Lab 6 Challenge

在Lab6实验中，我们在内核上搭载了第一个软件——**命令解析器shell**。Lab 6 Challenge是对于这个简单的shell的扩展，使其具有**更加灵活**的使用方式，并且能够支持**更多命令**。

输入make test lab=6_2 && make run可以进入shell，显示：

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
                ::
    ::
                ::
    ::
                ::                      MOS Shell 2023
     ::
                ::                  completed by singleApo
     ::
                ::
    ::
                ::
    ::

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

一行多命令

';'将命令分割为两个部分，依次执行。

_gettoken()函数用于寻找命令中的特殊符号并返回，可以发现宏SYMBOLS中已经包含了';': `#define SYMBOLS "<|>&;()"`。

```
// _gettoken()
    if (strchr(SYMBOLS, *s)) {
        int t = *s;
        *p1 = s;
        *s++ = 0;
        *p2 = s;
        return t;
    }
```

因此，我们只需要在原有的parsecmd()函数中，对于';'进行处理即可。

fork出子进程执行左边的命令，父进程**等待子进程结束后**再执行右边的命令，确保命令的**依次执行**。

```
// parsecmd()
    case ';':
        if ((*rightpipe = fork()) == 0) {   // child
            return argc;
        } else {                            // parent
            wait(*rightpipe);
            return parsecmd(argv, rightpipe);
        }
        break;
```

后台任务

同样，在原有的_gettoken()函数中，我们已经实现了对于'&'的读取。修改parsecmd()函数，对于'&'进行处理即可。

```
// parsecmd()
    case '&':
        if ((r = fork()) == 0) {            // child
            return argc;
        } else {                            // parent
            return parsecmd(argv, rightpipe);
        }
        break;
```

fork出子进程后无需等待，左右两边的命令将**同时执行**，其中子进程将在后台执行。

引号支持

在_gettoken()函数中有两个指针p1和p2，注意到在处理特殊符号时，p1指针指向了该符号，而p2指针指向的是该符号的后一个字符。

因此，在处理**一对引号**内的字符串时，首先读取第一个引号，随后将p1指向**引号后的第一个字符**，p2指向引号后的**第一个引号**即可。

```
// _gettoken()
    if (*s == '"') {
        s++;
        *p1 = s;
        while (*s && *s != '"') {
```

```
            s++;
        }
        *p2 = s;
        if (*s == 0) {
            return 0;
        }
        *s++ = 0;
        return 'w';
    }
```

键入命令时任意位置的修改

在readline()函数中，我们从控制台读入一行用户**输入的命令**，保存在char* buf中。用户输入的内容需要**经过处理**才能够被shell识别，因为其中可能包含了**不可见的转义字符与控制字符**，它们用于调整当前的输入位置，使得用户操作更加便捷。

- \033[D: left
- \033[C: right
- \b: backspace
- 0x7f: delete

例如，当用户输入1234后按下左键再输入5时，读取到的实际内容是1234\033[D5，在readline()函数中经过处理，将最终得到的12354**保存在buf**中。与此同时，我们还需要将处理后的结果12354**回显到控制台**，保证用户良好的使用体验。

我们使用r = read(0, &temp, 1)每次读取一个字符并处理，通过维护**实际的光标位置**，对buf进行修改。

```
int length = 0; // actual length
int i = 0;      // cursor position
```

在readline()函数中，定义了两个宏用于**手动移动**控制台光标，使用printf()函数输出控制字符即可。

```
#define MOVELEFT(y) printf("\033[%dD", (y))
#define MOVERIGHT(y) printf("\033[%dC",(y))
```

光标移动

左右键的实际输入由三个字符组成，其中\033对应的ASCII码为27，为**转义字符**，表示后面的字符是控制字符。

由于控制台的光标移动可以**自动完成**，我们只需在readline()函数中维护光标位置i即可。若光标移动超出**左右界限**，需要手动向反方向移动来**抵消控制台的移动**。

```
// readline()
    if (temp == '\033') {
        switch (direction())
        {
            case 3:             // left
                if (i > 0) {
                    i--;
                } else {
                    MOVERIGHT(1);
                }
                break;
            case 4:             // right
                if (i < length) {
                    i++;
                } else {
                    MOVELEFT(1);
                }
                break;
        }
    }
```

对左右键的判断封装在了direction()函数中，在之后我们还会用到这个函数来处理上下键。

```
int direction() {
    int r;
    char temp1, temp2;
    if ((r = read(0, &temp1, 1)) != 1) {
        if (r < 0) {
            debugf("read error: %d\n", r);
        }
        exit();
    }
    if ((r = read(0, &temp2, 1)) != 1) {
        if (r < 0) {
            debugf("read error: %d\n", r);
        }
        exit();
    }
    if (temp1 == '[') {
        switch (temp2) {
            case 'D':   // left:\033[D
                return 3;
            case 'C':   // right:\033[C
```

```
                return 4;
            default:
                return 0;
        }
    }
    return 0;
}
```

删除字符

\b对应的ASCII码为8，为**退格符**; 0x7f对应的ASCII码为127，为**删除符**。均用于删除当前光标位置**前**的字符。

- 光标位于末端
  删除buf中的最后一个字符，将**光标移动至开头**后使用printf()将更新后的buf输出到控制台。

  %s后面的空格用于清除控制台上残留的末尾字符，但需要手动向左移动光标到末端。

```
// readline()
    if (i == length) {  // cursor at the end
        buf[length - 1] = 0;
        MOVELEFT(i);
        printf("%s ", buf);
        MOVELEFT(1);
    }
```

- 光标位于中间
  为了删除光标位置前的字符，我们需要将光标及其以后的字符**整体向左移动**，这样可以覆盖掉光标前的字符。同样输出更新后的buf，随后注意维护控制台中的**光标位置**，使其位于被删除字符的位置。

```
// readline()
    else {              // cursor in the middle
        for (int j = i - 1; j < length - 1; j++) {
            buf[j] = buf[j + 1]; // shift left
        }
        buf[length - 1] = 0;
        MOVELEFT(i);
        printf("%s ", buf);
        MOVELEFT(length - i + 1);
    }
```

插入字符

输入**可见字符**时，若此时光标位于末端，将字符置于buf的末尾即可。

若光标位于中间，将光标及其以后的字符**整体向右移动**，为新字符腾出空间，其插入到光标位置。

输出buf后同样要维护控制台中的**光标位置**，使其位于新插入字符**后**的位置。

```
// readline()
    else {              // visible character
        if (i == length) {  // cursor at the end
            buf[i] = temp;
        } else {            // cursor in the middle
            for (int j = length; j > i; j--) {
                buf[j] = buf[j - 1]; // shift right
            }
            buf[i] = temp;
            buf[length + 1] = 0;
            MOVELEFT(i + 1);
            printf("%s", buf);
            MOVELEFT(length - i);
        }
        i++;
        length++;
    }
```

程序名称中 .b 的省略

在输入时若省略**可执行文件**的.b后缀，会导致在spawn()函数中打开该文件时路径不存在，装载程序失败。

解决的方法是共进行两次**打开判断**，第一次打开失败后，尝试**在路径后追加**".b"再次打开。

```
// spawn()
    if ((fd = open(prog, O_RDONLY)) < 0) {
        int length = strlen(prog);
        char suffix_path[1024];
        strcpy(suffix_path, prog);
        suffix_path[length++] = '.';
        suffix_path[length++] = 'b';
        suffix_path[length] = 0;
        if ((fd = open(suffix_path, O_RDONLY)) < 0) {
            return fd;
        }
    }
```

这样即使在输入时省略了.b后缀，也能够正常打开可执行文件。自此，我们能够向shell中输入**命令而不是文件名**了。

更多命令

以下三个命令属于**外部命令**，在runcmd()函数中，调用spawn()函数创建子进程，装载并执行对应的程序。

因此，我们在user目录下创建了tree.c、mkdir.c、touch.c三个文件，并在include.mk中添加了对应的编译规则。

tree [-adf] [directory]

实现了tree命令的三种常见模式：

- -a: 显示所有文件，包括隐藏文件(默认模式)
- -d：只显示目录文件
- -f: 显示完整路径

实现思路是仿照ls.c的做法遍历目录，若判断文件类型为**目录文件**，递归调用tree()函数，打印该目录下的所有文件。

mkdir [dirname] & touch [filename]

这两个命令的实现极为相似，都是在指定路径下创建一个新的文件，只有文件类型的区别。因此我们只需在**用户库**添加一个create()函数，使其能够创建两种类型的文件。

在此前的文件系统部分，已经实现了file_create()函数，需要将用户态下的create()函数与之对应。以下是create()函数的调用链：

- 用户进程：
  - user/lib/file.c create(): `fsipc_create(path, f_type);`
  - user/lib/fsipc.c fsipc_remove(): `fsipc(FSREQ_CREATE, req, 0, 0);`
  - user/lib/fsipc.c fsipc(): `ipc_send(envs[1].env_id, type, fsreq, PTE_D);`
- 文件系统进程：
  - fs/serv.c serve():

```
req = ipc_recv(&whom, (void *)REQVA, &perm);
switch (req) {
    case FSREQ_CREATE:
        serve_create(whom, (struct Fsreq_create *)REQVA);
        break;
}
```

  - fs/serv.c serve_create(): `file_create(rq->req_path, &f);`

以touch命令的实现为例，首先判断filename是否已经存在，若存在则报错，否则调用create()函数创建一个新文件即可，文件类型为FTYPE_REG。

```
// touch.c
    if (argc == 1) { // no filename
        usage();
    } else {
        for (i = 1; i < argc; i++) {
            if ((r = open(argv[i], O_RDONLY)) >= 0) {
                user_panic("file %s exists", argv[i]);
            }
            if ((r = create(argv[1], FTYPE_REG)) < 0) {
                user_panic("error create file %s: %d\n", argv[i], r);
            }
        }
    }
    return 0;
```

在serve_open()函数中添加创建文件的分支，现在open()函数能够支持文件打开模式O_CREAT，当文件不存在时创建一个新文件。

```
// serve_open()
    if ((rq->req_omode & O_CREAT) != 0) {
        if ((r = file_create(rq->req_path, &f)) < 0) {
            ipc_send(envid, r, 0, 0);
            return;
        }
    }
```

此外，在parsecmd()函数中，优化了**输出重定向**'>'的实现，当文件打开失败时，在该路径创建一个新文件。用户在使用输出重定向时不用预先创建文件，使用更加灵活。

```
// parsecmd()
    if ((fd = open(t, O_WRONLY)) < 0) {
        if ((r = create(t, FTYPE_REG)) < 0) {
            debugf("error create file %s\n", t);
            exit();
        }
        fd = open(t, O_WRONLY);
    }
```

历史命令功能

输出所有历史命令

基本思路是在每次输入命令时将其保存进.history文件中，当用户输入history命令时，从.history文件中读取历史命令并输出。

具体来说，savecmd()函数可分为创建、打开、写入.history文件三个步骤：

```
// savecmd()
    if (!history_init) {
        if ((r = create("/.history", FTYPE_REG)) < 0) {
            debugf("error creating .history\n");
            exit();
        }
        history_init = 1;
    }

    if ((fd = open("/.history", O_WRONLY | O_APPEND)) < 0) {
        debugf("error opening .history: %d\n", r);
        exit();
    }
```

```
    if ((r = write(fd, s, strlen(s))) != strlen(s)) {
        user_panic("error writng .history: %d\n", r);
    }
    write(fd, "\n", 1);
```

注意到open()函数使用了**追加写入模式**O_APPEND，在每次写入时从.history文件的末尾开始，不会覆盖之前已写入的命令。

为了实现这一点，我们需要将文件的**读写指针**移动到文件末尾，随后再写入新的命令。

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

在history.c中，实现了对于.history文件中内容的**按行输出**，并附带了历史命令的序号。

```
// history.c
    if ((fd = open("/.history", O_RDONLY)) < 0) {
            user_panic("error opening .history: %d\n", r);
    }

    while ((r = read(fd, &buf, 1)) == 1) {
        if (newline) {
            printf("%2d  ", line);
            newline = 0;
        }
        printf("%c", buf);

        if (buf == '\n') {
            newline = 1;
            line++;
        }
    }
```

上下键回溯历史命令

在savecmd()函数中，我们维护了**全局变量**linenum和offset，用于记录当前输入的命令的**数量**和**总偏移量**，可以快速在.history中定位对应的命令。

```
// savecmd()
    if (linenum == 0) {
        offset[linenum] = strlen(s) + 1;
    } else {
        offset[linenum] = offset[line - 1] + strlen(s) + 1;
    }
    linenum++;
```

getcmd()函数能够根据命令**序号**，从.history文件中读取对应的命令。

首先读取对应命令前的所有命令至temp中，剩余的内容就是要取出的命令，读取至cmd中。

```
// getcmd()
    if (line != 0) {
        if ((r = readn(fd, temp, offset[line - 1])) != offset[line -
1]) {
            user_panic("error reading .history: %d", r);
        }
        cmdlen = offset[line] - offset[line - 1];
    } else {
        cmdlen = offset[line];
    }

    if ((r = readn(fd, cmd, cmdlen)) != cmdlen) {
        user_panic("error reading .history: %d", r);
    }
    cmd[cmdlen - 1] = 0;
```

上下键的格式与左右键类似，分别为\033[A和\033[B，因此我们仍然在direction()函数中判断上下键的输入。

我们需要在按上下键时维护cmdline变量，使其对应.history中命令**序号**，随后调用getcmd()函数，读取cmdline对应的命令至buf中，并输出到控制台。

为了防止按上键时**当前输入丢失**，我们需要将当前输入的命令保存在curcmd中，当按下键返回到原位时再将其恢复。

```
// readline()
    case 1:             // up
        MOVEDOWN(1);
```

```
        if (cmdline == linenum) {
            buf[length] = 0;
            strcpy(curcmd, buf); // save current command
        }
        if (cmdline > 0) {
            cmdline--;
        } else {
            break;
        }
        getcmd(cmdline, buf);
        if (i > 0) {
            MOVELEFT(i);
        }
        printf("%s", buf);
        break;

    case 2:             // down
        if (cmdline == linenum - 1) {
            strcpy(buf, curcmd); // restore current command
        }
        if (cmdline < linenum) {
            cmdline++;
        } else {
            break;
        }
        getcmd(cmdline, buf);
        if (i > 0) {
            MOVELEFT(i);
        }
        printf("%s", buf);
        break;
```

注意控制台可能有残余的字符，通过输出空格将其覆盖，随后移动控制台中的光标回到末端。

```
// readline()
    if (strlen(buf) < length) {
        for (int j = 0; j < length - strlen(buf); j++) {
            printf(" ");
        }
        MOVELEFT(length - strlen(buf));
    }
```

支持相对路径

在我们的文件系统中，只支持以'/'开头的**绝对路径**，我们希望能够将以"./"开头或省略开头的**相对路径**，作为命令的参数传入shell。

cur_path与系统调用

如果我们想在命令中使用相对路径，需要知道**当前的目录位置**，随后将其与相对路径**拼接**，转换为绝对路径。

在内核中维护了一个**全局变量**cur_path，用于记录当前的目录位置，初始值为**根目录**'/'，并且在kern/syscall_all.c中定义了两个**系统调用**，实现了对于cur_path的**读取**和**修改**。

```
// syscall_all.c
    extern char cur_path[512];

    int sys_get_path(char *buf) {
        strcpy(buf, cur_path);
        return 0;
    }

    int sys_set_path(char *path) {
        if (strlen(path) >= 1024) {
            return -E_MAX_PATH;
        }
        strcpy(cur_path, path);

        return 0;
    }
```

新增库函数

在user/lib/path.c中，新增了三个用户库函数。

chdir()和getcwd()函数分别对应了系统调用sys_set_path()和sys_get_path()，用于**修改**和**读取**当前目录位置。

pathcat()函数将当前目录位置与**相对路径拼接**，转换为**绝对路径**。

```
// path.c
    int chdir(char *path) {
        return syscall_set_path(path);
    }
```

```
int getcwd(char *buf) {
    return syscall_get_path(buf);
}

void pathcat(char *path, const char *suffix) {
    int length = strlen(path);
    if (suffix[0] == '.') {
        suffix += 2;    // skip "./"
    }
    int suf_len = strlen(suffix);

    if (length != 1) {
        path[length++] = '/';
    }
    for (int i = 0; i < suf_len; i++) {
        path[length++] = suffix[i];
    }
    path[length] = 0;
}
```

注意在shell启动时，使用chdir()将cur_path初始化为根目录'/'。

(sh.c的main()函数对应启动过程)

```
if ((r = chdir("/")) < 0) {
    printf("error creating root path: %d\n", r);
}
```

cd & pwd

cd命令属于**内部命令**，因此不应该调用spawn()函数创建子进程执行，而是在直接执行后退出runcmd()函数。

- cd
  当cd命令没有参数时，代表直接返回到**根目录**。

```
if (argc == 1) {
    r = chdir("/");
    printf("back to the root directory\n");
    return;
}
```

- cd /path
  以'/'开头的路径是**绝对路径**，直接将绝对路径传入chdir()函数即可。

```
else if (argv[1][0] == '/') { // absolute path
    strcpy(path, argv[1]);
}
```

- cd path
  对于**相对路径**，需要将当前目录位置与相对路径拼接，转换为**绝对路径**。

```
else {                      // relative path
    getcwd(path);
    pathcat(path, argv[1]);
}
```

在检查路径是否存在以及是否为目录文件后，调用chdir()函数修改当前目录位置。

```
// runcmd()
    if ((r = open(path, O_RDONLY)) < 0) {
        printf("error opening path %s: %d\n", path, r);
        exit();
    }
    close(r);
    struct Stat st;
    if ((r = stat(path, &st)) < 0) {
        user_panic("stat %s: %d", path, r);
    }
    if (!st.st_isdir) {
        printf("path %s is not a directory\n", path);
        exit();
    }

    if ((r = chdir(path)) < 0) {
        printf("cd failed: %d\n", r);
        exit();
    }
    return;
```

pwd命令用于输出当前目录位置，只需调用库中的getcwd()函数读取cur_path，将其输出到控制台即可。

```
// pwd.c
    if ((r = getcwd(buf)) < 0) {
        printf("error getting path: %d\n", r);
        exit();
    }
    printf("%s\n", buf);
```

文件系统的修改

为了支持相对路径，我们需要修改部分文件系统的库函数，使其能够接受**相对路径**作为参数。

修改了user/lib/file.c中的open()函数与create()函数，思路类似。当path以'/'开头时无需处理，否则将当前目录位置与path拼接，转换为**绝对路径**。

```
// open()
    if (path[0] == '/') {
        try(fsipc_open(path, mode, fd));
    } else {
        char abs_path[512];
        getcwd(abs_path);
        pathcat(abs_path, path);
        try(fsipc_open(abs_path, mode, fd));
    }

// create()
    if (path[0] == '/') {
        return fsipc_create(path, f_type);
    }
    char abs_path[512];
    getcwd(abs_path);
    pathcat(abs_path, path);
    return fsipc_create(abs_path, f_type);
```

其它修改

- spawn
  由于修改后的open()函数默认不以'/'开头的路径为**相对路径**，因此在spawn()函数中需要将路径的开头设为'/'，准确对应到**根目录**下的命令文件。

```
// spawn()
    if (prog[0] == '/') {
        strcpy(suffix_path, prog);
    } else {
        suffix_path[0] = '/';
        strcpy(suffix_path + 1, prog);
    }
```

- ls
  将'/'修改为"./": `ls("./", "");`，代表从**当前目录**开始遍历。
- tree
  同样将'/'修改为"./"。

另外，tree命令的-f模式需要输出从根目录开始的**完整路径**，因此需要在tree()函数中将传入的参数转换为**绝对路径**。

```
// tree()
    if (path[0] == '/') {
        strcpy(abs_path, path);
    } else {
        getcwd(abs_path);
        pathcat(abs_path, path);
    }
```