华东师范大学数据学院上机实践报告

课程名称： 操作系统　　**年级：**2022 级　　**上机实践成绩：**

指导教师：翁楚良　　**姓名：**李芳

上机实践名称：Util　　**学号：**10214602404　　**上机实践日期：**2024.02.29

一、实验目的

熟悉如何在 xv6 环境下写代码

二、实验内容

完成 sleep、pingpong、primes、find、xargs 五个作业

三、使用环境

xv6

四、实验过程及结果

Sleep

首先，因为要求有“如果用户忘记传递参数，sleep 应该打印一条错误消息”，所以 main()函数中应首先判断命令行参数个数：argc！=2 时，利用 write(文件描述符为 2)打印出错情况下，正确的命令行使用方式：“Usage: sleep time\n”。

其次，要获取 time，根据提示使用 atoi()函数，将命令行参数第二个 argv[1]转化成整数，调用 sleep()函数。完成任务后 exit(0)安全退出程序。

最后，将$U/_sleep\加入到 Makefile 的 UPROGS 中。

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

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

    if (argc != 2)  //output wrong message
    {
        write(2, "Usage: sleep time\n", strlen("Usage: sleep time\n"));
        exit(1);
    }

    int time = atoi(argv[1]);  //get int time
    sleep(time);
    exit(0);
}
```

```
lifang@lifang-virtual-machine:~/xv6-labs-2021$ ./grade-lab-util sleep
make: “kernel/kernel”已是最新。
== Test sleep, no arguments == sleep, no arguments: OK (5.2s)
== Test sleep, returns == sleep, returns: OK (1.8s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.6s)
```

Pingpong

事先了解 pipe()函数的作用：pipe 是一种 IPC 机制，可以用来父子进程建立一个单向通道，一个进程可以向管道中写入数据，另一个可以从管道中读取数据：数组[0]读，数组[1]写。无名管道中，数据以先进先出的方式进行读取，无论写入和读取两过程的顺序如何，都不影响读取结果。

首先，初始两个数组建立两个管道，分别用于父进程和子进程；调用 fork()函数建立子进程后，根据 pid 分别操作。

Pid=0 子进程：关掉父到子的写，从父到子的读取端获取内容，子进程输出“received ping”。然后子进程要给父进程递信息：关掉子到父的读，向子到父的写入段写入“pong”，正常退出子进程。

Pid>0 父进程：关掉父到子的读，向父到子的写入段写入“ping”。然后父进程也要读取内容，这时，父必须等待子进程全部动作结束，写完了之后，再去读，所以调用 wait(0)。等完，关掉子到父的写，从子到父的读取端读内容，打印输出后正常退出。

Pid<0：fork()出错。

整个过程要注意，字符串以‘\n’结尾，所以 buf 数组长度为 5。将$U/_pingpong\加入到 Makefile 的 UPROGS 中。

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

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

    int p2c_fd[2], c2p_fd[2]; // parent & child
    pipe(p2c_fd), pipe(c2p_fd);
    char buf[5]; // save massage from pipe

    int size;
    int pid = fork();

    if (pid == 0)
    {
        // child,read massage from parent
        close(p2c_fd[1]);
        size = read(p2c_fd[0], buf, sizeof(buf));
        printf("%d: received ", getpid());
        write(1, buf, size);

        // write massage to parent
        close(c2p_fd[0]);
        write(c2p_fd[1], "pong\n", 5);
        exit(0);
    }
    else if (pid > 0)
    {
        // parent,write massage to child
        close(p2c_fd[0]);
        write(p2c_fd[1], "ping\n", 5);

        wait(0);
        // read massage from child
        close(c2p_fd[1]);
        size = read(c2p_fd[0], buf, sizeof(buf));
        printf("%d: received ", getpid());
        write(1, buf, size);
    }
    else
    {
        printf("fork error!\n");
    }

    exit(0);
}
```

```
lifang@lifang-virtual-machine:~/xv6-labs-2021$ ./grade-lab-util pingpong
make: “kernel/kernel”已是最新。
== Test pingpong == pingpong: OK (3.0s)
lifang@lifang-virtual-machine:~/xv6-labs-2021$
```

Primes

事先根据 wc.c 中的代码，基本了解了文件描述符的应用：通过文件描述符，程序可以打开文件、读取文件内容，并在使用完成后关闭文件。dup()用于复制文件描述符。文件描述符 1 只代表标准输出的内容，但是如果没有打印操作，也不会输出到终端。

main 函数：创建管道，建立进程分支：

子：将管道写端映射到标准输出 1，循环输出 2-35 到标准输出。

父：wait 子完成，将管道读映射到标准输入 0，开始调用 primes 查找素数

mapping 函数：处理文件描述符的映射。先关闭文件描述符 n，然后通过 dup(pd[n])复制 pd[n]到 n 上，形成映射，关闭通道中的两个描述符 pd[0]和 pd[1]。

primes 函数：找出小于 36 的所有素数。读取第一个整数，它一定是素数，所以直接打印输出。然后创建管道 fd，分进程处理：

子：先将管道写端映射到文件描述符 1，也就是标准输出上，然后通过 while 循环从标准输入读取下面的每一个数，不是第一个素数的倍数时，就输出。

父：wait 子进程全部结束，将管道读端映射到文件描述符 0，也就是标准输入上，然后递归调用，这样下来，皆能够输出打印输入范围内的所有素数。

最后，将$U/_primes\加入到 Makefile 的 UPROGS 中。

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

void mapping(int n, int pd[])
{
    close(n); // close file descriptor n

    dup(pd[n]); // generate a map fromn to pd[n]

    close(pd[0]); // close descriptor in the pipe
    close(pd[1]);
}

void primes()
{
    int pre, next;
    int fd[2];

    if (read(0, &pre, sizeof(int)))
    {
        printf("prime %d\n", pre); // print the first number directly
        pipe(fd);

        if (fork() == 0)
        {
            mapping(1, fd); // child maps the write port of the pipe to 1
            while (read(0, &next, sizeof(int)))
```

```
        {
            if (next % pre != 0)
            { // write who is not a multiple of the first number
                write(1, &next, sizeof(int));
            }
        }
    }
    else
    {
        wait(0);        // parent waits until child write all numbers
        mapping(0, fd); // parent maps the read port of the pipe to 0

        primes();
    }
}
}

int main(int argc, char *argv[])
{
    int fd[2];
    pipe(fd);

    if (fork() == 0)
    {
        mapping(1, fd);
        for (int i = 2; i < 36; i++)
        {
            write(1, &i, sizeof(int));
        }
    }
    else
    {
        wait(0);
        mapping(0, fd);
        primes();
    }
    exit(0);
}
```

```
lifang@lifang-virtual-machine:~/xv6-labs-2021$ ./grade-lab-util primes
make: “kernel/kernel”已是最新。
== Test primes == primes: OK (2.8s)
lifang@lifang-virtual-machine:~/xv6-labs-2021$
```

Find

事先阅读 ls.c 的代码，找出 find 和 ls 有联系的地方：同（都需要输入 path 参数、都需要判断 path 参数所指向的文件类型、都需要遍历目录项并读取目录项的名字），异（除 path 参数外 find 还需要 filename 参数、find 需要递归遍历指定目录中的所有子目录）

find()：首先尝试打开指定路径的文件，并获取其状态信息。根据文件类型执行不同的操作：如果是文件，则打印路径错误并返回。如果是目录，则遍历目录下的文件和子目录：对于每个子目录，递归调用 find 函数进行查找，对于每个文件，进行文件名匹配，匹配成功，则打印该文件的路径，最后关闭文件描述符。

main()：同样需要先判断命令行参数个数是否正确，然后再调用 find()函数。

注意：第一版代码没有意识到 while 分支递归调用 find 函数的必要性，因此无法通过测试，因为目录下可能有子目录，子目录下还有子子目录，一直调用到最后只有文件的那一层，查找过程才算结束，期间每遇到一次文件，就要比较下名称。

最后，把$U/_find\加入到 Makefile 的 UPROGS 中。

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

void find(char *path, char *filename)
{
    char buf[128], *p;
    int fd, fd1;
    struct dirent de;
    struct stat st, st1;

    // attempts to open the specified path file and retrieves its status information
    if ((fd = open(path, 0)) < 0)
    {
        fprintf(2, "path error\n");
        return;
    }

    if (fstat(fd, &st) < 0)
    {
        fprintf(2, "path stat failed\n");
        close(fd);
        return;
    }

    // performs different actions based on the file type
    switch (st.type)
    {
    case T_FILE:    //file
        fprintf(2, "path error\n");
        return;

    case T_DIR:     //directory
        strcpy(buf, path);  //copy path in buf
        p = buf + strlen(buf);  //make p point to the last position of buf
        *p++ = '/';  //add / to the last position
        while (read(fd, &de, sizeof(de)) == sizeof(de))  //judge if read the whole de
        {
            if (de.inum == 0)
            {
                continue;
            }
            if (!strcmp(de.name, ".") || !strcmp(de.name, ".."))  //skip this cycle
            {
                continue;
            }
            memmove(p, de.name, DIRSIZ);
            if ((fd1 = open(buf, 0)) >= 0)
            {
                if (fstat(fd1, &st1) >= 0)
                {
                    switch (st1.type)
                    {
                    case T_FILE:    //if it is file,compare its name
                        if (!strcmp(de.name, filename))
                        {
                            printf("%s\n", buf);
                        }
                        close(fd1);
                        break;
                    case T_DIR:    //if it is directory,recursive invocation find()
                        close(fd1);
                        find(buf, filename);
                        break;
                    case T_DEVICE:
                        close(fd1);
                        break;
                    }
                }
            }
        }

        break;
    }
    close(fd);
}

int main(int argc, char *argv[])
{
    if (argc != 3)  //input is wrong
    {
        fprintf(2, "Usage:find path fileName\n");
        exit(0);
    }
    find(argv[1], argv[2]);
    exit(0);
}
```

```
lifang@lifang-virtual-machine:~/xv6-labs-2021$ ./grade-lab-util find
make: “kernel/kernel”已是最新。
== Test find, in current directory == find, in current directory: OK (2.9s)
== Test find, recursive == find, recursive: OK (3.1s)
```

Xargs

事先阅读实验要求和 hint，知道了命令行标准输入的格式是 command | xargs [options] [command]，也就是类似于：执行 ls *.txt | xargs rm 相当于执行 rm file1.txt file2.txt file3.txt ...，即删除当前目录下所有以.txt 结尾的文件。

readline()：从标准输入读取一行数据，并将其分割成多个参数。通过循环读取每个字符，直到遇到换行符 \n 为止，将读取的字符存储在缓冲区 buf 中。如果读取的字符数超过 1023（buf 的大小减 1），则输出错误信息并退出程序。将读取的行按空格分割成多个参数，存储到 new_argv 数组中，并返回参数个数，更新参数数组为 cmd, arg[0], … , arg[k - 1], arg[k], … , arg[k + l - 1]。

main()：先检查命令行参数个数。然后动态分配内存保存第一个参数（即要执行的命令）到 command 中，将除第一个参数外的其他参数复制到 new_argv 数组中，形式为 cmd, arg[0], … , arg[k - 1]。进入循环，调用 readline() 函数读取一行参数，然后将参数数组传递给要执行的命令

子进程中，使用 exec() 执行指定的命令，如果执行失败，则输出错误信息。

父进程中，使用 wait() 等待子进程执行完成。

这样就能循环着一行一行地去执行命令了，一直到程序无法从命令行读取到有效信息时结束，保证每行命令都能够运行。

最后，把$U/_xargs\加入到 Makefile 的 UPROGS 中。

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

int readline(char *new_argv[32], int cur)
{
    char buf[1024];             // store input strings
    int n = 0;                  // record the number of char had read
    while (read(0, buf + n, 1)) // read a char from stdin, store in buf[n]
    {
        if (n == 1023) // is too long
        {
            fprintf(2, "argument is too long\n");
            exit(1);
        }
        if (buf[n] == '\n') // finish reading a line, exit cycle
            break;
        n++;
    }

    buf[n] = 0; // put 0 in end, to be a string
    if (n == 0)
    {
        return 0;
    } // no valid input

    int offset = 0;
    while (offset < n)
    {
        new_argv[cur++] = buf + offset;
```

```
        while (buf[offset] != ' ' && offset < n)
        {
            offset++;
        }
        while (buf[offset] == ' ' && offset < n)
        {
            buf[offset++] = 0;  //replace ' ' by'0'
        }
    }
    return cur; // the number of parameters
}

int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        fprintf(2, "Usage: xargs command (arg ...)\n");
        exit(1);
    }

    char *command = malloc(strlen(argv[1]) + 1); // save the first as command
    strcpy(command, argv[1]);

    char *new_argv[MAXARG]; // save others in new_argv
    for (int i = 1; i < argc; i++)
    {
        new_argv[i - 1] = malloc(strlen(argv[i]) + 1);
        strcpy(new_argv[i - 1], argv[i]);
    }

    int cur;
    while ((cur = readline(new_argv, argc - 1)) != 0)
    { // read standerd input
        new_argv[cur] = 0;
        if (fork() == 0)
        {
            exec(command, new_argv); // child execute command
            fprintf(2, "exec failed\n");
            exit(1);
        }
        wait(0);
    }
    exit(0);
}
```

```
lifang@lifang-virtual-machine:~/xv6-labs-2021$ ./grade-lab-util xargs
make: “kernel/kernel”已是最新。
== Test xargs == xargs: OK (4.1s)
lifang@lifang-virtual-machine:~/xv6-labs-2021$
```

Makefile:

```
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\
    $U/_primes\
    $U/_find\
    $U/_xargs\
```

Make grade

```
kernel.sym
make[1]: 离开目录“/home/lifang/xv6-labs-2021”
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (8.9s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (1.6s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (1.6s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (1.5s)
== Test primes ==
$ make qemu-gdb
primes: OK (1.8s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (2.1s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (3.0s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (3.5s)
== Test time ==
time: FAIL
    Cannot read time.txt
Score: 99/100
make: *** [Makefile:341：grade] 错误 1
lifang@lifang-virtual-machine:~/xv6-labs-2021$
```

五、**总结**

本次实验是操作系统这门课的第一次实验课，我完成了 vmware workstation 上 ubuntu 的配置，这让我印象比较深刻的是，它可以把这个工作环境具象化，形成类似 windows 桌面和系统，更加方便操作，之前用 wsl 都是类似 powershell 的命令行操作环境。我自己通过配置 ssh 连接，可以实现在本地 vscode 里远程连接 ubuntu 进行本次实验，比较方便，不过还是需要后台运行虚拟机，自己试着配置了下，我还不能实现挂起虚拟机还能本地远程连接，IP 地址会变化。

这次试验总共有 5 个小任务，通过阅读 user 里已经存在的代码来推任务代码怎么写，在最后一个任务时，比较困难，题意理解用了很长时间，还是不大熟悉命令行操作语句，这事以后要加强的地方。不过通过本次实验，我成功熟悉了 xv6 环境，还熟悉了一些系统调用函数的作用和使用方法，收获很大。