lab6-Shell 挑战性任务

实现相对路径

指令目录优化

基本实现思路

在我们原本实现的 Shell 环境中，所有指令的结构都必须从根目录 / 开始，也就是以绝对路径的形式给出，现在要求我们实现支持以下三个性质的相对指令：

1. 支持以当前目录作为起点的目录（比如当前处在 dir1，我们的指令目录可以以 dir1 作为开头）
2. 需要支持处理 . 和 .. 的跳转，也就是跳转到当前目录和上级父目录的特殊符号

我采用的核心实现思路是，不管是相对目录还是绝对目录，我都先进行预处理，把他变成绝对目录，这样能够最大程度上减少对原有目录支持逻辑的改动

因为涉及到目录的跳转和相对目录变成绝对目录，我在 syscall 中添加了下面两个系统调用，用来支持相关的功能：

```
// 重新设置当前进程所在的绝对目录
int syscall_set_rpath(char *newPath) {
  return msyscall(SYS_set_rpath, newPath);
}

// 获得当前进程所在的绝对目录
int syscall_get_rpath(char *dst) {
  return msyscall(SYS_get_rpath, dst);
}
```

同时，为了方便存储每个进程当前所在的绝对目录，我修改了 Env 结构体，在其中添加了一项 `r_path` 用来表示当前目录，通过这两个系统调用，我们就可以实现：

1. 当解析到相对路径的时候，就调用 `syscall_set_rpath` 获得对应的绝对路径
2. 当成功执行 cd 指令的时候，就调用 `syscall_set_rpath` 对应修改当前进程所在的目录，确保随调用随更新，能够时刻获得正确的当前绝对目录

目录预处理

通过 `path_pre_solve` 函数，我预先解析所有可能会遇到的相对路径，把他们一并转化成为绝对路径

```
int path_pre_solve(char* new_path, char* path)
{
  char cur[1024] = {0};
  getcwd(cur);
  strcpy(new_path, cur);
  // 绝对目录特判
  if (path[0] == '/')
  {
    memset(new_path, 0, sizeof new_path);
    new_path[0] = '/';
  }
  // 优化 ./ 和 ../ 结构
```

```
  for (int i = 0; path[i]; i++)
  {
    if (path[i] == '/')continue;
    else if (path[i] == '.' && (path[i + 1] == '/' || path[i + 1] ==
'\0'))continue;
    else if (path[i] == '.' && path[i + 1] == '.')
    {
      i++;
      struct Stat st;
      // 优化 ../ 前需要特判合法性，防止误删冗余路径
      if (stat(new_path, &st) < 0)return -1;
      del_path(new_path);
    }
    else
    {
      int j = i;
      while (path[j] && path[j] != '/')j++;
      add_path(new_path, path, i, j);
      i = j - 1;
    }
  }
  return 0;
}
```

这里`getcwd()`是对`syscall_get_rpath()`的顶层封装，这样的实现也更贴近真正 OS 的实现风格

这里我的主要处理思路是**路径拼接**和**流式遍历+处理**的思想，首先我获得进程当前所处的目录，存入`new_path`中，然后不断向该路径中拼接有可能得到的相对目录，分为以下两种情况：

1. 如果`path`传入的本来就是绝对路径，也就是以 / 开头，那没必要预处理，直接返回绝对路径即可
2. 如果是相对路径，我们遍历`path`，以 / 作为分隔，在遍历过程中处理 ./ 和 ../ 的情况
    - 如果遇到 ./ 的情况，它对于实际路径的转移没有影响，我们直接**跳过**，不予拼接
    - 如果遇到 ../ 的情况，我们就抵消一段前面已经拼接好的路径，比如我们当前拼接了 dir1 / dir2，现在遇到了一个 ../，我们就删掉之前拼接的 dir2
    - 如果都不是，则说明我们解析到一段正常的路径名称，我们就把他作为目录组成部分拼接到`new_path`中

其中`add_path`和`del_path`分别是添加一段以 / 分隔的路径，和删除一段以 / 分隔的路径（上一段添加的路径），具体实现比较简单，这里不再赘述

注意：这里有个很容易错的点，就是遇到 ../ 删除上一段路径的时候，我们有可能碰巧删除的是一段本来就不存在的路径，比如我们当前只有 dir1 没有 dir2，但是我们的路径是 dir1 / dir2 / .. /，这样 ../ 恰好帮我们把不合法的 dir2 给删掉了，就会导致这条指令出错，所以在每次删除的时候，都需要判断当前的**拼接出来**路径是否合法，只有合法的时候才可以放心删，如果不合法就直接返回 -1 即可，而判断目录存在性的函数`stat`课程组已经实现好了，我们直接调用即可

实现 cd 指令

这里课程组要求 cd 指令实现成内建指令的形式，所以我们在执行完 cd 指令以后直接从`runcmd`中返回即可，不需要调用`spawn`创建子进程，cd 指令的处理分支实现思路如下：

```
if (strcmp("cd", argv[0]) == 0) {
  int r;
  char cur[1024] = { 0 };
  struct Stat st;
```

```
  if (argc > 2) {
    printf("Too many args for cd command\n");
    return;
  }

  if (argc == 1) {
    cur[0] = '/';
  }
  else {
    // 直接调用路径预处理函数即可
    int t = path_pre_solve(cur, argv[1]);
    if (t < 0) {
      printf("cd: The directory '%s' does not exist\n", argv[1]);
      return;
    }
    printf("The cd path is %s to %s\n", argv[1], cur);
  }
  if ((r = stat(cur, &st)) < 0) {
    printf("cd: The directory '%s' does not exist\n", argv[1]);
    return;
  }
  if (!st.st_isdir) {
    printf("cd: '%s' is not a directory\n", argv[1]);
    return;
  }
  if ((r = chdir(cur)) < 0) {
    printf("cd: failed to change directory to '%s'\n", argv[1]);
    return;
  }
  return;
}
```

首先是一些边界特判（参数过多就直接报错，没有参数就默认跳转根目录，设置新目录为 / 即可），触发完特判以后，我们进入 cd 的主要处理流程：

1. 首先调用`path_pre_solve`把待跳转路径处理成绝对路径，同时去掉所有的 ./ 和 ../
2. 接着判断目录的存在性和合法性，仍然使用`stat`函数返回值判断存在性，通过检验 st 的置位符 is_dir 判断是否为目录，只有**合法存在的目录**才能支持跳转
3. 如果目录合法，我们就更新当前进程所在的目录，模拟目录跳转过程，这里用到先前实现的系统调用`syscall_set_rpath`，而`chdir`是对其进行的顶层封装

一定要注意，cd 是内建指令，执行完不管成不成功，都需要直接返回

实现 pwd 指令

如果前面的功能都实现完毕，那么 pwd 指令就很简单了，我们只需要调用`syscall_get_rpath`得到当前所处目录，直接输出即可

基于 pwd 命令的实现，我考虑对 shell 界面进行了优化，也就是在显示 $ 提示符的同时，也显示我们当前所处的目录，这更贴合我们实际使用的 Shell 环境，具体来讲，也就是把形如 pwd 的输出在每次输出命令提示的时候都输出一遍即可：

```
if (interactive) {
  char curPath[1024] = {0};
  if ((r = getcwd(curPath)) < 0) { printf("G");exit(); }
  printf("\n[%d] %s $ ", syscall_getenvid(), curPath);
}
```

实现环境变量

具体来讲，就是要求我们实现两类变量，一类全局变量，所有进程都可以访问，A 进程通过 sh 创建的子进程 B 也可以访问；另一类是局部变量，A 进程通过 sh 创建了子进程 B，那么 A 无法访问 B 的局部变量，B 也无法访问 A 的局部变量

局部变量

先说比较难实现的局部变量，这里很多学长实现的思路都是通过父子进程之间的共享内存来实现的，我个人采用了一种更简单的实现方式，也就是在每个 sh 进程内部都维护一个`localVar`数组，存放**这个 sh 进程**所创建的所有的局部变量，因为`localVar`仅定义在当前进程内部，那么不同进程之间自然就无法相互访问了

要支持这样的功能，我们就必须把所有和**环境变量**有关的操作都设置成内建指令，因为如果不是内建指令的话，需要设置局部变量的那个 sh 进程就会又额外创建一个子进程去执行设置局部变量的指令，而此时就发生了进程切换，要想让后者设置前者的局部变量，我们就又绕回了学长父子进程之间共享内存的老路，所以这里我的实现思路就是直接让父进程自己内建地完成和**环境变量**设置有关的所有指令，不进行任何进程切换

全局变量

全局变量的实现就相对容易，我们只需要找一个**所有进程**都可以访问到的地方，存一个`globalVar`的数组即可，我选择设在 syscall_all.c 文件中，让所有进程通过系统调用来设置或访问全局变量，因此这里又需要增加一个系统调用：

```
// 进程编号为 id 的进程，以 mode 对应的操作形式(如果是读写类型操作还需要考虑 rwmode)，去 globalVar 中按照 name 索引到对应的全局变量，做 val 的值操作
int syscall_getGlobalVar(char* name, char* val,  int mode, int id, int rwMode)
{
  return msyscall(SYS_getGlobalVar, name, val, mode, id, rwMode);
}
```

这里为了统一和环境变量有关的**初始化，修改，删除**操作，我只设计了一个系统调用`syscall_getGlobalVar`，而具体的操作类型由 mode 参数来加以区分，这样可以一定程度上简化环境变量的代码体量

核心实现流程

局部变量操作

在`getLocalVar`函数中，我实现了对局部变量操作的解析，主要是根据 mode 和 rmMode 完成对操作类型的解析（初始化，创建，修改，删除）

```
int getLocalVar(char* name, char* val, int mode, int rwMode) {
  int i;
  int flag = 0;
  // 查找合法变量
```

```
  if (mode == 0) {
    for (i = 0; i < 100; i++) {
      if (localVar[i].mode != 0 && strcmp(name, localVar[i].name) == 0) {
        flag = 1;
        break;
      }
    }
    if (flag == 1) {
      return 0;
    } else {
      return -1;
    }
    // 初始化创建变量
  } else if (mode == 1) {
    for (i = 0; i < 100; i++) {
      if (localVar[i].mode != 0 && strcmp(name, localVar[i].name) == 0) {
        if (localVar[i].mode == 1) {
          debugf("%s is read only\n", name);
          return -1;
        }
        strcpy(localVar[i].value, val);
        localVar[i].mode = rwMode > 0 ? 1 : 2;
        return 0;
      }
    }
    // 带权限修改变量
  } else if (mode == 2) {
    for (i = 0; i < 100; i++) {
      if (localVar[i].mode != 0 && strcmp(name, localVar[i].name) == 0) {
        if (localVar[i].mode == 1) {
          debugf("%s is read only\n", name);
          return -2;
        }
        localVar[i].mode = 0;
        return 0;
      }
    }
    // 获取环境变量的值
  } else if (mode == 3) {
    for (i = 0; i < 100; i++) {
      if (localVar[i].mode != 0 && strcmp(name, localVar[i].name) == 0) {
        strcpy(val, localVar[i].value);
        return 0;
      }
    }
    // 新加环境变量
  } else if (mode == 4) {
    for (i = 0; i < 100; i++) {
      if (localVar[i].mode == 0) {
        localVar[i].mode = rwMode > 0 ? 1 : 2;
        strcpy(localVar[i].value, val);
        strcpy(localVar[i].name, name);
        return 0;
      }
    }
    // 强制删除变量
  } else if (mode == 5) {
    for (i = 0; i < 100; i++) {
```

```
      if (localVar[i].mode != 0 && strcmp(name, localVar[i].name) == 0) {
        localVar[i].mode = 0;
        return 0;
      }
    }
  }
  return -1;
}
```

其中不同的 mode 分别对应不同的操作格式，具体设计如下（目前只支持以下五种操作，后续仍可进行功能扩展）：

1. mode == 0：检查是否存在名为 name 且**未被删除**的局部变量（核心功能其实就是 mode == 0 对应的查找过程）
    - loclaVar 数组权限位的含义：0 - 无效 / 已经被删除，1 - 只读变量，2 - 可写变量
2. mode == 1：修改已有环境变量的值和权限位，先进行查找，然后进行值 val 的复制，最后根据 rwMode 更新权限位
    - 这里 rwMode 的设计思路是：rwMode > 1 表示设置为只读 1，rwMode == 0 表示设置为可写 2
3. mode == 2：删除环境变量，注意这里删除前需要检查不是**只读**变量，因为只读变量是不允许删除的
4. mode == 3：获取环境变量的值
5. mode == 4：创建新的环境变量，在 localVar 数组中找到一个空位置，复制当前环境变量的信息进去即可
6. mode == 5：强制删除环境变量，无视任何权限，mode == 2 的加强版本，类似于 -f 的强制删除，只要能查找到就无条件立刻删除

全局变量操作

和`getLocalVar`完全是对偶函数，只不过前者是在进程内部实现，而`sys_getGlobalVar`是在全局实现的系统调用函数，前者用来实现对局部变量的操作，后者用来实现对全局变量的操作，前者操作的是`localVar`数组，后者操作的是`globalVar`数组

至于其余内容，该函数对 mode 的解析和`getLocalVar`完全相同，具体逻辑不再赘述

```
int sys_getGlobalVar(char* name, char* val, int mode, int id, int rwMode) {
  static int initFlag = 1;
  int i, j;
  int flag = 0;

  if (initFlag == 1) {
    initFlag++;
    for (i = 0; i < 7; i++) {
      for (j = 0; j < 100; j++) {
        globalVar[i][j].mode = 0;
      }
    }
  }

  if (mode == 0) {
    for (i = 0; i < 100; i++) {
      if (globalVar[id][i].mode != 0 && strcmp(name, globalVar[id][i].name)
== 0) {
        flag = 1;
```

```
        break;
      }
    }
    if (flag == 1) {
      return 0;
    } else {
      return -1;
    }
  } else if (mode == 1) {
    for (i = 0; i < 100; i++) {
      if (globalVar[id][i].mode != 0 && strcmp(name, globalVar[id][i].name)
== 0) {
        if (globalVar[id][i].mode == 1) {
          return -1;
        }
        globalVar[id][i].mode = rwMode > 0 ? 1 : 2;
        strcpy(globalVar[id][i].value, val);
        return 0;
      }
    }
  } else if (mode == 2) {
    for (i = 0; i < 100; i++) {
      if (globalVar[id][i].mode != 0 && strcmp(name, globalVar[id][i].name)
== 0) {
        if (globalVar[id][i].mode == 1) {
          return -2;
        }
        globalVar[id][i].mode = 0;
        return 0;
      }
    }
    return -1;
  } else if (mode == 3) {
    for (i = 0; i < 100; i++) {
      if (globalVar[id][i].mode != 0 && strcmp(name, globalVar[id][i].name)
== 0) {
        strcpy(val, globalVar[id][i].value);
        return 0;
      }
    }
  } else if (mode == 4) {
    for (i = 0; i < 100; i++) {
      if (globalVar[id][i].mode == 0) {
        globalVar[id][i].mode = rwMode > 0 ? 1 : 2;
        strcpy(globalVar[id][i].value, val);
        strcpy(globalVar[id][i].name, name);
        return 0;
      }
    }
  } else if (mode == 5) {
    if (id == 0) {
      return 0;
    }
    for (i = 0; i < 100; i++) {
      if (globalVar[id - 1][i].mode != 0) {
        globalVar[id][i].mode = globalVar[id - 1][i].mode;
        strcpy(globalVar[id][i].value, globalVar[id - 1][i].value);
        strcpy(globalVar[id][i].name, globalVar[id - 1][i].name);
```

```
      }
    }
    return 0;
  } else if (mode == 6) {
    int curVar = rwMode;
    if (curVar >= 100 || globalVar[id][curVar].mode == 0) {
      return -1;
    }
    strcpy(name, globalVar[id][curVar].name);
    strcpy(val, globalVar[id][curVar].value);
    return 0;
  } else if (mode == 7) {
    for (i = 0; i < 100; i++) {
      globalVar[id][i].mode = 0;
    }
    return 0;
  }
  return -1;
}
```

由于全局变量通过系统调用实现，所以我们真正调用的其实是用户态下的`syscall_getGlobalVar`接口，也就是系统调用的顶层接口，由于全局变量和局部变量的操作相同，这两个函数的实现也是相互对偶的

遍历展示函数

当 declare 不加参数的时候，需要打印当前进程所能访问到的所有环境变量，包括全局和局部，通过调用`showVar`函数实现

```
void showVar() {
  int i;
  int r;
  char name[16] = {0};
  char val[16] = {0};
  // global first
  for (i = 0; i < 100; i++) {
    r = syscall_getGlobalVar(name, val, 6, shellId, i);
    if (r == 0) {
      printf("%s=%s\n", name, val);
    }
  }
  // local next
  for (i = 0; i < 100; i++) {
    if (localVar[i].mode != 0) {
      printf("%s=%s\n", localVar[i].name, localVar[i].value);
    }
  }
}
```

依次遍历所有的全局变量数组和局部变量数组即可，如果数组对应位置有效，那么就说明是一个有效的环境变量，按课程组要求进行打印即可

declare 函数

declare 命令的具体实现，在该函数中完成，该函数的核心功能是对 declare 进行 name 和 val 的拆分，把环境变量名称和值进行分别映射，并且根据局部和全局变量的不同类型，完成变量在进程中真正的记忆过程

```
void declare(char* nameAndVal, int mode) {
  u_int x = mode & 0x1;
  u_int READONLY = mode & 0x2;
  char name[MAXENVLEN] = {0};
  char val[MAXENVLEN] = {0};
  int i, j;
  int r;
  if (mode == -1) {
    showVar();
    return;
  }
  for (i = 0; nameAndVal[i] != '=' && nameAndVal[i] != 0; i++){
    name[i] = nameAndVal[i];
  }
  if (nameAndVal[i]) {
    for (j = 0, i++; nameAndVal[i]; i++, j++){
      val[j] = nameAndVal[i];
    }
  }
  // 存在权限 -x 的情况
  if (x) {
    r = getLoaclVar(name, val, 0, 0);
    printf("%d\n", r);
    if (r == 0) {
      r = syscall_getGlobalVar(name, val, 0, shellId, 0);
      if (r == 0) {
        r = syscall_getGlobalVar(name, val, 1, shellId, READONLY);
        if (r) return;
        r = getLoaclVar(name, val, 5, 0);
        if (r) return;
      } else {
        r = syscall_getGlobalVar(name, val, 4, shellId, READONLY);
        if (r) return;
        r = getLoaclVar(name, val, 5, 0);
        if (r) return;
      }
    } else {
      r = syscall_getGlobalVar(name, val, 0, shellId, 0);
      if (r == 0) {
        r = syscall_getGlobalVar(name, val, 1, shellId, READONLY);
        if (r) return;
        r = getLoaclVar(name, val, 5, 0);
      } else {
        r = syscall_getGlobalVar(name, val, 4, shellId, READONLY);
        if (r) return;
      }
    }
    // 没有权限 -x 的情况
  } else {
    r = syscall_getGlobalVar(name, val, 0, shellId, 0);
    if (r == 0) {
```

```
      r = syscall_getGlobalVar(name, val, 1, shellId, READONLY);
      if (r) return;
    } else {
      r = getLoaclVar(name, val, 0, READONLY);
      if (r == 0) {
        r = getLoaclVar(name, val, 1, READONLY);
        if (r) return;
      } else {
        r = getLoaclVar(name, val, 4, READONLY);
        if (r) return;
      }
    }
  }
}
```

declare 函数流程解析：

1. 首先进行 name 和 val 的拆分，便于后边子函数的传参
2. 处理 `mode & 0x1`（强制更新变量），先检查本地变量是否存在：
   - 如果存在，检查全局变量是否存在，如果全局变量存在，更新全局变量，删除本地变；如果全局变量不存在，创建全局变量，删除本地变量
   - 如果本地变量不存在，检查全局变量是否存在，如果全局变量存在，更新全局变量，删除本地变量，如果全局变量不存在，创建全局变量
3. 处理 `!(mode & 0x1)`（非强制更新），先检查全局变量是否存在：
   - 如果全局变量存在，更新全局变量
   - 如果全局变量不存在，检查本地变量是否存在，如果本地变量存在，更新本地变量，如果本地变量不存在，创建本地变量

这里实现的逻辑稍微有一点绕，尤其是`syscall_getGlobalVar`在调用的时候，注意 mode 参数不要传错了，不然 debug 能找半天

declare-main 函数

该函数是对 declare 函数的封装，之所以设计又在顶层设计这个函数，主要原因是在实现完 declare 函数以后，发现 decalre 指令也会有参数（比如 -x），所以我们还需要设计一个参数解析和处理的流程，为了保证 declare 的简洁性和美观性，我把**命令参数解析**和具体封装交给更高一级的函数中实现，于是就出现了下面这个函数

```
void declareMain(int argc, char **argv) {
  int i;
  int flag[256] = {0};
  int mode = 0;
  ARGBEGIN
  {
    default:
    usage();
    case 'x':
    case 'r':
    flag[(u_char) ARGC()]++;
    break;
  }
  ARGEND
    if (flag['x']) {
      mode += 1;
    }
```

```
  if (flag['r']) {
    mode += 2;
  }
  if (argc == 0) {
    declare(0, -1);
    return;
  }
  for (i = 0; i < argc; i++){
    declare(argv[i], mode);
  }
}
```

该函数逻辑上没什么好说的，很经典的两步走，非常经典的逻辑：

1. 首先根据参数设置 flag 标志，进而设置操作 mode 的值，做好传参准备
2. 接着根据命令中的参数个数，选择是要打印所有现有变量`showVar`分支，还是依次设置命令中的环境变量`declare`分支

unset 函数

用于实现对某个环境变量删除的函数，本质上也是对`syscall_getGlobalVar`和`getLocalVar`的调用，如果已经实现了 declare，这个函数完全没有难度，仿照 declare 的实现即可

```
void unset(char* name) {
  int r;
  int i;
  r = syscall_getGlobalVar(name, 0, 2, shellId, 0);
  if (r == 0) {
    return;
  } else if (r == -2) {
    printf(" unset: '%s': cannot unset: readonly variable", name);
    return;
  }
  r = getLoaclVar(name, 0, 2, 0);
  if (r == -2) {
    printf(" unset: '%s': cannot unset: readonly variable", name);
    return;
  } else if (r == 0) {
    return;
  }
}
```

唯一需要注意的是，这里全局变量的优先级是高于局部变量的，如果有全局变量存在，我们要先删全局变量，再考虑去删局部变量

unset-main 函数

仿照 decalre 函数，我们为 unset 函数也写一个顶层封装，用来实现对指令参数的解析

```
void unsetMain(int argc, char **argv) {
  int i;
  int flag[256] = {0};
  ARGBEGIN
  {
    default:
    usage();
```

```
    case 'v':
    flag[(u_char) ARGC()]++;
    break;
  }
  ARGEND
    if (argc == 0) {
      printf("unset need a name of var\n");
      return;
    }
  for (i = 0; i < argc; i++){
    unset(argv[i]);
  }
}
```

这里需要解析的只有一个 -v 参数，以及空参数列表直接报错返回，其余情况直接调用 unset 执行删除命令即可

主执行流程

与 cd 指令同样需要注意的是，我们一开始就约定把 declare 设置成为内建指令，因此有关分支在执行完成后都需要直接 return，不需要另行创建子进程

```
if (strcmp(argv[0], "declare") == 0) {
  declareMain(argc, argv);
  return;
}
if (strcmp(argv[0], "unset") == 0) {
  unsetMain(argc, argv);
  return;
}
```

易错点总结

一个严重的错误

这么写看似我们所有的功能都实现了，但是交上去却过不了评测，其实我们在实现全局变量的时候还忽略了一个**重要**的要求，那就是子进程对于父进程全局变量的修改是作为副本修改的，也就是父进程的全局变量不会受到子进程修改的影响，子进程的修改仅保留在他本身

对于这个 bug 的修复，我考虑把 globalVar 开成二维数组，第一维存每个进程的进程号，也就是多传一个参数 shellId 进去，如果子进程要修改父进程中已经有的全局变量，我们就在 shellId(子) 对应的一维数组中先拷贝 shellId(父) 对应变量的副本，然后对这个副本进行修改，这样就实现了子进程对父进程影响的不可见性

那么有了 shellId 这个参数之后，我们就需要考虑父子进程之间进程编号的维护，我们显然不能用进程号本身来作为索引（主要是因为原始进程号过大，而离散化又不容易实现），我们考虑父进程 sh 对应的 shellId == 1，父进程每创建一个子进程，传给子进程的 shellId+1，子进程每退出一层回到父进程，父进程取出的 shellId-1，这样就能够实现父子进程之间进程的**步进式传递**

那么这个传递机制该如何实现呢？我们先考虑一个很熟悉的场景，假设我有一个父函数 sh 想调用递归调用它自己作为子函数 sh，此时 shellId 直接通过参数列表就可以实现上文的传递，我们可以理解成父函数把 shellId+1 放在栈空间，子函数去拿，而对于进程之间，这段共享区间我们很容易想到用一个**共享文件**来实现，也就是我们所创建在根目录下的 shellId 文件

实现思路

也就是说，父进程调用子进程，子进程回退父进程之前，都会把自己的 shellId 做相应更新以后写回这个共享文件（父进程写自己的 shellId+1，子进程写自己的 shellId-1），而每进入一个新的进程，他都要从上一级进程更新过的 shellId 文件中拿出对应的 shellId 作为自己当前的进程 Id

下面我们将详细展示这个机制的代码实现

进程 Id 获取

每个进程在被创建出来的时候，首先会调用 `getShellId` 得到来到他的那个进程所传给他的进程号

```
void getShellId() {
  int fd, n;
  int r;
  int i;
  char tmp[12] = {0};
  // 打开文件
  if ((fd = open("/shellId", O_RDWR | O_CREAT)) < 0) {
    printf("open %s: error in getShellId\n", fd);
    return;
  }
  // 修改偏移量为文件开头
  if ((r = seek(fd, 0)) < 0) {
    printf("error in seek getShellId\n");
    return;
  }
  // 读取文件
  n = read(fd, tmp, (long)sizeof tmp);
  int ans = 0;
  for (i = 0; tmp[i]; i++) {
    ans = ans * 10 + tmp[i] - '0';
  }
  close(fd);
  shellId = ans;
}
```

核心实现思路有两点：

1. 如果是第一次进程切换，有可能 shellId 文件还不存在，因此 open 失败的时候需要接着 create
2. 我们维护的 shellId 是数值，但是存入文件要是字符串形式，这里涉及到整型和字符串之间的转换，比较简单，直接参考上面代码即可

进程 Id 的更新写回

父进程调用子进程之前，要把自己的 shellId+1 后存入 shellId 文件；子进程退出到父进程之前，要把自己的 shellId-1 后存入 shellId 文件，二者都是覆盖式进行写入的，也就是说 shellId 文件任意时刻都只存储一个 shellId 所对应的字符串

```
void addShellIdToFile(int type) {
  int fd, n;
  int r;
  int i, j;
  int ans = shellId + type;
  printf("%d\n", ans);
  char tmp[12] = {0};
  i = 0;
  ans = ans < 0 ? 0 : ans;
  while (ans) {
```

```
    tmp[i++] = ans % 10 + '0';
    ans /= 10;
  }
  for (j = i - 1, i = 0; i < j; i++, j--){
    char c = tmp[i];
    tmp[i] = tmp[j];
    tmp[j] = c;
  }
  if ((fd = open("/shellId", O_RDWR)) < 0)
    printf("open %s: error in addShellIdToFile\n", fd);
  if ((r = seek(fd, 0)) < 0) {
    printf("error in seek addShellIdToFile\n");
  }
  r = write(fd, tmp, strlen(tmp));
  if (r < 0) {
    printf("error in writeback in addShellIdToFile\n");
  }
  close(fd);
}
```

至此，我们才算完全完成了**环境变量**有关的全部内容，这部分还是比较综合的一部分，涉及到进程，系统调用和文件操作，对应需要添加的代码量也比较大，算得上是整个 shell 中比较复杂的一环了

输入指令优化

这部分任务可以分成两部分，快捷键支持和指令格式的优化，这部分实现起来是相对比较简单的一部分：

1. 快捷键支持：支持上下左右箭头，5 个 ctrl 快捷键
2. 支持反引号，历史指令，一行多指令，不带后缀等指令格式的优化

下面也会分成这两大块进行讲解

方向键和快捷键

首先需要知道快捷键和方向键在终端会被解析成什么，这里问 chat 或参考往届学长都有说明，对于方向键和快捷键的修改主要集中在 readline 中

首先我们需要改一下 readline 的格式，让他一个一个字符的来读，读到的字符暂时存在 temp 中，根据这个字符到底是不是有效字符，来判断到底要不要把他真正加入到存储命令行的 buf 数组中

此外这里为了支持方向键带来的**光标**移动，我们还需要区分两个概念：

1. 变量 i：光标的位置，比如一条指令为 cd dir1/dir2，光标可以在 0 位置不超过指令长度 len 对应的位置之间的任意位置
2. 变量 len：回显的指令的实际长度

方向键

经过相关了解，上下左右四个箭头会被解析成 '\033' '[' 'A / B / C / D' 四个部分，因为会被解析成是那个字符，且不存在满足部分但不满足整体的情况，所以我们可以用**状态机**思想进行解析，也就是依次解析这三个字符，因为他们的区别无非是上下左右箭头的区别，一定不可能被解析成别的，具体代码示例如下：

```
case '\x1b': // first
read(0, &temp, 1);
if (temp == '[') { // second
```

```
  read(0, &temp, 1);
  if (temp == 'D') { // left
    if (i > 0) {
      i -= 1;
    } else {
      printf("\033[C");
    }
  } else if (temp == 'C') { // right
    if (i < len) {
      i += 1;
    } else {
      printf("\b");
    }
  }
```

左右箭头

这两个方向键的处理机制是相同的，他们不涉及到指令的修改后回显，只需要改变光标的位置，如果光标在指令的中间位置，我们不需要做任何事情，直接让光标改变就行，我们需要特判的是边界位置：

1. 如果光标在 0 位置，但又输入了一个左箭头，我们应该**人为输出**一个右箭头用来抵消这个左箭头，防止光标又向左跑到 $ 位置了
2. 如果光标在 len 位置，但又输入了一个右箭头，我们应该**人为输出**一个左箭头来抵消这个右箭头，防止光标再向右超过 len 位置

上下箭头

这两个方向键的处理比较复杂，需要结合 history 指令一起处理，我们留到 history 指令处再进行详解

快捷键 & 退格键

经过查阅，所有 ctrl+X 的快捷键都有对应的 ASCII 码，我们只需要按照对应的 ASCII 码来解析即可，五大快捷键的处理思路都差不多，只需要做好以下**三个**核心功能：

1. 搞对光标最终停留的位置
2. 实现指令回显（旧指令擦除和新指令重新输出）
3. 修改后指令在 buf 中同步修改

我们在这里给出快捷键实现和退格键实现的完整代码：

```
case 0x7f: // 退格键
  if (i <= 0) { break; }
  for (int j = (--i); j <= len - 1; j++) {
    buf[j] = buf[j + 1];
  }
  buf[--len] = 0;
  printf("\033[%dD%s \033[%dD", (i + 1), buf, (len - i + 1));
  break;
case 0x05: // ctrl+E 0x05
  for (int j = 0; j < i; j++)printf("\b");
  printf("%s", buf);
  i = len;
  break;
case 0x01: // ctrl+A 0x01
  for (int j = 0; j < i; j++)printf("\b");
  i = 0;
  break;
```

```
case 0x0B: // ctrl+K 0x0B
  for (int j = i; j < len; j++)printf(" ");
  for (int j = i; j < len; j++)printf("\b");
  for (int j = i; j < len; j++)buf[j] = '\0';
  len = strlen(buf);
  break;
case 0x15: // ctrl+U 0x15
  char tmp[1024] = {0};
  for (int j = i, t = 0; j < len; j++, t++)tmp[t] = buf[j];
  for (int j = 0; j < i; j++) { printf("\b"); }
  for (int j = 0; j < len; j++) { printf(" "); }
  for (int j = 0; j < len; j++) { printf("\b"); }
  printf("%s", tmp);
  len = strlen(tmp);
  for (int j = 0; j < len; j++) { printf("\b"); }
  i = 0;
  strcpy(buf, tmp);
  break;
case 0x17: // ctrl+W 0x17
  len = strlen(buf);
  int j = i;
  while(j > 0 && buf[j - 1] == ' ')j--;
  if (j > 0) {
    if (buf[j] == ' ')j--;
    while(j > 0 && buf[j] != ' ')j--;
    if (buf[j] == ' ')j++;
  }
  for (int t = j; t < i; t++)printf("\b");
  for (int t = j; t < len; t++)printf(" ");
  for (int t = j; t < len; t++)printf("\b");
  for (int t = i; t < len; t++)printf("%c", buf[t]);
  for (int t = i; t < len; t++)printf("\b");
  memset(tmp, 0, sizeof tmp);
  for (int t = i, k = 0; t < len; t++, k++)tmp[k] = buf[t];
  int t = 0;
  for (int k = j; tmp[t]; t++, k++)buf[k] = tmp[t];
  buf[j + t] = '\0';
  len = strlen(buf);
  i = j;
  break;
```

对应快捷键的真实功能可以在本地 VmWare 虚拟机上进行对照，OS 网站和 vscode 中似乎有保留快捷键，导致对应快捷键的功能无法实现，同时在提交的过程中，发现似乎五个快捷键的功能并没有被测到，不知道是不是测试点里漏写了

这里选择比较复杂的 ctrl+W 为例讲解快捷键的实现思路，其他的实现都是大同小异：

ctrl+W

该快捷键用来删除从光标当前位置向左所能找到的第一个非空格连续的字符串（如果找不到就不删），并且连带删除在向左查找过程中所遇到的空格，删除以后光标和指令相对位置保留，光标后的指令向前回填对齐

所以我们要做的就是以**当前光标位置 i** 为基准，向前寻找，找到第一个非空字符串的终点，再向前寻找找到这个非空字符串的起点，记录起点位置为 j，接下来演示指令的**回显**：

1. 首先从 i 到 j 位置，连续输出 \033[D，模拟光标的回退，让光标从 i 位置回到 j 位置

2. 接着从 j 位置到 len 位置，连续输出空格，模拟字符串的清空，此时控制台中只剩下 j 位置之前的内容
3. 接着再从 len 位置到 j 位置，连续输出 \033[D，模拟光标的回退，使得光标再次回到 j 位置
4. 接着从当前光标位置开始输出 buf 数组中从 i 下标到 len-1 下标的内容，模拟 i 位置后面内容的前移对齐
5. 最后从新指令 len 位置到 j 位置，连续输出 \033[D，模拟光标的回退，使得光标保持在 j 位置，和 shell 指令行为一致
6. 最后要在 buf 中同步删掉 j 到 i 位置的内容，因为我们不仅要保证回显到终端的指令被修改，buf 中的指令也要同步被修改，不然我们执行的就可能是一条错误的指令

其余快捷键指令的实现与 ctrl+W 大同小异，不再赘述，这一部分比较麻烦的地方就是光标回退和输出回显时候的对齐，多调一调应该问题不大，其中我们需要**注意**的就是反复调试，或者手动模拟，使得**回显**能够顺利进行

历史指令

该任务要求我们记录曾经成功执行（按过回车）的最多 20 条指令，并且能够通过上下箭头完成指令的滚转，通过 history 指令实现历史指令的输出

基本思路

这里我仍然考虑把 history 设置成内建指令，只需要在 sh 进程中开一个`all_lines`数组用来存储曾经执行过的数组即可，所需要的基础数据结构如下：

```
int all_line_count; // 表示指令总数量，上限20
int now_line_index; // 表示光标当前所在指令索引
char all_lines[25][1025]; // 存储历史指令的数组
```

具体实现

指令存储

每当我们成功获得一条指令，也就是 readline 得到回车的时候，buf 中就得到一条完整的指令，此时我们调用`save_line`函数，把 buf 中的指令存入 all_lines 中

```
void save_line(char* line) {
  int fd;
  if ((fd = open("/.mosh_history", O_TRUNC | O_WRONLY)) < 0) {
    all_line_count = 0;
    now_line_index = 0;
    if ((fd = open("/.mosh_history", O_WRONLY | O_TRUNC | O_CREAT)) < 0) {
      user_panic("open ./mosh_history: %d", fd);
    }
  }
  // 注意上限 20 条，不要自讨没趣
  if (all_line_count >= 20) {
    all_line_count = 20;
    for (int i = 0; i < 19; i++) {
      strcpy(all_lines[i], all_lines[i + 1]);
    }
    strcpy(all_lines[all_line_count - 1], line);
  }
  else {
    strcpy(all_lines[all_line_count], line);
    all_line_count++;
```

```
  }
  // 更新当前标签和历史最大标签相同
  now_line_index = all_line_count;
  int len = strlen(all_lines[now_line_index - 1]);
  all_lines[now_line_index - 1][len++] = '\n';
  all_lines[now_line_index - 1][len] = '\0';
  for (int i = 0; i < all_line_count; ++i) {
    len = strlen(all_lines[i]);
    write(fd, all_lines[i], len);
  }
  close(fd);
}
```

这里需要注意我们采用的是**头插法**的思想，对于数组来说就需要模拟 O(N) 的复杂度，当然因为最大上限不超过 20 条，时间复杂度完全说得过去，需要注意以下两个细节：

1. 每次插入一条新指令以后，需要同步更新`now_line_index`和`all_line_index`相同，确保我们下次回滚到的第一条指令是这次执行完的指令（这里曾经卡过很久）
2. 注意最多 20 条的上限，如果数量超了，我们牺牲的应该是 19 下标位置的指令，因为我们用的是头插法

回滚操作

这里我们结合上下箭头来进行讲解，当已经给出了一条指令 xxx 以后，我们如果要开始回滚，那么我们的指令就分成了两部分：

1. history 中的指令
2. 当前终端这条 xxx 指令（这条指令不能丢）

因此我们需要一个额外的 buf 来存储 xxx 这条指令，确保我们滚上去后再回来的时候，还能找到这条指令，这个 buf 我们命名为`now_cmd_buf, buf`

因此我们回滚的具体流程就是，如果是向上滚且不是第一条历史指令，我们就把对应的 now_line_index --，然后使用`getPast`索引 now_line_index 位置的指令输出；如果是向下且不是当前指令，我们就把 now_line_index++，然后使用`getPast`索引 now_line_index 位置的指令输出，而如果是当前指令，我们就输出 now_cmd_buf 中存储的内容

```
else if (temp == 'A') { // up
  // 这里很重要：需要抵消一个向上的箭头，不然光标会到处乱跑
  printf("\033[B");
  if (now_line_index != 0) {
    buf[len] = '\0';
    if (now_line_index == all_line_count) {
      strcpy(now_cmd_buf, buf);
    }
    if ((r = readPast(--now_line_index, buf)) < 0) { break; }
    sig++;
    for (int j = 0; j < i; j++) { printf("\b"); }
    for (int j = 0; j < len; j++) { printf(" "); }
    for (int j = 0; j < len; j++) { printf("\b"); }
    // 回显
    printf("%s", buf);
    i = strlen(buf);
    len = i;
  }
} else if (temp == 'B') { // down
  buf[len] = '\0';
```

```
  if (1 + now_line_index < all_line_count) {
    if ((r = readPast(++now_line_index, buf)) < 0) { break; }
  } else {
    if (sig == 0)break;
    strcpy(buf, now_cmd_buf);
    now_line_index = all_line_count;
    sig = 0;
  }
  for (int j = 0; j < i; j++) { printf("\b"); }
  for (int j = 0; j < len; j++) { printf(" "); }
  for (int j = 0; j < len; j++) { printf("\b"); }
  // 回显
  printf("%s", buf);
  i = strlen(buf);
  len = i;
}
}
break;
```

在具体实现过程中，我们还需要注意一些特判边界，比如上滚和下滚到边界的时候需要停住，不能无限向上或无限向下；以及切换到一条新指令以后，原有的 now_cmd_buf 需要及时清空，防止误判当前指令等，具体特判在代码中已经给出，这里不再赘述

history 列举指令

只需要逐行输出 `history` 文件中的内容即可，大一数据结构中的内容，我甚至没有封装成函数，唯一需要注意的是，history 在这里也是内建指令，因此执行完以后应该直接 return

```
if (strcmp(argv[0], "history") == 0) {
  int fd, r, n;
  char hist_buf[1025];
  if (argc > 1) {
    return;
  }
  if ((fd = open("/.mosh_history", O_RDONLY)) < 0) {
    printf("error in history.b\n");
    exit();
  }
  while ((n = read(fd, hist_buf, 1024)) > 0) {
    hist_buf[n] = '\0';
    printf("%s", hist_buf);
  }
  close(fd);
  return;
}
```

指令优化

这里我的基本思路是对指令特殊格式的解析都**尽量**阻断在指令语法层面，不让其进入语义层面，也就是通过预处理处理，把特殊格式的指令变回普通格式的指令，因为我们已经实现好了普通指令的接口，所以只需要让指令适合这个接口就行，没必要真的要改内部指令的实现逻辑

一行多指令输入

会给出形如`ls;ls;ls;cd dir1;ls`这样多个指令在一行中，由分号分开的输入形式，要求我们支持这样的指令

其实实现很简单，只需要写一个`while(check)`结构，解析当前指令中是否还有**分号**存在，如果有，就按照分号进行分割，切割出来的第一个子命令一定不含分号，正常执行，然后剩下的命令再进入`while(check)`，不断完成子命令的解析，对应函数如下：

```
while(parseFenHao(tmp_buf, buf) > 0){...}

// 看函数名就知道函数作用了
int parseFenHao(char *tmp_buf, char *buf)
{
  if (strlen(tmp_buf) == 0)return 0;
  int len = strlen(tmp_buf);
  for (int i = 0; i < len; i++)
  {
    if (tmp_buf[i] == ';')
    {
      for (int j = 0; j < i; j++)
      {
        buf[j] = tmp_buf[j];
      }
      buf[i] = '\0';
      for (int j = i + 1, t = 0; j < len; j++, t++)
      {
        tmp_buf[t] = tmp_buf[j];
      }
      tmp_buf[len - i - 1] = '\0';
      return 1;
    }
  }
  strcpy(buf, tmp_buf);
  for (int i = 0; tmp_buf[i]; i++)tmp_buf[i] = '\0';
  return 1;
}
```

注意：这里尽管有分号，但是还是要把他们当一条指令，也就是 history 显示的时候，需要当成一条指令来显示，也就是说，我们 history 用来 save_line 的操作应该放在分号解析之前

不带 .b 的输入

这个实现很简单，当我们尝试打开不带 .b 的文件失败以后，我们就给文件名加上 .b 再试着打开一遍就行，第一次打开文件失败，可能是因为没有 .b，我们就加上，如果加上 .b 还失败，才返回错误

```
// 判断两次，先判断没有 .b 再判断有 .b
if (((fd = open(path, O_RDONLY)) < 0) && ((fd = open_again_with_b(path)) < 0)) {
  printf("The path is %s\n", path);
  printf("spawn can't open file!\n");
  return fd;
}

int open_again_with_b(char *prog) {
  char temp[256];
```

```
  int i;
  for (i = 0; prog[i] != '\0'; ++i) {
    temp[i] = prog[i];
  }
  temp[i++] = '.';
  temp[i++] = 'b';
  temp[i] = '\0';
  return open(temp, O_RDONLY);
}
```

反引号解析

要求我们支持反引号的解析，也就是把反引号中命令执行的结果对原有命令做语法替换，替换后的整体指令作为新指令

比如 ls 的输出是 dir1，那么 echo ‘ ls ’ 的结果就应该是 echo dir1，因此输出的应该是 dir1

这里我们一个很自然的实现思路就是解析出反引号之后，先执行反引号中的内容，看看它的输出到底是什么，然后把这个输出拿回来，完全替换反引号和其中的内容即可，这个思路是很自然的，但关键是我们现在指令的输出结果都是直接输出到终端，我们能看到但是没法拿到

因此我们想到了先前实现过的管道，可以借助 lab6 中实现的**管道**，把子进程执行反引号内部指令的输出重定向到某个特殊的文件中，然后我们只需要再从这个文件中获取输出结果即可，以下是进行反引号替换的核心函数

```
void expand_backquote(char *input, char *output) {
  char *src = input;
  char *dst = output;
  while (*src) {
    if (*src == '`') {
      src++;
      char cmd[1024] = {0};
      int ci = 0;
      while (*src && *src != '`') {
        cmd[ci++] = *src++;
      }
      if (*src != '`') {
        debugf("error: unmatched backquote\n");
        exit();
      }
      src++;
      int p[2];
      // 创建管道
      if (pipe(p) < 0) {
        debugf("pipe error\n");
        exit();
      }
      // 创建子进程执行指令+写管道
      int child = fork();
      if (child < 0) {
        debugf("fork error\n");
        exit();
      } else if (child == 0) {
        // 父进程等待读管道内容
        close(p[0]);
        dup(p[1], 1);
        close(p[1]);
```

```
        runcmd(cmd);
        exit();
      } else {
        // 子进程循环写自己的输出进入管道
        close(p[1]);
        char result[1024] = {0};
        int offset = 0;
        int read_num = 0;
        while ((read_num = read(p[0], result + offset, sizeof(result) -
offset - 5)) > 0) {
          offset += read_num;
        }
        if (read_num < 0) {
          debugf("error in `\n");
          exit();
        }
        close(p[0]);
        // 父进程做输出语义替换
        int rlen = strlen(result);
        if (rlen > 0 && result[rlen - 1] == '\n') {
          result[rlen - 1] = '\0';
        }
        for (int i = 0; result[i]; i++) {
          *dst++ = result[i];
        }
      }
    } else {
      *dst++ = *src++;
    }
  }
  *dst = '\0';
}
```

该函数主要实现的是反引号内容的解析，把解析出来的子指令存入 cmd 中，然后创建子进程去执行这条指令，同时调用`pipe`创建一个管道，把子进程的写端，父进程的读端，分别记录在管道的两端，这样就能够另子进程执行结束后，父进程可以从管道中读出子进程执行的结果，存入 dst 数组中，接着我们使用 dst 中的内容对反引号之间的内容进行替换即可

注释功能

需要在 shell 中支持注释，具体执行过程中，指令中的注释会被忽略，很简单的一个小功能，只需要 readline 之后遍历我们读出来的指令，查找 # 然后把 # 后对应的内容自动忽略即可

```
for (int i = 0; i < strlen(buf); ++i) {
  if (buf[i] == '#') {
    buf[i] = '\0';
  }
}
```

新增外部指令

这里所说的新增指令都是 x.c -> x.b 需要创建子进程来执行的指令，内建指令我们不考虑，这些指令会在 runcmd 中被直接解析，并不会走 lab6 中我们使用 spawn 的这个流程解析

需要新增的内建指令：cd pwd history sh exit

需要新增的外部指令：mkdir touch rm，这三个指令其实本质都是在操作 open 函数，根据传入参数 O_OPEN，O_CREATE，O_MKDIR(新增) 来实现对文件不同的访问形式

mkdir

文件功能：创建目录

需要重点实现的是存在 -p 参数的时候，我们需要递归创造不存在的路径，比如 mkdir -p dir1 / dir2 / dir3，但是 dir1 和 dir2 都不存在，我们需要先创造 dir2，然后把 dir2 / dir3 作为参数，递归调用 mkdir 函数接着执行，然后在子函数中先创造 dir2，再把 dir3 作为及参数，递归调用 mkdir 接着执行，最后完成整个路径的递归创建

mkdir 主体函数如下：

```
int mkdir(char *path) {
  int fd;
  if (flag) {
    if ((fd = open(path, O_RDONLY)) >= 0) {
      close(fd);
      return 0;
    }
    int i = 0;
    char str[1024];
    for (int i = 0; path[i] != '\0'; ++i) {
      if (path[i] == '/') {
        str[i] = '\0';
        if ((fd = open(path, O_RDONLY)) >= 0) {
          close(fd);
        } else {
          break;
        }
      }
      str[i] = path[i];
    }
    for (; path[i] != '\0'; ++i) {
      if (path[i] == '/') {
        str[i] = '\0';
        fd = open(str, O_MKDIR);
        if (fd >= 0) {
          close(fd);
        } else {
          printf("other error when mkdir %s, error code is %d\n", path, fd);
          return -1;
        }
      }
      str[i] = path[i];
    }
    str[i] = '\0';
    fd = open(str, O_MKDIR);
    if (fd >= 0) {
      close(fd);
      return 0;
    } else {
      printf("other error when mkdir %s, error code is %d\n", path, fd);
      return -1;
    }
  } else {
```

```
    if ((fd = open(path, O_RDONLY)) >= 0) {
      close(fd);
      printf("mkdir: cannot create directory '%s': File exists\n", path);
      return -1;
    }
    fd = open(path, O_MKDIR | O_CREAT);
    if (fd == -10) {
      printf("mkdir: cannot create directory '%s': No such file or
directory\n", path);
      return -1;
    } else if (fd < 0) {
      printf("other error when mkdir %s, error code is %d\n", path, fd);
      return -1;
    } else {
      close(fd);
    }
    return 0;
  }
  return 0;
}
```

我们以 mkdir 函数为例重点解释一下实现流程，rm 和 touch 的实现思路大同小异，后面不再赘述：

1. 如果有 -p 参数，说明我们可以拥有更高的权限：
    - 首先根据路径打开目录，如果目录存在，则不需要重新创建，无事发生，也不需要报错，直接返回
    - 如果目录不存在，我们就创建一个新的目录，这里需要设置一个新的标志位 O_MKDIR 表示此时我们是在执行 mkdir 指令创建一个新的目录
    - 此时我们需要以 / 为分隔，一段一段递归调用 mkdir 进行目录的创建，每次只在上级父目录的基础上向下迭代创建一级目录即可
2. 如果没有 -p 参数，我们需要谨慎处理一些非法信息
    - 如果目录已经存在，则创建失败，报错并返回
    - 如果目录不存在，则我们正常创建，同样需要 O_MKDIR 参数的辅助

rm

文件功能：删除文件或目录，其实本质上还是操作 open 函数

```
int rm(char *path) {
  int fd;
  struct Stat st;
  if ((fd = open(path, O_RDONLY)) < 0) {
    if (!flag_f) {
      printf("rm: cannot remove '%s': No such file or directory\n", path);
      return -1;
    }
  }
  close(fd);
  stat(path, &st);
  if (st.st_isdir && !flag_r) {
    printf("rm: cannot remove '%s': Is a directory\n", path);
    return -1;
  }
  remove(path);
  return 0;
```

```
}
```

rm 的实现相对简单：

1. 先判断对象是否存在，删除不存在的文件会报错返回
2. 接着判断路径是否合法
3. 最后判断删除的是文件还是目录，删除对象和删除指令的参数必须一一对应，否则也要报错返回

touch

文件功能：创建文件，其实本质上还是操作 open 函数

```c
int touch(char *path) {
  int fd;
  if ((fd = open(path, O_RDONLY)) >= 0) {
    close(fd);
    return 0;
  }
  fd = open(path, O_CREAT);
  if (fd == -10) {
    printf("touch: cannot touch '%s': No such file or directory\n", path);
    return -1;
  } else if (fd < 0) {
    printf("other error when touch %s, error code is %d\n", path, fd);
    return -1;
  } else {
    close(fd);
  }
  return 0;
}
```

touch 的功能相对简单：

1. 首先判断文件是否存在，如果待创建的文件存在，不报错但直接返回
2. 接着判断路径是否合法
3. 最后直接创建文件即可，这里创建文件理论上一定会成功

sh & exit

sh 和 exit 虽然是内建指令，但是对于我们整个 shell 的功能（尤其是环境变量）的实现十分关键，因为它会负责子进程的创建和销毁，执行 sh 的时候我们要记得把 shellId+1 后存入对应的文件 shellId 中，然后**不返回**，接着调用 spawn 创建子shell 进程完成后续执行；执行 exit 的时候我们要记得把 shellId-1 后存入对应的文件 shellId 中，然后分父子进程依次处理

这里我的实现是把 sh 和 exit 都强制修改成内建的，因此即便指令中输入 sh.b 也会被强制解析成内建指令，可能对原始架构有一定修改

```c
// sh
if (strcmp(argv[0], "sh") == 0 || strcmp(argv[0], "/sh.b") == 0) {
  addShellIdToFile(1);
}

// exit
if (strcmp(buf, "exit") == 0) {
  r = syscall_getGlobalVar(0, 0, 7, shellId, 0);
  addShellIdToFile(-1);
```

```
  if (shellId == 0) {
    runcmd(buf);
  }
  else {
    exit();
  }
}
```

对于 exit 的指令，我们需要区分父子进程，若为子进程，就使用内建的 exit 指令返回上一级父进程而不会彻底退出，若为父进程，会执行 exit 这条**外部指令**，在我的实现中，exit 作为外部指令时等价于 halt 指令，也就是说，最顶层 sh 进程如果还执行 exit 指令的话等同于 halt，直接退出整个模拟环境

条件指令

又是一个难点，应该是我单个任务中花费时间最长的任务，需要我们解析形如 ls || ls 这样结构的指令，如果左边指令执行的结果正确或错误，结合 || 和 && 的**短路原则**，有可能我们不会执行右边的指令

实现思路

也就是说，在该任务中，我们需要支持父进程的 sh 能够拿到真正执行指令的那个子进程执行结果的**返回值**，最朴素的想法是直接用我们在 lab4 中写好的进程间 ipc，让子进程把自己的返回值通过 ipc_send 传给父进程，然后父进程使用 ipc_recv 等待返回结果，但是实际实现起来这么做很复杂，所以最后否掉了这个方案

这里我们仿照了全局变量 `globalVar` 的实现思路，在 syscall_all.c 中全局维护一个 `exit_status` 数组，用来完成对应进程编号 envid 和其返回值的映射，这里我们创建的进程编号最大是五位数，正常的数组完全存的下（否则要用 hash 映射优化）

（其实实际上开 1024 就能够存下，因为我们可以使用 ENVX 取 envid 的低位作为索引，根据 envid 的构建规则，只取低位也是完全可以保证去重的）

```
int exit_state[1024]; // 存储返回值

// 获取返回值
int sys_get_exit_status(u_int envid) {
  int res = 0;
  res = exit_state[ENVX(envid)];
  return res;
}

// 设置返回值
void sys_set_exit_status(u_int envid, u_int val) {
  exit_state[ENVX(envid)] = val;
  return;
}
```

这里因为需要访问全局内核中的 `exit_status`，我们需要实现两个系统调用 `syscall_set_exit_status` 和 `syscall_get_exit_status`，分别完成返回值的设置和返回值的获取，对应在内核中就是我们上面实现的这两个内核函数，这里为了**空间优化**，我用 ENVX 做了一层降维

实现时机

现在获取返回值和设置返回值的函数有了，还需要确定执行他们的时机，具体来讲，我们需要在子进程 exit 的时候设置它自己的返回值，在父进程 wait 子进程以后获取子进程的返回值，也就是实现所谓的**前V后P**原则，因此我们需要对 exit（这里指的是课程组给的 exit）和 wait 函数做如下的修改：

```
void libmain(int argc, char **argv) {
  int exit_status = main(argc, argv);
  // 子进程 exit 前设置返回值
  syscall_set_exit_status(env->env_id, exit_status);
  exit();
}

int wait(u_int envid) {
  const volatile struct Env *e;
  e = &envs[ENVX(envid)];
  while (e->env_id == envid && e->env_status != ENV_FREE) {
    syscall_yield();
  }
  u_int from_env;
  // 父进程 wait 后获取返回值
  int res = syscall_get_exit_status(envid);
  return res;
}
```

注意：在设计过程中，我还想过在每个 Env 控制块中增加一个 exit_status 中的字段，让每个进程记录自己的返回值，这么做理论可行，但忽略了一个很重要的事情，那就是子进程存完自己的返回值以后，在 exit 的时候会把自己的进程控制块**销毁**，那么父进程再尝试获取这个子进程控制块的时候，用子进程 envid 索引到的就是一个非法下标了，因此这种做法不可行

主实现流程

在 sh 中处理条件指令的时候，主要是在 parsecmd 函数中，如果我们解析到 || 或者 &&，那么 gettoken 就会返回代表 || 的 O，以及代表 && 的 A，此时我们就 fork 出来一个子进程来执行左边的指令，然后父进程直接进入 wait 等待子进程返回执行结果

```
case 'O':;
my_env_id = fork();
if (my_env_id < 0) {
  debugf("failed to fork in sh.c\n");
  exit();
}
else if (my_env_id == 0) {
  flag_next_is_condition = 1;
  return argc;
}
else {
  u_int caller;
  int res = wait(my_env_id);
  if (res == 0) {
    while (1) {
      int op = gettoken(0, &t);
      if (op == 0) {
        return 0;
      }
```

```
      else if (op == 'A') {
        return parsecmd(argv, rightpipe);
      }
    }
  }
  else {
    return parsecmd(argv, rightpipe);
  }
}
break;
```

我们以 || 为例解析流程，其中 && 基本对偶，不再赘述：

1. 首先当前 shell 进程 fork 出一个子进程来执行左边的指令
2. 如果是子进程（返回值 r == 0），直接返回当前解析出的参数个数 argc，进入 runcmd 开始执行相应的指令
3. 如果是父进程（返回值 r == 子进程 envid），则进入 wait 等待子进程执行完毕，并且获取子进程的返回值，如果此时子进程已经返回 1，那么根据 || 的短路法则，不需要执行后面的指令，直接返回 0 即可

追加重定向

也就是原有输出重定向的基础上实现**追加输出**的功能，> 是覆盖输出，新输出内容会覆盖原有文件内容，>> 是追加输出，新输出内容会接在原有文件内容之后

首先 >> 在 parsecmd 中的解析和 > 的解析完全相同，只不过我们读到两个 > 的时候就返回 a(append)，表示此时需要特殊处理追加重定向：

```
case 'a':
if (gettoken(0, &t) != 'w') {
  debugf("syntax error: > not followed by word\n");
  exit();
}
if ((fd = open(t, O_RDONLY)) < 0) {
  fd = open(t, O_CREAT);
  if (fd < 0) {
    debugf("error in open %s\n", t);
    exit();
  }
}
close(fd);
if ((fd = open(t, O_WRONLY | O_APPEND)) < 0) {
  debugf("error in open %s\n", t);
  exit();
}

if ((r = dup(fd, 1)) < 0) {
  debugf("failed to duplicate file to <stdout>\n");
  exit();
}
close(fd);
break;
```

这里我们需要在重定向的基础上增加一个新的标志位 O_APPEND，添加方式仿照 O_MKDIR 即可，该标志位表示此时我们是追加重定向，而不是从文件头开始覆盖写入，具体来讲就是在`serve_open`函数中，当识别到参数是 O_APPEND 的时候，就把文件描述符的 offset 改成文件大小，这样当前增加的位置就是上次添加的末尾了

```
#define O_APPEND 0x0004

// 特判追加，修改写入位置偏移量即可
if (rq->req_omode & O_APPEND) {
    ff->f_fd.fd_offset = ff->f_file.f_size;
}
```

其实要改的只有简单的两句代码

总结

最后是对 sh.c 文件中主流程的修改，主要修改的点有以下内容：

1. 首先进行了若语法层面的`while(check)`判断和替换机制，做了语法的预处理替换，用于处理分号，反引号等格式
2. 对内建指令的特判处理，如果是 cd，pwd，declare 等内建指令，我们不会 fork 子进程来处理，而是就在当前进程中实现

```
// 语义等价替换预处理
while(parseFenHao(tmp_buf, buf) > 0) {
  printf("We parsed '%s' \n", buf);
  while(parseFanYin(buf) > 0) {
    char _buf[1024] = {0};
    strcpy(_buf, buf);
    expand_backquote(_buf, buf);
  }
  ...
}

// 内建指令特殊处理
if (parseCD(buf) == 0 && parseDC(buf) == 0 && parseUN(buf) == 0) {
  if ((r = fork()) < 0) {
    user_panic("fork: %d", r);
  }
}
else {
  runcmd(buf);
  continue;
}
```

以上就是 lab6-shell 中的所有新增功能的实现思路，总体上所有新增任务的难度比较适中，并没有完全难到没有思路的新增指令，同时覆盖面也比较全面，对 lab4 的系统调用，lab3 的进程控制，lab5 的文件系统，lab6 的管道和基础 shell 全都有所覆盖，在实现过程中，也帮助我回顾了整个 OS 的实现过程，第一次打开虚拟机学 linux 命令的场景犹在眼前，整个实验跟下来还是有不少收获，OS 重点在于一种宏观的架构和思想，把几条重点的逻辑线搞清楚（lab3 的进程创建 / 调度逻辑，异常处理逻辑，lab4 的系统调用逻辑，lab5 的文件系统访问逻辑，lab6 的管道访问逻辑和 shell 加载子程序逻辑），对于后续相关功能的实现就一目了然了

在完成挑战性任务的时候，我参考了以下两位学长的博客

https://cookedbear.top/p/28193.html

https://blog.csdn.net/BlockInput?type=blog

他们的宝贵思路帮我节省了很多时间，也少走了很多弯路，在此表示由衷的感激和敬佩