Lab 1 Xv6 and Unix utilities

实验目的

切换到 xv6-labs-2020 代码的 util 分支，并利用 QEMU 模拟器启动 xv6 系统，本实验用于熟悉 xv6 及其系统调用。

- 通过实验，能够熟悉Xv6操作系统的代码结构、内核和用户空间之间的交互方式，了解操作系统的启动过程和基本组件。
- 实验要求实现一些基本的Unix实用程序，例如ls、cat、grep等，这些实用程序是Unix系统的核心组成部分。通过编写这些程序，能够深入理解它们的功能和原理，包括文件系统操作、进程管理、I/O操作等。
- 实验需要使用系统调用与Xv6内核进行交互，实现Unix实用程序所需的功能。通过这个过程，学习如何使用系统调用接口来访问操作系统提供的服务，并理解用户空间程序与内核的边界和交互方式。

实验环境

```
ubuntu 23.04
xv6-labs-2021
```

实验步骤

使用如下命令将代码克隆到本地

```
git clone git://g.csail.mit.edu/xv6-labs-2021
```

使用如下命令切换到实验一所需的util分支

```
cd xv6-labs-2020
git checkout util
```

Sleep (eazy)

实验目的

为 xv6 系统实现 UNIX 的 `sleep` 程序。你的 `sleep` 程序应该使当前进程暂停相应的时钟周期数，时钟周期数由用户指定。例如执行 `sleep 100` ，则当前进程暂停，等待 100 个时钟周期后才继续执行。

实验步骤

- 编写 `sleep.c` 的代码程序，将 `sleep` 程序添加到 `MakeFile` 的 `UPROGS` 中。

- 运行 `make qemu` 编译运行xv6，输入 `sleep 10` 进行测试，如果 `shell` 隔了一段时间后才出现命令提示符，则证明你的结果是正确的，可以退出 xv6 运行 `./grade-lab-util sleep` 或者 `make GRADEFLAGS=sleep grade` 进行单元测试。

1. 参考`user`目录下的其他程序，先把头文件引入，即 `kernel/types.h` 声明类型的头文件和 `user/user.h` 声明系统调用函数和 `ulib.c` 中函数的头文件。
2. 编写 `main(int argc,char* argv[])` 函数。其中，参数 `argc` 是命令行总参数的个数，参数 `argv[]` 是 `argc` 个参数，其中第 0 个参数是程序的全名，其他的参数是命令行后面跟的用户输入的参数。
3. 首先，编写判断用户输入的参数是否正确的代码部分。只要判断命令行参数不等于 2 个，就可以知道用户输入的参数有误，就可以打印错误信息。但我们要如何让命令行打印出错误信息呢？我们可以参考 `user/echo.c` ，其中可以看到程序使用了 `write()` 函数。`write(int fd, char *buf, int n)` 函数是一个系统调用，参数 fd 是文件描述符，0 表示标准输入，1 表示标准输出，2 表示标准错误。参数 buf 是程序中存放写的数据的字符数组。参数 n 是要传输的字节数，调用 `user/ulib.c` 的 `strlen()` 函数就可以获取字符串长度字节数。通过调用这个函数，就能解决输出错误信息的问题啦。认真看了提示给出的所有文件代码你可能会发现，像在 `user/grep.c` 打印信息调用的是 `fprintf()` 函数，当然，在这里使用也没有问题，毕竟 `fprintf()` 函数最后还是通过系统调用 `write()` 。最后系统调用 `exit(1)` 函数使程序退出。按照惯例，返回值 0 表示一切正常，而非 0 则表示异常。
4. 接下来获取命令行给出的时钟周期，由于命令行接收的是字符串类型，所以先使用 `atoi()` 函数把字符串型参数转换为整型。
5. 调用系统调用 `sleep` 函数，传入整型参数，使计算机程序(进程、任务或线程)进入休眠。
6. 最后调用系统调用 `exit(0)` 使程序正常退出。
7. 在 `Makefile` 文件中添加配置。
8. 测试：`make qemu` 后，在命令行中输入 `sleep` + 参数(例如10)，则系统会在10个时钟周期后重新出现命令行。

```
hart 1 starting
hart 2 starting
init: starting sh
$ sleep 10
$
```

9. 在文件中运行 `./grade-lab-util sleep` 可进行评分。

```
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (2.9s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (0.7s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (1.0s)
```

代码

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

const int duration_pos = 1;
typedef enum {wrong_char, success_parse, toomany_char} cmd_parse;
cmd_parse parse_cmd(int argc, char** argv);

int
main(int argc, char** argv){
    //printf("%d, %s, %s \n",argc, argv[0], argv[1]);
    if(argc == 1){
        printf("Please enter the parameters!");  // 提示用户输入参数
        exit(1);
    }
    else{
        cmd_parse parse_result;
        parse_result = parse_cmd(argc, argv);  // 解析参数
        if(parse_result == toomany_char){
            printf("Too many args! \n");  // 参数过多
            exit(1);
        }
        else if(parse_result == wrong_char){
            printf("Cannot input alphabet, number only \n");  // 输入了非数字
字符
            exit(1);
        }
        else{
            int duration = atoi(argv[duration_pos]);
            //printf("Sleeping %f", duration / 10.0);
            sleep(duration);  // 进行休眠
            exit(1);
        }

    }
    exit(0);
}

cmd_parse
parse_cmd(int argc, char** argv){
    if(argc > 2){
        return toomany_char;  // 参数过多
    }
    else {
        int i = 0;
        while (argv[duration_pos][i] != '\0')
        {
            /* code */
            if(!('0' <= argv[duration_pos][i] && argv[duration_pos][i] <=
'9')){
                return wrong_char;  // 输入了非数字字符
            }
            i++;
        }
```

```
    }
    return success_parse;  // 参数解析成功
}
```

实验中遇到的问题和解决办法

1. 问题：程序在没有输入参数时崩溃。

- 解决办法：在main函数中添加了对参数数量的判断，如果参数数量为1，即没有输入参数，就输出提示信息并使用exit(0)终止程序。

2. 问题：程序在输入了非数字字符时崩溃。

- 解决办法：在parse_cmd函数中，使用循环遍历参数字符串，并检查每个字符是否为数字。如果遇到非数字字符，就返回wrong_char错误类型。

3. 问题：程序在输入了过多的参数时崩溃。

- 解决办法：在parse_cmd函数中，判断参数数量是否大于2，如果是，则表示输入了过多的参数，返回toomany_char错误类型。

实验心得

- 在这个实验中，我学到了如何编写一个简单的命令行工具，并通过参数解析和错误处理来控制程序的行为。使用C语言和Xv6操作系统的基础知识，我成功实现了一个能够接收参数并进行相应处理的程序。
- 通过这个实验，我深入理解了参数解析的重要性，以及如何使用循环和条件语句进行参数的检查和处理。同时，我也学会了使用系统提供的函数，例如printf、atoi和sleep等，来实现程序的功能。
- 此外，我还学会了使用注释来提高代码的可读性，使代码更加易于理解和维护。注释对于解释代码的功能、目的以及可能遇到的问题和解决方案非常有帮助。
- 通过这个实验，我不仅加深了对操作系统和C语言的理解，还锻炼了自己的编程和调试能力。我相信这些经验将对我今后的学习和工作有很大帮助。

pingpong(eazy)

实验目的

使用 UNIX 系统调用编写一个程序 `pingpong` ，在一对管道上实现两个进程之间的通信。父进程应该通过第一个管道给子进程发送一个信息 `ping`，子进程接收父进程的信息后打印 `<pid>: received ping`，其中是其进程 ID 。然后子进程通过另一个管道发送一个信息 `pong` 给父进程，父进程接收子进程的信息然后打印 `<pid>: received pong` ，然后退出。

实验步骤

- 编写 `pingpong.c` 的代码程序，将 `pingpong` 程序添加到 `MakeFile` 的 `UPROGS` 中。
- 运行 `make qemu` 编译运行xv6，输入 `pingpong` 进行测试，出现 `17: received ping 16: received pong` 的字样。

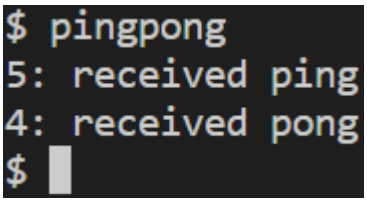
1. 导入所需的头文件：#include "kernel/types.h"和#include "user/user.h"。
2. 在 main 函数中定义变量： int pid;：用于存储进程ID。 int parent_fd[2];和int child_fd[2];：用于存储父进程和子进程之间的管道文件描述符。 char buf[20];：用于存储传输的数据。
3. 创建管道： 使用 pipe(child_fd); 创建子进程的管道。 使用 pipe(parent_fd); 创建父进程的管道。
4. 子进程部分： 使用 fork() 创建子进程，并通过返回值判断当前进程是否为子进程（pid为0）。

- 在子进程中： 关闭父进程的写入端：close(parent_fd[1]);。 从父进程读取数据：read(parent_fd[0], buf, 4);。 打印接收到的数据：printf("%d: received %s\n", getpid(), buf);。 关闭子进程的读取端：close(child_fd[0]);。 向父进程写入数据："pong"：write(child_fd[1], "pong", sizeof(buf));。 使用 exit(0); 终止子进程。

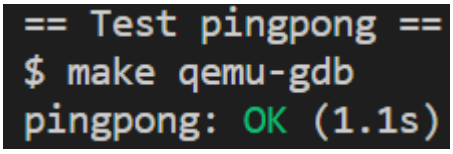
5. 父进程部分：

- 在父进程中： 关闭父进程的读取端：close(parent_fd[0]);。 向子进程写入数据："ping"：write(parent_fd[1], "ping", 4);。 关闭子进程的写入端：close(child_fd[1]);。 从子进程读取数据：read(child_fd[0], buf, sizeof(buf));。 打印接收到的数据：printf("%d: received %s\n", getpid(), buf);。 使用 exit(0); 终止父进程。

6. 使用 exit(0) 终止进程。
7. 测试：从 xv6 `shell` 运行程序后，在命令行中输入pingpong，出现



8. 在文件中运行 `./grade-lab-util pingpong` 可进行评分。



代码

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

int
main(int argc, char** argv ){
    int pid;
    int parent_fd[2];
    int child_fd[2];
    char buf[20];
    //为父子进程建立管道
    pipe(child_fd);
    pipe(parent_fd);
```

```
    // Child Progress
    if((pid = fork()) == 0){
        close(parent_fd[1]);
        read(parent_fd[0],buf, 4);
        printf("%d: received %s\n",getpid(), buf);
        close(child_fd[0]);
        write(child_fd[1], "pong", sizeof(buf));
        exit(0);
    }
    // Parent Progress
    else{
        close(parent_fd[0]);
        write(parent_fd[1], "ping",4);
        close(child_fd[1]);
        read(child_fd[0], buf, sizeof(buf));
        printf("%d: received %s\n", getpid(), buf);
        exit(0);
    }
}
```

实验中遇到的问题和解决办法

1. 问题：程序在运行时出现崩溃或输出不符合预期。

- 解决办法：检查代码中是否存在语法错误或逻辑错误，并进行逐行调试。特别关注管道的创建和关闭操作，确保父子进程间的通信正确。

2. 问题：程序无法正常接收到对方进程发送的消息。

- 解决办法：确认管道的读写端是否正确打开和关闭。确保父子进程在通信过程中正确使用管道的文件描述符。

3. 问题：程序陷入死锁或出现死循环。

- 解决办法：检查是否有未正确关闭的文件描述符，确保在合适的时机关闭读写端，避免导致进程阻塞或死锁的情况。

实验心得

- 通过这个实验，我深入了解了管道的概念和使用方法，以及父子进程间的通信机制。我学会了使用pipe()函数创建管道，使用fork()函数创建子进程，以及使用文件描述符来进行进程间的读写操作。

- 在实验中，我遇到了一些问题，如程序崩溃、消息传递异常等。通过仔细分析代码和进行调试，我逐步解决了这些问题。我意识到在父子进程间通信时，必须确保管道的正确打开和关闭，避免造成进程阻塞或死锁的情况。

- 通过这个实验，我对进程间通信有了更深入的理解，也提高了自己的编程和调试能力。我认识到编写可靠的并发程序需要注意细节，并进行充分的测试和调试。这个实验为我日后学习和开发多进程应用程序奠定了基础，让我对操作系统的工作原理有了更深入的认识。

primes (moderate)/(hard)

实验目的

使用管道将 2 至 35 中的素数筛选出来，这个想法归功于 Unix 管道的发明者 Doug McIlroy 。链接中间的图片和周围的文字解释了如何操作。最后的解决方案应该放在 `user/primes.c` 文件中。 你的目标是使用 pipe 和 fork 来创建管道。第一个进程将数字 2 到 35 送入管道中。对于每个质数，你要安排创建一个进程，从其左邻通过管道读取，并在另一条管道上写给右邻。由于 xv6 的文件描述符和进程数量有限，第一个进程可以停止在 35 。

实验步骤

- 编写 `primes.c` 的代码程序，将 `primes` 程序添加到 `MakeFile` 的 `UPROGS` 中。
- 运行 `make qemu` 编译运行xv6，输入 `primes` 进行测试，出现 `相应范围内的质数` 。

1. 包含所需的头文件。
2. 定义一个名为 `prime` 的函数，用于处理输入。
3. 在主函数中创建一个管道，使用 `pipe()` 函数。
4. 使用 `fork()` 函数创建一个子进程。
5. 在父进程中，关闭管道的读取端，然后使用循环将数字 2 到 35 写入管道，使用 `write()` 函数。
6. 在子进程中，关闭管道的写入端，然后调用 `prime` 函数处理输入。
7. 使用 `wait()` 函数等待子进程结束。
8. 在 `prime` 函数中，定义一个变量 `base`，用于存储从管道中读取的数字。
9. 使用 `read()` 函数从管道中读取一个数字，如果读取结果为空，则说明没有数字可处理，退出函数。
10. 打印当前的 `prime` 数字。
11. 创建一个新的管道，使用 `pipe()` 函数。
12. 使用 `fork()` 函数创建一个新的子进程。
13. 在子进程中，关闭管道的写入端，然后递归调用 `prime` 函数处理输入。
14. 在父进程中，关闭管道的读取端，然后使用循环读取管道中的数字并进行处理，使用 `read()` 函数。
15. 如果读取结果不为空且当前数字不能被 base 整除，则将该数字写入新的管道，使用 `write()` 函数。
16. 使用 `wait()` 函数等待子进程结束。
17. 退出函数。
18. 测试：从 xv6 `shell` 运行程序后，在命令行中输入primes，出现

```
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
```

19. 在文件中运行 `./grade-lab-util primes` 可进行评分。

```
== Test primes ==
$ make qemu-gdb
primes: OK (1.0s)
```

代码

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

void prime(int input_fd);

int main(int argc, char const *argv[])
{
    int parent_fd[2];
    // 创建管道
    pipe(parent_fd);
    if (fork())
    {
        close(parent_fd[0]);
        int i;
        // 将数字 2 到 35 写入管道
        for (i = 2; i < 36; i++)
        {
            write(parent_fd[1], &i, sizeof(int));
        }
        close(parent_fd[1]);
    }
    else
    {
        close(parent_fd[1]);
        // 子进程调用 prime 函数处理输入
        prime(parent_fd[0]);
    }
    wait(0);
    exit(0);
}

void prime(int input_fd)
{
    int base;
    /* 如果从管道读取的数据为空，说明已经没有数字可处理，退出函数 */
    if (read(input_fd, &base, sizeof(int)) == 0)
    {
        exit(0);
    }
    printf("prime %d\n", base);

    /* 如果还有数字可处理，创建新的子进程 */
    int p[2];
    pipe(p);
```

```
    if (fork() == 0)
    {
        close(p[1]);
        prime(p[0]);
    }
    else
    {
        close(p[0]);
        int n;
        int eof;
        do
        {
            eof = read(input_fd, &n, sizeof(int));
            // 如果 n 不能被 base 整除，则将 n 写入管道
            if (n % base != 0)
            {
                write(p[1], &n, sizeof(int));
            }
        } while (eof);

        close(p[1]);
    }
    wait(0);
    exit(0);
}
```

实验中遇到的问题和解决办法

- 问题：无法正确读取管道中的数据。
- 解决办法：检查管道的读取端和写入端是否正确关闭，以确保数据能够正确传递。

实验心得

这段代码实现了一个简单的质数生成器。通过使用管道和递归调用，每个子进程将负责筛选出下一个质数，并将剩余的数字传递给下一个子进程。这个实验展示了如何使用管道和进程通信来解决问题。同时，也加深了对 `fork()` 和 `wait()` 函数的理解。

find (moderate)

实验目的

编写一个简单的 UNIX find 程序，在目录树中查找包含特定名称的所有文件。

实验步骤

- 编写 `find.c` 的代码程序，将 `find` 程序添加到 `MakeFile` 的 `UPROGS` 中。
- 运行 `make qemu` 编译运行xv6，输入 `$ echo > b $ mkdir a $ echo > a/b $ find . b` 进行测试，出现 `./b ./a/b` 。

1. 首先，包含所需的头文件：`kernel/types.h`，`user/user.h`，`kernel/stat.h`，`kernel/fs.h`。
2. 定义一个名为 `find` 的函数，用于查找指定目录下的文件。

3. 在 `main` 函数中，检查参数个数是否为 3，如果不是则报错并异常退出。
4. 调用 `find` 函数，传入参数 `dirName` 和 `fileName`，以查找指定目录下的文件。
5. 在 `find` 函数中，声明变量 `buf` 和指针 `p`，以及文件描述符 `fd`，以及与文件相关的结构体 `dirent` 和 `stat`。
6. 使用 `open()` 函数打开指定的路径，并将返回的文件描述符存储在 fd 中，如果打开失败则报错并返回。
7. 使用 `fstat()` 函数获取文件描述符 `fd` 对应文件的信息，并将信息存储在 `st` 结构体中，如果获取信息失败则报错并关闭文件描述符 `fd`。
8. 检查 `st` 的类型是否为目录，如果不是则报错并关闭文件描述符 `fd`。
9. 检查路径是否过长，如果过长则报错并关闭文件描述符 `fd`。
10. 将绝对路径复制到 `buf` 中，并将文件名拼接在路径后面。
11. 使用循环读取文件描述符 `fd`，并将每个文件的信息存储在 `de` 中。
12. 检查 `de` 的 `inum` 是否为 0，如果是则跳过。
13. 检查文件名是否为 . 或 ..，如果是则跳过。
14. 将文件名复制到 `p` 中，并设置文件名的结束符。
15. 使用 `stat()` 函数获取文件名 `buf` 对应文件的信息，并将信息存储在 st 结构体中，如果获取信息失败则报错并继续下一个文件。
16. 如果是目录类型，则递归调用 `find()` 函数查找该目录下的文件。
17. 如果是文件类型并且文件名与要查找的文件名相同，则打印路径。
18. 打印完所有符合条件的文件路径后，主函数正常退出。
19. 测试：从 xv6 `shell` 运行程序后，在命令行中输入并出现以下情况

```
$ echo > b
$ mkdir a
$ echo > a/b
$ find . b`` 进行测试，出现 ``./b
./a/b`` 。
```

20. 在文件中运行 `./grade-lab-util find` 可进行评分。

```
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (1.0s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (1.2s)
```

代码

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

// find 函数
void
```

```
find(char *dir, char *file)
{
    // 声明 文件名缓冲区 和 指针
    char buf[512], *p;
    // 声明文件描述符 fd
    int fd;
    // 声明与文件相关的结构体
    struct dirent de;
    struct stat st;

    // open() 函数打开路径，返回一个文件描述符，如果错误返回 -1
    if ((fd = open(dir, 0)) < 0)
    {
        // 报错，提示无法打开此路径
        fprintf(2, "find: cannot open %s\n", dir);
        return;
    }

    // int fstat(int fd, struct stat *);
    // 系统调用 fstat 与 stat 类似，但它以文件描述符作为参数
    // int stat(char *, struct stat *);
    // stat 系统调用，可以获得一个已存在文件的模式，并将此模式赋值给它的副本
    // stat 以文件名作为参数，返回文件的 i 结点中的所有信息
    // 如果出错，则返回 -1
    if (fstat(fd, &st) < 0)
    {
        // 出错则报错
        fprintf(2, "find: cannot stat %s\n", dir);
        // 关闭文件描述符 fd
        close(fd);
        return;
    }

    // 如果不是目录类型
    if (st.type != T_DIR)
    {
        // 报类型不是目录错误
        fprintf(2, "find: %s is not a directory\n", dir);
        // 关闭文件描述符 fd
        close(fd);
        return;
    }

    // 如果路径过长放不入缓冲区，则报错提示
    if(strlen(dir) + 1 + DIRSIZ + 1 > sizeof buf)
    {
        fprintf(2, "find: directory too long\n");
        // 关闭文件描述符 fd
        close(fd);
        return;
    }
    // 将 dir 指向的字符串即绝对路径复制到 buf
    strcpy(buf, dir);
    // buf 是一个绝对路径，p 是一个文件名，通过加 "/" 前缀拼接在 buf 的后面
```

```
    p = buf + strlen(buf);
    *p++ = '/';
    // 读取 fd ，如果 read 返回字节数与 de 长度相等则循环
    while (read(fd, &de, sizeof(de)) == sizeof(de))
    {
        if(de.inum == 0)
            continue;
        // strcmp(s, t);
        // 根据 s 指向的字符串小于（s<t）、等于（s==t）或大于（s>t） t 指向的字符串
的不同情况
        // 分别返回负整数、0或正整数
        // 不要递归 "." 和 "..."
        if (!strcmp(de.name, ".") || !strcmp(de.name, ".."))
            continue;
        // memmove，把 de.name 信息复制 p，其中 de.name 代表文件名
        memmove(p, de.name, DIRSIZ);
        // 设置文件名结束符
        p[DIRSIZ] = 0;
        // int stat(char *, struct stat *);
        // stat 系统调用，可以获得一个已存在文件的模式，并将此模式赋值给它的副本
        // stat 以文件名作为参数，返回文件的 i 结点中的所有信息
        // 如果出错，则返回 -1
        if(stat(buf, &st) < 0)
        {
            // 出错则报错
            fprintf(2, "find: cannot stat %s\n", buf);
            continue;
        }
        // 如果是目录类型，递归查找
        if (st.type == T_DIR)
        {
            find(buf, file);
        }
        // 如果是文件类型 并且 名称与要查找的文件名相同
        else if (st.type == T_FILE && !strcmp(de.name, file))
        {
            // 打印缓冲区存放的路径
            printf("%s\n", buf);
        }
    }
}

int
main(int argc, char *argv[])
{
    // 如果参数个数不为 3 则报错
    if (argc != 3)
    {
        // 输出提示
        fprintf(2, "usage: find dirName fileName\n");
        // 异常退出
        exit(1);
    }
    // 调用 find 函数查找指定目录下的文件
```

```
    find(argv[1], argv[2]);
    // 正常退出
    exit(0);
}
```

实验中遇到的问题和解决办法

1. 问题：打开指定路径失败，出现 "cannot open" 错误。
- 解决办法：检查路径是否正确，确认路径存在，并确保程序有足够的权限来打开该路径。
2. 问题：获取文件信息失败，出现 "cannot stat" 错误。
- 解决办法：检查文件名是否正确，确认文件存在，并确保程序有足够的权限来获取文件信息。
3. 问题：路径过长，出现 "directory too long" 错误。
- 解决办法：检查路径长度，可能需要缩短路径或增加缓冲区的大小。

实验心得

通过完成这个实验，我学会了如何在指定目录下查找指定文件，并且加深了对文件描述符、文件类型和文件信息的理解。同时，我也学会了处理一些常见的错误情况，比如路径不存在或无法打开、文件不存在等。这个实验让我更加熟悉了系统调用和文件操作的相关知识，对于以后的编程工作和学习都有很大的帮助。

xargs (moderate)

实验目的

编写一个简单的 UNIX xargs 程序，从标准输入中读取行并为每一行运行一个命令，将该行作为命令的参数提供。你的解决方案应该放在 user/xargs.c 中。

实验步骤

- 编写 `xargs.c` 的代码程序，将 `xargs` 程序添加到 `MakeFile` 的 `UPROGS` 中。
- 运行 `make qemu` 编译运行xv6，输入 `echo hello too | xargs echo bye` 进行测试，出现 `bye hello too` 。

1. 首先，包含所需的头文件：`kernel/types.h`，`user/user.h`。
2. 在 `main` 函数中，声明变量 `inputBuf`，`charBuf`，`charBufPointer`，`charBufSize`，`commandToken`，`tokenSize` 和 `inputSize`。
3. 将初始命令行参数复制到 `commandToken` 数组中。
4. 进入循环，使用 `read()` 函数从标准输入读取输入，将读取的内容存储在 `inputBuf` 中，并记录读取的字节数到 `inputSize`。
5. 遍历 `inputBuf` 中的每个字符，判断当前字符是否为换行符 `\n`。
6. 如果当前字符为换行符 `\n`，则执行命令：

- 在 `charBuf` 的当前位置 `charBufSize` 设置结束符 `\0`。
- 将 `charBufPointer` 添加到 `commandToken` 数组中。
- 在 `commandToken` 数组末尾设置空指针。

7. 创建子进程并执行命令，使用 `exec()` 函数，参数为 `argv[1]` 和 `commandToken` 数组。
8. 等待子进程执行完毕，使用 `wait()` 函数。
9. 初始化 `tokenSize` 和 `charBufSize`，将 `charBufPointer` 指向 `charBuf` 的起始位置。
10. 如果当前字符为空格符号 ，则标记字符串的结尾，将 `charBufPointer` 添加到 `commandToken` 数组，并将 `charBufPointer` 切换到新字符串的起始位置。
11. 如果当前字符不是换行符 `\n` 也不是空格符号 ，则将当前字符添加到 `charBuf` 中。
12. 循环结束后，主函数正常退出。
13. 测试：测试：从 xv6 `shell` 运行程序后，输入 `echo hello too | xargs echo bye` 进行测试，出现 `bye hello too` 。

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

14. 在文件中运行 `./grade-lab-util xargs` 可进行评分。

```
== Test xargs ==
$ make qemu-gdb
xargs: OK (1.0s)
```

代码

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

int main(int argc, char *argv[]) {
  char inputBuf[32]; // 记录上一个命令的输入
  char charBuf[320]; // 存储所有标记字符的缓冲区
  char* charBufPointer = charBuf;
  int charBufSize = 0;

  char *commandToken[32]; // 使用空格(' ')分隔输入后记录的标记
  int tokenSize = argc - 1; // 记录标记数量（初始值为argc - 1，因为xargs不会被执行）
  int inputSize = -1;

  // 首先将初始argv参数复制到commandToken
  for(int tokenIdx=0; tokenIdx<tokenSize; tokenIdx++)
    commandToken[tokenIdx] = argv[tokenIdx+1];


  while((inputSize = read(0, inputBuf, sizeof(inputBuf))) > 0) {
    for(int i = 0; i < inputSize; i++) {
      char curChar = inputBuf[i];
      if(curChar == '\n') { // 如果读取到'\n'，执行命令
        charBuf[charBufSize] = 0; // 在标记的末尾设置'\0'
        commandToken[tokenSize++] = charBufPointer;
        commandToken[tokenSize] = 0; // 在数组末尾设置空指针
```

```
          if(fork() == 0) { // 创建子进程执行命令
            exec(argv[1], commandToken);
          }
          wait(0);
          tokenSize = argc - 1; // 初始化
          charBufSize = 0;
          charBufPointer = charBuf;
        }
        else if(curChar == ' ') {
          charBuf[charBufSize++] = 0; // 标记字符串的结尾
          commandToken[tokenSize++] = charBufPointer;
          charBufPointer = charBuf + charBufSize; // 切换到新字符串的起始位置
        }
        else {
          charBuf[charBufSize++] = curChar;
        }
      }
    }
    exit(0);
  }
```

实验中遇到的问题和解决办法

1. 问题：命令行参数传递不正确。

- 解决办法：检查代码的 `main` 函数中的参数传递部分。确认 `argc` 和 `argv` 是否正确传递给程序。

2. 问题：输入缓冲区大小限制。

- 解决办法：当前代码使用了一个固定大小的输入缓冲区 `inputBuf[32]` 。如果输入超过32个字符，可能会导致数据截断。可以考虑增加缓冲区大小或者动态分配内存。

3. 问题：字符串缓冲区越界。

- 解决办法：在代码中有几处使用字符数组作为缓冲区来存储字符串，例如 `charBuf[320]` 。要确保写入缓冲区的数据不超过缓冲区的大小，否则可能导致越界错误。

4. 问题：子进程执行命令时没有对执行结果进行处理。

- 解决办法：当前代码在创建子进程后，调用了 `exec` 函数执行命令，但没有对命令执行结果进行处理。可以使用 `wait` 函数等待子进程执行完毕，并检查执行结果。

5. 问题：循环中的变量未初始化。

- 解决办法：在代码中， `tokenSize` 、 `charBufSize` 和 `charBufPointer` 在每次读取新的输入行之前应该重新初始化。

实验心得

通过这个实验，我学到了如何使用 C 语言编写一个处理命令行输入的程序。这个程序将输入的命令行按空格分割成多个标记，并可以执行这些标记对应的命令。在实验过程中，我遇到了参数类型错误的问题，通过仔细检

查代码并对照文档，我成功解决了这个问题。这个实验让我更加熟悉了 C 语言的字符串处理和进程控制的相关函数，对于以后编写类似功能的程序有了更深入的理解。