# Lab 2 Report

## Big Picture

整个ush运行的大致流程是，由`command`函数接受输入指令，解析后交给`invoke`来执行。`invoke`实现如下：

```
int invoke(int argc, char **argv, int srcfd, char *srcfile, int dstfd, char *dstfile,
          BOOLEAN append, BOOLEAN bckgrnd)
{       /* invoke simple command */
    int pid;
    if (argc == 0 || builtin(argc, argv, srcfd, dstfd)) {
        return (0);
    }
    if ((pid = fork()) < 0) {
        syserr("fork failed");
    }
    if (pid == 0) {
        redirect(srcfd, srcfile, dstfd, dstfile, append, bckgrnd);
        if (echo(argc, argv)) exit(0);
        execvp(argv[0], argv);
        syserr("exec failed");
    }
    if (!bckgrnd) wait(0);
    return (pid);
}
```

`invoke`先尝试调用`builtin`函数，如果要运行的指令是内置指令，就可以在`builtin`中被处理（直接在当前进程执行）。如果不是，`invoke`会`fork`出一个子进程，在子进程下进行重定向`redirect`，然后使用`execvp`系统调用运行程序（`echo`因为可能涉及重定向，也放在了子进程下执行）。

除了`echo`的所有内置指令在`builtin`中执行：

```
static BOOLEAN builtin(int argc, char *argv[], int srcfd, int dstfd)
{       /* do built-in */
    char *arg;
    int i;
    BOOLEAN ret = TRUE;
    if (strchr(argv[0], '=') != NULL) {
        asg(argc,argv);
    }
    else if (strcmp(argv[0], "cd") == 0) {
        if (argc == 1) arg = EVget("HOME");
        else arg = argv[1][0] == '$' ? EVget(argv[1]+1) : argv[1];
        arg = arg == NULL ? "." : arg;
        if (chdir(arg) < 0)
            fprintf(stderr, "cd: no such file or directory: %s\n", arg);
    }
    else if (strcmp(argv[0], "exit") == 0) {
        exit(0);
```

```
    }
    else if (strcmp(argv[0], "export") == 0) {
        export(argc, argv);
    }
    else if (strcmp(argv[0], "pwd") == 0) {
        if ((arg = getcwd(NULL, 0)) == NULL) {
            syserr("pwd failed");
        }
        printf("%s\n", arg);
    }
    else if (strcmp(argv[0], "set") == 0) {
        set(argc, argv);
    }
    else if (strcmp(argv[0], "unset") == 0) {
        unset(argc, argv);
    }
    else {
        ret = FALSE;
    }
    return (ret);
}
```

# Environment Commands

## assignment, =

指令形式是`name=value`。找到`=`的下标把字符串分成两部分，然后调用`EVset`。

```
BOOLEAN EVset(char *name, char *val)
{       /* add name & value to environment */
    struct varslot * v;
    if ((v = find(name)) == NULL) {
        return FALSE;
    }
    return (assign(&v->name, name) && assign(&v->val, val));
}
...
void asg(int argc, char *argv[])
{       /* assignment command */
    char *part;
    if (argc != 1) {
        fprintf(stderr, "Usage: name=value\n");
        return;
    }
    part = strstr(argv[0], "=");
    if (part == NULL) {
        fprintf(stderr, "Usage: name=value\n");
        return;
    }
    *part = '\0';
    if (!EVset(argv[0], ++part)) {
```

```
        fprintf(stderr, "Assignment failed\n");
    }
}
```

## set

实现的指令形式是`set [name1=value1 [name2=value2 ...]]`.如果是单独一个`set`就打印当前所有环境变量，否则进行assignment。

```
void set(int argc, char *argv[])
{       /* set command */
    int i;
    if (argc == 1) {
        EVprint();
        return;
    }
    for (i = 1; i < argc; i++) {
        asg(1, argv+i);
    }
}
```

## unset

指令形式是`unset name1 [name2 ...]`，移除指定名称的环境变量。

```
void unset(int argc,char *argv[])
{       /* unset command */
    int i;
    struct varslot *v;
    if (argc < 2) {
        fprintf(stderr, "Usage: unset name1 name2 ...\n");
        return;
    }
    for (i = 1; i < argc; i++) {
        v = find(argv[i]);
        if (v != NULL) {
            free(v->name);
            free(v->val);
            v->name = v->val = NULL;
            v->exported = FALSE;
        }
    }
}
```

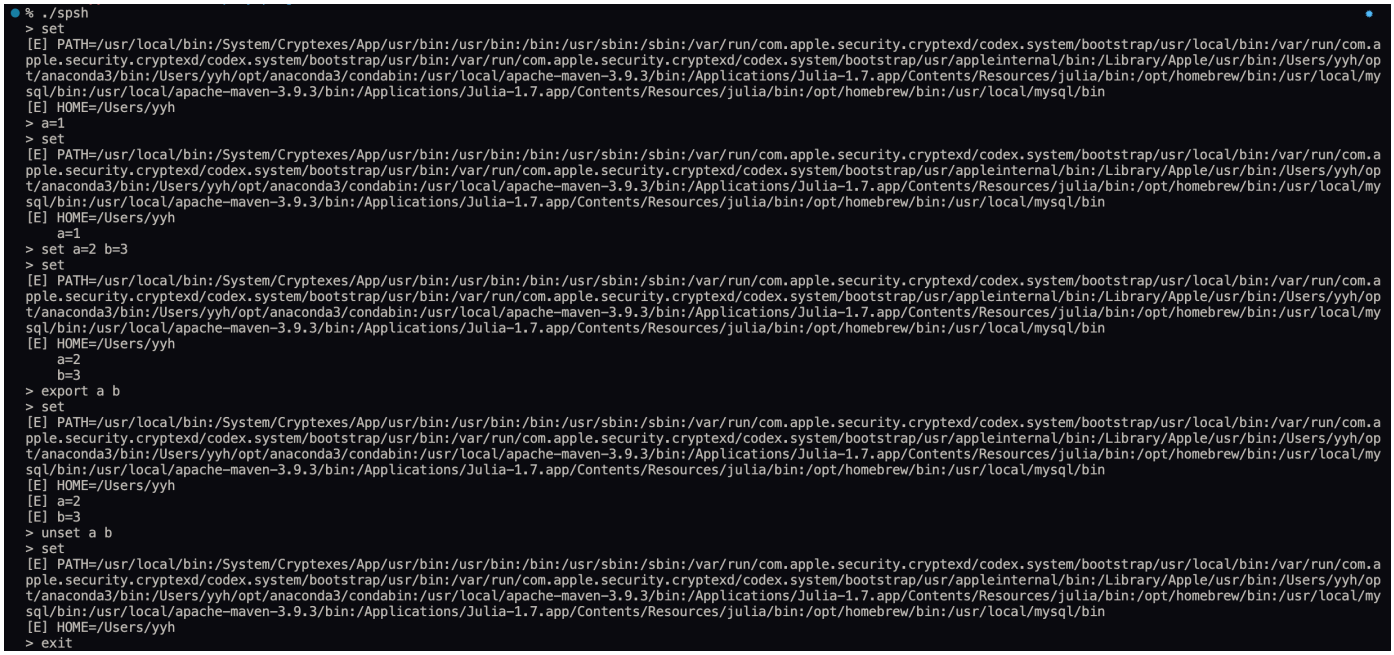
## export

指令形式是`export [name1 [name2 ...]]`，设置指定名称的环境变量为exported（会被传递到子进程）。如果不存在该变量则创建一个（值为空）。

如果单独一个`export`则执行与`set`一样的行为。

```
BOOLEAN EVexport(char *name)
{       /* set variable to be exported */
    struct varslot * v;
    if ((v = find(name)) == NULL) return(FALSE);
    if (v->name == NULL) {
        if (!assign(&v->name, name) || !assign(&v->val, "")) {
            return(FALSE);
        }
    }
    return (v->exported = TRUE);
}
...
void export(int argc, char *argv[])
{       /* export command */
    int i;
    if (argc == 1) {
      set(argc, argv);
      return;
    }
    for (i = 1; i < argc; i++)
    if (!EVexport(argv[i])) {
        printf("Cannot export %s\n", argv[i]);
        return;
    }
}
```

## testing




# Other Built-in Commands

## cd

指令形式是`cd [path]`。如果未指定路径，从环境变量中读取HOME的值。调用`chdir`切换到指定路径。

HOME未设置时不做任何事。

```
    if (argc == 1)arg = EVget("HOME");
    else arg = argv[1][0] == '$' ? EVget(argv[1]+1) : argv[1];
    arg = arg == NULL ? "." : arg;
    if (chdir(arg) < 0)
        fprintf(stderr, "cd: no such file or directory: %s\n", arg);
```

## echo

指令形式是`echo str1 [str2 ...]`。打印值到标准输出。如果一项以`$`开头，打印对应名称环境变量的值。如果一项有引号，打印引号内的内容，但分两种情况：若是双引号`"..."`，解析引号内的`$`；若是单引号`'...'`，不加解析地打印引号内的字符串（该行为与zsh一致）。

双引号的情况其实在`gettoken`中处理好了，因此这里只处理了`$`和单引号的情况。

```
static BOOLEAN echo(int argc, char *argv[])
{
    int i;
    char *arg;
    if (strcmp(argv[0], "echo") != 0) {
        return FALSE;
    }
    if (argc < 2) {
        fprintf(stderr, "Usage: echo ...\n");
```

```
        return TRUE;
    }
    for (i = 1; i < argc; i++) {
        if (argv[i][0] == '$') {
            arg = EVget(argv[i]+1);
            printf("%s", arg == NULL ? "" : arg);
        } else {
            if (argv[i][0] == '\'' && argv[i][strlen(argv[i])-1] == '\'') {
                strcpy(arg, argv[i]);
                strncpy(argv[i], arg+1, strlen(arg)-2);
                argv[i][strlen(arg)-2] = '\0';
            }
            printf("%s", argv[i]);
        }
        printf("%c", " \n"[i == argc-1]);
    }
    return TRUE;
}
```

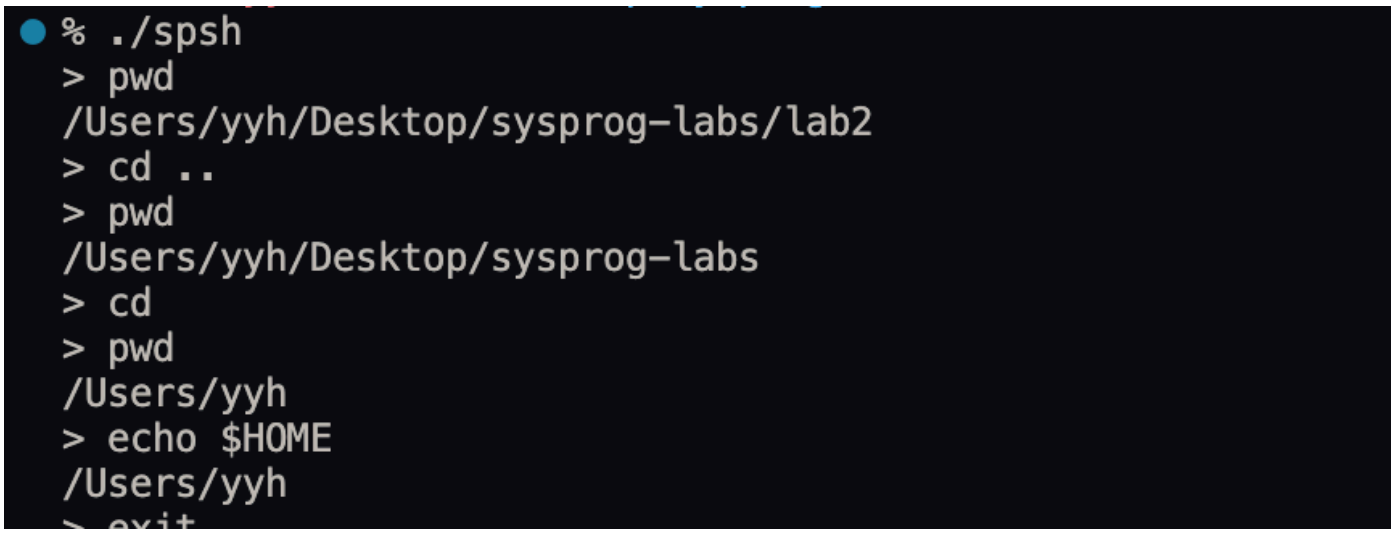
## exit

调用`exit(0)`结束进程。注意到这种便捷的实现只有在shell所在进程下执行才有效，这是有内置指令的意义之一。

## pwd

调用`getcwd`打印当前工作目录到标准输出。

## testing



```
% ./spsh
> echo HOME
HOME
> echo $HOME
/Users/yyh
> echo "$HOME"
/Users/yyh
> echo "AA BB"
AA BB
> echo 'AA BB'
'AA BB'
> echo '$HOME'
$HOME
> exit
```

# Features: redirect & pipe

## redirect, < > >>

重定向`redirect`实现如下：

```
static void redirect(int srcfd, char *srcfile, int dstfd, char *dstfile,
                     BOOLEAN append, BOOLEAN bckgrnd)
{       /* I/O redirection */
    if (srcfd == 0 && bckgrnd) {
        strcpy(srcfile, "/dev/null");
        close(0);
        srcfd = BADFD;
    }
    if (srcfd != 0) {
        close(0);
        if (srcfd == BADFD) {
            if (open(srcfile, O_RDONLY) < 0) {
                fprintf(stderr, "ERROR: failed to open %s\n", srcfile);
                exit(1);
            }
        } else {
            dup(srcfd);
            close(srcfd);
        }
    }
    if (dstfd != 1) {
        close(1);
        if (dstfd == BADFD) {
            if (append) { // >> dstfile
                if (open(dstfile, O_WRONLY | O_CREAT | O_APPEND, 0666) < 0) {
                    fprintf(stderr, "ERROR: failed to open %s\n", dstfile);
                    exit(1);
                }
            } else { // > dstfile
                if (open(dstfile, O_WRONLY | O_CREAT | O_TRUNC,  0666) < 0) {
```

```
                    fprintf(stderr, "ERROR: failed to create %s\n", dstfile);
                    exit(1);
                }
            }
        } else {
            dup(dstfd);
            close(dstfd);
        }
    }
}
```

- 如果指令是在后台运行，打开空设备作为标准输入。
- 重定向是利用`open`、`dup`会把打开的文件/指定fd 关联/复制到最小的空闲fd这一特性。
- 如果`srcfd`不为0（标准输入），需要重定向输入。如果`srcfd`是`BADFD`，表示需要打开`srcfile`指定的文件作为标准输入。
- 如果`dstfd`不为1（标准输出），需要重定向输出。如果`dstfd`是`BADFD`，表示需要打开`dstfile`指定的文件作为标准输出。输出情况分`>>`和`>`即是否append，对应的open flag不同；输出还可能涉及创建新文件，需指定权限（这里都定为`0666`即`rw-rw-rw-`）

使用如下：

```
% ./spsh
> echo a > out
> cat out
a
> echo aa >> out
> cat out
a
aa
> echo aaa > out
> cat < out
aaa
> exit
```

## pipe, |

初始代码中的pipe似乎没有效果，于是作了如下修改：


由于初始代码中使用的是递归，实际上是右边的指令先被执行（除非使用`fork`+`wait`等手段，但需要较大地改动`command`函数），更适合实现从右向左的pipe。因此把管道的方向反了一下，把右边指令的`dstfd`连到管道的写端，左边指令的`srcfd`连到管道的读端。这样实现的pipe使用如下：

```
% ./spsh
> sort | grep -e "ush" | ls
ush-env.c
ush-env.h
ush-env.o
ush-parse.c
ush-parse.h
ush-parse.o
ush-prt.c
ush-prt.h
ush-prt.o
ush-sig.c
ush-sig.h
ush-sig.o
ush.c
ush.h
ush.o
> wc | cat | echo "AA BB CC DD"
       1       4      12
> exit
```