Lab1-Xv6-and-Unix-utilities

Boot xv6

实验目的

- 学习配置实验环境
- 学习使用QEMU模拟器运行和调试操作系统
- 学习使用Git进行版本控制，掌握基本的Git操作，如克隆仓库、提交修改、查看日志和比较差异。

实验步骤

1. 安装必要依赖项

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

2. 克隆远程Xv6实验仓库

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

3. 查看git状态

```
$ git status
```

输出如下

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

nothing to commit, working tree clean
```

4. 查看版本差异

```
$ git log
```

输出如下：

```
        commit dc9153fcb9c9b762dfaae9f586aecdaf04fb68fe (HEAD -> util, origin/util,
origin/HEAD)
Author: Frans Kaashoek <kaashoek@mit.edu>
Date:   Mon Sep 19 09:27:22 2022 -0400

    Fix year

commit 4d7493893dcc8834c34518b511620ea24e0cca57
Author: Frans Kaashoek <kaashoek@mit.edu>
Date:   Sun Aug 28 10:18:51 2022 -0400

    2022 submission site

commit 0c477a6c84091415efe2ac15509adc4b518bd7ea
Merge: 7d47335 3d6ce9b
Author: Frans Kaashoek <kaashoek@mit.edu>
Date:   Thu Aug 25 13:52:48 2022 -0400

    Merge branch 'riscv' into util
```

5. 在自己的github上创建代码仓库，并同步代码

6. 使用QEMU模拟器运行Xv6，并使用ls命令

```
$ make qemu
```

输出如下：

```
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -
nographic -global 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
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2227
xargstest.sh   2 3 93
cat            2 4 32352
echo           2 5 31224
forktest       2 6 15328
grep           2 7 35704
init           2 8 32016
kill           2 9 31224
ln             2 10 31144
ls             2 11 34264
mkdir          2 12 31264
rm             2 13 31248
sh             2 14 53496
stressfs       2 15 32120
usertests      2 16 180608
grind          2 17 47328
wc             2 18 33344
zombie         2 19 30792
sleep          2 20 31136
console        3 21 0
```

7. 退出QEMU模拟器，Ctrl+A+X

实验中遇到的困难及解决办法

在 Ubuntu 中执行 `sudo apt-get install` 命令时，经常出现哈希错误并需要多次执行才能成功安装。这是由于网络连接不稳定或缓慢可能导致下载的软件包时出错，从而出现哈希校验失败的问题。安装过程中出现错误，可以尝试清理缓存并重新安装，直到安装成功。

实验心得

通过这次实验，我不仅掌握了基本的Git操作，还深入了解了QEMU模拟器的使用和操作系统的基本运行原理。在解决问题的过程中，我学会了如何查阅资料和文档，增强了自主学习和解决问题的能力。实验过程中遇到的困难和挑战，让我意识到在实际工作中，耐心和细致是非常重要的。

总的来说，这次实验对我来说是一次非常有价值的学习经历，它不仅提高了我的技术水平，还增强了我对操作系统和版本控制工具的信心。在未来的学习和工作中，我将继续深入研究这些工具和技术，不断提升自己的能力。

Sleep

实验目的

- **理解Xv6内核中的时间概念**：通过实现 `sleep` 程序，深入理解Xv6内核中“tick”的概念，即两次定时器中断之间的时间间隔。
- **熟悉命令行参数的处理**：学习如何在程序中获取和处理命令行参数，如果用户没有提供参数，程序需要输出错误信息。
- **使用系统调用**：通过调用Xv6内核提供的 `sleep` 系统调用，实现程序的暂停功能。
- **程序编写和编译**：编写 `user/sleep.c` 文件中的 `sleep` 程序，并将其添加到Makefile中的UPROGS部分，以便通过 `make qemu` 命令编译并在Xv6 shell中运行。

实验步骤

1. **阅读相关资料**：
    - 在开始编码前，先阅读Xv6书籍的第一章，了解基本概念。
2. **查看示例程序**：
    - 查阅 `user/` 目录下的其他程序（例如 `user/echo.c` 、 `user/grep.c` 和 `user/rm.c` ），了解如何获取和处理命令行参数。
3. **编写 `sleep` 程序**：
    - 在 `user/` 目录下创建 `sleep.c` 文件。
    - 使用 `atoi` 函数将命令行参数从字符串转换为整数。
    - 调用Xv6内核提供的 `sleep` 系统调用，暂停指定的时间。
    - 在程序结束时调用 `exit(0)` 。

    sleep.c代码如下

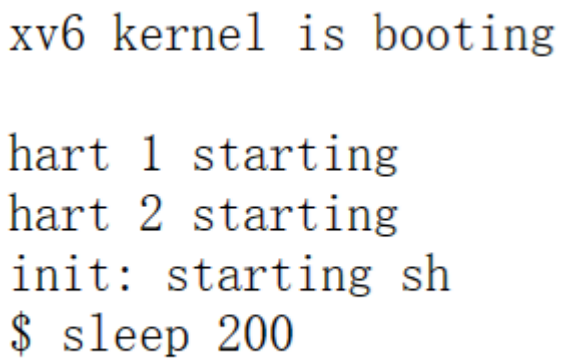
```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(int argc, char *argv[])
{
    // 检查参数数量，如果不是2个（包括程序名称和时间参数），则输出错误信息并退出
    if (argc != 2) {
        fprintf(2, "Usage: sleep <ticks>\n"); // 提示正确的使用方法
        exit(1); // 使用1作为错误代码
    }

    // 将命令行参数从字符串转换为整数，并调用sleep系统调用暂停执行
    int ticks = atoi(argv[1]);
    if (ticks < 0) { // 检查输入的ticks是否为负数
        fprintf(2, "Error: ticks should be a non-negative integer.\n");
        exit(1);
    }

    sleep(ticks);
    exit(0); // 正常退出程序
}
```

4. **编辑Makefile**：
    - 将`sleep`程序添加到Makefile的UPROGS部分，以便通过`make qemu`命令编译程序。
    - 添加后Makefile文件内容如下：

```
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\
```

5. **编译并运行程序**：

- 运行`make qemu`命令编译程序并启动QEMU模拟器。
- 在Xv6 shell中运行`sleep`程序，验证其是否按照指定时间暂停执行。
- 测试结果如下：



6. **测试程序**：

- 使用`make grade`命令运行所有测试，验证`sleep`程序的正确性。
- 或者使用`./grade-lab-util sleep`命令单独运行`sleep`程序的测试，确保其通过所有测试。
- 运行结果如下：

```
(base) root@876277deb135:~/workspace/personal_data/zwc/Xv6/xv6-labs-2022# ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.4s)
== Test sleep, returns == sleep, returns: OK (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
```

实验中遇到的困难及解决办法

调用`sleep`系统调用时，无法正确暂停程序。解决办法是检查`sleep`系统调用的实现是否正确，可以通过查阅`kernel/sysproc.c`文件中的`sleep`系统调用实现，确保其逻辑正确。此外，确保在Makefile中正确添加了`sleep`程序，注意注意，记得保存修改后的Makefile文件！！以便在编译时包含它。

实验心得

实现xv6系统中的`sleep`命令的实验，让我体会到了正确处理命令行参数的重要性，这是确保程序健壮性和用户友好性的关键步骤。我在程序中首先进行了参数数量的检查。这是一个基本而重要的步骤，通过判断`argc`的值，我确保用户输入了正确数量的参数。如果参数数量不正确，程序会通过标准错误输出提示正确的使用方法，并以错误代码`1`退出。这种做法可以防止用户由于错误的输入导致程序执行异常，同时也提供了清晰的用户引导，帮助他们正确使用程序。通过这次实验，我加深了对命令行程序处理逻辑的理解，尤其是如何通过代码处理和响应用户输入。虽然只是实现了一个简单的`sleep`命令，但实际上涵盖了许多基本但重要的编程技能，如参数解析、输

入验证、错误处理和用户交互。这不仅让我巩固了C语言的应用技能，也让我对编写更为复杂的系统级程序充满了信心。

---

pingpong

实验目的

- **掌握进程创建（Fork）**：使用 `fork()` 系统调用来创建一个子进程。这是UNIX和类UNIX系统中进程管理的基本方式。在这个实验中，父进程和子进程将通过管道通信。
- **理解管道通信（Pipes）**：管道是UNIX系统中用于进程间通信的一种机制。通过创建管道，进程可以单向或双向传输数据。本实验要求创建两个管道，一个用于父进程向子进程发送数据，另一个用于子进程回送数据给父进程。
- **熟悉使用数据读写（Read/Write）**：通过 `read()` 和 `write()` 系统调用，在管道中读取和发送数据。这些函数允许进程从管道中读取数据以及向管道写入数据，是实现进程间通信的关键操作。
- **理解进程标识（PID）**：使用 `getpid()` 系统调用获取当前进程的进程标识号（PID）。这在多进程编程中非常重要，特别是在需要区分不同进程或在进程间传递信息时。

实验步骤

编写实验程序，具体步骤如下：

- **初始化管道**：使用 `pipe()` 系统调用创建两个管道。每个管道由一对文件描述符（file descriptors）标识，一个用于读取，一个用于写入。你需要两个管道，一个让父进程向子进程发送数据，另一个让子进程回送数据给父进程。
- **分叉进程**：使用 `fork()` 系统调用复制当前进程，创建一个子进程。`fork()` 调用后，你会有两个几乎相同的进程：父进程和子进程。
- **配置管道**：
  - 父进程关闭它不需要的管道端口（子进程的写端和父进程的读端）。
  - 子进程关闭它不需要的管道端口（父进程的写端和子进程的读端）。
- **发送和接收数据**：
  - 父进程通过其管道的写端发送一个字节（ 'ping'）给子进程。
  - 子进程通过其管道的读端读取这个字节，打印其PID和接收到的消息（“received ping”）。

    - 子进程通过其管道的写端发送一个字节（ 'pong'）给父进程。
    - 父进程读取子进程发送的字节，并打印其PID和接收到的消息（“received pong”）。
- **结束进程**：
    - 子进程在发送数据后退出。
    - 父进程在接收数据后退出。
- 代码如下：

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

int main(int argc, char *argv[])
{
    // 确保没有提供不必要的命令行参数
    if (argc > 1) {
        fprintf(2, "No argument is needed!\n");
        exit(1);
    }

    int parent2child[2], child2parent[2]; // 分别代表父到子和子到父的管道  0是读端 1是写端
    char buf[5]; // 只需要足够的空间存储 "ping" 或 "pong"

    // 创建两个管道
    if (pipe(parent2child) < 0 || pipe(child2parent) < 0) {
        printf("pipe creation failed\n");
        exit(1);
    }
    // 在父进程中，fork() 返回新创建的子进程的进程ID（PID），这个值大于0。
    // 在子进程中，fork() 返回0。
    int pid = fork();

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

    if (pid == 0) { // 子进程代码
        close(parent2child[1]); // 关闭不需要的写端
        close(child2parent[0]); // 关闭不需要的读端

        // 从父进程读取字节
        if (read(parent2child[0], buf, sizeof(buf)) != 4) {
            printf("child process read failed\n");
            exit(1);
        }
        printf("%d: received %s\n", getpid(), buf);
        close(parent2child[0]); // 关闭读端

        // 回应父进程
        if (write(child2parent[1], "pong", 4) != 4) {
            printf("child process write failed\n");
            exit(1);
        }
        close(child2parent[1]); // 关闭写端
        exit(0);
    } else { // 父进程代码
```

```
        close(parent2child[0]); // 关闭不需要的读端
        close(child2parent[1]); // 关闭不需要的写端

        // 向子进程发送字节
        if (write(parent2child[1], "ping", 4) != 4) {
            printf("parent process write failed\n");
            exit(1);
        }
        close(parent2child[1]); // 关闭写端

        // 等待子进程结束
        wait(0);

        // 从子进程读取字节
        if (read(child2parent[0], buf, sizeof(buf)) != 4) {
            printf("parent process read failed\n");
            exit(1);
        }
        printf("%d: received %s\n", getpid(), buf);
        close(child2parent[0]); // 关闭读端

        exit(0);
    }
}
```

和Sleep实验一样，将pingpong.c加入Makefile文件中以编译使用，修改后的Makefile文件如下：

```
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\
    $U/_pingpong\
```

编译并运行程序，结果如下：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ pingpong
4: received ping
3: received pong
$
```

实验中遇到的困难及解决办法

如果父进程和子进程都在等待对方先发送数据，而没有释放任何资源，可能会导致死锁。 **解决办法**是确保进程间通信的顺序和时机是明确的，避免相互等待的情况。，比如子进程的代码先读后写，则父进程的代码先写后读即可

实验心得

在这次"ping-pong"进程间通信实验中，我深刻体会到了管道通信机制的实用性和效率。通过使用 `pipe()` 和 `fork()` 系统调用，我能够在父进程和子进程之间顺利地交换信息。这个实验让我理解到在设计并发系统时，如何重要的是确保通信的清晰和一致性。处理管道创建、数据传输和异常管理时，我遇到了一些挑战，例如确保在正确的时间关闭不必要的管道端。解决这些问题不仅增强了我的问题解决能力，还让我更加熟悉了UNIX环境下的系统级编程。整个过程是极具启发性的，它不仅提升了我的技术技能，还加深了我对操作系统底层工作原理的理解。

---

primes

实验目的

- **并发编程理解与实践**：通过实现一个并发版本的素数筛（Eratosthenes筛），学习如何使用并发处理提高程序的效率和性能。
- **进程间通信（IPC）**：学习如何使用 Unix 管道作为进程间通信的工具，这包括数据的传递和同步。

实验步骤

- **读取素数**：函数从左侧管道读取一个整数，这个整数是前一个筛选进程已经确认的素数。
- **输出素数**：读取的素数被打印出来，表示这是一个确定的素数。
- **管道和进程创建**：为每个确认的素数创建一个新的管道，并通过 `fork()` 创建一个新的进程。

- **子进程**:
  - 关闭不需要的管道写端。
  - 调用自身递归，继续素数筛选的工作。
- **父进程**:
  - 关闭子进程的管道读端。
  - 从左侧管道读取数据，筛选出非当前素数的倍数，将其写入新管道。
  - 关闭所有打开的管道和等待子进程完成。
- **退出条件**：当读操作返回0，即没有更多数据可读时，进程将关闭文件描述符并退出。

代码实现如下：

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


void findPrimes(int leftPipe[]) {
    int prime;
    // 从左侧管道读取素数
    int isRead = read(leftPipe[0], &prime, sizeof(prime));
    if (isRead == 0) {
        close(leftPipe[0]); // 关闭读端
        exit(0); // 无数据读取时退出
    } else if (isRead == -1) {
        printf("read failed"); // 读取失败
        exit(1);
    } else {
        printf("prime %d\n", prime); // 打印找到的素数
    }

    int parent2child[2];
    if (pipe(parent2child) < 0) {
        printf("pipe creation failed"); // 创建管道失败
        exit(1);
    }

    int pid = fork();
    if (pid < 0) {
        printf("fork failed"); // 创建进程失败
        exit(1);
    }

    if (pid == 0) {
        // 子进程逻辑
        close(parent2child[1]); // 关闭写端
        findPrimes(parent2child); // 递归调用以创建新的筛选进程
    } else {
        // 父进程逻辑
        close(parent2child[0]); // 关闭子进程的读端
        int tmp;
        while (read(leftPipe[0], &tmp, sizeof(tmp)) > 0) {
            if (tmp % prime != 0) {
                if (write(parent2child[1], &tmp, sizeof(tmp)) == -1) {
                    printf("write failed"); // 写入失败
                    exit(1);
                }
            }
        }
        close(leftPipe[0]); // 关闭读端
        close(parent2child[1]); // 关闭写端
        wait(0); // 等待子进程完成
```

```
    }

    exit(0);
}

int main(int argc, char *argv[]) {
    // 检查参数数量，确保不接受任何命令行参数
    if (argc > 1) {
        fprintf(2, "No argument is needed!\n");
        exit(1);
    }

    int nums[36];  // 创建数组以存储整数

    // 初始化整数数组
    for (int i = 2; i <= 35; i++) {
        nums[i] = i;
    }

    int parent2child[2];  // 管道数组
    // 创建管道
    if (pipe(parent2child) < 0) {
        printf("pipe creation failed");
        exit(1);
    }

    int pid = fork();  // 创建子进程
    if (pid < 0) {
        printf("fork failed");
        exit(1);
    }

    if (pid == 0) {
        // 子进程
        close(parent2child[1]);  // 子进程关闭写端
        findPrimes(parent2child);  // 调用递归处理函数
    } else {
        // 父进程
        close(parent2child[0]);  // 父进程关闭读端
        // 向管道写入整数
        for (int i = 2; i <= 35; i++) {
            if (write(parent2child[1], &nums[i], shi          close(parent2child[1]);
                wait(0);  // 等待子进程结束
                exit(1);
            }
        }
        close(parent2child[1]);  // 写入完成后关闭写端
        wait(0);  // 等待子进程结束
    }
    return 0;  // 正常退出
}
```

实验中遇到的困难及解决办法

使用 `wait(NULL)` 导致报错而使用 `wait(0)` 可以正常工作。 `wait()` 函数用于使父进程等待子进程结束，并回收子进程所使用的资源。如果您在使用 `wait(NULL)` 时遇到错误，需要确保包含正确的头文件如 `#include <sys/wait.h>`

实验心得

通过本次实验，我深刻体会到了进程间通信（IPC）技术的强大以及在实际编程中对资源管理的严格要求。通过使用Unix管道和进程创建实现的素数筛选算法让我更加理解了管道作为进程间通信手段的有效性。在编写和调试代码的过程中，我遇到了文件描述符管理和进程同步的挑战，特别是在正确使用 `wait()` 函数以确保父进程能够适当地等待子进程结束并回收资源方面。这次实验不仅加深了我的系统编程技能，也让我认识到在资源有限的系统中编程时，如何精确控制每一个系统调用的重要性。

---

find

实验目的

- **文件系统的遍历**：学习如何使用C语言来遍历目录和子目录，这是理解文件系统结构的基础。
- **递归函数的使用**：通过递归调用来搜索子目录，这有助于理解递归在实际编程中的应用。
- **字符串操作**：在C语言中处理字符串，特别是使用 `strcmp()` 函数来比较字符串，而不是使用 `==` 操作符，这是C语言中字符串操作的基本

实验步骤

- 学习 `user/ls.c` 文件
    - `char *fmtname(char *path)` 从一个给定的文件路径中提取出文件名部分，忽略路径中的目录部分。确保返回的文件名长度为 `DIRSIZ` 。如果文件名长度小于 `DIRSIZ` ，则用空格填充至 `DIRSIZ` 。
    - `void ls(char *path)` 用于列出指定目录下的所有文件。
- 改写 `char *fmtname(char *path)` 函数，使得返回的文件名后不自动补齐空格，否则在不同路径下的相同文件名无法通过 `strcmp()` 匹配成功

```c
// 改造ls.c中的fmtname,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];  // 静态缓冲区，用于存储和返回处理后的文件名
    char *p;

    // 寻找路径中最后一个斜杠后的第一个字符，即文件名的开始
    for (p = path + strlen(path); p >= path && *p != '/'; p--)
        ;
    p++;

    // 如果文件名的长度已经超过或等于 DIRSIZ，直接返回这个文件名
    if (strlen(p) >= DIRSIZ)
        return p;

    // 将文件名从 p 复制到 buf 中
    memmove(buf, p, strlen(p));

    // 在文件名后填充空格，直到达到 DIRSIZ 的长度
    memset(buf + strlen(p), ' ', DIRSIZ - strlen(p));

    // 在填充后的字符串中从后向前搜索，找到第一个非空格字符
    for (char *i = buf + strlen(buf);; i--) {
        if (*i != '\0' && *i != ' ' && *i != '\n' && *i != '\r' && *i != '\t') {
            *(i + 1) = '\0';  // 在这个字符后面放置字符串结束符 '\0'
            break;
        }
    }

    return buf;  // 返回处理后的字符串
}
```

- 改写`void ls(char *path)`,用于在指定目录及其子目录中搜索具有特定名称的文件

```c
// 定义find函数，用于在指定目录及其子目录中搜索具有特定名称的文件，从ls函数改造而来
void find(char *path, char *filename)
{
    char buf[MAX_PATH_LENGTH];  // 用于存储完整的文件或目录路径
    char *p;  // 辅助指针，用于操作路径字符串
    int fd;  // 文件描述符，用于打开目录
    struct dirent de;  // 目录项结构
    struct stat st;  // 文件状态结构

    // 尝试打开目录或文件
    if ((fd = open(path, 0)) < 0) {
        fprintf(2, "find: cannot open %s\n", path);  // 打开失败时输出错误信息
        return;
    }

    // 获取文件或目录的状态信息
    if (fstat(fd, &st) < 0) {
        fprintf(2, "find: cannot stat %s\n", path);  // 获取状态失败时输出错误信息
        close(fd);  // 关闭文件描述符
        return;
    }

    // 根据文件类型处理
    switch (st.type) {
    case T_FILE:  // 处理文件类型
        if (strcmp(fmtname(path), filename) == 0) {  // 比较处理后的文件名是否与目标文件名匹配
            printf("%s\n", path);  // 如果匹配，打印文件路径
        }
        break;

    case T_DIR:  // 处理目录类型
        // 检查路径长度是否超过缓冲区大小
        if (strlen(path) + 1 + DIRSIZ + 1 > sizeof(buf)) {
            fprintf(2, "find: path too long\n");  // 如果路径太长，输出错误信息
            break;
        }
        strcpy(buf, path);  // 将当前路径复制到缓冲区
        p = buf + strlen(buf);  // 设置指针到缓冲区末尾
        *p++ = '/';  // 在路径末尾添加斜杠，为拼接子目录或文件名做准备

        // 读取目录中的每个条目
        while (read(fd, &de, sizeof(de)) == sizeof(de)) {
            // 跳过无效的目录项以及 "." 和 ".."
            if (de.inum == 0 || strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0) {
                continue;
            }
            memmove(p, de.name, DIRSIZ);  // 将读取的目录项名称拼接到路径后
            p[DIRSIZ] = 0;  // 确保路径字符串正确结束
```

```
            if (stat(buf, &st) < 0) {
                fprintf(2, "find: cannot stat %s\n", buf);  // 如果无法获取子目录项的状
态，输出错误信息并继续
                continue;
            }
            find(buf, filename);  // 递归调用find函数，以检查子目录
        }
        break;
    }

    close(fd);  // 完成后关闭文件描述符
}
```

- 测试代码和结果如下：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ echo > b //新建名为b的空文件
$ mkdir a  // 新建名为a的空文件夹
$ echo > a/b // 在a文件夹下新建名为b的空文件
$ mkdir a/aa  // 在a文件夹下新建名为aa的空文件夹
$ echo > a/aa/b //在a/aa文件夹下新建名为b的空文件
$ find . b  // 查找所有名为b的文件
./b
./a/b
./a/aa/b
$
```

实验中遇到的问题及解决办法

直接使用`ls.c`中的`fmtname()`时`find . b`找不到任何结果，因为原来的fmtname()函数会将路径下的文件名使用空格自动补全，多余的空格导致文件名无法和单纯的`b`匹配成功。解决办法是改写原来的`fmtname()`函数的实现，把自动补全空格的逻辑去除。

实验心得

本次实验我深入学习了UNIX系统中的文件和目录操作，尤其是在实现简化版的`find`命令过程中，我遇到了一个具有挑战性的问题：原始的`fmtname()`函数由于使用空格填充导致文件名匹配失败。这个问题让我意识到在系统编程中处理字符串时的细节非常关键。通过修改`fmtname()`，去除了自动补齐空格的逻辑，我不仅解决了文件名匹配的问题，还学到了关于字符串处理的重要技术细节。这次实验不仅提升了我的编程技能，更加深了我对操作系统文件管理机制的理解，这是任何希望深入掌握系统编程的人必须经历的过程。

Xargs

实验目的

- **理解进程创建和管理**：通过使用 `fork`，`exec`，和 `wait` 系列函数，加深对操作系统进程管理方式的理解。
- **UNIX 工具的组合使用**：体验 UNIX 环境中多个工具（如 `xargs`，`find`，`grep`）联合使用的强大功能，理解其在实际应用中的灵活性和效率。

实验的步骤

- 自定义字符串复制函数 `my_strup`，功能类似于标准的 `strdup`，用于分配内存并复制字符串。
- `allocForArgs` 为每个参数分配内存。如果分配失败，则释放已分配的内存并退出程序。
- `freeArgs` 释放参数数组中所有字符串的内存并将指针设为 `NULL` 以防悬空指针。
- 程序首先检查是否提供了足够的命令行参数。
- 初始化参数数组，并为每个可能的参数预先分配内存。
- 将命令行参数复制到参数数组中。
- 通过标准输入读取额外参数，使用空格或换行符作为分隔。
- 使用 `fork` 创建一个子进程，在子进程中执行通过参数数组指定的命令。
- 父进程等待子进程执行结束后，清理所有动态分配的内存。
- 代码如下：

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "kernel/param.h"
#include "user/user.h"
#include <stddef.h>
#define STDIN 0
#define STDOUT 1
#define MAXARGLEN 32

char* myStrdup(const char* s) {
    if (s == NULL) return NULL;
    char* new_str = malloc(strlen(s) + 1);  // 加 1 为 null 终结符分配空间
    if (new_str == NULL) return NULL;
    strcpy(new_str, s);  // 复制字符串包括 null 终结符
    return new_str;
}


void allocForArgs(char *args[], int argNum, int argLen)
{
    for (int i = 0; i < argNum; i++) {
        args[i] = malloc(argLen * sizeof(char));
        if (args[i] == 0) {
            printf("Memory allocation failed for argument %d\n");
            // 如果分配失败，释放之前已分配的内存并退出
            while (i-- > 0) {
                free(args[i]);
            }
            exit(1);
        }
    }
}

void freeArgs(char *args[], int argNum)
{
    for (int i = 0; i < argNum; i++) {
        free(args[i]);
        args[i] = NULL;  // 避免悬空指针
    }
}


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

    // 确保至少提供了一个命令
    if (argc < 2) {
        fprintf(STDOUT, "Usage: %s <command> [args]\n");
        exit(1);
    }
```

```
    char *args[MAXARG];  // 存储命令和参数的数组
    allocForArgs(args, MAXARG, MAXARGLEN);  // 为参数数组分配内存
    int argNum = 0;  // 参数索引
    char buffer[MAXARGLEN];  // 用于临时存储标准输入的数据
    int bufferLen = 0;  // 缓冲区当前长度

    // 处理argv中的命令和参数
    for (int i = 1; i < argc && argNum < MAXARG; i++, argNum++) {
        strcpy(args[argNum], argv[i]);
        args[argNum][MAXARGLEN - 1] = '\0';  // 确保字符串结束
    }

    // 从标准输入读取额外的参数
    char ch;
    while (read(STDIN, &ch, 1) > 0 && argNum < MAXARG) {
        if (ch == '\n' || ch == ' ') {
            if (bufferLen > 0) {
                args[argNum] = myStrdup(buffer);  // 复制缓冲区到args
                bufferLen = 0;  // 重置缓冲区长度
                if (++argNum >= MAXARG) break;  // 检查参数数量限制
            }
        } else {
            if (bufferLen < MAXARGLEN - 1) {
                buffer[bufferLen++] = ch;
                buffer[bufferLen] = '\0';  // 保持buffer为有效的C字符串
            }
        }
    }

    // 设置参数数组的结束标志
    args[argNum] = 0;

    // 创建子进程执行命令
    if (fork() == 0) {
        exec(args[0], args);
        freeArgs(args, argNum);
    } else {
        wait(0);  // 父进程等待子进程结束
        freeArgs(args, argNum);  // 释放分配的内存
    }

    return 0;
}
```

实验中遇到的问题和解决办法

在 C 语言中，`NULL` 是一个指针常量，通常用来初始化指针或作为指针操作的标记。但实验中遇到错误信息指出 `NULL` 未定义，这可能是因为相应的头文件没有被包含进来，加入 `#include`

`<stddef.h>` 即可解决。

`strdup` 函数是 POSIX 标准的一部分，而非 ISO C 标准的一部分。在本实验环境中，默认可能不包括对 POSIX 函数的支持，导致 `strdup` 无法使用。解决办法是实现自定义的`strdup` 函数，使用 `myStrdup` 替代 `strdup` 可以避免依赖 POSIX 环境。

实验心得

这次实验我尝试实现了一个简化版的 UNIX `xargs` 程序，通过动手构建 `xargs` 功能，我从底层实现细节上理解了它如何从标准输入读取数据并传递给其他命令，这是我之前在使用这些命令时从未深入思考过的。

特别地，通过这次实验，我对进程创建和管理有了更实际的体验。使用 `fork()` 创建新进程，并通过 `exec()` 系列函数在子进程中运行新的程序，这些都是操作系统课程中的理论知识。在实验中亲自编码实现，让我对这些概念有了更深刻的认识，特别是处理命令行参数和从标准输入中动态读取数据的过程，让我认识到在实际应用中如何处理和传递参数的重要性和复杂性。

总结

本实验系列旨在通过实际操作来加深对操作系统基本概念的理解和应用，包括进程管理、文件系统、进程间通信等。同时，学习使用Unix工具和Git进行版本控制，提升实际开发和调试能力。

- **Boot Xv6**：
  - 目标：配置实验环境，使用QEMU模拟器运行和调试Xv6操作系统，学习Git基本操作。
  - 通过克隆Xv6实验仓库，使用QEMU运行Xv6，并通过ls命令验证系统运行情况，掌握了基本的环境配置和版本控制工具的使用。
- **Sleep**：
  - 目标：理解Xv6内核中的时间概念，处理命令行参数，使用系统调用。
  - 通过编写和运行一个简单的sleep程序，深入理解了命令行参数处理和系统调用的使用。
- **PingPong**：
  - 目标：掌握进程创建，理解管道通信，使用数据读写系统调用，理解进程标识。
  - 通过实现进程间的ping-pong通信，学会了使用管道进行进程间通信，以及在多进程环境中进行数据读写和进程同步。
- **Primes**：
  - 目标：理解并发编程，掌握进程间通信。

    - 通过并发版本的素数筛选算法，理解了使用管道进行进程间通信的有效性和并发编程的基本方法。

- **Find**：
    - 目标：文件系统遍历，递归函数使用，字符串操作。
    - 通过实现简化版的find命令，学会了如何递归遍历目录和处理文件名字符串，增强了对文件系统操作的理解。

- **Xargs**：
    - 目标：理解进程创建和管理，组合使用Unix工具。
    - 通过实现xargs命令，学习了如何动态读取标准输入并传递给其他命令，理解了Unix工具组合使用的灵活性和效率。