OS-lab6-challenge

EASY

实现后台运行

shell的原理是umain 从控制台读取一行后fork，把这一行命令传递给子进程。子进程执行完毕后退出，父进程调用wait函数等待子进程执行结束被摧毁。

因此，后台运行所需要的任务便是**shell 不需要等待此命令执行完毕后再继续执行**，即当存在&时，不进行wait。

在user/sh.c中添加一个局部变量

```
void runcmd(char *s) {
...
   int nowait = 0;
...
}
```

在解析到&时将其赋值为1。

```
case '&':
            nowait = 1;
            break;
        }
```

在后续，shell 等待 spawn 出来的进程执行完命令。这里根据noWait变量进行判断，是否进行等待。

```
if (r >= 0) {
        if (debug_) writef("[%08x] WAIT %s %08x\n", env->env_id, argv[0], r);
        if (!nowait) wait(r);
    }
```

实现一行多命令

查阅user/sh.c，可发现在runcmd函数中，对于特殊的token，进行switch，那么只需要将；视为一个特殊token即可。

```
case ';':
            forktemp = fork();
            if (forktemp) {
                wait(forktemp);
                goto again;
            } else {c
                goto runit;
            }
            break;
```

可以注意到，这部分可以有很多处理方法，类似 | token也可以。

我选择了一种比较简练的方法，当读到；时，强制结束当前输入，直接启用fork，待其执行结束后，继续读接下来的命令。

虽然这种方式会用到goto语句，显得很不雅致。但既然其他case都使用了这点，也就不在意了。

测试

输入

```
echo.b 123;echo.b 321
```

输出

如无特殊说明，此后的测试均不再输出多余内容

```
serve_open 00000400 ffff000 0x105
[00002404] SPAWN: echo.b 123
serve_open 00002404 ffff000 0x0

::::::::::spawn size : 20  sp : 7f3fdfd8::::::::
123
[00002c05] destroying 00002c05

[00002c05] free env 00002c05

i am killed ...

[00002404] destroying 00002404

[00002404] free env 00002404

i am killed ...
```

```
[00001c03] SPAWN: echo.b 321
serve_open 00001c03 ffff000 0x0

::::::::::spawn size : 20  sp : 7f3fdfd8::::::::
321
[00002404] destroying 00002404

[00002404] free env 00002404

i am killed ...

[00001c03] destroying 00001c03

[00001c03] free env 00001c03

i am killed ...
```

实现引号支持

与一行多命令相仿。查阅user/sh.c，有int _gettoken()函数。作用是从字符串中读取下一个token。

对于特殊的SYMBOLS（例如 | & ;等，上文便是对其进行操作），会直接返回，对于结束，会返回0，对于一个单词，会返回 w 。同时将两个指针 *p1 *p2分别设置为word开头字符和结尾后字符。

那么本任务的目的，就是使得这个函数，对于 "" 内的内容，视为同一个token，即返回 w 。

仿照对于空白符的处理

```
while(strchr(WHITESPACE, *s))*
    s++ = 0;
```

可以得到

```
if(*s == '"')
{
    *s = 0;
    *p1 = ++s;
    while(s != 0 && *s != '"')
    {
        s++;
    }
    *s = 0;
    *p2 = s;
    return 'w';
}
```

注意到需要进行两次 `*s=0` ,这是为了防止将 "" 本身识别为token。

测试

输入c

```
echo.b "123|321"
```

输出

```
123|321
```

实现额外命令

命令添加的流程如下：

- 在user里面写xx.c，其中保留`void umain(int argc, char **argv)`
- 在user/Makefile里面添加xx.b
- 在fs/Makefile里面添加xx.b

如果lab6有课上exam的话，我想这就是内容之一吧

tree

这其实是一道dfs搜索算法题目，难度并不大。

虽然tree的打印看起来复杂，但如果按照深度排序，则可以发现明显的规律，为四字一组。

```
for (i = 0; i < depth; i++) {
    if (i == depth - 1) {
        if (finalFile[i]) {
            fwritef(1, "`-- ");
        } else {
            fwritef(1, "|-- ");
        }
    } else {
        if (finalFile[i]) {
            fwritef(1, "    ");
        } else {
            fwritef(1, "|   ");
        }
    }
}
```

按照深度逐层输出即可

注意到结构体中f_type这个属性，表示为文件夹或者文件。可以根据此来设计输出的颜色，把文件夹设置为蓝色。

```
if (fileFd->f_file.f_type == 1) {
        fwritef(1, "\033[34m%s\033[m\n", fileFd->f_file.f_name);
    } else {
        fwritef(1, "%s\n", fileFd->f_file.f_name);
    }
```

mkdir/touch

其实主要难点复制某次exam便可

include/fs.h 新增结构体和对应的宏：

```
struct Fsreq_create {
    u_char req_path[MAXPATHLEN];c
    int type;
};
#define FSREQ_CREATE    8
```

在fs/serv.c实现serve_create函数

注意到此处的type便是上文提到的标识符，是否为文件夹

```
void serve_create(u_int envid, struct Fsreq_create *rq)
{
    writef("serve_create: %s\n", rq->req_path);
    int r;
    char *path = rq->req_path;
    struct File *file;
    if((r = file_create(path, &file)) < 0) {
        ipc_send(envid, r, 0, 0);
        return;
    }
    file->f_type = rq->type;
    ipc_send(envid, 0, 0, 0);
}
```

在 fs/serv.c 中 修改serve函数，添加如下：

```
case FSREQ_CREATE:
    serve_create(whom, (struct Fsreq_create *)REQVA);
    break;
```

在fsipc.c/fsipc_create中添加

```
int fsipc_create(const char* path, int type)
{
    struct Fsreq_create *req;
    req = (struct Fsreq_create*) fsipcbuf;
    req->type = type;
    strcpy((char*) req->req_path, path);
    return fsipc(FSREQ_CREATE, req, 0, 0);
}
```

这样就可以在用户态的file.c添加

```
int create_file(const char* path)
{
    return fsipc_create(path, 0);
    //create_dir同理
    //return fsipc_create(path, 1);
}
```

而二者的umain函数便十分简单

```
void
umain(int argc, char **argv) {
   if (argc == 2) {
        if (create_file(argv[1]) < 0) {
        //mkdir同理
        //if (create_dir(argv[1]) < 0) {
            fwritef(1, "create file failed!\n");
        } else {
            fwritef(1, "\033[32mSuccess touch\033[m\n");
        }
   } else {
        fwritef(1, "usage: touch [filename]\n");
   }
}
```

测试

输入

```
mkdir.b s1;mkdir.b s1/s2;mkdir.b s3;touch.b s1/s2/s4;touch.b s3/s5;tree.b;tree.b
s1
```

输出

如无特殊说明，与跳板机中不同，实验报告中的输出反馈不带颜色

```
Success mkdir

Success mkdir

Success mkdir

Success touch

Success touch

Success touch

/
|-- motd
|-- newmotd
|-- testarg.b
|-- init.b
|-- num.b
|-- echo.b
```

```
|-- ls.b
|-- sh.b
|-- cat.b
|-- touch.b
|-- mkdir.b
|-- tree.b
|-- history.b
|-- declare.b
|-- unset.b
|-- testptelibrary.b
|-- .history
|-- s1
|   `-- s2
|       `-- s4
`-- s3
    `-- s5
Success tree

s1
`-- s2
    `-- s4
Success tree
```

Normal

简要分析，此任务可以被分为五个主要内容

- 起始时创建.hitsoty
- 对.history进行**不覆盖**写入
- 监听上下键
- 替换输入字符
- 以一定格式读取输出.history

其中任务1过于简单，参见touch的实现，任务5的在任务3中已经实现，均不再赘述。

不覆盖写入

Lab5-2-exam中其实已经对此有实现，只需要复制当时的课上代码即可

在user/lib.h中添加

```
#define O_APPEND    0x0004
```

在user/sh.c中添加

```
void save_cmd(char* cmd) {
   int r = open(".history", O_CREAT | O_WRONLY | O_APPEND);
   if (r < 0) {
       fwritef(1, "open .history failed!");
       return r;
   }
   write(r, cmd, strlen(cmd));
   write(r, "\n", 1);
   return 0;
}
```

并在umain函数中添加调用

```
for(;;){
       if (interactive)
           fwritef(1, "\n$ ");
       readline(buf, sizeof buf);
       save_cmd(buf);
```

值得注意的是，我此处是在readline之后立刻进行添加。这是因为我对于上下键的处理，是通过在readline函数中做手脚实现的。这样是为了避免上下键对.history的写入。

监听上下键

上下键在linux中会被编码为

上：`27 '[' 'A'`

下：`27 '[' 'B'`

需要对于连续字符进行特判，这里使用read对接下来的字符进行特殊处理。

```
if(buf[i] == 27) {
    char tmp;
    read(0, &tmp, 1);
    char tmp2;
    read(0, &tmp2, 1);
    if (tmp == 91 && tmp2 == 65) {
        //上键处理
        ...
    } else if (tmp == 91 && tmp2 == 66) {
        //下键处理
        ...
    }
```

这里定义了一个全局变量 keyboardSPL 用来记录当前是否已经更新过指令。即，连续两次上键会追溯倒二条指令，但是如果中间夹杂了其他按键，第二次上键应该只追溯倒一指令。

```
if(buf[i] == 27) {
    ...
    if (tmp == 91 && tmp2 == 65) {
            //上键处理
            ...
            keyboardSPL = 0;
```

```
        } else if (tmp == 91 && tmp2 == 66) {
            //下键处理
            ...
            keyboardSPL = 0;
        } else {
            keyboardSPL = 1;
        }
} else {
    keyboardSPL = 1;
}
```

对于.history文件的读取主体逻辑如下，并不复杂，只需要注意特判空文件即可。

此外，需要注意，读取字符串后，不应该以\0结尾。这是因为通过上键追溯指令后，用户还可能以对其进行增添。因此此处不能像往常一样，以\0结束字符串。

```
int fdnum = open(".history", O_RDONLY);
struct Fd *fd = num2fd(fdnum);
char *c;
char *begin = fd2data(fd);
char *end = begin + ((struct Filefd*)fd)->f_file.f_size;
...
int now = 0;
while ((*c) != '\n' && (*c) != '\0' && (*c) < end) {
    buf[now] = *c;
    now++;
    c++;
}
```

替换输入字符

替换输入字符串并不复杂，但难点在于如何使得前一条指令正确的回显。

这里用到了linux对于左键和清空键的编码

左：`27 '[' 'D'`

清空：`27 '[' 'K'`

```
fwritef(1, "\x1b[B");
int j;
for (j = 0; j < i-1; j++) {
    fwritef(1, "\x1b[D");
}
fwritef(1, "\x1b[K");
......
fwritef(1, "%s", buf);
```

- `fwritef(1, "\x1b[B")` 使得光标在上移后移回
- `fwritef(1, "\x1b[D")` 使得光标左移回起点，注意此处 i 是当前输入指令的长度，而不是回溯之后的长度
- `fwritef(1, "\x1b[K")` 使得光标所在行的输入清空，也就是清空先前的指令。
- `fwritef(1, "%s", buf)` 将从.history中读取到的指令回显

至此，Normal部分完成

测试

输入

```
echo.b 1
echo.b 2
echo.b 13 //注:此条为上键两次后再输入3与回车
history.b
```

输出

```
1

2

13

History 0 : echo.b 1
History 1 : echo.b 2
History 2 : echo.b 13
History 3 : history.b
```

Challenge

此部分选择的内容为**实现 shell 环境变量**

需求实现

对问题进行分析，可以发现一条基本的思路。

首先，全局变量和局部变量是逻辑一致的。或者说，实现局部变量后，全局变量是局部变量的特例。

那么，环境变量并不能放置在用户态，而是必须放入内核态，通过系统调用的方式进行获取。

既然需要系统调用，那么就需要对shell进行表示，用来作为变量寻访的唯一标识。

变量承载了如此多的属性（所属shell的id标识符，可见性，name，value）等等，那么最好类比OO中的思想，使用一个结构体来储存（没办法c不是c++，不支持面向对象）。

```
struct ENV_VALUE{
   char name[MAX_NAME_LEN];
   int value;
   int shell_id;    //0:global，否则对应各自的shellid。
   int rdonly;      //由于题目并没给出其他权限要求，仅用0/1表示是否只读
   int alive;       //此处做了简化，并没有实际unset变量，而是将其deactivate
} env_value[MAX_VALUE_NUM];
```

相应的输入输出方法并不复杂，这里不再赘述。只是一些很基本的c语言知识，新建结构体，添加到结构体数组中，查询全部或者单个等等。

值得注意的是输出，此处我做了些许简化，以下是lib/env_value部分示例

```
...
printf("\033[36mName : \033[m\033[32m%s\033[m ", env_value[i].name);
```

```
printf("\033[36mValue : \033[m\033[32m%d\033[m ", env_value[i].value);
if (env_value[i].rdonly) {
    printf("\033[36mReadOnly : \033[m\033[32mYES\033[m ");
} else {
    printf("\033[36mReadOnly : \033[m\033[31mNo\033[m  ");
}
if (env_value[i].shell_id) {
    printf("\033[36mVisibility : \033[m\033[32mLOCAL\033[m \n");
} else {
    printf("\033[36mVisibility : \033[m\033[31mGLOBAL\033[m\n");
}
...
```

注意到截取函数的来源是lib/env_value.c。所以不能使用传统的 `fwritef` 或者 `writef`。只能使用 `printf` 函数。因为参数的返回值只能是一个int，这可能稍微有些遗憾。

在构建完成这些内容后，需要设置对应的系统调用。

在user/syscall_lib.c中添加四个系统调用

```
int
syscall_create_shell_id() {
    return msyscall(SYS_create_shell_id, 0, 0, 0, 0, 0);
}

int
syscall_declare_env_value(char* name, int value, int shell_id, int type) {
    return msyscall(SYS_declare_env_value, name, value, shell_id, type, 0);
}

int
syscall_unset_env_value(char* name, int shell_id) {
    return msyscall(SYS_unset_env_value, name, shell_id, 0, 0, 0);
}

int
syscall_get_env_value(char* name, int type, int shell_id) {
    return msyscall(SYS_get_env_value, name, type, shell_id, 0, 0);
}
```

在user/lib.h 声明

```
int syscall_create_shell_id();
int syscall_declare_env_value(char *name, int value, int shell_id, int type);
int syscall_unset_env_value(char *name, int shell_id);
int syscall_get_env_value(char *name, int type, int shell_id);
```

在include/unistd.h中添加对应的系统调用

```
#define SYS_create_shell_id ((__SYSCALL_BASE ) + (17) )
#define SYS_declare_env_value ((__SYSCALL_BASE ) + (18) )
#define SYS_unset_env_value ((__SYSCALL_BASE ) + (19) )
#define SYS_get_env_value ((__SYSCALL_BASE ) + (20) )
```

在lib/syscall_all.c中完成

```
int sys_create_shell_id(int sysno) {
    return create_shell_id();
}

int sys_declare_env_value(int sysno, char* name, int value, int shell_id, int
type) {
    return declare_env_value(name, value, shell_id, type);
}

int sys_unset_env_value(int sysno, char* name, int shell_id) {
    return unset_env_value(name, shell_id);
}

char* sys_get_env_value(int sysno, char* name, int type, int shell_id) {
    return get_env_value(name, type, shell_id);
}
```

而这四个函数，对应的调用位置和功能如下：

user/sh.c

- create_shell_id()： 在shell创建时，为其申请一个独一的id，并输出。

```
void umain(int argc, char **argv) {
    ...
    shell_id = syscall_create_shell_id();
    writef("\033[34mThis shell's id :\033[m \033[32m%d\033[m\n", shell_id);
    ...
}
```

user/declare.c

- get_env_value()： 获取环境变量，输入参数为1，即输出全部
- declare_env_value()： 声明环境变量

```
void umain(int argc, char **argv) {
    ...
    if (argc == 2) {
        syscall_get_env_value(NULL, 1, shell_id);
    } else {
        ... //进行各种处理获得token
        syscall_declare_env_value(name, value, shell_id, rdonly);
    }
}
```

user/unset.c

- unset_env_value()： 删除环境变量（在我的实现中为不激活环境变量）

```
void umain(int argc, char **argv) {
    ...
    syscall_unset_env_value(name, shell_id);
}
```

user/echo.c

- get_env_value()：获取环境变量，输入参数为0，即输出对应名称。

```
void umain(int argc, char **argv) {
    ...
    int value = syscall_get_env_value(&argv[i][1], 0, shell_id);
    ...
}
```

由此，实现完毕

和其他同学交流后，发现也有仿照normal，设置.envValue文件，通过对文件读写的方式，完成环境变量挑战的思路。

测试

输入

```
declare.b v1 =1          //无参定义
declare.b -r v2 =2       //定义只读
declare.b -r v2 =3       //对只读进行重定义尝试
declare.b -x v3 =3       //定义全局变量
declare.b -xr v4         //定义全局只读变量，默认值为0
declare.b                //输出
sh.b                     //新建子shell
declare.b                //输出
unset.b v3               //删除v3
unset.b v4               //删除只读变量尝试
declare.b                //输出
echo.b $v4               //echo输出单变量
```

输出

```
This shell's id : 1

Success create value

Success create value

WARNING : Redeclare readonly value

Success create value

Success create value

Name : v1 Value : 1 ReadOnly : No  Visibility : LOCAL

Name : v2 Value : 2 ReadOnly : YES Visibility : LOCAL
```

```
Name : v3 Value : 3 ReadOnly : No  Visibility : GLOBAL

Name : v4 Value : 0 ReadOnly : YES Visibility : GLOBAL

This shell's id : 2

Name : v3 Value : 3 ReadOnly : No  Visibility : GLOBAL

Name : v4 Value : 0 ReadOnly : YES Visibility : GLOBAL

Success unset value

WARNING : Unset readonly value

Name : v4 Value : 0 ReadOnly : YES Visibility : GLOBAL

Name : v4 Value : 0
```

后记

修改user/sh.c可以输出一个漂亮的LOGO（甚至彩色）

但要修改user/printf.c中的user_myoutput，使得每次换行，不再多换一行

效果展示：

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::  __          __            _ _                 _   _           _  ::
::  \ \        / /           (_) |               | | (_)         | | ::
::   \ \  /\  / /__ _ __ ___  _| |_ _ ____      _| |_ _ _ __   __| | ::
::    \ \/  \/ / __| '_ ` _ \| | __| '_ \ \ /\ / / __| | '_ \ / _` | ::
::     \  /\  /\__ \ | | | | | | |_| |_) \ v  v /| |_| | | | | (_| | ::
::      \/  \/ |___/_| |_| |_|_|\__| .__/ \_/\_/  \__|_|_| |_|\__,_| ::
::                                 | |                               ::
::                                 |_|       github.com/wsmitpwtind  ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```