

一、实验内容

1.1. sleep

1.1.1 实验步骤

a. 首先明确sleep的功能和命令行参数。Usage: sleep ticks 即函数接受两个参数，第一个参数为sleep，第二个参数为ticks，ticks表示睡眠时间。

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

    // 检查参数个数
    if (argc != 2) {
        fprintf(2, "Usage: sleep ticks\n");
        exit(1); // 提供一个错误代码
    }
}
```

b. 然后对于传入的ticks参数进行数据类型转换和合法性检查，ticks必须大于0。

```
// 将传入的睡眠时间从字符串转换为整数
ticks = atoi(argv[1]);
if (ticks < 0) {
    fprintf(2, "sleep: ticks should be non-negative\n");
    exit(1);
}
```

c. 最后调用sleep函数进行睡眠。

```
// 调用sleep系统调用，传入ticks数量
sleep(ticks);

exit(0); // 成功退出
```

1.1.2 实验测试结果

```
yuyi@LAPTOP-PFU0V0VL:~/Operating_System/lab/xv6-oslab24$ python3 grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (4.1s)
== Test sleep, returns == sleep, returns: OK (1.4s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.4s)
```

1.2 pingpong

1.2.1 实验要求

- 父进程向管道中写入数据，子进程从管道将其读出并打印<pid>: received ping from pid <father pid>，其中<pid>是子进程的进程ID, <father pid>是父进程的进程ID。
- 子进程从父进程收到数据后，通过写入另一个管道向父进程传输数据，然后由父进程从该管道读取并打印<pid>: received pong from pid <child pid>，其中<pid>是父进程的进程ID, <child pid>是子进程的进程ID。

根据提示可知：

1. 需要使用两个管道(可以命名为c2f和f2c)；
2. 其中c2f用于子进程向父进程传输数据；
3. f2c用于父进程向子进程传输数据。

1.2.2 实验步骤

a. 首先设置两个管道，分别命名为c2f（子进程向父进程传输数据）和f2c（父进程向子进程传输数据）。

```
//将读端设为0，写端设为1
#define PIPE_READ 0
#define PIPE_WRITE 1

int main(void) {
    int c2f[2], f2c[2];
    int pid; // 进程ID
    char buf[512];
    int n;

    // 创建两个管道
    if (pipe(c2f) < 0 || pipe(f2c) < 0) {
        fprintf(2, "pipe() failed\n");
        exit(1);
    }
}
```

b. 利用fork()函数创建子进程和父进程。利用在不同进程中pid值不同进行分类讨论。

- 首先需要注意在父进程和子进程中都要能够获取两者的pid。
 - 对于父进程而言，可以通过pid和getpid()函数获取；
 - 而对于子进程而言，本应该通过getppid()函数获取父进程的pid，但是由于user.h中没有定义getppid()函数，所以只能通过管道传递父进程的pid。

因此代码过程如下：

- 对于子进程(pid == 0)，它要先从父进程读取数据，通过管道读取父进程的PID，输出<pid>: received ping from pid <father pid>，然后向父进程发送数据，即“pong”。
- 对于父进程(pid > 0)，要先将父进程的PID写入管道，然后向子进程发送数据，再从子进程读取数据，输出<pid>: received pong from pid <child pid>，最后调用wait()函数等待子进程退出。

```
pid = fork();
if (pid < 0) {
```

```
    fprintf(2, "fork() failed\n");
    exit(1);
} else if (pid == 0) { // 子进程
    // 关闭子进程不需要的管道端
    close(c2f[PIPE_READ]);
    close(f2c[PIPE_WRITE]);

    // 从父进程读取数据
    n = read(f2c[PIPE_READ], buf, sizeof(buf));
    if (n > 0) {
        // 子进程通过管道读取父进程的 PID
        int parent_pid = *(int*)buf;
        fprintf(1, "%d: received ping from pid %d\n", getpid(), parent_pid);
        // 向父进程发送数据
        n = write(c2f[PIPE_WRITE], "pong", 5);
        if (n < 0) {
            fprintf(2, "child write to pipe failed\n");
        }
    }
    exit(0);
} else { // 父进程
    // 关闭父进程不需要的管道端
    close(c2f[PIPE_WRITE]);
    close(f2c[PIPE_READ]);

    // 将父进程的 PID 写入管道
    int parent_pid = getpid();
    if (write(f2c[PIPE_WRITE], &parent_pid, sizeof(parent_pid)) !=
        sizeof(parent_pid)) {
        fprintf(2, "parent write to pipe failed\n");
    } else {
        // 向子进程发送数据
        n = write(f2c[PIPE_WRITE], "ping", 4);
        if (n < 0) {
            fprintf(2, "parent write to pipe failed\n");
        } else {
            // 从子进程读取数据
            n = read(c2f[PIPE_READ], buf, sizeof(buf));
            if (n > 0) {
                fprintf(1, "%d: received pong from pid %d\n", getpid(), pid);
            }
        }
    }
    wait(0); // 等待子进程退出
}
exit(0);
}
```

1.2.3 实验测试结果

```
yuyi@LAPTOP-PFU0V0VL:~/Operating_System/lab/xv6-oslab24$ python3 grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong lenient testing == pingpong lenient testing: OK (3.7s)
== Test pingpong strict testing with changing pids == pingpong strict testing with changing pids: OK (1.5s)
```

1.3 find

1.3.1 实验要求

在XV6上实现用户程序find，即在目录树中查找名称与字符串匹配的所有文件或目录，输出文件的相对路径。该程序的命令格式为find <path> <name>。将代码写在user/find.c文件中。

1.3.2 实验步骤

1.3.2.1 首先观察所需要的头文件和结构体

1. stat结构体 在/kernel/stat.h中。根据stat结构体所示，可以根据tyoe字段判断文件类型，其中目录类型为T_DIR，文件类型为T_FILE，设备类型为T_DEVICE。这个用于文件夹递归判断。

```
#define T_DIR      1    // Directory
#define T_FILE     2    // File
#define T_DEVICE   3    // Device

struct stat {
    int dev;           // File system's disk device
    uint ino;          // Inode number
    short type;        // Type of file
    short nlink;       // Number of links to file
    uint64 size;       // Size of file in bytes
};
```

2. dirent结构体 在/kernel/fs.h中。这个结构体用于存储从目录中读取的每个条目的信息

```
struct dirent {
    ushort inum; //文件的inode编号，用于唯一标识文件系统中的文件或目录。
    char name[DIRSIZ]; //一个字符串数组，包含文件路径目录字符串。
};
```

3. strncpy函数 在/kernel/string.c中。这个函数用于复制字符串，可以利用参数达到拼接字符串的目的。

```
char *strncpy(char *s, const char *t, int n) {
    char *os;

    os = s;
    while (n-- > 0 && (*s++ = *t++) != 0)
        ;
    while (n-- > 0) *s++ = 0;
```

```
    return os;
}
```

1.3.2.2 主函数

首先需要判断传入的参数是否合法，满足用法要求。若满足，则调用find函数。

```
int main(int argc, char *argv[]) {
    if (argc < 3) {
        fprintf(2, "Usage: find <path> <name>\n");
        exit(1);
    }

    find(argv[1], argv[2]);
    exit(0);
}
```

1.3.2.3 实现find函数

1. 首先根据需要初始化变量，包括存储路径的字符串，存储文件名的字符串，以及存储文件信息的结构体。其中buf为更改的路径字符串，fullpath为完整路径字符串，originpath为原路径字符串，保证同一目录下循环探查文件时原目录路径不变。

```
void find(char *path, char *name) {
    char buf[MAX_PATH];
    char fullpath[MAX_PATH];
    char originpath[MAX_PATH];
    int fd;
    struct dirent de;
    struct stat st;

    strncpy(fullpath, path, MAX_PATH - 1);

    fullpath[MAX_PATH - 1] = '\0'; // 确保字符串以空字符终止
    int len = strlen(fullpath);
    if (len + 1 < MAX_PATH) { // 确保有足够的空间添加 "/"
        fullpath[len] = '/';
        fullpath[len + 1] = '\0';
    }

    //保存原来的路径，递归的时候使用这个路径
    strncpy(originpath, path, MAX_PATH - 1);
    originpath[MAX_PATH - 1] = '\0'; // 确保字符串以空字符终止
    int len1 = strlen(originpath);
    if (len1 + 1 < MAX_PATH) { // 确保有足够的空间添加 "/"
        originpath[len1] = '/';
        originpath[len1 + 1] = '\0';
    }
}
```

2. 调用open函数打开目录，并判断是否成功。

```
// 打开目录
fd = open(fullpath, O_RDONLY);
if (fd < 0) {
    fprintf(2, "find: cannot open %s\n", path);
    return;
}
```

3. 调用read函数遍历目录。

- 使用memset函数清空buf，并拼接字符串，用于存储文件或目录的完整路径。
- 使用stat结构体获取buf中存储的文件或目录的状态，用于判断buf是否为正确的路径。
- 调用strcmp函数判断buf中的文件名是否与name匹配，如果匹配则输出buf中的路径。
- 如果buf为目录，则递归调用find函数。
- 注意循环时，是针对相同目录下的所有文件和目录进行查找，拼接路径使用originpath；对于递归，则使用更新后的fullpath。

```
// 读取目录内容
while (read(fd, &de, sizeof(de)) > 0) {

    if (de.inum == 0)
        continue;

    // 构建文件或目录的完整路径
    memset(buf, 0, MAX_PATH);
    strncpy(buf, originpath, MAX_PATH - 1);
    buf[MAX_PATH - 1] = '\0'; // 确保字符串以空字符终止
    int buf_len = strlen(buf);

    if (buf_len + 1 + strlen(de.name) < MAX_PATH) { // 确保有足够的空间添加名称
        // 如果不是根目录才增加一个斜杠
        if (strcmp(path, ".") != 0){
            buf[buf_len] = '/';
        }
        strncpy(buf + buf_len + 0, de.name, MAX_PATH - buf_len - 1); // 拼接字符串

        buf[MAX_PATH - 1] = '\0'; // 确保字符串以空字符终止
    }

    // 获取文件状态
    if (stat(buf, &st) < 0) {
        fprintf(2, "find: cannot stat %s\n", buf);
        continue;
    }

    // 检查名称是否匹配
```

```

        if (strcmp(de.name, name) == 0) {
            printf("%s\n", buf);
        }

        // 如果是目录，则递归查找
        if (st.type == T_DIR && strcmp(de.name, ".") != 0 && strcmp(de.name, "..")
            != 0) {
            strncpy(fullpath, buf, MAX_PATH - 1);
            fullpath[MAX_PATH - 1] = '\0'; // 确保字符串以空字符终止
            find(fullpath, name);
        }
    }
    close(fd);
}

```

1.3.3 实验测试结果

```

yuyi@LAPTOP-PFU0V0VL:~/Operating_System/lab/xv6-oslab24$ python3 grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory and create a file == find, in current directory and create a file: OK (9.9s)
== Test find, in current directory and create a dir == find, in current directory and create a dir: OK (6.3s)
== Test find, find file recursive == find, find file recursive: OK (3.4s)
== Test find, find dir recursive with no duplicates == find, find dir recursive with no duplicates: OK (2.6s)

```

1.3.4 实验结果总结

所有测试都通过，分数60分。

```

== Test sleep, no arguments == sleep, no arguments: OK (16.2s)
== Test sleep, returns == sleep, returns: OK (3.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (2.6s)
== Test pingpong lenient testing == pingpong lenient testing: OK (2.4s)
== Test pingpong strict testing with changing pids == pingpong strict testing with changing pids: OK (1.7s)
== Test find, in current directory and create a file == find, in current directory and create a file: OK (1.7s)
== Test find, in current directory and create a dir == find, in current directory and create a dir: OK (2.2s)
== Test find, find file recursive == find, find file recursive: OK (2.1s)
== Test find, find dir recursive with no duplicates == find, find dir recursive with no duplicates: OK (2.9s)
Score: 60/60
yuyi@LAPTOP-PFU0V0VL:~/Operating_System/lab/xv6-oslab24$

```

1.4 xv6启动流程实验

当执行完exec系统调用后，XV6会从initcode进程切换导init进程。

commands.gdb如下：

```

b exec
c
p cpus[$tp]->proc->name
c
p cpus[$tp]->proc->name

```

因此在调试时，给exec加一个断点即可。手动调试GDB：

```
(gdb) b exec
Breakpoint 1 at 0x80004d6c: file kernel/exec.c, line 12.
(gdb) c
Continuing.

Breakpoint 1, exec (path=path@entry=0x3ffffffdef0 "/init", argv=argv@entry=0x3ffffffddf0) at kernel/exec.c:12
12   int exec(char *path, char **argv) {
(gdb) p cpus[$tp]->proc->name
$1 = "initcode\000\000\000\000\000\000\000"
(gdb) c
Continuing.

Breakpoint 1, exec (path=path@entry=0x3ffffffbef0 "sh", argv=argv@entry=0x3ffffffbdf0) at kernel/exec.c:12
12   int exec(char *path, char **argv) {
(gdb) p cpus[$tp]->proc->name
$2 = "init", '\000' <repeats 11 times>
(gdb) |
```

脚本自动执行GDB调试：

```
(gdb) source user/commands.gdb
Breakpoint 1 at 0x80004d6c: file kernel/exec.c, line 12.

Breakpoint 1, exec (path=path@entry=0x3ffffffdef0 "/init", argv=argv@entry=0x3ffffffddf0)
at kernel/exec.c:12
12   int exec(char *path, char **argv) {
$1 = "initcode\000\000\000\000\000\000\000"

Breakpoint 1, exec (path=path@entry=0x3ffffffbef0 "sh", argv=argv@entry=0x3ffffffbdf0)
at kernel/exec.c:12
12   int exec(char *path, char **argv) {
$2 = "init", '\000' <repeats 11 times>
(gdb)
```

二、问题解答

了解管道模型，回答下列问题：

2.1 问题1

简要说明在pingpong实验中，你是怎么创建管道的？结合fork系统调用说明你是怎么使用管道在父子进程之间传输数据的。

1. 管道的创建：

- 首先初始化两个int数组c2f和f2c，分别用于存储管道的读端和写端。
- 使用pipe()函数创建管道，返回0表示创建成功，返回-1表示创建失败。

```
#define PIPE_READ 0
#define PIPE_WRITE 1

int c2f[2], f2c[2];
// 创建两个管道
if (pipe(c2f) < 0 || pipe(f2c) < 0) {
    fprintf(2, "pipe() failed\n");
    exit(1);
}
```

2. 父子进程之间的数据传输：

- 利用 `pid = fork()` 的值分开子进程和父进程。
- 子进程向父进程发送数据：
 - 首先关闭子进程不需要的管道端，即关闭 `c2f[PIPE_READ]` 和 `f2c[PIPE_WRITE]` 端。
 - 从父进程读入数据和父进程 `pid`。
 - 向父进程写入数据，即“pong”。
- 父进程向子进程发送数据：
 - 首先关闭父进程不需要的管道端，即关闭 `c2f[PIPE_WRITE]` 和 `f2c[PIPE_READ]` 端。
 - 向子进程写入父进程的 `pid` 和“ping”。
 - 从子进程读入数据 `pong` 并输出。

2.2 问题2

试解释，为什么要提前关闭管道中不使用的一端？（提示：结合管道的阻塞机制）

- 当两个进程通过管道进行通信时，如果一个进程试图写入一个管道，而另一个进程没有读取，写操作可能会阻塞，直到有进程读取数据。同样，如果一个进程试图从管道中读取数据，而另一个进程没有写入，读操作也会阻塞。
- 如果父进程和子进程都试图写入对方管道的写端，而没有读取对方的读端，那么它们都会阻塞，导致死锁。

2.3 问题3

阅读如下示例程序。

```
int p[2];          /* 存储管道的两个文件描述符 */
char *argv[2];
argv[0] = "wc";    /* 设置要执行的命令为"wc" */
argv[1] = 0;       /* 参数数组以NULL结尾，表示没有更多参数 */
pipe(p);           /* 创建管道，p[0]是管道的读端，p[1]是管道的写端 */

if (fork() == 0) {
    /* 子进程 */
    close(0);       /* 关闭标准输入（文件描述符0） */
    dup(p[0]);       /* 复制管道的读端p[0]，让文件描述符0指向管道的读端 */
    close(p[0]);     /* 关闭重复的管道读端 */
    close(p[1]);     /* 关闭不再需要的管道写端 */
    exec("/bin/wc", argv); /* 执行"wc"程序 */
} else {
    /* 父进程 */
    close(p[0]);     /* 关闭管道的读端 */
    write(p[1], "hello world\n", 12); /* 向管道写入"hello world\n" */
    close(p[1]);     /* 关闭管道的写端，表示写入完成 */
}
```

假设子进程没有关闭管道写端，运行该程序后，尝试描述会发生什么？为什么会有这样的结果？

- 如果子进程没有关闭管道的写端，父进程的 `write` 调用可能会阻塞，因为管道的缓冲区满了，而子进程没有读取数据。
- 这将导致死锁，因为父进程在等待写入操作完成，而子进程在等待读取操作，但两者都无法继续进行。

三、实验感想

3.1 实验中碰到何种问题？如何解决？

- 在pingpong实验中，最麻烦的问题就是不能直接在子进程中通过getppid()函数获取父进程的进程ID，因为头文件中没有相应的系统调用，只能通过管道来传递父进程的进程ID。
- 在find实验中，也是由于头文件中对字符串处理函数的缺乏，导致只能使用一些其他函数来代替字符串处理（例如字符串拼接）的功能。

3.2 实验感想

无