

- 一、实验内容
  - 1 sleep
    - 1.1 实验步骤
    - 1.2 实验测试结果
  - 2. pingpong
    - 2.1 实验步骤
    - 2.2 实验测试结果
  - 3. find
    - 3.1 实验步骤
    - 3.2 实验测试结果
  - sleep、pingpong、find测试结果
  - 4. xv6启动流程实验
    - 4.1 实验步骤
    - 4.2 实验测试结果
- 二、问题解答
  - 2.1 问题1内容
  - 2.2 问题2内容
  - 2.3 问题3内容

# 一、实验内容

---

## 1 sleep

---

### 1.1 实验步骤

a. 首先检查参数量是否正确，若错误则打印程序使用方法并退出。

```
if(argc != 2){  
    fprintf(2, "Usage: sleep <ticks>\n");  
    exit(1);  
}
```

b. 使用atoi将第二个参数转为睡眠时间，而后调用user/user.h中声明的系统调用sleep睡眠指定时间，并打印文字，最后退出。

```
int ticks = atoi(argv[1]);
sleep(ticks);
printf("(nothing happens for a little while)\n");

exit(0);
```

## 1.2 实验测试结果

```
● piacesy@piacesy:~/Code/xv6-oslab24$ ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.3s)
== Test sleep, returns == sleep, returns: OK (1.1s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
```

## 2. pingpong

### 2.1 实验步骤

a. 首先利用user.h中声明的系统调用pipe创建两个分别用于从父进程向子进程传输数据和从子进程向父进程传输数据的管道，并创建子进程。

```
int f2c[2];
int c2f[2];

pipe(f2c);
pipe(c2f);

int pid = fork();
```

b. 若 $pid > 0$ ，则表明该进程为父进程， $pid$ 为子进程的 $pid$ 。此时关闭 $f2c$ 的读端，关闭 $c2f$ 的写端。而后通过 $getpid$ 获取父进程的 $pid$ ，将"ping"和父进程 $pid$ 写入 $f2c$ 。之后读 $c2f$ 等待子进程传输数据，接收到数据后进行打印。

```
if(pid > 0){
    close(f2c[0]);
    close(c2f[1]);

    int ppid = getpid();
    write(f2c[1], "ping", 4);
```

```

write(f2c[1], (char*)&ppid, sizeof(ppid));
close(f2c[1]);

char buf[5];
read(c2f[0], buf, 4);
buf[4] = '\0';

printf("%d: received %s from pid %d\n", ppid, buf, pid);
close(c2f[0]);
}

```

c. 若pid=0，则表明该进程为子进程。此时关闭c2f的读端，f2c的写端。而后通过getpid获取子进程pid，并在f2c的读端读取父进程传输的信息以及父进程pid。读取后先进行打印再向父进程传输数据以保持打印顺序。

```

else{
    close(c2f[0]);
    close(f2c[1]);

    int cpid = getpid();
    int ppid;

    char buf[5];
    read(f2c[0], buf, 4);
    buf[4] = '\0';
    read(f2c[0], (char*)&ppid, sizeof(ppid));

    printf("%d: received %s from pid %d\n", cpid, buf, ppid);

    write(c2f[1], "pong", 4);
    close(c2f[1]);
}

```

## 2.2 实验测试结果

```

piacesy@piacesy:~/Code/xv6-oslab24$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong lenient testing == pingpong lenient testing: OK (0.7s)
== Test pingpong strict testing with changing pids == pingpong strict testing with changing pids: OK (1.0s)

```

## 3. find

### 3.1 实验步骤

a. 对于程序主体，首先检查参数量，而后调用实现的find函数进行递归查找。这里若path的最后一个字符为'/'则将其替换为'\0'是为了防止后续调用get\_basename函数出错。

```
int main(int argc, char* argv){
    if(argc != 3){
        fprintf(2, "Usage: find <path> <name>\n");
        exit(1);
    }
    char* path = argv[1];
    if(path[strlen(path)] == '/'){
        path[strlen(path)] = '\0';
    }
    char* name = argv[2];
    find(path, name);
    exit(0);
}
```

b. find函数接受path作为查询路径以及name作为查找的文件或文件夹名称。首先通过open系统调用获取path路径对应的文件描述符，而后再通过fstat函数获取stat结构体用于判断path对应的是文件还是文件夹。

```
if((fd = open(path, O_RDONLY)) < 0){
    fprintf(2, "cannot open %s\n", path);
    exit(1);
}

if(fstat(fd, &st) < 0){
    fprintf(2, "cannot stat %s\n", path);
    close(fd);
    exit(1);
}
```

c. 若对应的是文件，则通过实现的get\_basename函数获取path的文件名，并使用strcmp与name进行对比，若一致则进行打印。

```
if(st.type == T_FILE){
    char* filename = get_basename(path);
    if(strcmp(filename, name) == 0){
        printf("%s\n", path);
    }
}
```

d. 其中get\_basename函数使用strchr不断查找path中的'/'字符，若不存在则说明整个path为文件或文件名，否则将最后一个'/'之后的字符串返回作为文件名或文件名。

```
char* get_basename(char* path){
    char* prev_slash = 0;
    char* first_slash = strchr(path, '/');
    while (first_slash != 0){
        prev_slash = first_slash;
        first_slash = strchr(first_slash+1, '/');
    }
    return prev_slash? prev_slash+1: path;
}
```

e. 若对应的是文件夹，则同样先通过get\_basename函数获取文件夹名并进行比较，若一致则进行打印。之后使用read函数从文件夹中读取dirent结构体表示文件夹项，并跳过"."和".."(分别表示当前文件夹和上一级文件夹)。对于每一个文件夹项，将一个'/'和其名称拼接在path后作为新路径并调用find函数进行递归查询。

```
else if(st.type == T_DIR){
    char* dirname = get_basename(path);
    if(strcmp(dirname, name) == 0){
        printf("%s\n", path);
    }
    strcpy(buf, path);
    buf[strlen(path)] = '/';
    buf[strlen(path)+1] = '\0';
    while(read(fd, &de, sizeof(de)) == sizeof(de)){
        if(de.inum == 0 || strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0)
        {
            continue;
        }
        strcpy(buf+strlen(path)+1, de.name);
        find(buf, name);
    }
}
```

## 3.2 实验测试结果

```
● piacesy@piacesy:~/Code/xv6-oslab24$ ./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 (1.0s)
== Test find, in current directory and create a dir == find, in current directory and create a dir: OK (1.0s)
== Test find, find file recursive == find, find file recursive: OK (1.1s)
== Test find, find dir recursive with no duplicates == find, find dir recursive with no duplicates: OK (1.0s)
```

# sleep、pingpong、find测试结果

```
piacesy@piacesy:~/Code/xv6-oslab24$ ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (0.5s)
== Test sleep, returns == sleep, returns: OK (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.1s)
== Test pingpong lenient testing == pingpong lenient testing: OK (1.0s)
== Test pingpong strict testing with changing pids == pingpong strict testing with changing pids: OK (1.0s)
== Test find, in current directory and create a file == find, in current directory and create a file: OK (1.1s)
== Test find, in current directory and create a dir == find, in current directory and create a dir: OK (1.0s)
== Test find, find file recursive == find, find file recursive: OK (1.2s)
== Test find, find dir recursive with no duplicates == find, find dir recursive with no duplicates: OK (0.9s)
Score: 60/60
```

## 4. xv6启动流程实验

### 4.1 实验步骤

a. 由elearning给出的initcode调试流程可知，在scheduler.c的c->proc=p执行结束后，进程名变为initcode，故在其下一段代码处(proc.c第433行)打上临时断点后运行直到断点处并进行打印。

```
tbreak proc.c:433
continue
p cpus[$tp]->proc->name
```

b. 由xv6启动过程可知，sysfile.c在int ret=exec(path, argv)调用了init程序，故在其下一段代码处(sysfile.c第384行)打上临时断点后运行直到断点处并进行打印。

```
tbreak sysfile.c:384
continue
p cpus[$tp]->proc->name
```

### 4.2 实验测试结果

```
(gdb) source user/commands.gdb
Temporary breakpoint 1 at 0x800020d6: file kernel/proc.c, line 433.

Temporary breakpoint 1, scheduler () at kernel/proc.c:433
433          swtch(&c->context, &p->context);
$1 = "initcode\000\000\000\000\000\000\000"
Temporary breakpoint 2 at 0x80005bf8: file kernel/sysfile.c, line 384.

Temporary breakpoint 2, sys_exec () at kernel/sysfile.c:384
384      for (i = 0; i < NELEM(argv) && argv[i] != 0; i++) kfree(argv[i]);
$2 = "init\000\000de\000\000\000\000\000\000"
```

## 二、问题解答

### 2.1 问题1内容

- 在pingpong实验中使用pipe创建两个分别用于从父进程向子进程传输数据(f2c)和从子进程向父进程传输数据(c2f)的管道。而后使用fork创建子进程。
- 若pid>0，则表明该进程为父进程，pid为子进程的pid。此时关闭f2c的读端，关闭c2f的写端。而后通过getpid获取父进程的pid，将"ping"和父进程pid写入f2c。之后读c2f等待子进程传输数据，接收到数据后进行打印。
- 若pid=0，则表明该进程为子进程。此时关闭c2f的读端，f2c的写端。而后通过getpid获取子进程pid，并在f2c的读端读取父进程传输的信息以及父进程pid。读取后先进行打印再先父进程传输数据以保持打印顺序。

### 2.2 问题2内容

- 若读进程未及时关闭其写端，则在写进程在写完数据并关闭写端时，读进程若继续进行读操作，则无法在读端读到EOF，最终阻塞在read调用上。
- 若写进程未及时关闭其读端，则若写进程在读进程关闭读端后继续写入，则会导致其阻塞在write调用上。

### 2.3 问题3内容

- 结果：若子进程没有关闭写端，则子进程会持续阻塞。
- 原因：子进程在调用wc后，程序会持续等待来自管道读端的输出直到EOF。然而由于子进程未关闭写端，所以即使父进程关闭写端后，管道的读操作仍然不会返回

EOF，这就会导致子进程一直阻塞。