

lab6-challenge-report

题目

Lab6 挑战性任务分为必做与选做两个部分。需要完成必做部分的所有功能，以及两项选做任务的其中一项。

在 Lab6 基础部分的实验中我们实现了一个简单的 shell。在本次挑战性任务，我们将通过实现一些难度层次递进的小组件或附加功能，来丰富我们的 shell，从而加深对整个 MOS 的理解，让我们的 MOS 更加完整。

你可以参考 [Bash Reference Manual](#) 官方文档来理解并完成以下任务。对于涉及的 Linux 命令和库函数，你可以使用 `man` 命令，或通过 [Linux man pages](#) 查询其手册，为实现提供参考。

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。

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。

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`。

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。

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。

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。

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。

选做部分 1：实现 shell 环境变量

1. 支持 `declare [-xr] [NAME [=VALUE]]` 命令，其中：
 2. `-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。
 3. `-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。
 4. 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`。
 5. 其中 `VALUE` 为可选参数，缺省时将该变量赋值为空字符串即可。
 6. 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。
 7. 支持 `unset NAME` 命令，若变量 `NAME` 不是只读变量，则删除变量 `NAME`。
 8. 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。

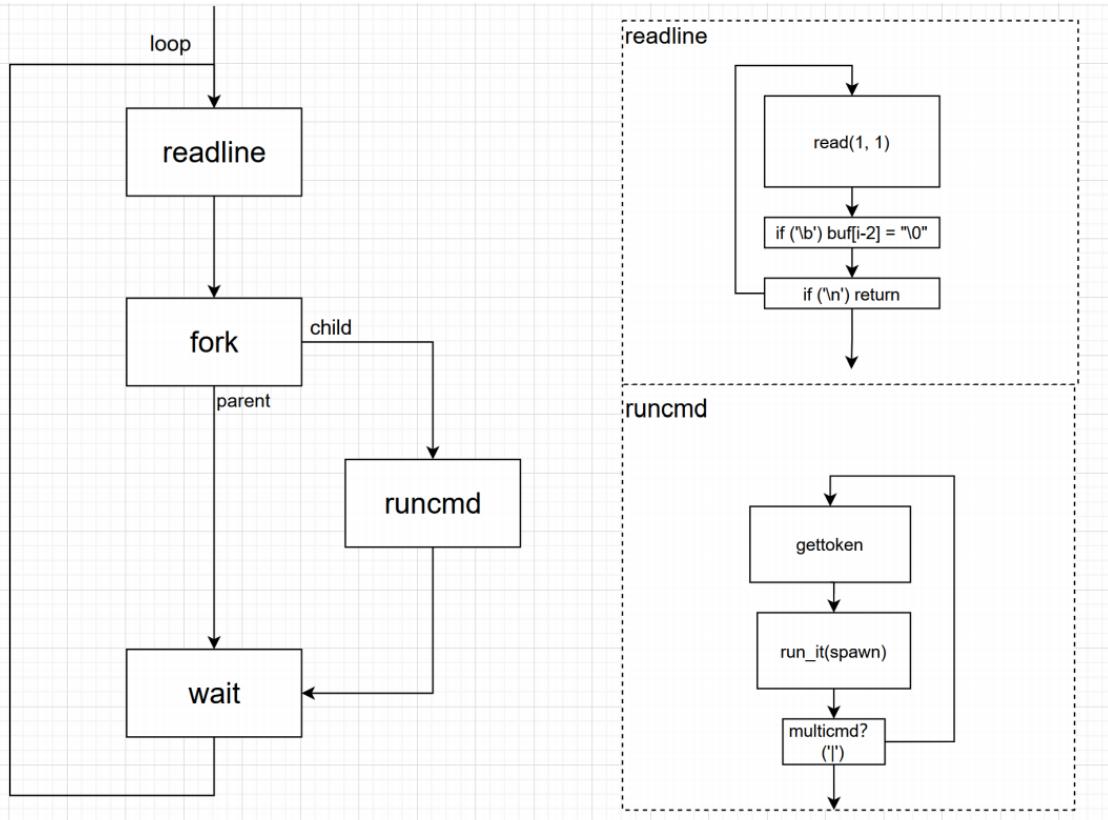
选做部分 2：支持相对路径

MOS 中现有的文件系统操作并不支持相对路径，对于一切路径都从根目录开始查找，因此在 shell 命令中也需要用绝对路径指代文件，这为命令的描述带来了不便。你需要为每个进程维护**工作目录**这一状态，并为 `open()` 等用户库函数增加对参数中相对路径的支持，将不以 `/` 开头的路径视为相对路径，从当前进程的工作目录开始查找。同时，你需要添加 `chdir()` 和 `getcwd()` 等库函数，以支持切换当前进程的工作目录，并使进程的工作目录在 `fork()` 或 `spawn()` 时被子进程继承，从而实现以下功能：

1. 支持内部命令 `cd <path>`，切换工作目录到 `<path>`，其中 `<path>` 可以是绝对路径或相对路径；
2. 支持 `pwd` 命令，输出当前工作目录；
3. 在切换工作目录后，测试 `cat.b`、`ls.b`、`touch.b` 等接受文件参数的命令，确保其参数中的相对路径能够正常工作。

实现思路

如图为 Tab6 架构的简图，也是我们修改的主战场（剩下的用户态操作基本都要自己添加文件实现了）



我在完成了上述必做任务后，选择了实现相对路径的选做任务，除此之外，为了使shell使用更加流畅，我选择了加入clear和彩色输出等来进一步使其更加便于使用，下面为我各部分的实现思路。

添加文件、命令

添加新的命令时，需要修改 `/user/include.mk`，在 `USERAPPS` 下添加对应文件的 `.b`。

想要将自己的 C 语言代码添加到编译过程（用户库）中，需要在 `/user/include.mk` 的 `USERLIB` 后添加对应文件的 `.o`。

一行多命令

shell的原理是main 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。

因为在 `/user/sh.c` 中的保留 `SYMBOLS` 里已经预留有 `;` 字符，`gettken` 函数本身就能够解析下一个 `;` 字符，所以我们只需要将 `;` 作为一个特殊的token即可，我们无需改变词法部分，只需改变语法解释部分。

```

1 // user/sh.c parsecmd
2 int forktemp = 0;
3 case ';':
4     forktemp = fork();
5     if (forktemp) {
6         wait(forktemp);
7         return parsecmd(argv, rightpipe);
8     } else {
9         return argc;
10    }
11    break;

```

当 `parsecmd` 函数解析到 `;` 时，对 `shell` 进程进行 `fork`，`;` 左侧指令作为子进程直接返回执行，右侧指令作为父进程则是先 `wait` 保证左侧指令执行完毕后，继续向右进行解析。

实现后台任务

前面我们已经提到了 shell 原理是将一行命令传递到子进程，父进程 `wait` 到子进程结束。而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` 操作。

所以同理，`gettken` 函数本身就能够解析下一个 `&` 字符。而所谓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` 即可。

```

1 // user/sh.c parsecmd
2
3 case '&':
4     if ((r = fork()) == 0) {
5         return argc;
6     } else {
7         return parsecmd(argv, rightpipe);
8     }
9     break;

```

当 `parsecmd` 函数解析到 `&` 时，对 `shell` 进程进行 `fork`，`&` 左侧指令作为子进程直接返回执行，右侧指令则是作为父进程继续向右进行解析。

实现引号支持

与一行多命令相仿。查阅 `user/sh.c`，有 `int _gettken()` 函数，作用是从字符串中读取下一个 token。所以为了将引号内的内容视为单独的字符串，作为一个参数传递给要执行的命令，我们需要在读到第一个 `"` 时，将 `*p1` 指向该 `"` 的后一位，将 `s`（字符串指针）不断后移，直到读到与前一个引号相应匹配的 `"`，作为需要读入的参数。`*s++ = 0` 是为了让字符串截止，同时将字符串指针后移一位，使得后续解析能够进行。

```

1 // user/sh.c _gettken
2
3 /* TODO:lab6-challenge */
4 if (*s == '"') {
5     *s = 0;
6     *p1 = ++s;
7     while (s != 0 && *s != '"') {
8         s++;
9     }
10    if (s == 0) {
11        debugf("\\"don't match!!!\\n");

```

```
12         return 0;
13     }
14     *s = 0;
15     *p2 = s;
16     return 'w';
17 }
18 //
```

实现键入命令时任意位置的修改

实现左右键移动光标和指令任意位置的删改。以在当前光标位置进行字符的增加与删除。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。

这部分进行的工作比较多。我重写了一遍读取行的部分。因为我认为在原有代码基础上新增功能比较复杂且容易出 bug，还好原本 `readline()` 的内容就不算多，因此重构得还算顺利。

这里需要维护三个序列：

- `buf[]` 数组：是我们需要的指令行返回值，也是我们得到的真实结果，其中不能有不可见字符；
- 显示的输入栏：显示出来的结果，我们需要维护这个现实的结果是符合要求的；
- 真正的标准输入：真正的标准输入，其中包括用户维持显示结果的大量不可见字符和空格等。

重构后的逻辑是：保存当前光标的位置，每次键入时都会保存光标之后的字符串（若有的话），依次输出光标前的字符串、输入字符、光标后的字符串。在实现过程中要注意维护光标的位置。

具体代码实现如下（这个代码由于后续history使用到了上下键，所以两部分代码耦合在了一起）

```
1 // user/sh.c
2 int offset;                                     // 0:empty line, -1:last cmd,
3 -2:...
4 int solveDirCmd(char* buf, int type) { // type: 0 means up, 1 means down
5     if (newDirCmd == 1) {
6         offset = 0;
7     }
8     if (type == 0) {
9         offset--;
10    } else if (offset < 0) {
11        offset++;
12    }
13    int x = 0;
14    if (offset == 0) {
15        while (buf[x] != '\0') {
16            buf[x] = '\0';
17            x++;
18        }
19        return -1;
20    }
21    int fdnum = open(".history", O_RDONLY);
22    if (fdnum < 0) {
23        debugf("open .history failed in solveDir!\n");
24        return 0;
25    }
26    struct Fd* fd = num2fd(fdnum);
27    char* c;
28    char* begin = fd2data(fd);
29    char* end = begin + ((struct FileFd*)fd)->f_file.f_size;
30    c = end - 1;
```

```

31     while (((*c) == '\n' || (*c) == 0) && (c > begin)) {
32         c--;
33     }
34
35     if (c == begin) { // no history cmd
36         buf[0] = '\0';
37         return 0;
38     }
39
40     c++; // last \n or \0
41     int i;
42     for (i = 0; i > offset; i--) {
43         while ((*c) != '\n' && (*c) != '\0') {
44             c--;
45             if (c <= begin) {
46                 break;
47             }
48         }
49         c--;
50         if (c <= begin) {
51             break;
52         }
53     }
54     offset = i; // avoid offset too bigger than real cmd num
55     if (c > begin) {
56         while (c > begin && (*c) != '\n') {
57             c--;
58         }
59         if ((*c) == '\n') {
60             c++;
61         }
62     } else {
63         c = begin;
64     }
65     int now = 0;
66     while (buf[now] != '\0') {
67         buf[now] = '\0';
68         now++;
69     }
70     now = 0;
71     while ((*c) != '\n' && (*c) != '\0' && (*c) < end) {
72         buf[now] = *c;
73         now++;
74         c++;
75     }
76     return now;
77 }
78 /**
79 #define MOVELEFT(y) printf("\033[%dB", (y))
80 #define MOVERIGHT(y) printf("\033[%dC", (y))
81 void readline(char* buf, u_int n) {
82     /* TODO:lab6-challenge */
83     int r;
84     int off = 0;
85     int len = 0;
86     char op;
87     while (off < n) {
88         if ((r = read(0, &op, 1)) != 1) {

```

```

89         if (r < 0) {
90             debugf("read error: %d\n", r);
91         }
92         exit();
93     }
94     if (op == '\b' || op == 0x7f) {
95         /* TODO:lab6-challenge */
96         if (off > 0) {
97             if (off == len) {
98                 buf[--off] = 0;
99                 printf("\033[D \033[D");
100            } else {
101                for (int j = off - 1; j < len - 1; j++) {
102                    buf[j] = buf[j + 1];
103                }
104                buf[len - 1] = 0;
105                MOVELEFT(off--);
106                printf("%s ", buf);
107                MOVELEFT(len - off);
108            }
109            len -= 1;
110        }
111        // 
112    } else if (op == '\r' || op == '\n') {
113        buf[len] = 0;
114        return;
115    }
116    /* TODO:lab6-challenge */
117    else if (op == 27) {
118        char tmp;
119        read(0, &tmp, 1);
120        char tmp2;
121        read(0, &tmp2, 1);
122        if (tmp == 91 && tmp2 == 65) {
123            debugf("\x1b[B"); // down to cmd line
124            int j;
125            for (j = 0; j < off; j++) {
126                debugf("\x1b[D"); // left to line head
127            }
128            debugf("\x1b[K"); // clean line
129            off = solveDirCmd(buf, 0);
130            len = strlen(buf);
131            debugf("%s", buf);
132        } else if (tmp == 91 && tmp2 == 66) {
133            int j;
134            for (j = 0; j < off; j++) {
135                debugf("\x1b[D");
136            }
137            debugf("\x1b[K");
138            off = solveDirCmd(buf, 1);
139            len = strlen(buf);
140            debugf("%s", buf);
141        }
142        /* TODO:lab6-challenge */
143        else if (tmp == 91 && tmp2 == 67) {
144            if (off < len) {
145                off++;
146            } else {

```

```

147         MOVELEFT(1);
148     }
149 } else if (tmp == 91 && tmp2 == 68) {
150     if (off > 0) {
151         off--;
152     } else {
153         MOVERIGHT(1);
154     }
155 }
156 /**
157 newDirCmd = 0;
158 } else {
159     newDirCmd = 1;
160     if (off == len) {
161         buf[off++] = op;
162     } else { // i < len
163         for (int j = len; j > off; j--) {
164             buf[j] = buf[j - 1];
165         }
166         buf[off] = op;
167         buf[len + 1] = 0;
168         MOVELEFT(++off);
169         printf("%s", buf);
170         MOVELEFT(len - off + 1);
171     }
172     len += 1;
173 }
174 }
175 debugf("line too long\n");
176 while ((r = read(0, buf, 1)) == 1 && buf[0] != '\r' && buf[0] != '\n')
{
177     ;
178 }
179 buf[0] = 0;
180 }

```

这里需要着重强调一下Linux对于上下左右键的编码问题

上下左右键在linux中会被编码为

上: 27 '[' 'A'

下: 27 '[' 'B'

右: 27 '[' 'C'

左: 27 '[' 'D'

(更多编码可以参考博客[Terminal Control Escape Sequences](#))

所以我们需要在读到27 [Esc] 后连续读取两个字符以判断指令的类型,从而分别调用属于左右键和上下键的不同功能.

实现程序名称中 .b 的省略

spawn函数与 fork 函数类似, 其最终效果都是产生一个子进程, 不过与 fork 函数不同的是, spawn 函数产生的子进程不再执行与父进程相同的程序, 而是装载新的 ELF 文件, 执行新的程序。

如果我们再结合之前提到的shell的原理，就会发现spawn就是尝试执行shell里的命令，所以在这里尝试追加执行是最佳位置。所以当用户指定的程序路径不存在时，可以在这里尝试在路径后追加 .b 再打开。

```
1 // user/lib/spawn.c spawn
2
3 if ((fd = open(prog, O_RDONLY)) < 0)
4 {
5     char fd_default[128];
6     strcpy(fd_default, prog);
7     int fd_len = strlen(fd_default);
8     fd_default[fd_len] = '.';
9     fd_default[fd_len + 1] = 'b';
10    fd_default[fd_len + 2] = 0;
11    if ((fd = open(fd_default, O_RDONLY)) < 0)
12    {
13        return fd;
14    }
15 }
```

这里我一开始由于对于lab6的架构不太熟悉，所以采用了特判的方式进行，也可以在一定程度达到预期效果，不过还是会相对比较麻烦的。（具体而言就是对输入字符串进行buf的修改）

```
1 void clarify_cmd(char* cmd) {
2     int len = strlen(cmd);
3     for (int i = 0; i < len; i++) {
4         if ((cmd[i] == 'l' && cmd[i + 1] == 's' &&
5             (i + 3 >= len || cmd[i + 2] != '.' && cmd[i + 3] != 'b')) ||
6             (cmd[i] == 'c' && cmd[i + 1] == 'd' &&
7             (i + 3 >= len || cmd[i + 2] != '.' && cmd[i + 3] != 'b'))) {
8                 len += 2;
9                 for (int j = len - 1; j > i + 1; j--) {
10                     cmd[j] = cmd[j - 2];
11                 }
12                 cmd[i + 2] = '.';
13                 cmd[i + 3] = 'b';
14             }
15         }
16         for (int i = 0; i < len; i++) {
17             if ((cmd[i] == 'c' && cmd[i + 1] == 'a' && cmd[i + 2] == 't' &&
18                 (i + 4 >= len || cmd[i + 3] != '.' && cmd[i + 4] != 'b')) ||
19                 (cmd[i] == 'p' && cmd[i + 1] == 'w' && cmd[i + 2] == 'd' &&
20                 (i + 4 >= len || cmd[i + 3] != '.' && cmd[i + 4] != 'b'))) {
21                     len += 2;
22                     for (int j = len - 1; j > i + 2; j--) {
23                         cmd[j] = cmd[j - 2];
24                     }
25                     cmd[i + 3] = '.';
26                     cmd[i + 4] = 'b';
27                 }
28             }
29             for (int i = 0; i < len; i++) {
30                 if ((cmd[i] == 'e' && cmd[i + 1] == 'c' && cmd[i + 2] == 'h' &&
31                     cmd[i + 3] == 'o' &&
32                     (i + 5 >= len || cmd[i + 4] != '.' && cmd[i + 5] != 'b')) ||
33                     (cmd[i] == 'h' && cmd[i + 1] == 'a' && cmd[i + 2] == 'l' &&
```

```

34         cmd[i + 3] == 't' &&
35         (i + 5 >= len || cmd[i + 4] != '.' && cmd[i + 5] != 'b')) ||
36         (cmd[i] == 't' && cmd[i + 1] == 'r' && cmd[i + 2] == 'e' &&
37         cmd[i + 3] == 'e' &&
38         (i + 5 >= len || cmd[i + 4] != '.' && cmd[i + 5] != 'b'))) {
39     len += 2;
40     for (int j = len - 1; j > i + 3; j--) {
41         cmd[j] = cmd[j - 2];
42     }
43     cmd[i + 4] = '.';
44     cmd[i + 5] = 'b';
45 }
46 }
47 for (int i = 0; i < len; i++) {
48     if ((cmd[i] == 'm' && cmd[i + 1] == 'k' && cmd[i + 2] == 'd' &&
49           cmd[i + 3] == 'i' && cmd[i + 4] == 'r' &&
50           (i + 6 >= len || cmd[i + 5] != '.' && cmd[i + 6] != 'b')) ||
51           (cmd[i] == 't' && cmd[i + 1] == 'o' && cmd[i + 2] == 'u' &&
52           cmd[i + 3] == 'c' && cmd[i + 4] == 'h' &&
53           (i + 6 >= len || cmd[i + 5] != '.' && cmd[i + 6] != 'b'))) {
54     len += 2;
55     for (int j = len - 1; j > i + 4; j--) {
56         cmd[j] = cmd[j - 2];
57     }
58     cmd[i + 5] = '.';
59     cmd[i + 6] = 'b';
60 }
61 }
62 for (int i = 0; i < len; i++) {
63     if ((cmd[i] == 'h' && cmd[i + 1] == 'i' && cmd[i + 2] == 's' &&
64           cmd[i + 3] == 't' && cmd[i + 4] == 'o' && cmd[i + 5] == 'r' &&
65           cmd[i + 6] == 'y' &&
66           (i + 8 >= len || cmd[i + 7] != '.' && cmd[i + 8] != 'b'))) {
67     len += 2;
68     for (int j = len - 1; j > i + 6; j--) {
69         cmd[j] = cmd[j - 2];
70     }
71     cmd[i + 7] = '.';
72     cmd[i + 8] = 'b';
73 }
74 }
75 }

```

实现更丰富的命令

tree

tree本身的三种类型的判断可以参考课程组源码中对于ls指令的实现，还是比较简单的，而具体对于程序的遍历，也就只是dfs的过程。

实现了 `tree` 的三种模式：

- `-a` 是显示所有的文件和目录，模式缺省时默认为 `-a`；
- `-d` 显示所有的目录；
- `-f` 为显示文件和目录的完整路径。

同时记录了获取的文件及目录数量，在对参数中每个目录（默认根目录）实行 `tree` 指令后输出。（根据手册行为，初始的目录是不计入的）

实现方式是递归地判断文件类型：如果读到文件，就输出文件名（无 `-d` 模式）；如果读到目录文件，就继续对目录文件进行处理。

```
1  /* TODO:lab6-challenge */
2  #include <lib.h>
3  #define MAXDEPTH 50
4  int flag[256];
5  int file_cnt;
6  int dir_cnt;
7  void tree(char* path) {
8      int r;
9      struct Stat st;
10
11     if ((r = stat(path, &st)) < 0) {
12         user_panic("stat %s: %d", path, r);
13     }
14
15     if (!st.st_isdir) {
16         printf("%s [error opening dir]\n", path);
17         printf("0 directories, 0 files\n");
18         exit();
19     }
20
21     printf("\033[33m%s\033[m\n", path);
22     treedir(path, 0);
23 }
24
25 void treedir(char* path, int step) {
26     int fd;
27     int n;
28     struct File f;
29
30     if ((fd = open(path, O_RDONLY)) < 0) {
31         return;
32     }
33     while ((n = readn(fd, &f, sizeof(f))) == sizeof(f)) {
34         if (f.f_name[0]) {
35             tree1(path, f.f_type == FTYPE_DIR, f.f_name, step + 1);
36         }
37     }
38     if (n > 0) {
39         user_panic("short read in directory %s", path);
40     }
41     if (n < 0) {
42         user_panic("error reading directory %s: %d", path, n);
43     }
44 }
45
46 void tree1(char* path, u_int isdir, char* name, int step) {
47     char* sep;
48
49     if (flag['d'] && !isdir) {
50         return;
51     }
52     //
53     if (step > MAXDEPTH) {
54         debugf("tree is too deep");
```

```

55         return;
56     }
57     // 
58     for (int i = 0; i < step - 1; i++) {
59         printf("\033[34m    \033[m");
60     }
61     printf("\033[34m-- \033[m");
62
63     if (path[0] && path[strlen(path) - 1] != '/') {
64         sep = "/";
65     } else {
66         sep = "";
67     }
68
69     if (flag['f'] && path) {
70         printf("\033[34m%s%s\033[m", path, sep);
71     }
72     printf("\033[34m%s\n\033[m", name);
73
74     if (isdir) {
75         dir_cnt += 1;
76         char newpath[256];
77         strcpy(newpath, path);
78         int namelen = strlen(name);
79         int pathlen = strlen(path);
80         if (strlen(sep) != 0) {
81             newpath[pathlen] = '/';
82             for (int i = 0; i < namelen; i++) {
83                 newpath[pathlen + i + 1] = name[i];
84             }
85             newpath[pathlen + namelen + 1] = 0;
86             treedir(newpath, step);
87         } else {
88             for (int i = 0; i < namelen; i++) {
89                 newpath[pathlen + i] = name[i];
90             }
91             newpath[pathlen + namelen] = 0;
92             treedir(newpath, step);
93         }
94     } else {
95         file_cnt += 1;
96     }
97 }
98 void usage(void) {
99     printf("\033[31musage: tree [-adf] [directory...]\n\033[m");
100    exit();
101 }
102 int main(int argc, char** argv) {
103     int i;
104     ARGBEGIN {
105     default:
106         usage();
107     case 'a':
108     case 'd':
109     case 'f':
110         flag[(u_char)ARGC()]++;
111         break;
112     }

```

```

113     ARGEND
114     int file[MAXDEPTH] = {0};
115     if (argc == 0) {
116         tree("./");
117     } else {
118         tree(argv[1]);
119     }
120     if (flag['d']) {
121         printf("\033[32m\n%d directories\033[m\n", dir_cnt);
122     } else {
123         printf("\033[32m\n%d directories, %d files\033[m\n", dir_cnt,
124         file_cnt);
125     }
126     printf("\n");
127     return 0;
128 }
```

这里使用了彩色输出使输出的树结构更加的美观简洁。

mkdir 和 touch

这里的mkdir()和touch()用户函数可以模仿咱们最后一次上机时的exam进行编写。

首先在相应头文件中添加需要用到的宏、函数和结构体声明。 (这里直接将整个挑战性任务所有的设置全部一次列出了，具体含义可以直接参考函数名)

```

1 // user/include/fsreq.h
2
3 #define FSREQ_CREATE_FILE 8
4 #define FSREQ_CREATE_DIR 9
5 #define FSREQ_OPENAT 10
6 struct Fsreq_create_file {
7     u_char req_path[MAXPATHLEN];
8 };
9
10 struct Fsreq_create_dir {
11     u_char req_path[MAXPATHLEN];
12 };
13 struct Fsreq_openat {
14     u_int dir_fileid;
15     char req_path[MAXPATHLEN];
16     u_int req_oemode;
17 };
18
19 // user/include/lib.h
20
21 int fsipc_create_file(const char* );
22 int fsipc_create_dir(const char* );
23 int fsipc_openat(u_int dir_fileid,
24                   const char* path,
25                   u_int oemode,
26                   struct Fd* fd);
27 int create_file(const char* path);
28 int create_dir(const char* path);
29 int openat(int dirfd, const char* path, int mode);
30
31 #define O_APPEND 0x0004
32 int syscall_get_cur_path(char* buf);
```

```

33 int syscall_set_cur_path(char* path);
34 int chdir(char* path);
35 int getcwd(char* buf);
36 void pathcat(char* path, const char* suffix);
37
38 // fs/serve.h
39
40 int walk_path_at(struct File* par_dir,
41                   char* path,
42                   struct File** pdir,
43                   struct File** pfile,
44                   char* lastelem);
45 int file_openat(struct File* dir, char* path, struct File** pfile);
46 char* skip_slash(char* p);

```

接下来从用户到文件服务依次实现调用链：

创建用户库函数，调用 `fsipc_create_file` 和 `fsipc_create_dir`。

```

1 // user/lib/file.c
2
3 int create_file(const char* path) {
4     if (path[0] == '/') {
5         return fsipc_create_file(path);
6     }
7     char dpath[128];
8     try(getcwd(dpath));
9     pathcat(dpath, path);
10    return fsipc_create_file(dpath);
11 }
12
13 int create_dir(const char* path) {
14     if (path[0] == '/') {
15         return fsipc_create_dir(path);
16     }
17     char dpath[128];
18     try(getcwd(dpath));
19     pathcat(dpath, path);
20     return fsipc_create_dir(dpath);
21 }

```

实现 `fsipc_create_file` 和 `fsipc_create_dir`，将请求需要的参数（文件路径，文件类型）赋值给设定好的 `struct Fsreq_create_file` 和 `struct Fsreq_create_dir` 结构体中，传给 `fsipc` 函数。

```

1 // user/lib/fsipc.c
2
3 int fsipc_create_file(const char* path) {
4     struct Fsreq_create_file* req;
5     req = (struct Fsreq_create_file*)fsipcbuf;
6     if (strlen(path) >= MAXPATHLEN) {
7         return -E_BAD_PATH;
8     }
9     strcpy((char*)req->req_path, path);
10    return fsipc(FSREQ_CREATE_FILE, req, 0, 0);
11 }

```

```

12
13 int fsipc_create_dir(const char* path) {
14     struct Fsreq_create_dir* req;
15     req = (struct Fsreq_create_dir*)fsipcbuf;
16     if (strlen(path) >= MAXPATHLEN) {
17         return -E_BAD_PATH;
18     }
19     strcpy((char*)req->req_path, path);
20     return fsipc(FSREQ_CREATE_DIR, req, 0, 0);
21 }
```

实现文件服务端，调用已经实现的 `file_create` 函数即可。返回0即表示正常，否则返回相应的报错值。

```

1 // fs/serv.c
2
3 void serve_create_file(u_int envid, struct Fsreq_create_file* req) {
4     char* path = req->req_path;
5     int r;
6     struct File* file;
7     if ((r = file_create(path, &file)) < 0) {
8         ipc_send(envid, r, 0, 0);
9     } else {
10        file->f_type = FTYPE_REG;
11        ipc_send(envid, 0, 0, 0);
12    }
13 }
14
15 void serve_create_dir(u_int envid, struct Fsreq_create_dir* req) {
16     char* path = req->req_path;
17     int r;
18     struct File* file;
19     if ((r = file_create(path, &file)) < 0) {
20         ipc_send(envid, r, 0, 0);
21     } else {
22        file->f_type = FTYPE_DIR;
23        ipc_send(envid, 0, 0, 0);
24    }
25 }
```

在 `serve` 函数中新建 ipc 请求的情况。

```

1 // fs/serv.c serve
2
3 /* TODO:lab6-challenge */
4 case FSREQ_CREATE_FILE:
5     serve_create_file(which, (struct Fsreq_create_file*)REQVA);
6     break;
7
8 case FSREQ_CREATE_DIR:
9     serve_create_dir(which, (struct Fsreq_create_dir*)REQVA);
10    break;
```

实现了文件创建功能后，`mkdir` 和 `touch` 只需要相应的调用即可，并传入相应的文件类型。(这里同样使用了彩色输出)

```

1 // user/mkdir.c
2
3 /* TODO:lab6-challenge */
4 #include <lib.h>
5
6 int main(int argc, char** argv) {
7     if (argc == 2) {
8         if (create_dir(argv[1]) < 0) {
9             printf("create dir failed!\n");
10        } else {
11            printf("\033[35mSuccess mkdir\033[m\n");
12        }
13    } else {
14        printf("usage: mkdir [dirname]\n");
15    }
16    return 0;
17}
18
19
20 // user/touch.c
21
22 /* TODO:lab6-challenge */
23 #include <lib.h>
24
25 int main(int argc, char** argv) {
26     if (argc == 2) {
27         if (create_file(argv[1]) < 0) {
28             printf("create file failed!\n");
29         } else {
30             printf("\033[35mSuccess touch\033[m\n");
31         }
32     } else {
33         printf("usage: touch [filename]\n");
34     }
35     return 0;
36}
37

```

最后需要修改 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。

实现方式是在文件没打开时尝试创建文件，然后打开即可。

```

1 // user/sh.c parsecmd
2
3 case '>':
4     if (gettoken(0, &t) != 'w') {
5         debugf("syntax error: > not followed by word\n");
6         exit();
7     }
8
9     if ((r = open(t, O_WRONLY)) < 0) {
10        if (create(t, FTYPE_REG) < 0) {
11            debugf("> open and create failed\n");
12            exit();
13        }
14        r = open(t, O_WRONLY);
15    }

```

```
16     fd = r;
17     dup(fd, 1);
18     close(fd);
```

实现历史命令功能

不覆盖写入

由于我们需要不断向history文件中写入新的指令信息，所以文件的打开方式应该为可添加写入，这个的具体实现可以参考去年lab5实验的一个exam——[BUAA OS Lab5-2 课上测试](#)

```
1 //user/include/lib.h
2 #define O_APPEND 0x0004
3 //user/lib/file.c open
4 if ((mode & O_APPEND) != 0) {
5     fd->fd_offset = size;
6 }
7 //user/lib/sh.c
8 /* TODO:lab6-challenge */
9 void save_cmd(char* cmd) {
10     int r = open(".history", O_CREAT | O_WRONLY | O_APPEND);
11     if (r < 0) {
12         debugf("open .history failed! in save");
13         return r;
14     }
15     write(r, cmd, strlen(cmd));
16     write(r, "\n", 1);
17     return 0;
18 }
```

之后直接在main中调用save_cmd即可。

监听上下键

这部分在上文中已经提及了，主要就是对linux编码的判断和处理。

```
1 //user/lib/sh.c
2 /* TODO:lab6-challenge */
3 int offset; // 0:empty line, -1:last cmd, -2:...
4 int solvedirCmd(char* buf, int type) { // type: 0 means up, 1 means down
5     if (newDirCmd == 1) {
6         offset = 0;
7     }
8     if (type == 0) {
9         offset--;
10    } else if (offset < 0) {
11        offset++;
12    }
13    int x = 0;
14    if (offset == 0) {
15        while (buf[x] != '\0') {
16            buf[x] = '\0';
17            x++;
18        }
19        return -1;
20    }
21    int fdnum = open(".history", O_RDONLY);
```

```

22     if (fdnum < 0) {
23         debugf("open .history failed in sloveDir!\n");
24         return 0;
25     }
26     struct Fd* fd = num2fd(fdnum);
27     char* c;
28     char* begin = fd2data(fd);
29     char* end = begin + ((struct Filefd*)fd)->f_file.f_size;
30     c = end - 1;
31
32     while (((*c) == '\n' || (*c) == 0) && (c > begin)) {
33         c--;
34     }
35
36     if (c == begin) { // no history cmd
37         buf[0] = '\0';
38         return 0;
39     }
40
41     c++; // last \n or \0
42     int i;
43     for (i = 0; i > offset; i--) {
44         while ((*c) != '\n' && (*c) != '\0') {
45             c--;
46             if (c <= begin) {
47                 break;
48             }
49         }
50         c--;
51         if (c <= begin) {
52             break;
53         }
54     }
55     offset = i; // avoid offset too bigger than real cmd num
56     if (c > begin) {
57         while (c > begin && (*c) != '\n') {
58             c--;
59         }
60         if ((*c) == '\n') {
61             c++;
62         }
63     } else {
64         c = begin;
65     }
66     int now = 0;
67     while (buf[now] != '\0') {
68         buf[now] = '\0';
69         now++;
70     }
71     now = 0;
72     while ((*c) != '\n' && (*c) != '\0' && (*c) < end) {
73         buf[now] = *c;
74         now++;
75         c++;
76     }
77     return now;
78 }
```

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。具体细节看上面部分代码即可。

这里用到了linux对于左键和清空键的编码

左: 27 '[' 'D'

清空: 27 '[' 'K'

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。

```
1 //user/history.c
2 /* TODO:lab6-challenge */
3 #include <lib.h>
4 int main(int argc, char** argv) {
5     if (argc != 1) {
6         debugf("usage: history\n");
7     }
8     int fdnum = open(".history", O_RDONLY);
9     if (fdnum < 0) {
10         debugf("open .history failed\n");
11         return;
12     }
13     char buf;
14     int r;
15     int cnt = 0;
16     int newline = 1;
17     while ((r = read(fdnum, &buf, 1)) != 0) {
18         if (newline) {
19             debugf("\033[32mHistory \033[m");
20             debugf("\033[34m%d \033[m", cnt);
21             debugf("\033[32m: \033[m");
22             debugf("%c", buf);
23             cnt++;
24             newline = 0;
25         } else {
26             debugf("%c", buf);
27         }
28         if (buf == '\n') {
29             newline = 1;
30         }
31     }
32 }
```

支持相对路径

这部分参考了大佬的博客——[「操作系统」challenge实验报告](#)

访问路径的系统调用

相对路径的实现思路是在内核维护一个“当前目录”（绝对路径），并通过系统调用访问和修改。因此实现了两个系统调用以及添加相应的变量如下。

```
1 // include/env.h
```

```

2  extern char cur_path[128];
3
4 // include/env.c
5 char cur_path[128];
6
7 // include/syscall.h
8 SYS_get_cur_path,
9 SYS_set_cur_path,
10
11 // kern/syscall_all.c
12 #ifndef _CUR_PATH_
13 #define _CUR_PATH_
14
15 #define MAX_PATH_LEN      128
16 #define E_CUR_PATH        21
17
18 int sys_get_cur_path(char *buf) {
19     strcpy(buf, cur_path);
20     return 0;
21 }
22
23 int sys_set_cur_path(char *path) {
24     if (strlen(path) >= MAX_PATH_LEN) {
25         return -E_CUR_PATH;
26     }
27     strcpy(cur_path, path);
28
29     return 0;
30 }
31
32#endif // _CUR_PATH_
33
34 [SYS_get_cur_path] = sys_get_cur_path,
35 [SYS_set_cur_path] = sys_set_cur_path,
36
37 // user/include/lib.h
38 int syscall_get_cur_path(char *buf);
39 int syscall_set_cur_path(char *path);
40
41 // user/lib/syscall_lib.c
42 int syscall_get_cur_path(char *buf) {
43     return msyscall(SYS_get_cur_path, buf);
44 }
45
46 int syscall_set_cur_path(char *path) {
47     return msyscall(SYS_set_cur_path, path);
48 }

```

添加库函数

实现 `chdir` 和 `getcwd` 封装两个系统调用，实现 `pathcat` 用于拼接路径。

```

1 // user/lib/path.c
2
3 #include <lib.h>
4
5 int chdir(char *path) {

```

```

6     return syscall_set_cur_path(path);
7 }
8
9 int getcwd(char *buf) {
10    return syscall_get_cur_path(buf);
11 }
12
13 void pathcat(char *path, const char *suffix) {
14    int pre_len = strlen(path);
15
16    if (suffix[0] == '.') {
17        suffix += 2;
18    }
19    int suf_len = strlen(suffix);
20
21    if (pre_len != 1) {
22        path[pre_len++] = '/';
23    }
24    for (int i = 0; i < suf_len; i++) {
25        path[pre_len + i] = suffix[i];
26    }
27
28    path[pre_len + suf_len] = 0;
29 }
30
31 // user/include/lib.h
32 int chdir(char *path);
33 int getcwd(char *buf);
34 void pathcat(char *path, const char *suffix);

```

cd 和 pwd 命令

cd 命令是内部命令，因此不应该创建一个子进程，而是在检测到 cd 命令时直接执行并退出。

在进程创建的时候需要初始化路径。

```

1 // user/sh.c/
2
3 /* TODO:lab6-challenge */
4 void chpwd(int argc, char** argv) {
5     int r;
6     if (argc == 1) {
7         if ((r = chdir("/")) < 0) {
8             printf("cd failed: %d\n", r);
9             exit();
10        }
11        printf("/\n");
12    } else {
13        if (argv[1][0] == '/') {
14            if ((r = chdir(argv[1])) < 0) {
15                printf("cd failed: %d\n", r);
16                exit();
17            }
18        } else {
19            // Get current direct path.
20            char path[128];
21            if ((r = getcwd(path)) < 0) {
22                printf("cd failed: %d\n", r);

```

```

23         exit();
24     }
25     if (strcmp(argv[1], "..") == 0) {
26         int len = strlen(path);
27         while (path[len - 1] != '/') {
28             path[len - 1] = '\0';
29             len--;
30         }
31         if (strcmp(path, "/") != 0) {
32             path[len - 1] = '\0';
33         }
34     } else {
35         // Parse the target indirect path into direct.
36         pathcat(path, argv[1]);
37     }
38
39     // Confirm the path exists and is a directory.
40     if ((r = open(path, O_RDONLY)) < 0) {
41         printf("path %s doesn't exist: %d\n", path, r);
42         exit();
43     }
44     close(r);
45     struct Stat st;
46     if ((r = stat(path, &st)) < 0) {
47         user_panic("stat %s: %d", path, r);
48     }
49     if (!st.st_isdir) {
50         printf("path %s is not a directory\n", path);
51         exit();
52     }
53
54     if ((r = chdir(path)) < 0) {
55         printf("cd failed: %d\n", r);
56         exit();
57     }
58 }
59
60 return;
61 }
62 void do_incmd(int argc, char** argv) {
63     if (strcmp(argv[0], "cd") == 0 || strcmp(argv[0], "cd.b") == 0) {
64         chpwd(argc, argv);
65     } else {
66         /* TODO:command clear */
67         debugf("\033[2J");
68         debugf("\033[1;1H");
69         //
70     }
71 }
72
73 // user/sh.c/main
74 if ((r = chdir("/")) < 0) {
75     printf("created root path failed: %d\n", r);
76 }

```

pwd 命令直接获取当前路径即可。

```

1 // user/pwd.c
2 #include <lib.h>
3
4 void usage(void) {
5     printf("usage: pwd\n");
6     exit();
7 }
8
9 int main(int argc, char **argv) {
10    int r;
11    char buf[128];
12
13    if (argc != 1) {
14        usage();
15    } else {
16        if ((r = getcwd(buf)) < 0) {
17            printf("get path failed: %d\n", r);
18            exit();
19        }
20        printf("%s\n", buf);
21    }
22
23    return 0;
24 }
25

```

相对路径支持

在上述思路下，支持相对路径最重要的是修改 `open` 函数的逻辑。

我的思路是除了 `/` 开头的绝对路径以外，默认打开相对路径，用 `getcwd` 获取当前绝对路径，在 `open` 函数内对相对路径进行拼接。这样做就不需要修改文件系统的内部逻辑了。

相应地，在必做部分实现的 `create_file` 和 `create_dir` 也需要修改。

```

1 // user/lib/file.c
2
3 /* TODO:lab6-challenge */
4 if (path[0] == '/') {
5     try(fsipc_open(path, mode, fd));
6 } else {
7     char dpath[128];
8     try(getcwd(dpath));
9     pathcat(dpath, path);
10    try(fsipc_open(dpath, mode, fd));
11 }
12
13 int create_file(const char* path) {
14     if (path[0] == '/') {
15         return fsipc_create_file(path);
16     }
17     char dpath[128];
18     try(getcwd(dpath));
19     pathcat(dpath, path);
20     return fsipc_create_file(dpath);
21 }
22
23 int create_dir(const char* path) {

```

```

24     if (path[0] == '/') {
25         return fsipc_create_dir(path);
26     }
27     char dpath[128];
28     try(getcwd(dpath));
29     pathcat(dpath, path);
30     return fsipc_create_dir(dpath);
31 }

```

原有命令和函数的修改

因为默认路径变成了相对路径，因此在很多命令中默认的路径应从 `/` 改为 `./`。

`spawn` 中默认打开的是根目录下的命令，需要对此做相应的调整。

```

1 // user/tree.c
2 tree("./");
3
4 // user/ls.c
5 ls("./", "");
6
7 // user/lib/spawn.c
8 if (prog[0] != '/') {
9     fd_default[0] = '/';
10    strcpy(fd_default + 1, prog);
11 } else {
12     strcpy(fd_default, prog);
13 }

```

彩色输出

Code	Text color	Code	Text color
30	black	90	dark grey
31	red	91	light red
32	green	92	light green
33	yellow	93	yellow
34	blue	94	light blue
35	magenta	95	light purple
36	cyan	96	turquoise
37	white	97	white

这里我们只需要按照对应的颜色调用相应的printf即可，例如：

```
1 | printf("\033[35mSuccess touch\033[m\n");
```

这里就可以直接修改35为其他颜色对应编号以完成彩色输出。

清空操作

增加清空操作可以让你的shell更加便捷和赏心悦目，而且添加非常简单，可以直接作为一个内部命令添加，具体即在之前添加cd指令的地方进行判断是否为clear，再进行简单操作即可。

```
1 | void do_incmd(int argc, char** argv) {
2 |     if (strcmp(argv[0], "cd") == 0 || strcmp(argv[0], "cd.b") == 0) {
3 |         chpwd(argc, argv);
4 |     } else {
5 |         /* TODO:command clear */
6 |         debugf("\033[2J");//清屏
7 |         debugf("\033[1;1H");//控制光标回到第一行第一列
8 |         //
9 |     }
10 | }
```

这样我们shell的功能测试告一段落，撒花！！！

实验难点

在本lab代码编写过程中，我出现过几个难以解决的bug，不过好在最终都顺利解决了，现在下方简要汇总：

1. 对 `readline` 的重构过程中，如果遇到连续的'\b'指令，需要进行特判处理。
2. 在编写相对路径的 `cd` 指令时，需要对 `argv[1]` 是否为 `..` 进行特判，如果是则不能通过相对路径链接的方式形成绝对路径。与此同时，绝对路径和相对路径使用在 `create` 时的判断也非常容易忽略。
3. shell的许多行为都经过了linux的二次编码，如果不知道这部分知识，很可能会对于实现对应功能无所适从，例如我就是在学长博客中才了解到linux对于上下左右键重新编码这件事的。
4. 由于原本的文件系统并没有实现追加功能，因此对 `.history` 的追加（append）写入需要实现追加的打开方式。

功能测试

一行多命令

；分隔符的实验命令和实验现象如下：

```
$ ls;cat newmotd;mkdir test
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
testfdsharing.b
testshell.sh
script
ls.b
pwd.b
echo.b
sh.b
tree.b
halt.b
testptelibrary.b
istory
```

```
[00003805] destroying 00003805
[00003805] free env 00003805
i am killed ...
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
This is a different message of the day!
[00004805] destroying 00004805
[00004805] free env 00004805
```

```
i am killed ...
[00004004] destroying 00004004
[00004004] free env 00004004
i am killed ...
Success mkdir
[00005004] destroying 00005004
[00005004] free env 00005004
```

```
$ 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
mkdir.b
testfdsharing.b
testshell.sh
script
ls.b
pwd.b
echo.b
sh.b
tree.b
halt.b
testptelibrary.b
istory
test
```

```
[00006004] destroying 00006004
[00006004] free env 00006004
i am killed ...
[00005002] destroying 00005002
```

```
[00005803] destroying 00005803
[00005803] free env 00005803
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
mkdir.b
testfdsharing.b
testshell.sh
script
ls.b
pwd.b
echo.b
sh.b
tree.b
halt.b
testptelibrary.b
history
```

istory test

```
[00009005] destroying 00009005
[00009005] free env 00009005
i am killed ...
[00008804] destroying 00008804
[00008804] free env 00008804
i am killed ...
[00008003] destroying 00008003
[00008003] free env 00008003
i am killed ...
```

从现象中可以看出，; 分隔符实现了一行多命令的功能，在; 左侧和右侧没有指令以及一行多个; 的情况下都能正常运行。

实现后台任务

& 分隔符的实验命令和实验现象如下：

```
$ ls & cat newmotd
[0000d005] destroying 0000d005
[0000d005] free env 0000d005
i am killed ...
[0000c004] destroying 0000c004
[0000c004] free env 0000c004
i am killed ...
This is a different message of the day
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
testfdsharing.b
testshell.sh
[0000e805] destroying 0000e805
[0000e805] free env 0000e805
i am killed ...
script
ls.b
[0000d803] destroying 0000d803
[0000d803] free env 0000d803
i am killed ...
```

两条指令都正常进行并退出，其中 `ls.b` 指令（进程）在后台进行。

实现引号支持

引号支持的实验命令和实验现象如下：

```
$ echo "123|321"
123|321
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...

$ cat "newmotd"
This is a different message of the day!
[00004004] destroying 00004004
[00004004] free env 00004004
i am killed ...
[00003803] destroying 00003803
[00003803] free env 00003803
i am killed ...
```

实现键入命令时任意位置的修改

这个在我实际输入时经常打错，其实时刻都在验证这个功能。

实现程序名称中 `.b` 的省略

前面的实验和实验现象中已经验证了。

实现更丰富的命令

`tree`、`mkdir` 和 `touch` 指令以及相应的选项验证如下：

```
$ mkdir test;touch test/txt1;touch test/txt2
Success mkdir
[00005805] destroying 00005805
[00005805] free env 00005805
i am killed ...
[00005004] destroying 00005004
```

```
[00005004] destroying 00005004  
[00005004] free env 00005004  
i am killed ...  
Success touch  
[00006805] destroying 00006805  
[00006805] free env 00006805  
i am killed ...  
[00006004] destroying 00006004  
[00006004] free env 00006004  
i am killed ...  
Success touch  
[00007004] destroying 00007004  
[00007004] free env 00007004  
i am killed ...  
[00004803] destroying 00004803  
[00004803] free env 00004803  
i am killed ...
```

```
$ tree  
./  
|-- 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  
|-- testfdsharing.b  
|-- testshell.sh  
|-- script  
|-- ls.b  
|-- pwd.b  
|-- echo.b
```

```
|-- sh.b  
|-- tree.b  
|-- halt.b  
|-- testptelibrary.b  
|-- istory  
|-- test  
    |-- txt1  
    |-- txt2
```

1 directories, 28 files

创建文件目录和创建文件成功。tree指令用于查看创建效果。

下面验证tree的相关参数：

```
$ tree -f
./
|-- ./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
|-- ./testfdsharing.b
|-- ./testshell.sh
|-- ./script
|-- ./ls.b
|-- ./pwd.b
|-- ./echo.b
|-- ./sh.b
|-- ./tree.b
|-- ./halt.b
|-- ./testptelibrary.b
|-- ./istory
|-- ./test
```

```
|-- ./test/txt1  
|-- ./test/txt2
```

```
1 directories, 28 files
```

```
$ tree -d  
./  
|-- test
```

```
1 directories
```

```
$ tree -a
./
|-- 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
|-- testfdsharing.b
|-- testshell.sh
|-- script
|-- ls.b
|-- pwd.b
|-- echo.b
|-- sh.b
|-- tree.b
```

```
|-- halt.b  
|-- testptelibrary.b  
|-- istory  
|-- test  
|   |-- txt1  
|   |-- txt2
```

1 directories, 28 files

`tree` 指令的基本功能正常，`-f` 选项输出完整路径正常，`-d` 选项只输出文件目录功能正常，指定路径功能正常。并且能够输出文件目录和文件的计数信息。

修改的重定向功能运行如下：

```
$ cat newmotd >newfile  
[0000f004] destroying 0000f004  
[0000f004] free env 0000f004  
i am killed ...  
[0000e803] destroying 0000e803  
[0000e803] free env 0000e803  
i am killed ...  
  
$ cat newfile  
This is a different message of the day!  
[00010004] destroying 00010004  
[00010004] free env 00010004  
i am killed ...  
[0000f803] destroying 0000f803  
[0000f803] free env 0000f803  
i am killed ...
```

可以看到，原本没有 `newfile` 这一文件，重定向会创建这个文件并将内容写入。

实现历史命令功能

`history` 指令比较容易形式化验证。

```
$ history
History 0 : clear
History 1 : echo "123|321"
History 2 : cat "newmotd"
History 3 : clear
History 4 : mkdir test;touch test/txt1;touch test/txt2
History 5 : tree
History 6 : tree -f
History 7 : tree -d
History 8 : tree -a
History 9 : cat newmotd >new file
History 10 : cat newmotd >newfile
History 11 : ls
History 12 : cat newmotd >newfile
History 13 : cat newfile
History 14 : history
[00011004] destroying 00011004
[00011004] free env 00011004
i am killed ....
[00010803] destroying 00010803
[00010803] free env 00010803
i am killed ...
```

而上下移动在使用过程中已经多次验证了。可以看到，历史记录的查看和运行，以及回到原本输入内容的功能均正确。

支持相对路径

相对路径的测试如下：

```
$ cd test
```

```
$ tree
```

```
./  
|-- txt1  
|-- txt2  
|-- istory
```

```
0 directories, 3 files
```

```
[00012004] destroying 00012004
```

```
[00012004] free env 00012004
```

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

```
[00011803] destroying 00011803
```

```
[00011803] free env 00011803
```

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

```
$ pwd
```

```
/test
```

```
[00013004] destroying 00013004
```

```
[00013004] free env 00013004
```

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

```
[00012803] destroying 00012803
```

```
[00012803] free env 00012803
```

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

测试 cd .. :

```
$ mkdir test
Success mkdir
[00003004] destroying 00003004
[00003004] free env 00003004
i am killed ...
[00002803] destroying 00002803
[00002803] free env 00002803
i am killed ...

$ cd test

$ cd ..

$ pwd
/
[00004004] destroying 00004004
[00004004] free env 00004004
```

后记

最后是对logo的一个花活，OS完结撒花！！！整体来说lab6-challenge的难度还是比较大的，我也参考了一些学长的博客才完成了这次任务，但相比于其他challenge而言，lab6的测试可以直接依托shell实现而不必自己编写程序，可以说lab6-challenge带给我的收获还是非常大的。

Charles Shell 2023
21373191

charles2530@github.io

Charles \$ |

以上是我对于lab6-challenge的实现思路和部分源码，可能存在bug，欢迎讨论交流！！！