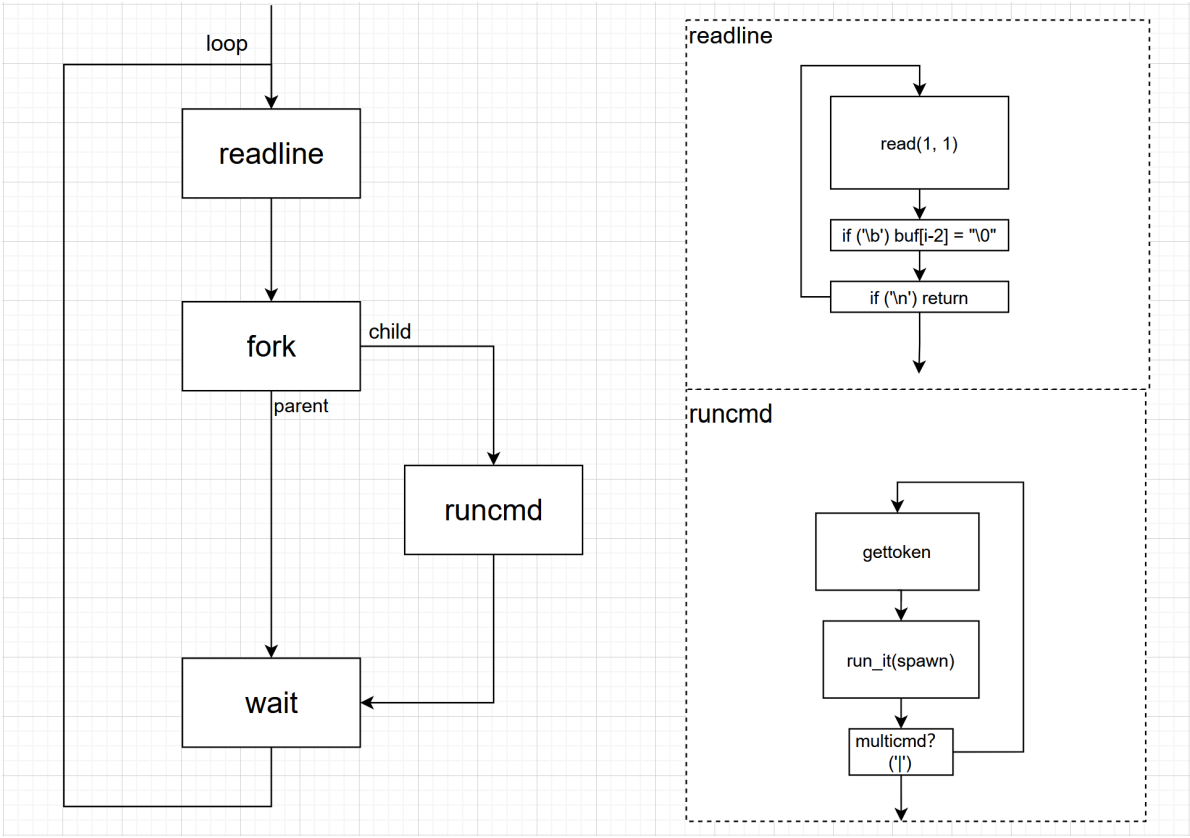
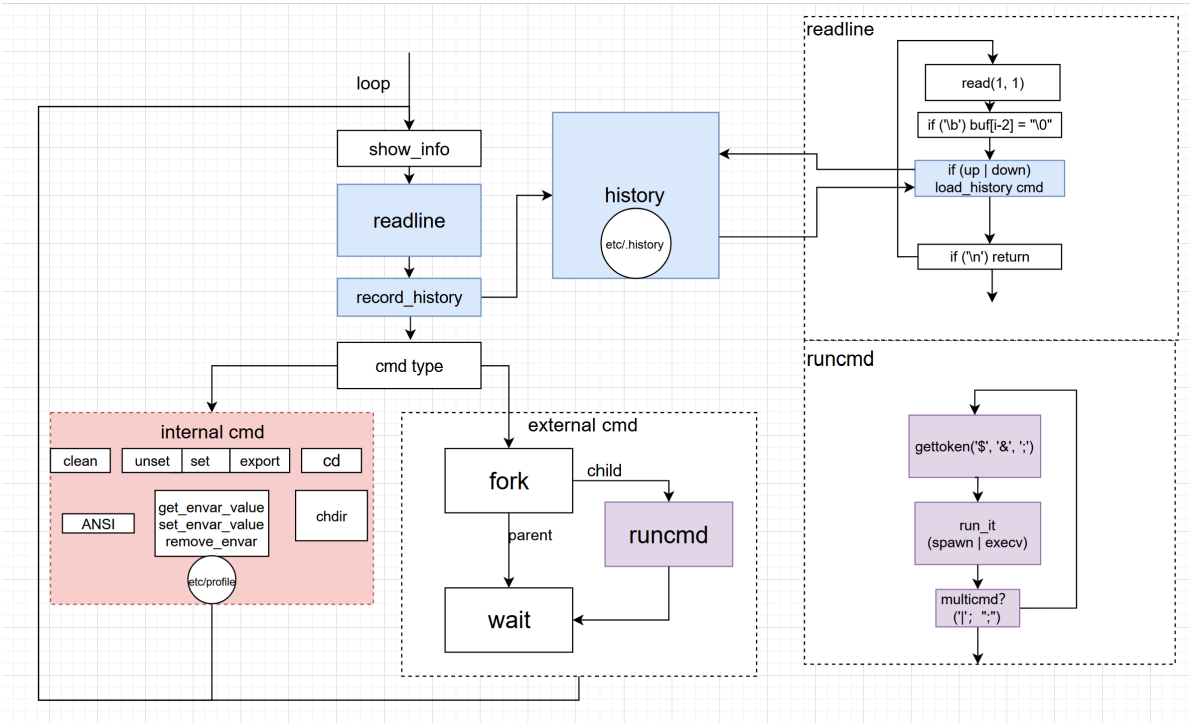
Lab6-Challenge Report

0. 总述



如图是lab6中大部分由官方实现的shell，在此基础上，笔者增加部分内容后总体如下：



红色部分是内部命令的支持，在本次任务中主要支持了`clean`、`unset set export` `cd`这五个命令，均是以直接调用相关函数实现的。由于内部命令的执行不会新开一个进程，而是直接在sh进程中执行。因此它们应当区别于外部命令，在fork之前被执行，返回到等待指令的初始状态，如图所示即和externalcommand平级。

蓝色部分为对历史命令的记录和上下键导出的支持，在每读取一行指令后，会将指令以字符串的形式读入`etc/.history`中。在从键盘读取指令时，会对`up` `down`键进行特判，将历史命令写入buffer，并在屏幕上回显，从而实现效果。

紫色部分主要是对特殊字符的处理，将这些特殊字符转化为正常指令的形式。核心为对`gettoken`函数的改写。

上述三个核心的实现在一、二、三部分中进行详述，而具体的实现需要对现有的操作系统进行修改，将在第四部分中进行详述，除以上三个核心的实现之外。还有颜色控制相关的实现及对外部命令进行扩展，将分别在第五和第六部分进行详述。

一. 内部命令

以内部命令的形式简单实现了`cd`、`export`、`set`、`clean`、`unset`.

1. cd

`cd`将直接改变shell中的全局变量`pwd`，在确定目标路径可以打开且为目录文件后，会将目录文件名附在原`pwd`之后，而pwd作为指令执行时的环境信息被加入参数。

2. export、set、unset

在sh.c中新增`envar`结构体：

```
struct envar {
    char name[1024];
    char value[1024];
    u_int type;
};
```

新增全局的envar结构体数组`envars`，并增加如下函数对其进行管理：

```
init_envar()
get_envar_value(char* name)
set_envar_value(char* name, char* value)
remove_envar(char* name)
```

这个结构体数组与`etc/profile`中的内容实时保持同步。sh刚刚启动时的几个初始的环境变量也是从该磁盘文件中读出的。

3. clean

clean主要ANSI控制字符实现：

```
 } else if (strcmp(argv[0], "clean") == 0) {
    PRINT_CLEAN //清屏
    BACK_TO_TOP //控制光标回到第一行第一列
 }

//在color.h中定义了如下宏函数
/* clean cmd */
#define PRINT_CLEAN  writef("\033[2J"); //clean_screen
#define BACK_TO_TOP writef("\033[1;1H"); //back to (1, 1)
```

二. 历史命令的记录与导出

历史命令记录于`etc/.history`文件中。在shell启动并执行第一行指令后，会在`etc`目录下创建该文件，并在每一次读取完指令后向其中写入历史命令。当然实现在文件末尾添加内容需要文件系统对于`O_APPEND`的支持。需要对文件系统进行修改。

```
void record_history(char* buf, int n) {
    if (first_run == 0) {
        user_create("etc/.history", 0);
        first_run = 1;
    }
    // record current cmd into .history
    int fd = open("etc/.history", O_WRONLY | O_APPEND);
    int k = write(fd, buf, n);
    close(fd);
    fd= open("etc/.history", O_WRONLY | O_APPEND);
    k = write(fd, "\n", 1);
    close(fd);
    history_size += 1;
}
```

实现通过上下键切换历史指令需要改写`readcmd`函数，使其在接受到上下键时，清空当前记录的命令缓冲(buffer)，导入特定的历史命令，并在console上回显。

```
if (buf[i] == 0x41) {
    flush(buf);
    historyget(index, history_cmd);
    strcpy(buf, history_cmd);
    fwritef(1, " %s", buf);
    i = strlen(buf)-1;
    index = ((index-1)<0? (index-1+history_size) : (index-1))%history_size;
}
else if (buf[i] == 0x42) {
    flush(buf);
    historyget(index-1, history_cmd);
    strcpy(buf, history_cmd);
    fwritef(1, " %s", buf);
    index = (index+1)%history_size;
    i = strlen(buf)-1;
```

```
}
```

这里`index`的计算通过mod实现了一个循环，会在`up`和`down`键入后增减，在其他键键入后恢复初值`history_size - 1`，即最后一条历史命令。

三. 特殊字符的处理与指令的环境配置

1. 修改`_gettoken`函数

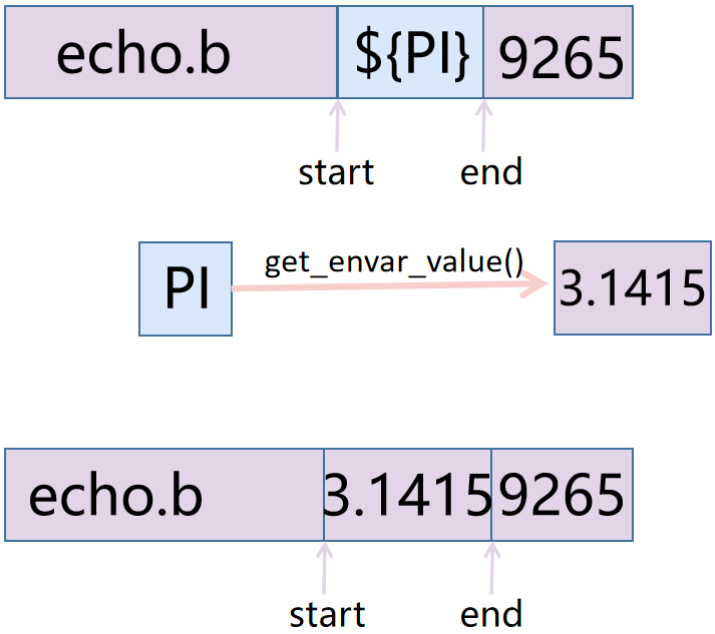
首先向其中增加一个参数`index`，通过修改index中的内容增加一个返回对应字符位置下标的"返回值"。

进一步地，修改其内部对于特殊字符的判断，对于`'`字符，读到此字符，则将sh的一个全局变量`mask`置1，在mask为1时，**对于除`'`之外的其他字符不做特殊处理**，当做普通字符继续向后读取，在读取到下一个`'`时，将mask置0。从而表示强引用的结束。

```
if (index == '\'') {
   if (mask == 1) mask = 0;
   else mask = 1;
}
```

这个`_gettoken`中的循环是修改的核心，除了bool逻辑的判断中加入mask之外，还实现了对于`$`的支持——将`$`后空格前，或者`${ }`之中的内容根据环境变量直接进行替换。

如图展现的是一个指令中对`$`后的变量进行替换的过程



在实现时，根据特殊字符设置start和end指针的位置，将name`PI`拷贝至新建的字符串中，通过`get_envar_value(buffer)`获得环境变量`PI`的值，在根据start和end指针的位置进行字符串的拼接。注意这里当`mask`为1时，不进入`if`分支，即将`$`当做普通字符看待。

```
while(*s && ((!strchr(WHITESPACE SYMBOLS, *s)) || mask) && *s != '\'') {
        if (*s == '$' && !mask) {  // if true, start use envar{name} to replace
name in CmdString
            char* start;
            char end[1024];
            cleanbuf(end);
```

```
            start = s;
            s++;
            char buffer[1024];
            cleanbuf(buffer);
            char ch = ' ';
            if (*s == '{') {
                ch = '}';
                s++;
            }
            int k = 0;
            while (*(s+k) != ch && *(s+k) != '\n' && *(s+k)!= '\0') {
                buffer[k] = *(s+k);
                k++;
            }
            buffer[k] = '\0';
            if (*(s+k) == '}') strcpy(end, s+k+1);
            else strcpy(end, s+k);
            char* value = get_envar_value(buffer); //get value = envar[name]
            for (k=0; value[k]!=0; k++) {
                *(start+k) = value[k];
            }
            *(start+k) = 0;
            int size = k;
            for (k=0; end[k]!=0; k++) {
                *(start+k+size) = end[k];
            }
            *(start+k+size) = 0;
        } else {
            s++;
        }
    }
```

2. 特殊字符产生的特定标记变量与指令环境的配置

对于其他特殊字符，则在进行处理后设置一些标记变量。在`runit`中根据相应的值配置指令的执行：

```
case ';':
   multicmd = 1;
   goto runit;
   break;

case '&':
   backorder = 1;
   break;

runit:
spawn or execv...
if (!backorder) wait(r);
if (multicmd) {
    multicmd = 0;
    goto again;
}
exit();
```

如代码所示，当multicmd被标记为1时，将在执行完当前指令后`goto again`执行下一条指令。

当backorder被标记为1时，将不等待spawn或execv的子进程执行完毕，直接`exit`返回。

除此之外，指令需要被添加一些'环境'。对于外部指令，在本次challeng中为了更好地模拟实际情况，将各个`.b`APP放入了`bin`文件夹中，此时，`/bin`就类似于linux环境变量中的`$PATH`，默认这些外部指令需要到其中寻找，因而，对于每一条外部指令，都需要在实际调用时添加其所在的路径，
即`/bin/xxx.b`

```
strcpy(outcmd, "/bin/");
strcpy((outcmd+strlen(outcmd)), argv[0]);
```

同时，由于设置了当前目录`pwd`的全局变量，对于argv[1-x]，需要在其前添加pwd，当然，是否添加pwd还需要判断该参数是否为文件，在本次任务中笔者只是粗糙地对`-[mode]`参数和`echo.b`后的参数进行特判，在复杂情况下不够全面，但能够体现核心的思想。

```
} else if (strcmp(argv[0], "echo.b")!= 0){
    int i = 1;
    for (i=1; (argv[i] != 0 && argv[i][0] != '-'); i++) {
        for (ind = 0; ind<512; ind++) pargv[ind] = 0;
        strcpy(pargv, pwd);
        strcpy((pargv + strlen(pargv)), "/");
        strcpy((pargv + strlen(pargv)), argv[i]);
        strcpy(argv[i], pargv);
    }
}
```

四. 基础配置

上面这些sh中的功能的实现，需要对操作系统进行一些修改：

- 对于.history的记录，需要能以`O_APPEND`的形式打开文件
- 对于打印中被打断的情况，需要新建系统调用`sys_print_string`用以实现防打断的writef
- 为了增加文件目录的结构性，需要添加write_directory在磁盘中烧录目录结构
- 为了支持`mkdir` `touch`命令，需要增加create_file函数并在fserve进程中提供相应的支持
- 为了减少进程的开销(fork后自进程再次spawn一个新进程是没有必要的)，实现了`execv`系统调用使得进程自己加载新的二进制程序下面逐一对他们进行介绍。

下面逐一对他们进行介绍。

1. write_directory

完成了lab5中残留的`write_directroy`函数。

```
struct File* write_directory(struct File *dirf, char *name) {
    struct File* target = create_file(dirf);
    const char* fname = strrchr(name, '/');
    if(fname) ++fname;
    else fname = name;
    strcpy(target->f_name, fname);
    target->f_size = 0;
    target->f_type = FTYPE_DIR;
    return target;
}
```

此后，对于需要烧录的目录结构和需要烧录进特定目录的文件，如下进行烧录即可

```
bin = write_directory(&super.s_root, argv[i]);
write_file(bin, argv[i]);
```

完成烧录后的文件目录情况如下(通过`tree.b`指令展示):

```
SZY@superShell:/usr$ tree.b -a

/
├──bin
│   ├──num.b
│   ├──echo.b
│   ├──ls.b
│   ├──cat.b
│   ├──history.b
│   ├──touch.b
│   ├──tree.b
│   ├──mkdir.b
│   └──testbackod.b
├──usr
├──etc
│   ├──profile
│   └──.history
├──motd
├──newmotd
├──testarg.b
├──init.b
├──sh.b
└──testptelibrary.b
```

2. O_APPEND的支持

实现较为简单，对user/file/open函数进行简单修改即可。

```
if ((O_APPEND & mode) != 0) {
    char buffer[1];
    buffer[0] = 0;
    while (file_read(fd, buffer, 1, fd->fd_offset)) {
        if (buffer[0] == 0) {
            break;
        }
        fd->fd_offset++;
    }
    buffer[0] = 0;
}
```

3. 防中断的打印

内核态下屏蔽中断，首先实现一个内核态中打印字符串的系统调用：

```
void sys_print_string(int sysno, char* str)
{
#if SAFEPRINT_ON == 1
    str = (char*)(PRINTADDR);
    str[MAXPRINTLEN - 1] = '\0';
#endif /*safeprint_on == 1*/
    printf("%s", str);
}
```

此后，对user的writef函数进行修改，改变user_lp_Print中的参数，使得被转化后需要打印的字符串被存在一个字符数组中，将该字符数组的内容复制到内核地址空间，并将首地址作为参数进行系统调用。

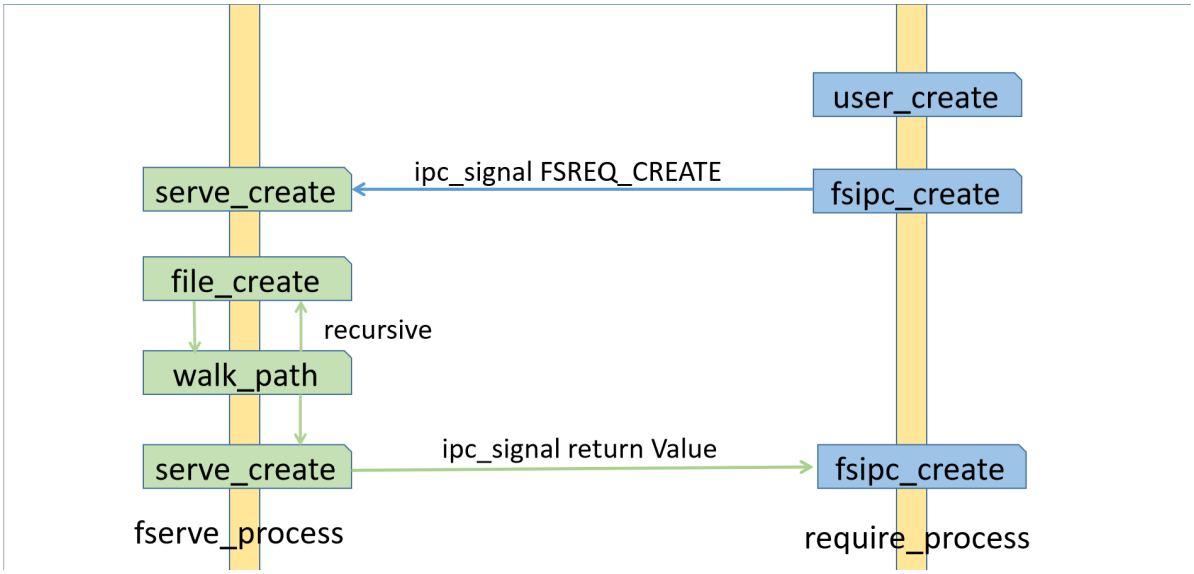
```
void writef(char *fmt, ...)
{
    user_bzero((void*)writefx_buf, 1024);

    va_list ap;
    va_start(ap, fmt);
    user_lp_Print(user_out2str, writefx_buf, fmt, ap);
    va_end(ap);
    writefx_buf[1023] = '\0';

#if SAFEPRINT_ON == 1
    user_bcopy(writefx_buf, PRINTADDR, MAXPRINTLEN);
#endif /* safeprint_on = 1 */
    syscall_print_string(writefx_buf);
}
```

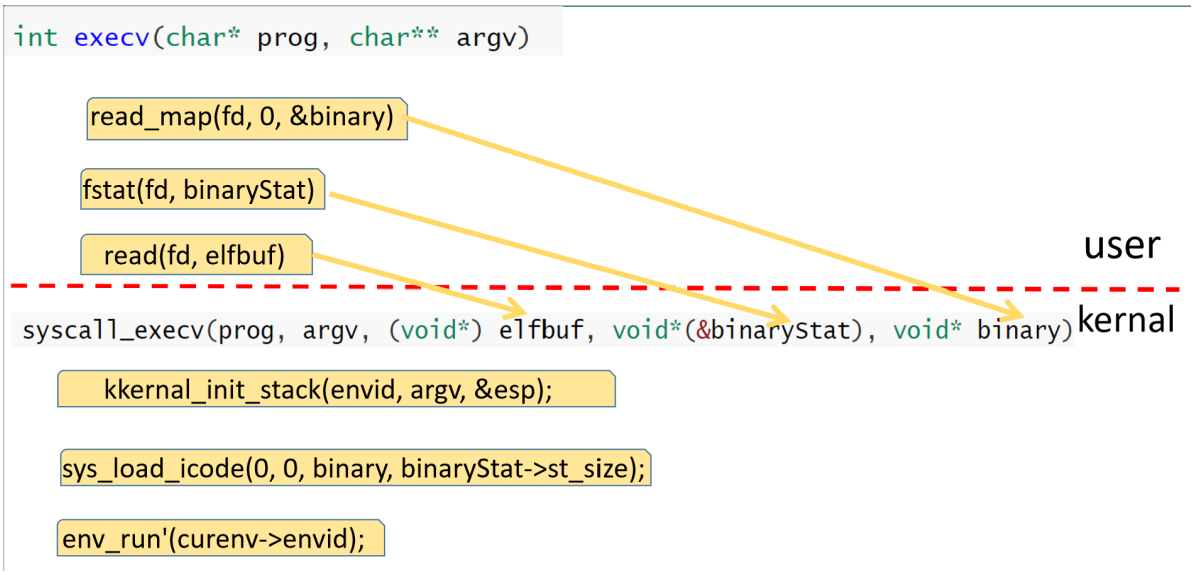
4. user_create函数于fs_serve的支持

在我们的操作系统中，进程对文件操作进行请求，由fs_serve进程统一对其进行管理。其实user_create函数已经在lab5课上测试中进行了较为完备的实现(在extra中还实现了递归式地创建文件)，这里的实现与之相同：



核心的修改在`walk_path`和`file_create`函数中。

5. execv



笔者是基本仿照spawn函数实现的，由于execv需要替换当前进程的执行内容，故**对其所加载的镜像进行的重载的操作都需要在内核态下的内核函数中完成**，因此，内核态下需要增加一个init_stack函数用于重载用户栈。execv在用户态下需要加载好需要传递各个参数，并放入内核空间。在内核态下完成重载镜像并调用特殊的`env_run`函数重新执行。

五. ANSI颜色配置

在本次设计中实现了不同粗细、颜色的字符输出， 在`user/color.h`头文件中定义了不同打印方式的宏:

```
/*set_front_color*/
#define PRINT_FONT_RED  printf("\033[31m"); //red
#define PRINT_FONT_GRE  printf("\033[32m"); //green
#define PRINT_FONT_YEL  printf("\033[33m"); //yellow

/*set_background_color*/
#define PRINT_BACK_RED  printf("\033[41m"); //red
#define PRINT_BACK_GRE  printf("\033[42m"); //green
#define PRINT_BACK_YEL  printf("\033[43m"); //yellow
...
```

为了在主程序中的代码逻辑更为清晰，为此定义了一些宏函数，用于灵活地显示颜色:

```
#define PRINT_INIT do{            \
        PRINT_ATTR_BOL              \
        PRINT_FONT_CYA              \

 writef("\n:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::\n
"); \
        writef("::
::\n"); \
        writef("::                    Super Shell  V0.0.0_2
::\n"); \
```

```
        writef("::
::\n"); \

 writef("::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::");
\
        PRINT_ATTR_REC                \
}while (0)

#define PRINT_COLOR(info, color, type) do{    \
    if((color)==1) PRINT_FONT_RED                \
    else if((color)==2) PRINT_FONT_YEL        \
    else if((color)==3) PRINT_FONT_BLU        \
    else if((color)==4) PRINT_FONT_GRE         \
    if ((type) == 0) printf("%s", (info));    \
    else if((type) == 1) printf("%s\n", (info)); \
    else printf("%s ", (info));                \
    PRINT_ATTR_REC                            \
}while (0)
```

使用这些宏函数时就能方便地实现不同的打印，如为了模拟ubantu中的shell的prompt，在打印prompt `$` 之前

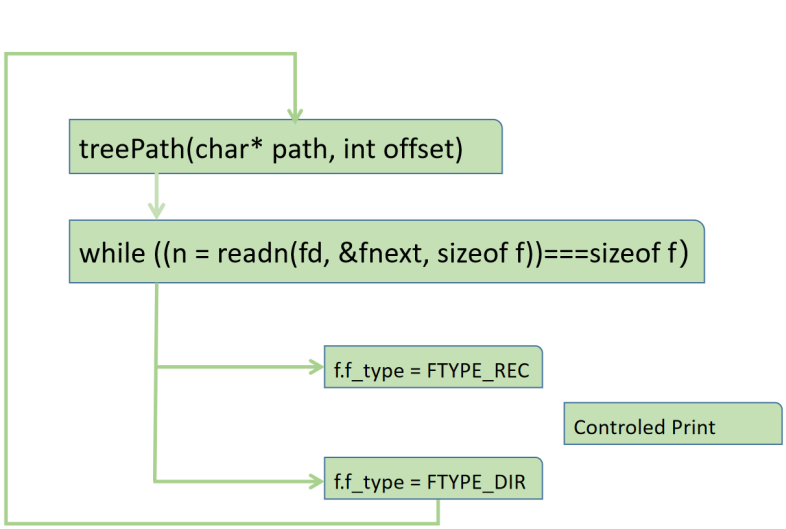
```
char* info = get_envar_value("USER");
PRINT_USERNAME(info);
if (strcmp(pwd, "") == 0) PRINT_CURRENT_DIR("/");
else PRINT_CURRENT_DIR(pwd);
```

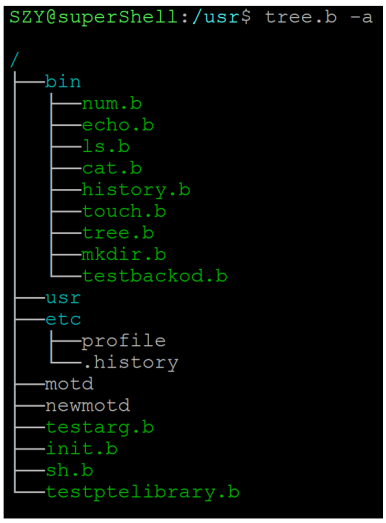
在返回错误信息时：

```
PRINT_FONT_RED
writef("command '%s' cannot be implemented!\n\033[1A", argv[0]);
PRINT_ATTR_REC
```

六. 外部命令的扩展

应要求实现了`mkdir`、`touch`、`history`、`tree`外部命令，前三个指令实际上仅需打开文件进行读取，或者调用`user_create`函数创建文件。而仅有`tree`命令的实现较为复杂，故以之为例进行简述





如图所示为其调用的核心函数`treePath`，其函数主体为一个循环，对打开的目录文件进行读取，并将结果传入一个Srtuct File的指针指向的结构体中，从而遍历该目录下的文件。对于此目录下的文件，如果为非目录文件则直接打印，对于目录文件则进行递归调用。从而实现文件的遍历。此外，为了更好地模拟linux的tree函数展现其文件系统的树形结构，还需要维护并传递参数`offset`标记其所在的递归层级，并根据实际情况进行打印。实现效果如右图所示。

总结

感觉lab6的challenge在所有挑战性任务中实现难度是较大的。但笔者依然选择这个挑战性任务，原因是：lab6的challeng能通过敲自己熟悉的linux命令就能展示结果，通过一点一滴的反馈逐步实现较为复杂的功能。整个challenge完成下来花费了大量的时间精力，同时分别找到lab5和lab2以及lab3的祖传bugs。但是收获也是与付出成正比的。笔者对于实验的操作系统MOS的整体性理解可以说更上一层楼(也许本来在1楼，现在上到2楼了)。通过完善shell，我们的操作系统不再那么冰冷，而是可以进行简单的和甚至稍稍复杂一点交互。结合CO来看，某种程度上说，我们已经**在理论上**可以从逻辑门开始，直至完成一个具有交互能力的现代计算机了。