/_grep user/_init user/_kill user/_ln user/_ls
user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind
user/_wc user/_zombie
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1)
blocks 954 total 1000
balloc: first 591 blocks have been allocated
balloc: write bitmap block at sector 45
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
```

6. 使用`ls`命令查看：

```
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2227
xargstest.sh   2 3 93
cat            2 4 32864
echo           2 5 31720
forktest       2 6 15856
grep           2 7 36240
init           2 8 32216
kill           2 9 31680
ln             2 10 31504
ls             2 11 34808
mkdir          2 12 31736
rm             2 13 31720
sh             2 14 54168
```

```
stressfs        2 15 32608
usertests       2 16 178800
grind           2 17 47528
wc              2 18 33816
zombie          2 19 31080
console         3 20 0
```

7. 按下Ctrl+A后再按下X键以退出QEMU模拟器。

1.3 实验中遇到的问题和解决办法

- **问题**：使用Git提交至远程仓库时，遇到鉴权失败的情况。

  解决方法：若是通过HTTP协议进行传输，GitHub在2021年就停止了使用密码进行验证的方式，需要使用用户名+Token的方式进行验证。更好的方式是使用SSH进行传输，只需在GitHub账户设置中配置服务器的公开密钥完成配对即可。

- **问题**：提交至远程仓库后，GitHub上无法打开项目文件夹。

  解决方法：在克隆原项目仓库后，其目录下默认存在一个`.git`文件夹和相关配置文件，在高一级项目文件夹的git预备过程中，会将其视为一个子模块，采取“忽视”的方法进行处理。只需删除git相应文件即可。

1.4 实验心得

通过本次实验，我成功地启动了xv6内核，并且能够在命令行界面进行基本操作。我学会了使用Git进行代码版本控制，能够使用常见的Git命令进行提交和查看代码变更。在实验过程中，我遇到了一些小问题，但通过仔细阅读文档和查找解决方法，我成功地解决了这些问题。这次实验使我更加熟悉了xv6内核的构建和运行过程，并为以后的实验打下了基础。

2. sleep (easy)

2.1 实验目的

- 在xv6系统中实现UNIX程序sleep
- 熟悉内核函数的使用
- 熟悉项目的文件结构和逻辑关系

2.2 实验步骤

1. 阅读xv6文档书的第1章。
2. 查看`user/`目录中的其他一些程序（例如`user/echo.c`、`user/grep.c`和`user/rm.c`），以了解如何获取传递给程序的命令行参数。查看xv6内核代码`kernel/sysproc.c`，其中实现了sleep系统调用，以及`user/user.h`中定义的从用户程序调用的sleep的C语言定义，还有`user/usys.s`中的汇编代码，用于从用户代码跳转到内核执行sleep。
3. 编写sleep程序的代码，创建`user/sleep.c`并在其中实现，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) {
        fprintf(2, "Only 1 argument is needed!\nUsage: sleep <ticks>\n");
        exit(-1);
    }
    sleep(atoi(argv[1]));
    exit(0);
}
```

4. 将sleep程序添加到`Makefile`的`UPROGS`列表中，这样运行`make qemu`命令时将编译新增的sleep程序，修改后如下：

```
UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $U/_grep\
    $U/_init\
    $U/_kill\
    $U/_ln\
    $U/_ls\
    $U/_mkdir\
    $U/_rm\
    $U/_sh\
    $U/_stressfs\
    $U/_usertests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_sleep\ # 新增的sleep函数
```

5. 在xv6 shell中运行程序进行测试，结果如下：

```
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-utilities$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -g
lobal virtio-mmio.force-legacy=false -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 1 starting
hart 2 starting
init: starting sh
$ sleep 100
```

6. 运行以下命令以进行grade：

```
./grade-lab-util sleep
```

结果如下：

```
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-utilities$ ./grade-lab-util sleep
make: “kernel/kernel”已是最新。
== Test sleep, no arguments == sleep, no arguments: OK (1.2s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-utilities$
```

2.3 实验中遇到的问题和解决方法

- **问题：** 使用grade进行测试时，提示无法连接到QEMU

  解决方法： 误删除了项目目录下的`.gdbinit.tmpl-riscv`等一些文件，从仓库拉取备份即可。

- **问题：**打开QEMU模拟后使用sleep报错，提示`exec sleep failed`

  解决方法： 打开Makefile文件，找到UPROGS变量，并将sleep程序的名称添加到列表中。

- **问题：**QEMU模拟进行编译时出错，提示出现了未定义的类型

  解决方法： `user/user.h`使用到了`kernel/types.h`中的定义，因此在包含头文件时`kernel/types.h`必须放在`user/user.h`的前面。鉴于clang-format会对include进行排序，需要在其格式化配置文件`.clang-format`中设置`SortIncludes`为false。

2.4 实验心得

通过完成本实验，我学到了如何在xv6系统中调用内核函数实现一个简单的sleep程序。我了解了命令行参数的处理、系统调用的使用以及在用户程序和内核之间进行跳转的方法。本实验对于加深对操作系统和系统编程的理解非常有帮助。通过实践，我更好地掌握了xv6系统的构建和编程技巧。

3. pingpong (easy)

3.1 实验目的

- 在xv6系统中实现UNIX程序pingpong
- 加深对管道、进程的理解
- 熟悉`wait()`、`read()`等函数的使用

3.2 实验步骤

1. 编写pingpong实验程序，使用`pipe`创建管道，使用`fork`创建子进程，使用`read`从管道中读取字节，使用`write`将字节写入管道，使用`getpid`获取调用进程的进程ID，实现父子进程之间的字节交换和输出。代码如下：

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

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

    if (argc > 1) {
        fprintf(2, "No argument is needed!\n");
        exit(1);
    }

    int p_to_c[2], c_to_p[2];
    char buf[10];
```

```
    if (pipe(p_to_c) < 0) {
        printf("pipe create failed\n");
        exit(1);
    }

    if (pipe(c_to_p) < 0) {
        printf("pipe create failed\n");
        exit(1);
    }

    int pid;
    pid = fork();

    if (pid < 0) {
        printf("fork failed\n");
        exit(1);
    }

    if (pid == 0) {
        // 子进程
        close(c_to_p[0]);
        close(p_to_c[1]);
        // 从管道读取字节
        if (read(p_to_c[0], buf, 4) == -1) {
            printf("child process read failed\n");
            exit(1);
        }
        close(p_to_c[0]);

        printf("%d: received %s\n", getpid(), buf);

        strcpy(buf, "pong");
        // 向管道发送字节
        if (write(c_to_p[1], buf, 4) == -1) {
            printf("child process write failed\n");
            exit(1);
        }
        close(c_to_p[1]);
        exit(0);
    }
    else {
        strcpy(buf, "ping");
        // 父进程
        close(c_to_p[1]);
        close(p_to_c[0]);
        // 向管道发送字节
        if (write(p_to_c[1], buf, 4) == -1) {
            printf("parent process write failed\n");
            exit(1);
        }

        // 等待子进程结束
        wait(0);

        close(p_to_c[1]);
        // 从管道读取字节
        if (read(c_to_p[0], buf, 4) == -1) {
            printf("parent process read failed\n");
```

```
            exit(1);
        }

        close(c_to_p[0]);
        printf("%d: received %s\n", getpid(), buf);

        exit(0);
    }
}
```

2. 同sleep实验，将pingpong程序添加到`Makefile`的`UPROGS`列表中，运行，结果如下：

```
init: starting sh
$ pingpong
4: received ping
3: received pong
$ QEMU: Terminated
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-utilities$ ./grade-lab-util pingpong
make: “kernel/kernel”已是最新。
== Test pingpong == pingpong: OK (0.6s)
```

3.3 实验中遇到的问题和解决方法

- **问题：** 调试时错误输出是一堆打乱顺序的字符

  解决方法： 问题在于同时启动了两个线程进行输出，所以导致了乱序的字符，控制好运行逻辑即可。

- **问题：** `read()`和`write()`两个函数的返回值始终为-1

  解决方法： 原因在于没有正确理解管道的第一个元素为读取端，第二个元素为写入端。纠正即可。

3.4 实验心得

通过完成本实验，我学会了如何使用UNIX系统调用在父子进程之间进行通信。我了解了管道的创建和使用，以及在父子进程之间传递字节的方法。本实验对于加深对操作系统和系统编程的理解非常有帮助。通过实践，我更好地掌握了xv6系统的构建和编程技巧。

4. primes (moderate)/(hard)

4.1 实验目的

- 编写一个使用管道实现并发的素数筛选器

4.2 实验步骤

1. 编写primes实验程序，程序核心代码如下：

```
void recursive_com(int left_pipe[])
{
    int prime;
    int is_read = read(left_pipe[FDS_READ], &prime, sizeof(prime));
    if (is_read == 0) {
        close(left_pipe[FDS_READ]);
        exit(1);
    }
    else if (is_read == -1) {
        printf("read failed\n");
        exit(1);
```

```
    }
    else {
        printf("prime %d\n", prime);
    }

    int p2c[2];
    if (pipe(p2c) < 0) {
        printf("pipe create failed\n");
        exit(1);
    }

    int pid;
    pid = fork();

    if (pid < 0) {
        printf("fork failed\n");
        exit(1);
    }

    if (pid == 0) {
        // 子进程
        close(p2c[FDS_WRITE]);
        close(left_pipe[FDS_READ]);
        close(left_pipe[FDS_WRITE]);
        int ttt = 0;
        if (read(p2c[FDS_READ], &ttt, sizeof(ttt)) == -1) {
            printf("read is closed");
        }
        else {
            printf("read is not closed");
        }
        recursive_com(p2c);
    }
    else {
        // 父进程
        int tmp;
        while (read(left_pipe[FDS_READ], &tmp, sizeof(tmp)) > 0) {
            if (tmp % prime != 0) {
                int flag = write(p2c[FDS_WRITE], &tmp, sizeof(tmp));
                if (flag == -1) {
                    printf("write %d failed\n", tmp);
                    exit(1);
                }
            }
        }
        close(left_pipe[FDS_READ]);
        close(p2c[FDS_WRITE]);
        wait(0);
    }

    exit(0);
}
```

这是一个递归函数，从左侧管道获取数字，通过筛选后，发送到右侧管道。

2. 同上述实验，将程序添加到`Makefile`的`UPROGS`列表中，运行，结果如下：

```
init: starting sh
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
$ QEMU: Terminated
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-utilities$ ./grade-lab-util primes
make: “kernel/kernel”已是最新。
== Test primes == primes: OK (0.6s)
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-utilities$
```

4.3 实验中遇到的问题和解决方法

- **问题：** 出现无法预估的错误，将数字3发送到新的递归函数时，读取失败，`read()`函数的返回值为-1

  解决方法： 问题在于在父进程中已经关闭了管道的写入端，在子进程中若再次关闭，由于xv6系统的某些机制，会关闭另一管道的写入端，导致无法预估的错误。

4.4 实验心得

本次实验在编写代码时遇见了极不寻常的错误，但在长时间排查后最终发现是由于重复关闭某一管道端口而导致的越界行为，导致另一管道的端口被关闭。通过此次实验，我更加深化了对管道、进程的认知，对xv6内核有了更为深入的了解。

5. find (moderate)

5.1 实验目的

- 实现简单程序，查找目录树中具有特定名称的所有文件
- 加深对xv6文件结构的理解

5.2 实验步骤

1. 查看user/ls.c文件，了解如何读取目录。其中有两个函数：
   - `void ls(char *path)`：用于显示指定路径下的所有文件（包含文件夹）
   - `char *fmtname(char *path)`：提取出给定路径的文件名并返回规范化字符串
2. 根据上述两个函数进行改写，编写`find()`核心函数如下：

   对于`char *fmtname(char *path)`函数，进行了改写，使得返回的字符串后不自动补齐空格：

```
char *fmtname(char *path)
{
    static char buf[DIRSIZ + 1];
    char *p;

    // Find first character after last slash.
    for (p = path + strlen(path); p >= path && *p != '/'; p--)
        ;
    p++;

    // Return blank-padded name.
```

```
    if (strlen(p) >= DIRSIZ)
        return p;
    memmove(buf, p, strlen(p));
    memset(buf + strlen(p), ' ', DIRSIZ - strlen(p));
    for (char *i = buf + strlen(buf);; i--) {
        if (*i != '\0' && *i != ' ' && *i != '\n' && *i != '\r' && *i != '\t') {
            *(i + 1) = '\0';
            break;
        }
    }
    return buf;
}
```

对于`void ls(char *path)`函数进行改写，不直接打印所有文件，而是打印符合条件的文件名：

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

    if ((fd = open(path, 0)) < 0) {
        fprintf(2, "find: cannot open %s\n", path);
        return;
    }

    if (fstat(fd, &st) < 0) {
        fprintf(2, "find: cannot stat %s\n", path);
        close(fd);
        return;
    }

    switch (st.type) {
    case T_FILE:
        if (strcmp(fmtname(path), filename) == 0) {
            printf("%s\n", path);
        }
        break;

    case T_DIR:
        if (strlen(path) + 1 + DIRSIZ + 1 > sizeof(buf)) {
            fprintf(2, "find: path too long\n");
            break;
        }
        strcpy(buf, path);
        p = buf + strlen(buf);
        *p++ = '/';

        while (read(fd, &de, sizeof(de)) == sizeof(de)) {
            if (de.inum == 0 || strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0) {
                continue;
            }
            memmove(p, de.name, DIRSIZ);
```

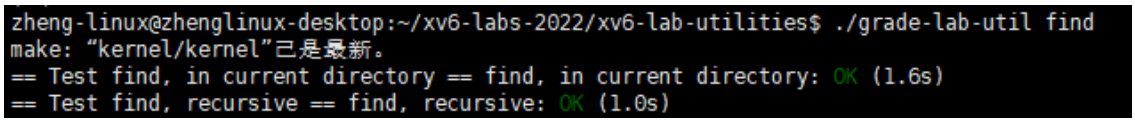
```
            p[DIRSIZ] = 0;
            if (stat(buf, &st) < 0) {
                fprintf(2, "find: cannot stat %s\n", buf);
                continue;
            }
            find(buf, filename);
        }
        break;
    }

    close(fd);
}
```

3. 将 `find` 程序添加到 `Makefile` 的 `UPROGS` 列表中。
4. 利用以下命令创建相应文件和文件夹进行测试：

```
$ echo > b
$ mkdir a
$ echo > a/b
$ mkdir a/aa
$ echo > a/aa/b
```

结果如下：

进行grade测试，结果如下：



5.3 实验中遇到的问题和解决方法

- **问题：** 输入 `find . b` 无结果

  解决方法： 对于该问题事实上先后是由两种原因导致：1. `fmtname()` 函数返回的是一个规范化的字符串，长为32，后用空格补齐，因此直接使用 `strcmp()` 比较无法得到预期值，需要改写 `fmtname()` 函数。2.在使用 `make qemu` 命令进行模拟编译得到的虚拟机中，每一次都是依据本地文件夹重新生成的，因此上一次实验中创建的文件和文件夹不会保留。

5.4 实验心得

通过完成本实验，我学会了如何编写一个简单的UNIX的find程序，实现对目录树中特定名称文件的查找。我了解了目录的读取、递归算法和字符串比较的方法。本实验对于加深对文件系统和目录操作的理解非常有帮助。

6. xargs (moderate)

6.1 实验目的

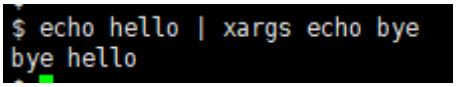
- 编写xargs程序
- 了解shell中管道 | 的原理和使用方法

6.2 实验步骤

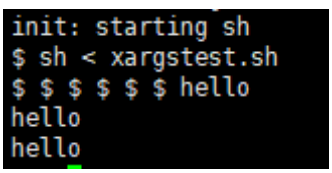
1. 使用 `fork` 和 `exec` 来调用每行输入的命令。使用 `wait` 函数在父进程中等待子进程完成命令的执行。编写实验代码如下：

```
char *args[MAXARG];
    alloc_args(args, MAXARG, MAXARGLEN);
    int retval;
    char tmp;
    int arg_num = 0, arg_len = 0;
    args[arg_num] = argv[1];
    arg_num++;

    for (int i = 2; i < argc; i++) {
        strcpy(args[arg_num], argv[i]);
        arg_num++;
    }

    while ((retval = read(STIN, &tmp, sizeof(tmp))) > 0 && arg_num < MAXARG)
{
        if (tmp == '\n' || tmp == ' ') {
            args[arg_num][arg_len] = '\0';
            arg_len = 0;
            arg_num++;
        }
        else {
            args[arg_num][arg_len] = tmp;
            arg_len++;
        }
    }

    // 置NULL
    args[arg_num] = 0;

    if (fork() == 0) {
        exec(args[0], args);
        free_args(args, arg_num);
    }
    else {
        wait(0);
        free_args(args, arg_num);
    }
```

2. 将 `xargs` 程序添加到 `Makefile` 的 `UPROGS` 列表中，执行测试，结果如下：



3. 运行grade程序，结果如下：



6.3 实验中遇到的问题和解决方法

- **问题：** 最后只输出了一行`hello`

  解决方法： 问题在于误解了题目的要求，面对`find . b`返回的多行输出，遇到换行应该继续往下读取，直到没有内容可读取。

- **问题：** 多次尝试`exec()`均失败

  解决方法： 参数列表指针数组末尾应当有个空指针，否则exec读取会失败。

6.4 实验心得

通过完成本实验，我学会了如何编写一个简单的UNIX xargs程序，实现对标准输入行的处理并调用命令执行。我熟悉了fork和exec函数的使用、标准输入的读取以及命令执行的处理。更理解了shell中程序的调用逻辑，对深入理解操作系统起到了较大帮助。