实验报告

遇到的问题

使用`chdir()`后参数发生变化

事情是这样的,当我使用后想要输出一条错误信息:

```
if (chdir(path) != 0) {
    NshError("cd : %s: no such file or directory\n", path);
    return -1;
}
```

当且仅当`path`被`~`替换后, **神奇**的引发了**段错误**.使用gcc自带的`-fsanitize=address`选项后得到了:

```
=================================================================
==18359==ERROR: AddressSanitizer: stack-use-after-scope on address
0x7ffe9985b450 at pc 0x7f1f29e78f89 bp 0x7ffe9985b2c0 sp 0x7ffe9985aa38
READ of size 16 at 0x7ffe9985b450 thread T0
==18359==ABORTING
```

发现问题是出在`printf`上, 这个错误可以复现,

发现是 `stack-use-after-scope` 导致的, 是我在函数外部仍然使用了函数内部的局部变量, 然后就寄了

`readline`库内存泄漏问题

运行时发现了

```
=================================================================
==46857==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 18 byte(s) in 3 object(s) allocated from:
    #0 0x7f9e60fef887 in __interceptor_malloc
../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:145
    #1 0x7f9e60f20bac in xmalloc (/lib/x86_64-linux-
gnu/libreadline.so.8+0x39bac)

SUMMARY: AddressSanitizer: 18 byte(s) leaked in 3 allocation(s).
```

后来想起来再ICS PA中也遇到过相同的问题,具体详见这篇博客.

> **更新:**
> 发现 OJ 平台的编译选项并不支持我使用 `readline` 库, 也就是在Makefile 中没有写
>
> ```
> -lreadline
> ```
>
> 遂投入`getline()`的怀抱, 不再面临 `readline` 库祖传内存泄漏的问题.

`execvp()`不执行程序

没有好好看`man`导致的(

在调用函数的时候**胡乱传参**, 必然运行不起来hhhhh

多个`pipe()`使用时子进程接收不到输入

这个应该是我 lab 1 中印象最深, 解决时间最久的 bug.

简单来说, 就是我发现原来实现 `|` 的函数最多只能够支持一个 `|`, 试图将原来实现 `|` 的函数修改为 **一个子进程** 同时支持 两个 `|`(也就是一个 **READ** , 一个 **WRITE**)

然后我就发现运行类似于

```
ls | wc | sortgi
```

这样的命令时, 中间的 `wc` 死活接收不到输入, 就在那里干等.

我试了很久, 终于在深度了解 `pipe()` 的使用后, 找到了原因.

具体来说, `pipe()` 中的信息属于 "阅后即焚" 类型的, 我重点排查 **没有使用**

```
close(pipe[WRITE]);
```

关闭的进程, 发现运行 `wc` 的父进程恰好没有 `close()`, 从而顺利解决问题.

额外实现的功能

> **Hint:** 本部分功能在开启`#define DEMO`, 重新编译
>
> ```
> gcc -fsanitize=address -g -o nsh nsh.c
> ```
>
> 后才会启动, 也可以根据 Makefile 构建.

语法树分析

在使用 `getline()` 接受输入后, 使用 `excute()` 函数将读取字符串,
先进行 **正则表达式** 匹配, 得到除去空格类型 `TK_NOTYPE` 的
`tokens`.

> 在正则匹配的过程中, 支持对常见的错误类型 (如 `||`, `;;` 等)
> 进行匹配, 类型为 `TK_ERROR`.

然后进行形式审查 `format_check()` 之后, 使用 `work()` 函数

```
int work(Token *tokens, int begin, int end, int ground, int pipes[2], int rw,
int pipe2[2], int rw2)
```

进行递归的语法树分析, 并在递归结尾的情况调用 `cmd_run()` 函数进行
内部和外部指令的分发, 对外部指令调用 `run()` 执行.

回到递归分析的过程中, 首先根据不同符号的 "运算" 优先级进行对应的处理,
就像 ICS 里面的表达式求值一样, 将之前的 `*`, `+` 转化为 `|` 和 `&` 等.

- `&` 优先级最高
- `|`, `;`, `>` 优先级次之

因此在处理的时候要首先使用 `get_first_op()`, `get_second_op()` 两个函数
找到 `|`, `;`, `>` 的下标,

从而能够将长式子划分为短的子式子.如此递归往复.

语法高亮

基于上述的语法树分析, 我们可以利用语法树来高亮代码.

在构建 `nsh-demo` 中, 我们使用 `print_back()` 函数来返回语法高亮后的输入.

> 本来是想在输入的时候即时反馈, 像我常用的 `fish` 一样的,
> 后来发现读取输入的 `getline()` 函数并不支持这么做, 只能搞成输入后打印高亮的结果.
>
> 因本人主观原因 (时间仓促)以及 客观原因 (OJ 不支持 `-lreadline` 编译选项), 最后就选择使用
> `getline()` 读取输入

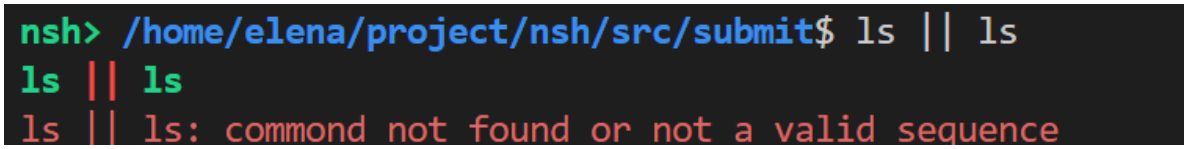
- 对输入进行语法高亮

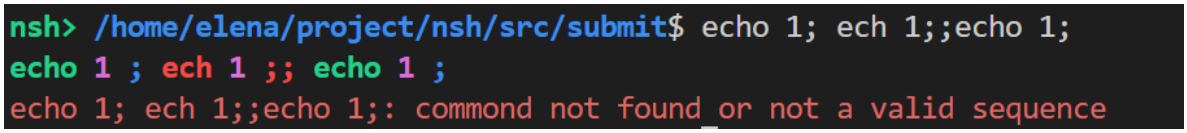
```
nsh> /home/elena/project/nsh/src/submit$ du -ha | sort -rh | head -n 1
du -ha | sort -rh | head -n 1
508K    .
```

```
nsh> /home/elena/project/nsh/src/submit$ du -sh .
du -sh .
508K    .
```

```
nsh> /home/elena/project/nsh/src/submit$ gcc -fsanitize=address -g -DDEMO -o nsh-demo
nsh.c
gcc -fsanitize=address -g -DDEMO -o nsh-demo nsh.c
```

- 非法输入 warning





Acknowledge

通过本次实验, 我对系统调用和 shell 有了更深入的认识, 并且~~毫无意外的掉入了 **并发** 的大坑(bushi)~~,

本次实验全部代码都是本人编写的(~~因此有点屎山~~), 下列链接中的内容在实验过程中提供了很多参考:

- OS lab1 提供了实验的要求以及基本设计思路
- CSAPP 上的 shell lab 提供了一些设计 shell 的基本思路
- jyy老师的精彩讲解 提供了对输入进行语法树分析的一些启发
- ICS pa 的一些启发 一些代码编写的 trick (比如使用 宏, 写 Makefile), 以及表达式求值部分的一些启发