Shell Lab Report

胡译文
2021201719
March 28, 2023

Solutions

在本次实验中，我们实现了命令行解析和执行函数 `eval()`、`builtin_cmd()`、`do_bgfg()`、`waitfg()` 等，以及信号处理函数 `sigchld_handler()`、`sigtstp_handler()`、`sigint_handler()` 等。同时，我们还拓展了一些原有函数的功能或返回值，并引入了 `csapp` 库中的一些函数。

`eval()`, `builtin_cmd()` and `do_bgfg()`

`eval()` 函数的功能是解析命令行输入（`cmdline`），主要由三个步骤组成：

1. **解析**。调用 `parsline()` 将 `cmdline` 解析成以参数分割的二维数组 `argv`，并得到进程状态（前台或后台）。
2. **Fork**。使用安全的 `Fork()` 函数复制当前进程，分别处理子进程（执行命令）和父进程（记录新进程）。我们会暂时阻断 `SIGCHLD` 信号。
3. **后处理**。根据进程状态后处理。

在第一步解析中，如果发现 `argv[0]` 是一个内置命令，那么**将调用转发给 `builtin_cmd()` 处理**。该函数处理的内容很简单，即将命令转化为对应的函数调用。比如当 `strcmp(argv[0], "fg") == 0` 时调用 `do_bgfg()` 函数输出。

`do_bgfg()` 函数处理了前后台进程的转换，也主要分为三步：

1. **合法性检查**。首先检查传入的 `argv` 是否合法，包括参数数量、类型等。
2. **获取 `job_t`**。将 `jid` 或 `pid` 转化为 `job_t`，如果没有找到则报错。
3. **状态转换**。在最后一步中，改变 `job_t` 中的状态，并发送 `SIGCONT` 继续信号。

Signal handler

这一部分主要实现了三种信号的处理函数：`sigint_handler()` 对应键盘信号 `ctrl-c`，`sigtstp_handler()` 处理键盘信号 `ctrl-z`，而 `sigchld_handler()` 对应 `Terminated` 或 `Stopped` 状态。

我们先来看最复杂的 `sigchld_handler()` 的情况：退出有三种情况，除了从主程序返回以外的两种情况都会以 `SIGCHLD` 信号的形式被捕捉；当子进程收到 `SIGSTP` 信号时会进入 `STOP` 状态，也会被捕捉到。

针对正常退出（`exit`），我们只需删除该进程；对于接受到终止信号退出（terminated），我们不仅需要删除进程，还需要在 shell 中显示退出进程和信号；而接受信号停止（stop）只需要修改 `job_t` 中的状态即可，后续的操作会通过读取该状态实现。

值得注意的是，由于 `job_t` 是一个结构体，根据安全的信号处理原则，我们在处理时会暂时阻断所有信号：

```
sigset_t mask_all, prev_all;
Sigfillset(&mask_all);
Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
getjobpid(jobs, _pid)->state = ST;
Sigprocmask(SIG_SETMASK, &prev_all, NULL);
Sputjob(_jid, _pid);
sio_puts(" stopped by signal ");
```

```
sio_putl(WSTOPSIG(status));
sio_puts("\n");
```

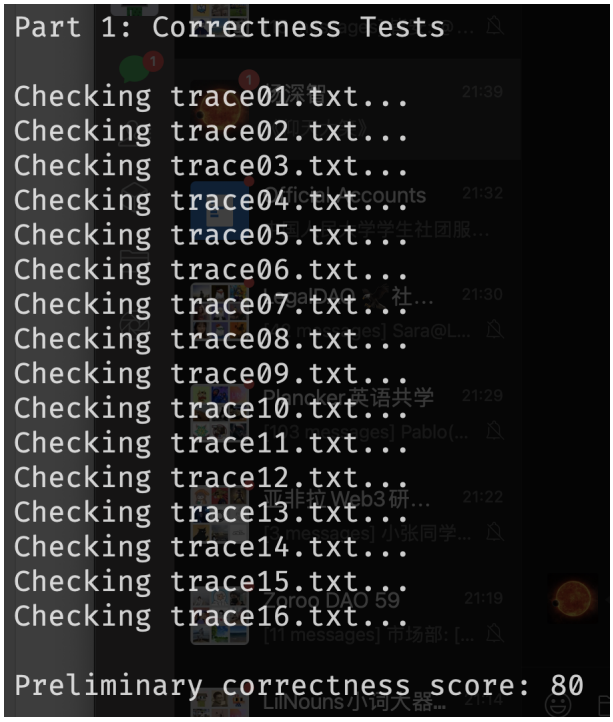
`waitfg()`

最后等待前台的操作很容易实现。注意由于前后台转化是 shell 内部的机制，并没有对应的信号表示，因此我们使用 `sleep()` 手动阻塞。

```
void waitfg(pid_t pid) {
    struct job_t* _job = getjobpid(jobs, pid);
    while (_job->state == FG) {
        sleep(1);
    }
    Vprintf("waitfg: Process (%d) no longer the fg process\n", (int) pid);
    return;
}
```

Results

我们的测试结果如下：



Appendix

Helper Function

我们实现了一些帮助函数/宏，比如用于 `handler` 中安全地输出：

```
#define Sputjob(jid,pid) sio_puts("Job ["); \
                         sio_putl(jid); \
                         sio_puts("] ("); \
                         sio_putl(pid); \
                         sio_puts(")") \
// end of Sputjob
```

我们还使用了 `csapp.c` 中的部分帮助函数来实现这些安全的输出：

```
static void sio_reverse(char s[]);
static void sio_ltoa(long v, char s[], int b);
static size_t sio_strlen(char s[]);
ssize_t sio_puts(char s[]);
ssize_t sio_putl(long v);
void sio_error(char s[]);
ssize_t Sio_putl(long v);
ssize_t Sio_puts(char s[]);
```

Expansion

实现管道和 I/O 重定向有比较清晰的方法:

1. 拓展 `parseline()` 读取 I/O 重定向符号和管道符。
2. 首先实现 I/O 重定向，即将所有的 `printf()` 改为 `fprintf()` 并且将 `fprintf()` 和 `write()` 中的源改为打开的文件。最后关闭文件。
3. 管道符可以在重定向的基础上实现。分别创建两个进程，并将第一个进程的输出重定向至一个零食文件，并将第二个进程的输入重定向为该文件。