

必做部分

实现一行多命令

用 ; 分开同一行内的两条命令，表示依次执行前后两条命令。; 左右的命令都可以为空。

提示：在 user/sh.c 中的保留 SYMBOLS 里已经预留有 ; 字符。

在 user/sh.c 中的 parsecmd 函数中的 switch 增加 ; 的情况

```
case ';':  
    if((*rightpipe = fork()) == 0) {  
        return argc;  
    } else {  
        wait(*rightpipe);  
        do{  
            close(0);  
            if((r = opencons())<0){  
                user_panic("1");  
            }  
        }while(r!=0);  
        dup(0,1);  
        return parsecmd(argv, rightpipe);  
    }  
    break;
```

这里主要参考了|的写法。

- 但是要注意这里需要依次执行前后两条命令。所以这里需要先执行左侧命令（子进程），再执行右侧命令（父进程）。
- 且这里假如不重新设置父进程的标准输入和输出，这里会出现 ls.b|cat.b;cat.b;这一函数无法运行；之后的函数。

测试

- 一行多命令

ls.b;ls.b

```

$ ls.b;ls.b;
[00003805] destroying 00003805
[00003805] free env 00003805
i am killed ...
aaa.txt testarg.b cat.b pingpong.b testbss.b newmotd testpiperace.b testpipe.b motd init.b num.b lorem testfdsharing.b testshell.sh script ls.b echo.b sh.b halt.b testptelibrary.b
[00004005] destroying 00004005
[00004005] free env 00004005
i am killed ...
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
aaa.txt testarg.b cat.b pingpong.b testbss.b newmotd testpiperace.b testpipe.b motd init.b num.b lorem testfdsharing.b testshell.sh script ls.b echo.b sh.b halt.b testptelibrary.b
[00004804] destroying 00004804
[00004804] free env 00004804
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...

```

- 与重定向适配

```
ls|cat;cat;
```

```

$ ls|cat;cat;
[00005003] pipecreate
aaa.txt testarg.b cat.b pingpong.b testbss.b newmotd history.b testpiperace.b testpipe.b motd init.b num.b lorem touch.b mkd
ir.b testbackend.b testfdsharing.b testshell.sh declare.b script ls.b echo.b sh.b tree.b unset.b halt.b testptelibrary.b .hi
story
[00006806] destroying 00006806
[00006806] free env 00006806
i am killed ...
[00006005] destroying 00006005
[00006005] free env 00006005
i am killed ...
[00007007] destroying 00007007
[00007007] free env 00007007
i am killed ...
aaaaaa|

```

实现后台任务

用 & 分开同一行内的两条命令，表示同时执行前后两条命令。& 左侧的命令应被置于后台执行，Shell 只等待 & 右侧的命令执行完毕，然后继续执行后续语句，此时用户可以输入新的命令，并且可能同时观察到后台任务的输出。你需要自行设计测试，以展现此功能的运行效果。& 左侧的命令不能为空。

提示：在 user/sh.c 中的保留 SYMBOLS 里已经预留有 & 字符。

- 处理&字符

在 user/sh.c 中的 parsecmd 函数中的 switch 增加&的情况

```

case '&':
    if((*rightpipe = fork()) == 0) {
        return argc;
    } else {
        return parsecmd(argv, rightpipe);
    }
    break;

```

- 解决后台忙等问题

在 kern/syscall_all.c 中解决忙等问题

```
int sys_cgetc(void) {
    int ch;
    ch = scancharc();
    return ch;
}
```

- 测试函数

testbackend.b

```
#include <lib.h>
int main(){
    printf("testbackend begin\n");
    for(int i=0;i<1000000000;i++){
        if (i % 100000000 == 0) {
            printf("hello, i is %d\n\n", i);
        }
    }
}
```

测试

testbackend.b&
ls.b;

```
$ testbackend.b&
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...

$ testbackend begin
hello, i is 0

[00004005] destroying 00004005
[00004005] free env 00004005
i am killed ...

$ ls.b
aaa.txt testarg.b cat.b pingpong.b testbss.b newmotd testpiperace.b testpipe.b motd init.b num.b lorem testbackend.
b testfdsharing.b testshell.sh script ls.b echo.b sh.b halt.b testptelibrary.b

[00005006] destroying 00005006
[00005006] free env 00005006
i am killed ...
[00004805] destroying 00004805
[00004805] free env 00004805
i am killed ...

$ hello, i is 100000000
hello, i is 200000000
```

实现引号支持

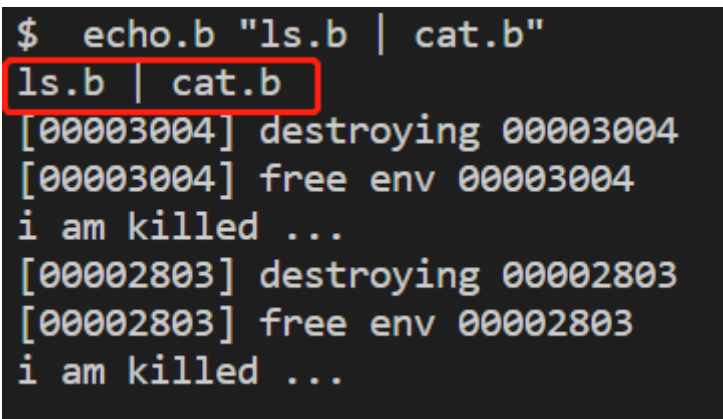
实现引号支持后，shell 可以处理如： `echo.b "ls.b | cat.b"` 这样的命令。即 shell 在解析时，会将双引号内的内容看作单个字符串，将 `ls.b | cat.b` 作为一个参数传递给 `echo.b`。

在 `user/sh.c` 中修改了 `_gettoken` 函数，补充了这一个条件，使得双引号中间的内容识别为统一参数，返回 `w`。

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

测试

```
echo.b "ls.b | cat.b"
```



```
$ echo.b "ls.b | cat.b"
ls.b | cat.b
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...
```

实现键入命令时任意位置的修改

现有的 shell 不支持在输入命令时移动光标。你需要实现：键入命令时，可以使用 `Left` 和 `Right` 移动光标位置，并可以在当前光标位置进行字符的增加与删除。要求每次在不同位置键入后，可以完整回显修改后的命令，并且键入回车后可以正常运行修改后的命令。

在 `user/sh.c` 中修改了 `readline` 函数，一方面对于 `buf` 进行维护，另外一方面对于回显进行维护。对于 `buf` 的维护主要是通过修改 `i`，并在最后截断 `buf` 函数。对于回显进行维护则是实现了 `MOVELEFT`

和 MOVERIGHT 函数来显示光标的移动和在修改后 printf 新的 buf 来修改回显。

```

void readline(char *buf, u_int n)
{
    int len = 0;
    int r;
    char temp;
    for (int i = 0; i < n; i++)
    {
        if ((r = read(0, &temp, 1)) != 1)
        {
            if (r < 0)
            {
                debugf("read error: %d\n", r);
            }
            exit();
        }
        if (temp == '\b' || temp == 0x7f)
        {
            if (i > 0)
            {
                if (i == len)
                {
                    buf[i - 1] = 0;
                    printf("\b \b");
                }
                else
                {
                    for (int j = i - 1; j < len - 1; j++)
                    {
                        buf[j] = buf[j + 1];
                    }
                    buf[len - 1] = 0;
                    MOVELEFT(i);
                    printf("%s ", buf);
                    MOVELEFT(len - i + 1);
                }
                len--;
                i -= 2;
            }
            else
            {
                i = -1;
            }
        }
        else if (temp == '\033')
        {
            char temp1, temp2;
            if ((r = read(0, &temp1, 1)) != 1)
            {
                if (r < 0)
                {

```

```

        debugf("read error: %d\n", r);
    }
    exit();
}
if (temp1 != '[')
    user_panic("\\033 is not followed by '['");
if ((r = read(0, &temp2, 1)) != 1)
{
    if (r < 0)
    {
        debugf("read error: %d\n", r);
    }
    exit();
}

if (temp1 == '[')
{
    switch (temp2)
    {
        case 'D': // 左键
            if (i > 0)
            {
                i -= 2;
            }
            else
            {
                i = -1;
                MOVERIGHT(1);
            }
            break;
        case 'C': // 右键
            if (i < len)
            {
                i = i;
            }
            else
            {
                i = len - 1;
                MOVELEFT(1);
            }
            break;
        default:
            i--;
            break;
    }
}

}
else if (temp == '\n' || temp == '\r')
{
    buf[len] = 0;
    return;
}

```

```

    }
    else
    {
        if (i == len)
        {
            buf[i] = temp;
        }
        else
        {
            for (int j = len; j > i; j--)
            {
                buf[j] = buf[j - 1];
            }
            buf[i] = temp;
            buf[len + 1] = 0;
            printf("%s", buf + i + 1);
            MOVELEFT(len - i);
        }
        len += 1;
    }
    if (len >= n)
    {
        break;
    }
}
debugf("line too long\n");
while ((r = read(0, buf, 1)) == 1 && buf[0] != '\r' && buf[0] != '\n')
{
    ;
}
buf[0] = 0;
}

void MOVELEFT(int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("\b");
    }
}

void MOVERIGHT(int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("\033[C");
    }
}

```


测试

通过输入 12345'\033[D'\033[D'\b'a'\033[C'b'\n'来测试。简而言之则是输入 12345，然后按左方向键两次，再按 backspace 键一次，输入 a，再按右方向键一次，输入 b，回车。

```
$ 12a4b5
spawn 12a4b5: -10
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...
```

;

实现程序名称中 .b 的省略

目前的用户程序被烧录到文件系统中后，其可执行文件以 .b 为后缀，为 shell 中命令的输入带来了不便。你需要修改现有的实现，以允许命令中的程序名称省略 .b 后缀，例如当用户指定的程序路径不存在时，尝试在路径后追加 .b 再打开。

在 user/lib/spawn.c 中的 spawn 函数中，当用户指定的程序路径不存在时，尝试在路径后追加 .b 再打开。

```
int fd;
if ((fd = open(prog, O_RDONLY)) < 0)
{
    char tmp[1024];
    int i;

    for (i = 0; *(prog + i) != '\0'; i++)
    {
        *(tmp + i) = *(prog + i);
    }
    *(tmp + i) = '.';
    *(tmp + i + 1) = 'b';
    *(tmp + i + 2) = '\0';
    if ((fd = open(tmp, O_RDONLY)) <= 0)
    {
        return fd;
    }
}
```

测试

ls

```
$ ls
aaa.txt testarg.b cat.b pingpong.b testbss.b newmtd history.b testpiperace.b testpipe.b mtd init.b num.b lorem touch.b mkd
ir.b testbackend.b testfdsharing.b testshell.sh declare.b script ls.b echo.b sh.b tree.b unset.b halt.b testptelibrary.b .hi
story
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...
```

这里要注意不能直接修改 prog 这个 char 指针，会把 prog 后面的内容覆盖掉。我们需要用 tmp 数组来存这个内容。

实现更丰富的命令

参考实验环境中的 Linux 命令 tree、mkdir、touch 来实现这三个命令，请尽可能地实现其完整的功能。

为了实现文件和目录的创建，你需要实现用户库函数 mkdir() 和文件打开模式 O_CREAT。

实现文件的创建后，你需要修改 shell 中输出重定向 > 的实现，使其能够在目标路径不存在时自动创建并写入该文件。

- tree 函数

利用 dfs 生成树。主要函数如下。

```

void dfs(char *path, int depth, int finalFile[])
{
    struct Fd *fd;
    struct FileFd *fileFd;
    int r;
    if ((r = open(path, O_RDONLY)) < 0)
    {
        debugf("%s [error opening dir]\n", path);
        return;
    }

    for (int i = 0; i < depth; i++)
    {
        if (i == depth - 1)
        { // 到文件那层了
            if (finalFile[i])
            { // 是最后一个
                printf("└─ ");
            }
            else
            {
                printf("├─ ");
            }
        }
        else
        { // 仍未到
            if (finalFile[i])
            {
                printf("  ");
            }
            else
            {
                printf("│  ");
            }
        }
    }

    fd = num2fd(r);
    fileFd = (struct FileFd *)fd;
    if (fileFd->f_file.f_type == FTYPE_REG)
    { // 是文件
        printf("%s\n", fileFd->f_file.f_name);
        num_file++;
        return;
    }
    else
    { // 是目录
        printf("\033[34m%s\033[m\n", fileFd->f_file.f_name);
        num_dir++;
    }
}

```

```

u_int size = fileFd->f_file.f_size;
u_int num = ROUND(size, sizeof(struct File)) / sizeof(struct File);
struct File *file = (struct File *)fd2data(fd);
u_int file_num = 0; // 记录目前文件块数量
for (int i = 0; i < num; i++)
{
    // 如果是一个文件
    if (file[i].f_name[0] != '\0')
    {
        file_num++;
    }
}

u_int j = 0; // 已处理的有效子文件
for (int i = 0; i < num; i++)
{
    if (file[i].f_name[0] == '\0')
    {
        continue;
    }

    // 制作新的path
    char nextPath[1024];
    memset(nextPath, 0, 1024);
    strcpy(nextPath, path);
    int len = strlen(nextPath);
    if (path[len - 1] == '/' && len != 1)
    {
        return;
    }
    // debugf("path = %s depth = %d len = %d file_name = %s\n\n", path, depth, len, file[i].
    nextPath[len] = '/';
    len++;
    strcpy(nextPath + len, file[i].f_name);

    if (j == file_num - 1)
    {
        finalFile[depth] = 1;
    }

    dfs(nextPath, depth + 1, finalFile);
    j++;
}
finalFile[depth] = 0;
}

```

- mkdir

在 user/include/lib.h 中添加 fsipc_create 函数

```
int fsipc_create(const char *path, int type);
```

在 fsreq.h 文件中增加 CREATE 的相关信息

```
#define FSREQ_CREATE 8

struct Fsreq_create
{
    char req_path[MAXPATHLEN];
    int type;
};
```

同时在 user/lib/fsipc.c 中参考 fsipc_open 增加 fsipc_create 函数

```
int fsipc_create(const char *path, int type)
{
    struct Fsreq_create *req;

    req = (struct Fsreq_create *)fsipcbuf;

    // The path is too long.
    if (strlen(path) >= MAXPATHLEN)
    {
        return -E_BAD_PATH;
    }

    strcpy((char *)req->req_path, path);
    req->type = type;
    return fsipc(FSREQ_CREATE, req, 0, 0);
}
```

对应的，在 serve.c 文件中需要在 serve 函数中添加 case FSREQ_CREATE。同时添加 serve_create 函数

```

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

void serve_create(u_int envid, struct Fsreq_create *rq)
{
    struct File *f;
    int r;

    if ((r = file_create(rq->req_path, &f)) < 0)
    {
        ipc_send(envid, r, 0, 0);
        return;
    }
    f->f_type = rq->type;
    ipc_send(envid, 0, 0, 0);
}

```

最后，创建一个 mkdir.c 文件就可以了

```

#include <lib.h>

int main(int argc, char **argv)
{
    int fd;
    if (argc != 2)
    {
        printf("Usage: mkdir directory_name\n");
        return;
    }
    if ((fd = open(argv[1], O_RDONLY)) >= 0)
    {
        printf("Directory already exists\n");
        return;
    }
    if ((fd = create(argv[1], FTYPE_DIR)) < 0)
    {
        printf("Fail to create directory %s\n", argv[1]);
    }
    return 0;
}

```

- touch

touch 的命令跟 mkdir 相差不大，主要是假如没有找到输入的文件，就要创建一个新的文件。所以将 mkdir 中 create 的类型变成 FTYPE_REG 就可以了。

```
#include <lib.h>

int main(int argc, char **argv)
{
    int fd;
    if (argc != 2)
    {
        printf("Usage: touch file_name\n");
        return;
    }
    if ((fd = open(argv[1], O_RDONLY)) >= 0)
    {
        printf("File %s touch success\n", argv[1]);
        return;
    }
    if ((fd = create(argv[1], FTYPE_REG)) < 0)
    {
        printf("Fail to create directory %s\n", argv[1]);
    }
    return 0;
}
```

测试

tree, mkdir,touch 功能

```
mkdir 1;mkdir 1/2;touch 1/a.txt;touch 1/2/b.txt;tree 1 /;
```

```

1
├── 2
│   ├── a.txt
│   └── b.txt
└── a.txt
/
├── aaa.txt
├── testarg.b
├── cat.b
├── pingpong.b
├── testbss.b
├── newmotd
├── history.b
├── testpiperace.b
├── testpipe.b
├── motd
├── init.b
├── num.b
├── lorem
├── touch.b
├── mkdir.b
├── testbackend.b
├── testfdsharing.b
├── testshell.sh
├── declare.b
├── script
├── ls.b
├── echo.b
├── sh.b
├── tree.b
├── unset.b
├── halt.b
├── testptelibrary.b
├── .history
└── 1
    ├── 2
    │   ├── a.txt
    │   └── b.txt
    └── a.txt
2 directories, 34 files

```

实现历史命令功能

在 Linux 的 shell 中我们输入的命令都会被保存起来，并可以通过 Up 和 Down 键回溯，这为我们的 shell 操作带来了极大的方便。在此项任务中，需要实现保存所有输入至 shell 的命令，并可以通

过 history.b 命令输出所有的历史命令，以及通过上下键回溯命令并运行。

任务提示：

要求我们将在 shell 中输入的每步命令，在解析前/后保存进一个专用文件（如 .history）中，每行一条命令。

通过编写一个用户态程序 history.b 文件并写入磁盘中，使得每次运行 history.b 时，能够将文件（.history）的内容全部输出。

键入 Up 和 Down 时，切换历史命令。键入上下键后，并且按回车，可以执行当前显示的这条命令。

注意

禁止使用局部变量或全局变量的形式实现保存历史命令，即不能用进程的堆栈区保存历史命令。

禁止在烧录 fs.img 时烧录一个 .history 文件，即你需要在第一次写入时，创建一个 .history 文件，并在随后每次输入时在 .history 文件末尾写入。

- 实现 O_APPEND

需要修改 file.c 的 open 文件，对 O_APPEND 进行特判。

```
if ((mode & O_APPEND) != 0)
{
    fd->fd_offset = size;
}
```

且需要在 user/include/lib.h 中加入 O_APPEND。

- 实现 history 命令

该命令等同于"cat /.history"，所以只要将 cat 命令复制过来，并做一些小修改就可以实现了。

```

#include <lib.h>
char buf[8192];

void history(int f, char \*s)
{
    long n;
    int r;

    while ((n = read(f, buf, (long)sizeof buf)) > 0)
    {
        if ((r = write(1, buf, n)) != n)
        {
            user_panic("write error copying %s: %d", s, r);
        }
    }
    if (n < 0)
    {
        user_panic("error reading %s: %d", s, n);
    }
}

int main(int argc, char \**argv)
{
    int f, i;

    if (argc != 1)
    {
        printf("Usage: history\n");
        return;
    }

    f = open("/.history", O_RDONLY);
    if (f < 0)
    {
        user_panic("can't open .history: %d", f);
    }
    else
    {
        history(f, argv[i]);
        close(f);
    }

    return 0;
}

```

- 实现上下键切换

- 首先创建了三个用于管理 history 文件的函数, hist_init, hist_store, get_hist。
- 然后在 readline 中修改上下键的判断

```
case 'A': // 上键
    MOVEDOWN(1);
    flush(strlen(buf) + 2); //去掉$
    printf("$ ");
    if (offset < row)
    {

    }
    strcpy(buf, history[row - offset]);
    len = strlen(buf);
    printf("%s", buf);
    i = len-1;
    break;
case 'B': // 下键]
    flush(strlen(buf) + 2); //去掉$
    printf("$ ");
    if (offset > 0)
    {
        offset--;
    }
    if (offset != 0)
    {
        strcpy(buf, history[row - offset]);
        len = strlen(buf);
        printf("%s", buf);
    }
    i = len-1;
    break;
```

测试

- history 功能测试

```
ls;
ls;
history;
```

```

$ ls
aaa.txt testarg.b cat.b pingpong.b testbss.b newmotd history.b testpiperace.b testpipe.b motd init.b num.b lorem touch.b mkd
ir.b testbackend.b testfdsharing.b testshell.sh declare.b script ls.b echo.b sh.b tree.b unset.b halt.b testptelibrary.b .hi
story 1
[00005006] destroying 00005006
[00005006] free env 00005006
i am killed ...
[00004805] destroying 00004805
[00004805] free env 00004805
i am killed ...

$ ls
aaa.txt testarg.b cat.b pingpong.b testbss.b newmotd history.b testpiperace.b testpipe.b motd init.b num.b lorem touch.b mkd
ir.b testbackend.b testfdsharing.b testshell.sh declare.b script ls.b echo.b sh.b tree.b unset.b halt.b testptelibrary.b .hi
story 1
[00006006] destroying 00006006
[00006006] free env 00006006
i am killed ...
[00005805] destroying 00005805
[00005805] free env 00005805
i am killed ...

$ history
ls
ls
history
[00007000] destroying 00007000
[00007000] free env 00007000
i am killed ...
[00006805] destroying 00006805
[00006805] free env 00006805
i am killed ...

```

截图(Alt + A)

- 上下键功能测试

```

ls;
ls;
history;
'\033[A'\033[A'Enter' //上方向键+上方向键+回车键
'\033[A'\033[A'\033[A'\033[B'Enter'
//上方向键+上方向键+上方向键+下方向键+回车键

```

```

$ ls
aaa.txt testarg.b cat.b pingpong.b testbss.b newmotd history.b testpiperace.b testpipe.b motd init.b num.b lorem touch.b mkd
ir.b testbackend.b testfdsharing.b testshell.sh declare.b script ls.b echo.b sh.b tree.b unset.b halt.b testptelibrary.b .hi
story
[00000004] destroying 00000004
[00006004] free env 00006004
i am killed ...
[00005803] destroying 00005803
[00005803] free env 00005803
i am killed ...

$ history
ls
ls
history
ls
history
[00007004] destroying 00007004
[00007004] free env 00007004
i am killed ...
[00006803] destroying 00006803
[00006803] free env 00006803
i am killed ...

```

- 返回原函数测试

```

ls;
'\033[A'\033[B'ls'Enter' //上方向键+下方向键+ls+回车键

```

```
$ ls
aaa.txt testarg.b cat.b pingpong.b testbss.b newmotd history.b testpipeace.b testpipe.b motd init.b num.b lorem touch.b mkd
ir.b testbackend.b testfdsharing.b testshell.sh declare.b script ls.b echo.b sh.b tree.b unset.b halt.b testptelibrary.b .hi
story
[00005804] destroying 00005804
[00005804] free env 00005804
i am killed ...
[00005003] destroying 00005003
[00005003] free env 00005003
i am killed ...
```

选做部分 1：实现 shell 环境变量

- 支持 declare [-xr] [NAME [=VALUE]] 命令，其中：
 - -x 表示变量 NAME 为环境变量，否则为局部变量。
 - 环境变量对子 shell 可见，也就是说在 shell 中输入 sh.b 启动一个子 shell 后，可以读取并修改 NAME 的值，即支持环境变量的继承。
 - 局部变量对子 shell 不可见，也就是说在 shell 中输入 sh.b 启动一个子 shell 后，没有该局部变量。
- -r 表示将变量 NAME 设为只读。只读变量不能被 declare 重新赋值或被 > unset 删除。
- 如果变量 NAME 不存在，需要创建该环境变量；如果变量 NAME 存在，将该变> 量赋值为 VALUE。
- 其中 VALUE 为可选参数，缺省时将该变量赋值为空字符串即可。
- 如果没有 [-xr] 及 [NAME [=VALUE]] 部分，即只输入 declare，则输出当> 前 shell 的所有变量，包括局部变量和环境变量。
- 支持 unset NAME 命令，若变量 NAME 不是只读变量，则删除变量 NAME。
- 支持在命令中展开变量的值，如使用 echo.b \$NAME 打印变量 NAME 的值。
- 在 sh.c 中增加 shell_id 这一变量，作为区别不同的 shell。
- 该任务的实现主要是在 kern/sysctl_all.c 文件中添加这一函数。这一函数根据 op 的五种情况，分别进行五个对环境变量的操作。

```
int sys_envar(char *name, char *value, u_int op, u_int isGlobalEnv)
```

```
#define VAR_Create 0
#define VAR_GetOne 1
#define VAR_SetValue 2
#define VAR_Unset 3
#define VAR_GetAll 4
#define VAR_SetRead 5
```

这五种情况分别处理 生成一个环境变量、根据 name 得到一个环境变量的 value、设置一个环境变量的 value、取消一个环境变量的设置、得到所有当前 shell 可观察到的环境变量、设置一个环境变量的

readonly 性质。

- 在 user/include/lib.h 、 user/lib/syscall_lib.c、 include/syscall.h 文件中增加相关的信息。
- 新建一个 declare 命令，根据参数的特性去使用相应的 op。需要注意，我们通常而已把 create 和 setValue 统一起来（在 create 发现没有这个文件时，则 setValue）。

```
void setValue(char *name, char *value, u_int isGlobalEnVar)
{
    int r;
    if ((r = syscall_envar(name, value, VAR_Create, isGlobalEnVar)) == -E_FILE_EXISTS)
    {
        syscall_envar(name, value, VAR_SetValue, isGlobalEnVar);
    }
    else if (r < 0)
    {
        user_panic("can't set Value: %d\n", r);
    }
}
```

- 在 echo 函数中，假如识别到\$，则使用系统调用去获得对应的环境变量。

```
if (argv[i][0] == '$')
{
    char value[1024];
    syscall_envar(&argv[i][1], value, VAR_GetOne, 0);
    printf("%s", value);
}
```

- 创建 unset 函数

```
if (argc != 2)
{
    printf("Usage: unset file_name\n");
    return;
}
syscall_envar(argv[1], "", VAR_Unset, 0);
```

测试函数

- 设置环境变量、局部变量，设置可读性，输出所有变量。

```
declare a =1;declare -x b =2;declare -xr c =3;declare;
```

```

$ declare a =1;declare -x b =2;declare -xr c =3;declare;
[00003805] destroying 00003805
[00003805] free env 00003805
i am killed ...
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
[00004805] destroying 00004805
[00004805] free env 00004805
i am killed ...
[00004004] destroying 00004004
[00004004] free env 00004004
i am killed ...
[00005805] destroying 00005805
[00005805] free env 00005805
i am killed ...
[00005004] destroying 00005004
[00005004] free env 00005004
i am killed ...
Name: a          Value: 1          --[LOCAL VARIABLE]
Name: b          Value: 2          --[ENVIRONMENT VARIABLE]
Name: c          Value: 3          --[ENVIRONMENT VARIABLE] --[READ_ONLY]
[00006805] destroying 00006805
[00006805] free env 00006805
i am killed ...
[00006004] destroying 00006004
[00006004] free env 00006004
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...

```

- 环境变量可见，局部变量不可见。

```

declare a =1;declare -x b =2;declare;sh;
declare;

```

```

$ declare a =1;declare -x b =2;declare;sh;
[00004805] destroying 00004805
[00004805] free env 00004805
i am killed ...
[00004004] destroying 00004004
[00004004] free env 00004004
i am killed ...
[00005805] destroying 00005805
[00005805] free env 00005805
i am killed ...
[00005004] destroying 00005004
[00005004] free env 00005004
i am killed ...
Name: a                Value: 1                --[LOCAL VARIABLE]
Name: b                Value: 2                --[ENVIRONMENT VARIABLE]
[00006805] destroying 00006805
[00006805] free env 00006805
i am killed ...
[00006004] destroying 00006004
[00006004] free env 00006004
i am killed ...

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::                                                                    ::
::                               MOS Shell 2023                        ::
::                                                                    ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
shell_id is 1

$ declare
Name: b                Value: 2                --[ENVIRONMENT VARIABLE]
[00008807] destroying 00008807
[00008807] free env 00008807
i am killed ...
[00008006] destroying 00008006
[00008006] free env 00008006
i am killed ...

```

- 普通变量可以被 declare 和 unset

```
declare a =1;declare b =1;declare;declare a =2;declare;unset a;declare;
```



```

$ declare a =1;declare b =1;declare;declare a =2;declare;unset a;declare;
[00003805] destroying 00003805
[00003805] free env 00003805
i am killed ...
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
[00004805] destroying 00004805
[00004805] free env 00004805
i am killed ...
[00004004] destroying 00004004
[00004004] free env 00004004
i am killed ...
Name: a                      Value: 1                      --[LOCAL VARIABLE]
Name: b                      Value: 1                      --[LOCAL VARIABLE]
[00005805] destroying 00005805
[00005805] free env 00005805
i am killed ...
[00005004] destroying 00005004
[00005004] free env 00005004
i am killed ...
envar a has existed
[00006805] destroying 00006805
[00006805] free env 00006805
i am killed ...
[00006004] destroying 00006004
[00006004] free env 00006004
i am killed ...
Name: a                      Value: 2                      --[LOCAL VARIABLE]
Name: b                      Value: 1                      --[LOCAL VARIABLE]
[00007805] destroying 00007805
[00007805] free env 00007805
i am killed ...
[00007004] destroying 00007004
[00007004] free env 00007004
i am killed ...
[00008805] destroying 00008805
[00008805] free env 00008805
i am killed ...
[00008004] destroying 00008004
[00008004] free env 00008004
i am killed ...
Name: b                      Value: 1                      --[LOCAL VARIABLE]
[00009805] destroying 00009805
[00009805] free env 00009805
i am killed ...
[00009004] destroying 00009004
[00009004] free env 00009004
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...

```

- -r 变量不可以被 declare 和 unset

```
declare -r a =1;declare;declare -r a =2;declare;unset -r a;declare;
```

```
$ declare -r a =1;declare;declare -r a =2;declare;unset -r a;declare;
```

```
[00003805] destroying 00003805
```

```
[00003805] free env 00003805
```

```
i am killed ...
```

```
[00003004] destroying 00003004
```

```
[00003004] free env 00003004
```

```
i am killed ...
```

```
Name: a Value: 1
```

```
--[LOCAL VARIABLE] --[READ ONLY]
```

```
[00004805] destroying 00004805
```

```
[00004805] free env 00004805
```

```
i am killed ...
```

```
[00004004] destroying 00004004
```

```
[00004004] free env 00004004
```

```
i am killed ...
```

```
envvar a has existed
```

```
This variable can't be changed because it's read only!
```

```
[00005805] destroying 00005805
```

```
[00005805] free env 00005805
```

```
i am killed ...
```

```
[00005004] destroying 00005004
```

```
[00005004] free env 00005004
```

```
i am killed ...
```

```
Name: a Value: 1
```

```
--[LOCAL VARIABLE] --[READ ONLY]
```

```
[00006805] destroying 00006805
```

```
[00006805] free env 00006805
```

```
i am killed ...
```

```
[00006004] destroying 00006004
```

```
[00006004] free env 00006004
```

```
i am killed ...
```

```
Usage: unset file_name
```

```
[00007805] destroying 00007805
```

```
[00007805] free env 00007805
```

```
i am killed ...
```

```
[00007004] destroying 00007004
```

```
[00007004] free env 00007004
```

```
i am killed ...
```

```
Name: a Value: 1
```

```
--[LOCAL VARIABLE] --[READ ONLY]
```

```
[00008805] destroying 00008805
```

```
[00008805] free env 00008805
```

```
i am killed ...
```

```
[00008004] destroying 00008004
```

```
[00008004] free env 00008004
```

```
i am killed ...
```

```
[00002803] destroying 00002803
```

```
[00002803] free env 00002803
```

```
i am killed ...
```

- 缺省 value

```
declare a;declare;
```

```

$ declare a;declare;
[00003805] destroying 00003805
[00003805] free env 00003805
i am killed ...
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
Name: a Value: --[LOCAL VARIABLE]
[00004805] destroying 00004805
[00004805] free env 00004805
i am killed ...
[00004004] destroying 00004004
[00004004] free env 00004004
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...

```

- echo 输出环境变量

```
declare a =1;echo $a;
```

```

$ declare a =1;echo $a;
[00003805] destroying 00003805
[00003805] free env 00003805
i am killed ...
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
1
[00004805] destroying 00004805
[00004805] free env 00004805
i am killed ...
[00004004] destroying 00004004
[00004004] free env 00004004
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...

```

至此，我们完成了 lab6 的挑战性任务

遇到的问题

本次作业感觉遇到比较容易忽视的问题有四个

- 一行多命令和重定向的冲突，这里需要利用文件系统知识进行修改。
- 后台运行的忙等现象，这里需要对忙等进行处理。
- tree 函数对同一个文件夹进行输出假如不对上一次打开的文件进行关闭会出现 bug。
- 上下键和左右键的适配问题，需要通过修改 i 来实现。