

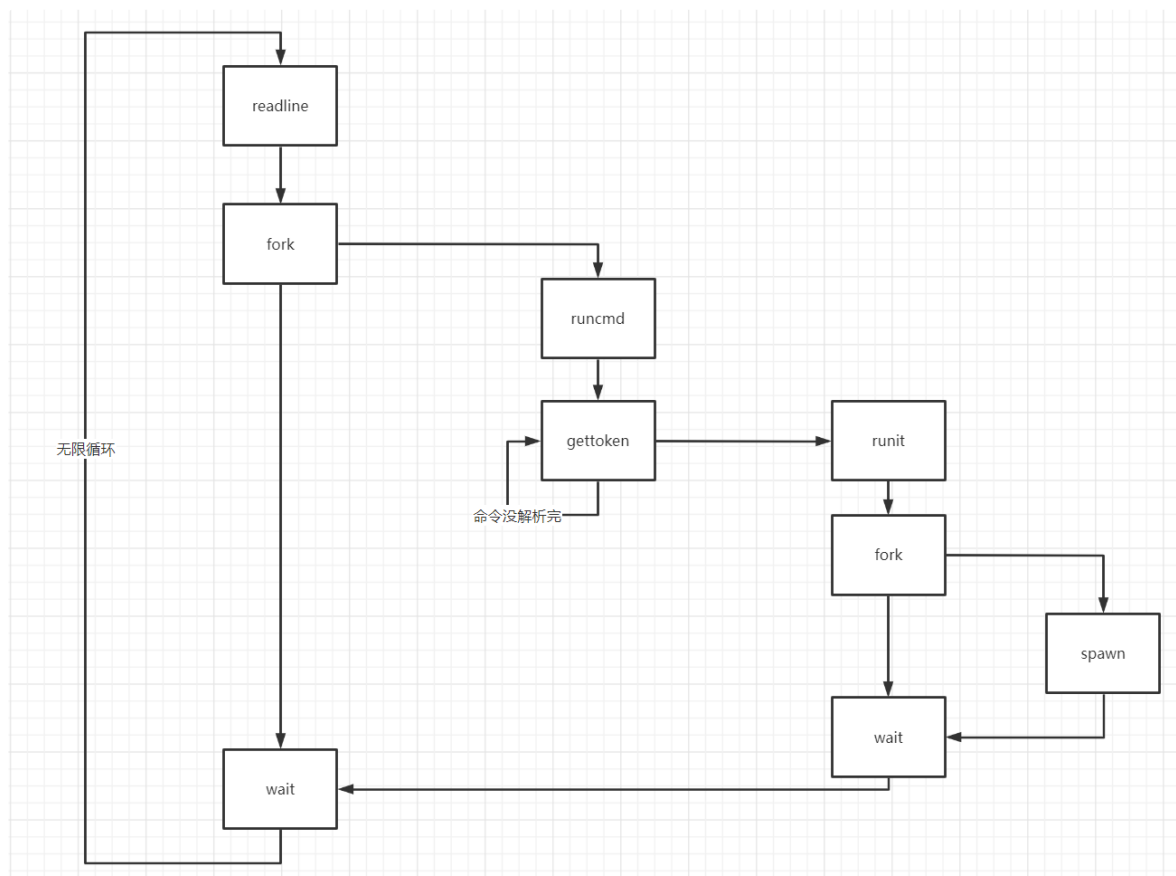
Lab6-Challenge 实验报告

扶星辰

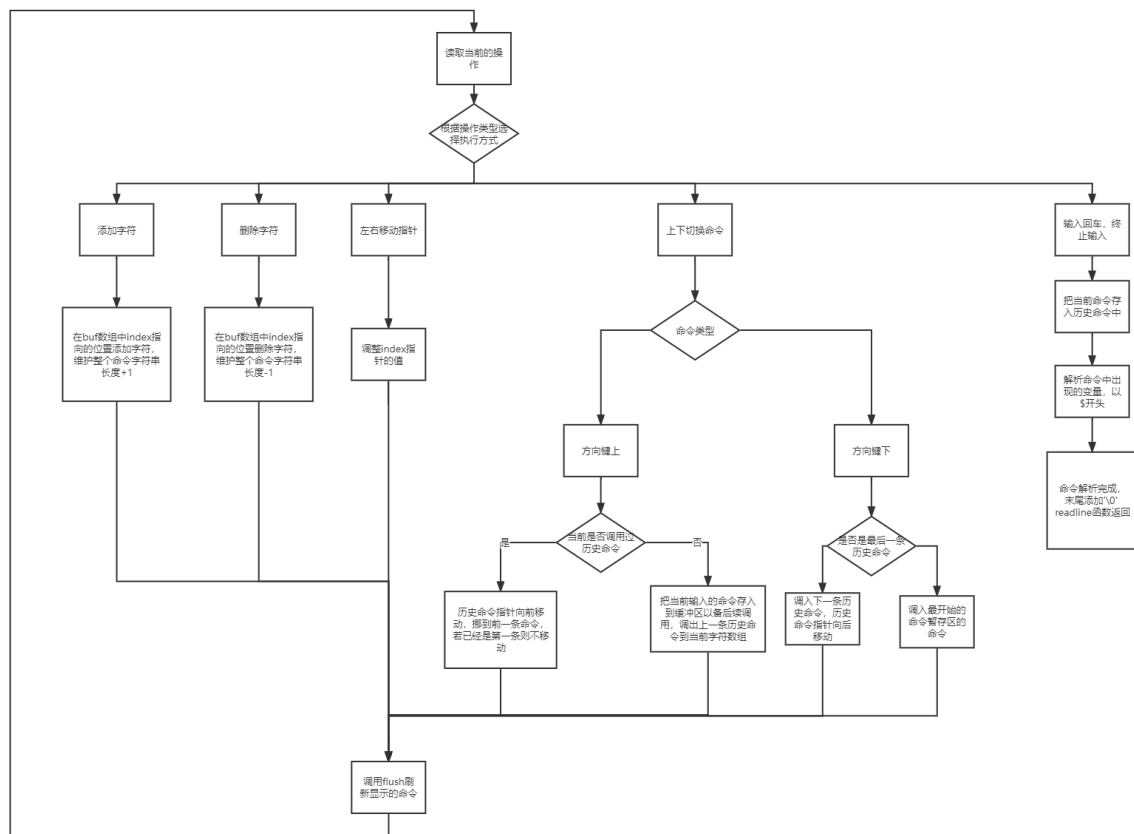
19377251

实现思路

原先的lab6中 运行指令的流程是



在实现完challenge的功能后，这部分的流程主体没有太大变化，变化主要体现在readline的实现，下面给出readline函数的处理逻辑 图片分辨率有点大



后台运行功能(easy)

说明

取消spawn后的wait, 让父进程提前返回readline处即可实现后台运行

实现方式

实现代码

```

again:
    .....
    case '&':
        backstage=1;
        break;
runit:
    .....
    if (r ≥ 0) {
        if (debug_) writef("[%08x] WAIT %s %08x\n", env→env_id,
argv[0], r);
        // if(isrun==0)
        if(backstage==0)
            wait(r);
        else
            backstage=0;
    }

```

实现一行多条命令(easy)

说明

原先的lab6中，一条命令解析完就会跳转到runit，runit运行后就会返回至readline读取下一条命令。

实现方式

实现该方法为在runcmd函数中增加符号判断';'，当读取到';'后直接跳转到runit，在runit运行完后进行判断，如果是由；跳转而来，就跳转回again继续解析命令。

实现代码

```
again:
    .....
    case ';':
        runnow=1;
        goto runit;
runit:
    .....
    if(runnow==1){
        runnow=0;
        goto again;
    }
```

实现引号支持(easy)

说明

在原先的命令解析函数gettoken中，他分词的逻辑是根据WHITESPACE来分，然后如果没有遇到指针就一直++，直到遇到下一个分词符号就跳转回，不同参数间字符更改为\0来截断。因此只需要破坏对引号的截断逻辑即可。该操作可以实现一个参数内包含空格；等特殊字符，且被引号保护的内容不会被declare声明的变量解析

实现方式

gettoken中添加的核心代码如下

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

遇到"后就直接往后读取下一个"，中间的字符都归为一个参数，最后返回'w'表明这一段是一个参数整体。

实现文件命令(easy)

说明

主要实现了 `touch` `mkdir` `tree` 函数，同时附带完成了sh进程与serv进程的通信、初始化磁盘的一些内容。

实现方式

由于代码过于冗长，这里就不妨代码了，主要说一下思路。

为了实现文件交互命令 `tree` `mkdir` `touch`，首先需要完善文件系统的功能。

在原先的文件系统中，新建文件是用了 `serv` 进程中的 `open` 功能，通过添加 `O_CREAT` 标识符，在文件系统进程判断后通过调用 `file_create` 来实现新建文件，但是如果这样新建文件有几个坏处

- 功能冗余 `open` 是为了打开文件读写 如果用它来新建文件会有冗余操作 降低效率
- 功能不全 当前实现的 `file_create` 不包含新建文件夹的操作，只能在根目录的级别下新建文件

为了解决这两个问题，使用了一个新的进程通信方式

步骤如下

- 在 `include/fs.h` 中新建服务号 `FSREQ_CREATE` 新建结构体 `Fsreq_create` 来完成与文件系统间的信息传递
- 在 `user/file.c` 中新建函数 `user_create` 来实现新建文件的可用接口
- 在 `user/fsipc.c` 中实现 `user_create` 函数所需要传递的信息，实现把信息传递给文件进程 `fs/serv`
- 在 `fs/serv.c` 中添加接收消息的判断，读取服务号 `FSREQ_CREATE`，并将对应参数传递给 `serve_create`，通过 `serve_create` 调用 `file_create` 函数来完成新建文件的操作
- `file_create` 函数调用 `walk_path` 来完成查找文件的操作，其中的用法为调用自己递归地查找，并在最后完成文件创建，`walk_path` 主要是用来遍历文件夹，查找所需新建的文件是否存在，其中调用 `dir_lookup` 来进一步查找文件

由于实现了文件夹的对应功能，因此在所有函数的参数中都有同时 `isdir` 参数来判断是否需要新建文件夹的操作，实现了在没有对应文件夹的情况下可以正确完成 `mkdir` 和 `touch`，例如 `touch a/a/a/b` 这种操作

最后 `tree` 的实现需要用到稍微繁杂的递归，主体逻辑是读取并按行输出当前目录下的所有内容，如果读取到了目录，就递归的对这个目录调用 `tree` 函数，同时应该多传递一个参数来表明当前目录所嵌套的层级数，这样就能让最后的输出和正常 `tree` 的指令一样比较好看。

实现历史命令功能(normal)

说明

实现上下键切换到历史命令和当前命令的功能

实现方式

在根目录下新建文件 `history` 文件，在其中按行记录每次输入的命令，通过改写 `readline`，在每次回车运行命令后把新的命令利用 `O_APPEND` 添加到文件尾，对于命令切换，我是实现的逻辑如下

- 当用户输入 `↑` 时，会判断当前用户的命令属于哪个部分，如果已经在最开始的历史指令则无反馈；如果此前没有输入过 `↑`，那么就把当前已经输入的东西存入命令缓冲区，在用户在历史最后

一条命令并输入`[↑]`后会调处原来的命令缓冲区的东西，如果有上一条指令则会切到上一条指令，并把指针移动到这条指令的末尾

- 在用户输入`[↑]`时，会判断下一条有没有历史指令，如果有就切过去，如果没有就切回到用户命令缓冲区的内容，如果已经在用户命令缓冲区则无反馈。

此外，为了实现更好的用户交互，在此基础上还实现了 `backspace` 退格 左右键切换用户当前指针来实现插入输入的操作。此外还支持命令的tab补全功能。

这些的实现原理用到了ANSI的控制码，每次命令更新后刷新当前行的命令，重新打印这些命令，并把当前用户的指针移动到对应位置。这样就能在插入、删除字符时实现应有的效果。

对于具体的实现方式，这里展示几个函数

保存当前命令 使用`O_APPEND`来向文件中追加内容

```
void save_command(char * buf)
{
    // writef("saved: %d %s\n",strlen(buf),buf);
    int fd = open("etc/history", O_WRONLY|O_CREAT|O_APPEND|O_PROTECT);
    write(fd,buf,strlen(buf));
    write(fd,"\n",1);
    close(fd);
}
```

初始化历史命令buffer，读取history文件到buffer里，初始化指令指针等待后续的输出

```
void init_buf()
{
    history_buf[0]=0;
    temp_buf[0]=0;
    int fd=open("etc/history",O_RDONLY|O_CREAT|O_PROTECT);
    int n=read(fd,history_buf,4096);
    close(fd);
    // writef("ok");
    // writef("-%d-\n",strlen(history_buf));
    if(n<0)
        user_panic("error reading history");
    if(history_buf[0]==0){
        history_maxl=0;
        history_ptr=0;
        return;
    }
    history_maxl=history_buf;
    while(*history_maxl!='\0'){
        if(*history_maxl=='\n'){
            *history_maxl='\0';
            // writef("ok");
        }
        history_maxl++;
    }
    history_maxl--;
    history_ptr=history_maxl+1;
    return;
}
```

切换到上一条命令，主要操作 保存当前命令到暂存区（如果是要输入的指令） 调入当前指向的指令
刷新指令缓存区

```
int history_last(char *buf,int length)
{
    buf[length]=0;
    // writef("up");
    if(history_ptr==history_maxl){
        return 0;
    }
    if(history_ptr==history_buf){
        return 0;
    }
    int i;
    for(i=0;buf[i]!=0;i++){
        writef("\b \b");
    }
    if(history_ptr==history_maxl+1){
        strcpy(temp_buf,buf);
    }
    history_ptr-=2;
    while(*history_ptr!=0){
        history_ptr--;
    }
    history_ptr++;
    strcpy(buf,history_ptr);
    return 1;
}
```

实现shell环境变量(challenge)

说明

为了实现这一功能，使用了一个文件variables进行变量存储

在变量中区分是否全局变量，如果是局部变量则用当前进程的pid唯一标识这个变量。

如果是只读变量则以后不允许修改。

在实现的环境变量功能里，全局变量和局部变量是不冲突的概念，可以同时有同名的全局和局部变量，但是每个进程中的局部变量名称是非重复的，全局变量的名称也是非重复的。当一个进程找变量名时，会先找他对应的局部变量，没有才会找全局变量。

declare 命令的参数与挑战性任务中的描述基本一致，单独输入**declare**会输出所有的变量，**declare**加参数会新建变量或者覆盖原先变量（原先变量没有-r）

此外还支持了**unset**撤销，当**unset**带-x时会删除全局变量，否则会删除局部变量。

对于**declare**出的变量，可以使用**declare**命令查看所有已声明的变量，也可以在命令胡中使用\$name的方式调用变量的值，例如

```
declare path =home
echo $path
//会输出home
echo "$path"
//会输出$path
```

同时在查找变量时会遵循一个原则：局部优先，如果查找到了局部变量则直接使用局部变量，没有则会寻找全局变量，如果从未声明则改变了不会替换它的值。

实现方式

主要通过读写 `etc/variables` 来完成操作

下通过代码简要说明

这是输出所有变量的代码，主体是一个while循环，其中调用的函数会解析当前行变量的各种信息，通过指针的形式存储在name、r、x等变量中，同时会让缓存区指针p前进到下一个，当到末尾时就会返回0

```
void print_all_variable()
{
    int fd=open("etc/variables",O_RDONLY|O_CREAT|O_PROTECT);
    read(fd,variable_buf,4096);
    close(fd);
    char *p=variable_buf;
    int r,x,pid;
    char *name,*value;
    if(*p=='\0'){
        writef("no variables now");
        return;
    }else{
        writef("name    r    x    pid value\n");
    }
    while(get_detail(&p,&name,&r,&x,&pid,&value)){
        writef("%s\t",name);
        if(r==1) writef("1\t");
        else writef("0\t");
        if(x==1)writef("1\t");
        else writef("0\t");
        writef("%d\t",(pid==-1?0:pid));
        if(value==0){
            writef("UNDEFINED\n");
        }else{
            writef("%s\n",value);
        }
    }
    writef("declare variable list end");
}
```

删除变量

该操作是建立在查找变量之后，如果查找到了要替换的变量，会传入该变量信息在暂存区的起始位置，把当前位置设为'\0'，并调整当前位置为下一个变量位置，向文件中写入两部分内容即可完成删除操作。遇到冲突变量 或者 **unset** 都会调用这个函数

```
void remove_variable(char *stop)
{
    int fd=open("etc/variables",O_RDONLY|O_PROTECT);
    read(fd,variable_buf,4096);
    close(fd);
    // writef("%d\n",strlen(variable_buf));
    // writef("replace: %s\n",stop);
    *stop=0;
    // writef("replace1: %s %d\n",variable_buf,strlen(variable_buf));
    while(*stop!='\n')stop++;
    stop++;
    // writef("replace2: %s %d\n",stop,strlen(stop));
    fd=open("etc/variables",O_WRONLY|O_PROTECT);
    write(fd,variable_buf,strlen(variable_buf));
    write(fd,stop,strlen(stop));
    close(fd);
}
```

判断变量是否冲突的片段

该片段首先查找变量名，变量名相同则对比其作用域，作用域相同则根据x判断，都是局部变量就判断pid是否相同，相同或者都是全局变量就会进入是否是可读变量的判断，在这里维护了stop指针，这个指针就是上面删除变量所用到的

```
while(get_detail(&p,&name,&r,&x,&pid,&value)){
    if(strcmp(name,pname)==0){
        if(x!=px){
            // continue;
        }else{
            if(x==0){
                if(syscall_getenvpid()==pid){
                    change=1;
                    break;
                }
            }else{
                change=1;
                break;
            }
        }
    }
    stop=p;
}
if(change==0){
    add_to_tail(pname,pr,px,pvalue);
    return;
}
if(r){
    if(x) writef("global ");
}
```





```

        else writef("local ");
        writef("variable %s can only be read",name);
        return;
    }
    remove_variable(stop);

```

其他功能

在challenge中easy、normal、一个challenge任务的基础上，还实现了以下功能

- 在输入命令时支持 **tab**（命令补全，文件名补全尚未实现），**backspace**（删除一个已输入的字符）， （左右键移动光标，配合输入字符或backspace可以实现更好的编辑要使用的命令）
- 对磁盘中文件重新组织，所有命令（初始文件）均放在根目录的**bin**目录下，环境变量存储文件和历史命令文件存储在**etc**目录下，用户可以操作的文件目录为 **/home**

由于对文件组织进行了重构，因此一些指令的实际效果也会发生变化

对于 **touch mkdir** 等新建文件、文件夹的指令或 **>** 等会导致新建文件、文件夹的操作，新建的文件都会在 **home** 目录下，例如 **touch a** 会新建 **home/a** 文件

ls tree 指令若没有其他参数 会输出home目录下的文件 加上参数 **/** 后会从根目录开始输出

由于上述权限限制，用户无法读写home文件夹以外的内容（只能读到文件名，写操作完全屏蔽）

实现方式

对于输入指令的实现方式可以查看上面readline流程图

对于文件的重新组织，在创建磁盘镜像时替换成为如下代码，把初始化的.b可执行文件放入bin文件夹中，同时创建etc和home文件夹

```

struct File *bin;
bin = write_directory(&super.s_root, "/bin");
write_directory(&super.s_root, "/etc");
write_directory(&super.s_root, "/home");

if(strcmp(argv[2], "-r") == 0) {
    for (i = 3; i < argc; ++i) {
        write_directory(&super.s_root, argv[i]);
    }
}
else {
    for(i = 2; i < argc; ++i) {
        write_file(bin, argv[i]);
    }
}

```

在这里修改并补全了 **write_directory** 函数

函数如下通过 **create_file** 来在对应文件夹下新建一个文件块 然后修改文件块的信息并返回其指针

```

struct File* write_directory(struct File *dirf, char *name) {
    // Your code here
    printf("%s\n", name);
    struct File* res=create_file(dirf);
    const char * filename=strrchr(name, '/');
    if(filename) filename++;
    else filename=name;
    strcpy(res->f_name, filename);
    res->f_size=0;
    res->f_type=FTYPE_DIR;
    return res;
}

```

功能测试

由于测试的样例过多，功能的测试会在答辩时——展示。

遇到的问题

第一个问题是命令行交互，在最开始的MOS命令行中输入`[↑]`等方向键会导致指针乱跑，而且也不支持退格tab等操作。

通过网上查询到了ASNI控制码来实现命令交互，为了更好的实现，在这里引入了一个命令行buf来记录当前输入的命令，通过维护length（已输入的命令长度）和index（用户当前指针的位置）来记录当前命令的状态，当需要插入字符或者删除字符时，通过index的位置来在buf中删除、添加字符，通过这个数组，也能实现history、tab、替换环境变量等功能。

此外，为了避免原先`[↑]`等方向键导致指针乱跑的现象，在每次调用readline前使用控制码`\033[s`记录当前指针的位置，每次命令buffer有变动时都会调用flush函数，该函数会首先返回原先存储的指针位置，然后删除当前行右边的所有字符，然后把buf中的字符依次打印出来，最后根据index的值把指针挪动到合适的位置，这样就实现了每次命令刷新命令。

第二个问题是实现文件系统的create接口，在当初做lab5时对这方面只是有个大概的印象，没有一个完整的概念。在完成lab5-2-exam时实现了O_CREAT的操作，这次上机让我大概了解了用户进程与文件系统进程的交互过程，并通过向往届学长提问咨询，最后实现了这样一个新建文件的交互方式。

在user中实现函数user_create来给出用户操作文件的接口，同时在fsipc中进一步完善user_create的所需要传递的信息，并发送给文件管理进程，文件管理进程通过读取服务号来判断当前传递的信息类型，并把这些信息传递给serve_create函数，该函数解析传来的信息后调用file_create来实现创建文件、文件夹的功能。

在这里改动了原先的file_create函数，在其中支持了文件夹参数，该参数会根据命令新建对应的文件夹来满足新建文件、文件夹时他的父文件夹不存在的情况。

第三个问题是文件的写入问题，在实现declare的过程中发现了文件系统的一个小bug。

当open一个现有的文件且没有O_APPEND时，应该是从头重新写入这个文件，但是如果新写入的内容比原先的内容字符少，文件的size并不会正确的减小，这是由于user/file.c中函数file_write更新文件大小时机的问题，在这里只有新写入的文件大于原文件时才会更新文件的size，这与实际的文件写入规则应该是不匹配的。更改完这个bug后，declare的相关内容被正确的实现了。