ShellLab 报告

姓名：卢虹宇
学号：2023202269

Part A: 思路简述

1.核心方法流程

依次实现了内置命令、外部命令、IO重定向、pipe管道、eval_script，最后添加了history功能，值得一提的是：实现了交互性程序（如gdb等等）的运行、IO重定向实现了追加输出、pipe管道支持多级管道链、history支持用户选择查看几条指令以及选择指令执行的功能。

一些测试的截图

- pipe管道测试

```
(base) root@LAPTOP-IFFR0KNH:/home/Courses/大二下/ICS2/2025-ruc-ics2-shelllab-whiteman333# ./tsh
tsh> /bin/echo "hello world" | /bin/grep "hello"
hello world
tsh> /bin/ps aux | /bin/grep "bash"
root       77643  0.0  0.0   8240  4516 pts/6    Ss   21:44   0:00 /bin/bash --init-file /root/.cursor
-server/bin/82ef0f61c01d079d1b7e5ab04d88499d5af500e0/out/vs/workbench/contrib/terminal/common/scripts/
shellIntegration-bash.sh
tsh> /bin/echo -e "line1\nline2\nline3" | /bin/wc -l
3
```

- history功能测试
   历史功能测试截图

- 标准测试

Report results

```
▶ Run panjd123/autograding-grading-reporter@v1
Processing: tests
tests
Test code:
python3 grader.py --write-result

Total points for tests: 100.00/100

Test runner summary
┌──────────────────┬────────────┬─────────────┐
| Test Runner Name | Test Score | Max Score   |
├──────────────────┼────────────┼─────────────┤
| tests            | 100        | 100         |
├──────────────────┼────────────┼─────────────┤
| Total:           | 100        | 100         |
└──────────────────┴────────────┴─────────────┘
🏆 Grand total tests passed: 1/1

Workflow Run Response: https://api.github.com/repos/RUCICS/2025-ruc-ics2-shelllab-whiteman333/check-suites/36014878158
```

2.关键数据结构

整个框架中比较重要的数据结构有很多，比如`job`、`command_t`等等,自己实现的数据结构只有`history`结构体比较重要，通过维护一个全局变量`hist`来实现查看历史指令，相关代码添加到了`shell.h`和`history.c`中。

Part B: 具体实现分析

命令解析与执行

<!-- 300字以内,描述:

1. 如何处理不同类型的命令（内建命令/外部命令）
2. 如何实现作业控制（前台/后台）
3. 实现中的关键优化
4. 关键的错误处理，一些边界情况与 sanity check
   -->

1.处理不同类型的命令（内建命令/外部命令）

在`eval`中首先判断命令是否是内建命令，如果是则在shell主进程中处理，反之则`fork`子进程在子进程中调用`exevc`运行可执行文件。

2.实现前台/后台控制

在`fork`出子进程后，父进程通过`bg`变量判断命令是否是前台程序，如果是前台程序则通过`tcsetpgrp`函数转交中断控制权，并等待前台程序结束，最后依然通过`tcsetpgrp`函数收回终端控制权；如果是后台程序则打印一条运行消息后继续等待下一条终端指令。

3.实现中的关键优化

1实现了一个`write_format`函数，可以异步安全地输出格式化字符串。
2`waitfg`
3控制权移交中的信号屏蔽

4.关键错误处理

对`parse_command_line`返回值进行检查，处理命令解析错误。
所有系统调用（`fork`、`pipe`、`dup2`等）后检查错误状态，确保资源正确分配。
内置命令如`cd`、`kill`等实现了参数有效性检查，防止无效输入。
处理`exec`函数失败的情况，打印错误信息并正确退出子进程。

信号处理

<!-- 300字以内,描述:

1. 支持的信号类型
2. 信号处理方法
3. 关键的错误处理
   -->

1.支持的信号类型

· `SIGINT` 终止前台程序组
· `SIGTSTP` 暂停前台程序组
· `SIGQUIT` 杀死所有子程序并退出shell
· `SIGCHLD` 回收子进程并打印子进程终止原因

2.信号处理方法

`SIGCHLD`处理器使用`waitpid`循环检查所有结束的子进程，根据其退出状态更新作业列表。
`SIGINT`/`SIGTSTP`处理器通过`fg_pid`获取前台进程组PID，将信号转发给整个进程组。
信号处理过程中使用信号屏蔽（`sigprocmask`）防止竞态条件。
`SIGQUIT`处理器遍历作业列表，终止所有进程并等待它们结束，确保干净退出。

3.关键错误处理

- 在`SIGCHLD`中为了防止可能同时有多个子进程结束的情况用`while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)`来回收子进程，并统一判断和打印子进程的停止原因。
- 保存和恢复`errno`值，确保信号处理不干扰主程序流程。
- 处理`waitpid`可能的错误，区分`ECHILD`和其他错误情况。
- 回收完所有子进程后再退出shell，避免内存泄露。

I/O重定向与管道

<!-- 300字以内,描述:

1. 如何实现输入输出重定向
2. 管道的实现方法
3. 对齐等特殊处理
   -->

1.如何实现输入输出重定向

输入重定向通过检查`command_t`结构体中的`infile`字段实现。如果存在，则使用`open`函数以只读模式打开文件，然后用`dup2`将标准输入（`STDIN_FILENO`）重定向到该文件描述符。输出重定向则检查`outfile`字段，根据`append`标志决定是覆盖写入还是追加写入。使用适当的`flags`打开文件后，用`dup2`将标准输出（`STDOUT_FILENO`）重定向到文件。重定向操作在子进程执行命令前完成，并确保资源正确关闭。

2.管道的实现方法

管道实现使用`pipe()`系统调用创建管道，获取读写端文件描述符。采用循环处理命令链表的方式支持多级管道。对每个命令，判断其在管道中的位置：如有前一命令，将标准输入重定向到前一个管道的读端；如有下一命令，创建新管道并将标准输出重定向到写端。在每次迭代中，关闭已用管道并保存当前管道供下次使用。这种设计允许数据从一个命令流向下一个命令，形成完整的管道链。

3.对齐等特殊处理

为确保IO重定向与管道正确结合，实现了以下特殊处理：

- 在子进程中及时关闭所有不需要的管道文件描述符，避免读取或写阻塞。
- 使用`saved_stdout`和`saved_stdin`保存shell原始的标准输入输出，确保命令执行后能正确恢复。
- 处理多个命令时，仅对第一个命令处理输入重定向，仅对最后一个命令处理输出重定向。
- 使用`prev_pipe`和`current_pipe`数组跟踪管道状态，实现复杂的多级管道结构。

Part C: 关键难点解决

<!-- 选择2-3个最有技术含量的难点:

1. 具体难点描述
2. 你的解决方案
3. 方案的效果
   示例难点:

- 作业控制实现
- 信号处理的边界情况
- 管道与重定向的结合
  -->

难点1：多级管道的实现

难点描述

实现支持任意数量命令的管道链（如`cmd1 | cmd2 | cmd3 | ...`）是一个挑战，需要正确管理多个管道和文件描述符。

解决方案

- 采用迭代处理命令链表的方式，为每对相邻命令创建一个管道。
- 使用`prev_pipe`和`current_pipe`数组跟踪当前和上一个管道的文件描述符。
- 在处理每个命令时，根据其位置（是否有前序/后续命令）决定重定向设置。
- 确保在适当的时机关闭管道文件描述符，避免资源泄露或管道堵塞，比如在只读取管道数据的子进程中需要及时把写端口关闭。

效果

成功实现了支持任意长度的命令管道链，如`cmd1 | cmd2 | cmd3 | cmd4`，数据能够正确地从一个命令流向下一个命令。

难点2：前台作业控制与终端管理

难点描述

实现前台作业控制需要正确管理进程组和终端控制权，并且在处理终端控制权时总是会遇到异常信号，查阅资料后解决。

解决方案

- 使用`setpgid`创建新的进程组，确保子进程在自己的进程组中。
- 对前台作业使用`tcsetpgrp`转移终端控制权，让子进程能直接接收终端信号。
- 实现`waitfg`函数等待前台进程完成，避免竞态条件。
- 使用`SIGTTOU`信号控制处理终端控制权转移过程，由于在收回终端控制权时shell主进程已经不是前台程序，非前台程序调用`tcsetpgrp`函数会给所有后台进程发送`SIGTTOU`信号，造成异常，所以在收回进程时需要屏蔽该信号。

效果

Shell能够正确运行交互式程序如`gdb`、`vim`等，用户可以直接与这些程序交互，`Ctrl+C`、`Ctrl+Z`等控制命令能够正确地作用于前台进程。

难点3：历史命令功能实现

难点描述

需要设计数据结构存储历史命令，并实现各种历史命令访问方式。

解决方案

- 设计`History`结构体存储命令历史，包含命令数组、当前大小和容量。
- 实现`add_history`函数添加新命令，处理重复命令、换行符和数组溢出。
- 实现`history`内置命令显示历史记录，支持显示指定数量的历史。
- 实现`!`前缀命令执行历史记录，包括`!!`（上一条）、`!n`（第n条）和`!prefix`（前缀匹配）。
- 在`shell_loop`中集成历史记录功能。

效果

用户可以方便地查看历史命令，使用简洁的语法重新执行之前的命令，大大提高了Shell的易用性

Part D: 实验反馈

<!-- 你的反馈对我们至关重要
可以从实验设计，实验文档，框架代码三个方面进行反馈，具体衡量：

1. 实验设计：实验难度是否合适，实验工作量是否合理，是否让你更加理解Shell，Shell够不够有趣
2. 实验文档：文档是否清晰，哪些地方需要补充说明
3. 框架代码：框架代码是否易于理解，接口设计是否合理，实验中遇到的框架代码的问题（请引用在 repo 中你提出的 issue）
   -->

- 实验设计：比较合理，完成作业时间倒是足够但是感觉要做额外探索的话感觉时间比较紧张。
- 实验文档：个人感觉写的已经很清晰了
- 框架代码：大体上是CSAPP的原版代码，框架肯定没问题，但是测试的时候貌似bug很多，而且由于我应该算是最先开始做这个lab的一批人，被测试的一些bug困扰了很久，不过我的困扰能帮助各位发现bug减少后面同学完成这个lab的时间也是值得的。至于具体有哪些bug,我只开了一个issue,其他都是单独跟彭文博师兄交流的，其中主要是检查程序输出是否匹配时测试程序会发疯，至于师兄具体怎么解决的不太了解，不过估计是缓冲区之类的问题，还有就是测试的脚本中比如`/bin/echo`只写了个`/echo`之类的。

总体来讲这个lab还是很好玩，能自己实现一个shell来使用，体验感如果没有测试的bug的话会更好，由衷地感谢各位助教师兄！！

参考资料

1. [tcgetpgrp(3) — Linux manual page] - [https://www.man7.org/linux/man-pages/man3/tcsetpgrp.3.html] - [SIGTOUT的出现原因]