

操作系统专题实践 - 实验2

71123329 段嘉文

shell的实现

操作系统专题实践 - 实验2

shell的实现

实验目的

实验内容

具体要求

设计思路

主要数据结构及其说明

编写shell部分代码

1. 数据结构定义

2. 宏定义与常量

3. 工具函数模块实现

4. 命令执行模块实现

5. 主程序实现

6. 编译配置

编写测试程序及测试

测试环境准备

基本命令测试

内置命令测试

输出重定向测试

输入重定向测试

单级管道测试

多级管道测试

管道与重定向组合测试

批处理模式测试

错误处理测试

实验体会

实验目的

通过本次实验，深入理解操作系统中Shell命令行解释器的工作原理，掌握进程创建（`fork`）、进程替换（`exec`）、管道（`pipe`）、文件重定向等核心系统调用的使用方法。通过从零开始实现一个功能完备的Shell程序，提升对操作系统进程管理、进程间通信、文件系统接口等机制的理解，增强系统级编程能力和问题分析能力。

实验内容

1. 设计并实现一个简单的Shell命令行解释器，支持交互模式和批处理模式两种运行方式。
2. 实现基本的命令解析功能，能够正确识别命令、参数以及特殊符号（如管道、重定向符号等）。
3. 实现进程创建与执行机制，通过fork和exec系统调用族执行外部命令程序。
4. 实现输入输出重定向功能，支持标准输入重定向（<）、标准输出重定向（>）以及追加输出（>>）。
5. 实现管道功能，支持单级管道和多级管道链，实现进程间的数据流传递。
6. 实现内置命令（built-in commands），如cd（切换目录）、exit（退出Shell）、path（设置搜索路径）、help（显示帮助信息）等。
7. 编写测试脚本，验证Shell各项功能的正确性和稳定性。

具体要求

- Shell应支持交互模式：启动后显示提示符（如user@host:path\$），等待用户输入命令并执行，执行完毕后继续显示提示符等待下一条命令。
- Shell应支持批处理模式：接收一个文本文件作为参数，按顺序执行文件中的每一条命令，执行完毕后自动退出。
- 实现完整的命令解析逻辑，正确处理命令、参数、空格、制表符等字符，支持任意数量的命令参数。
- 实现输入重定向（<）：从指定文件读取数据作为命令的标准输入。
- 实现输出重定向（>）：将命令的标准输出写入指定文件，若文件存在则覆盖。
- 实现追加输出（>>）：将命令的标准输出追加到指定文件末尾，若文件不存在则创建。
- 实现单级管道（|）：将前一个命令的标准输出作为后一个命令的标准输入。
- 实现多级管道链：支持多个命令通过管道串联执行（如cmd1 | cmd2 | cmd3）。
- 实现管道与重定向的组合使用（如ls | grep txt > result.txt）。
- 内置命令必须由Shell进程自身执行，不能通过fork子进程的方式实现。
- cd命令应能正确切换当前工作目录，并更新环境变量。
- path命令应能动态修改可执行文件的搜索路径。
- 实现基本的错误处理机制，对于命令不存在、文件无法打开、权限不足等错误给出明确的提示信息。
- 代码应具有良好的模块化结构，将不同功能封装成独立的函数或模块。
- 编写完整的测试用例，验证各项功能在不同场景下的正确性。

设计思路

本实验的设计基于Unix Shell的经典实现思路，采用模块化设计，将命令解析、进程执行、管道处理、重定向等功能分离成独立模块，便于开发、调试和维护。

1. 整体架构设计

Shell的核心工作流程可以概括为"读取-解析-执行"三个阶段：

- **读取阶段：**从标准输入或批处理文件中读取一行命令文本。
- **解析阶段：**对命令文本进行词法分析，识别命令、参数、管道符、重定向符号等元素。
- **执行阶段：**根据解析结果，通过系统调用创建进程、设置管道和重定向、执行目标程序。

为了支持交互模式和批处理模式，主程序采用统一的循环结构：检测命令来源（键盘或文件），读取命令行，调用命令执行模块，等待执行完成后继续下一轮循环。当遇到EOF（批处理文件结束或用户按Ctrl-D）或exit命令时退出循环。

2. 环境管理策略

定义全局的环境结构体（`Environment`），用于维护Shell的运行时状态。在Shell启动时，通过`init_environment()`函数初始化环境结构体，读取系统PATH环境变量并解析成字符串数组，同时获取当前工作目录。这种设计使得环境状态在整个Shell生命周期内保持一致，并可被各模块共享访问。

3. 命令解析设计

命令解析是Shell的核心模块之一，负责将用户输入的字符串转换为可执行的数据结构。解析流程分为以下几个步骤：

- (1) **管道分割：**首先检测命令行中是否存在管道符|，若存在则按管道符分割成多个子命令，每个子命令单独解析。
- (2) **重定向识别：**在每个子命令中查找<、>、>>符号，提取输入输出文件名，并记录重定向类型。为了正确处理>>追加重定向，需要检测连续的两个>符号。
- (3) **命令与参数提取：**去除重定向部分后，剩余的字符串按空格和制表符分割，第一个词作为命令名，后续词作为参数。使用`strtok()`函数进行分词，并将结果存储在动态分配的字符串数组中。
- (4) **可执行文件查找：**对于非内置命令，需要在PATH路径中查找对应的可执行文件。若命令包含/则视为绝对路径或相对路径直接使用，否则遍历PATH中的每个目录，拼接命令名并检查文件是否存在且可执行。

这种分层解析的方式使得每个步骤的职责明确，便于处理复杂的命令组合（如管道+重定向）。

4. 进程执行模型

Shell执行外部命令采用经典的fork-exec模型：

- 父进程（**Shell本身**）：调用 `fork()` 创建子进程，然后调用 `waitpid()` 等待子进程执行完毕，期间保持阻塞状态。
- 子进程：在 `fork` 后立即设置重定向（通过 `dup2()` 修改标准输入输出的文件描述符），然后调用 `execv()` 替换为目标程序。若 `execv` 失败则输出错误信息并退出。

对于内置命令（如 `cd`、`exit`、`path`），不能通过 `fork` 子进程的方式执行，因为子进程的环境修改无法影响父进程。因此在解析阶段先判断是否为内置命令，若是则直接在当前进程中执行相应操作并返回，不进入 `fork-exec` 流程。

5. 管道实现机制

管道是进程间通信的重要手段，本实验采用匿名管道（`pipe()`）实现命令之间的数据流传递。对于包含 n 个命令的管道链，需要创建 $n-1$ 个管道，每个管道包含读端和写端两个文件描述符。

管道的连接遵循以下规则：

- 第一个命令：标准输出重定向到第一个管道的写端，标准输入保持不变（或来自输入重定向）。
- 中间命令：标准输入来自前一个管道的读端，标准输出重定向到下一个管道的写端。
- 最后一个命令：标准输入来自最后一个管道的读端，标准输出保持不变（或写入输出重定向）。

在所有子进程创建完毕后，父进程必须关闭所有管道的文件描述符，否则管道读端会一直等待数据导致进程阻塞。同时，父进程需要按顺序等待所有子进程结束，确保命令执行完毕。

6. 重定向实现机制

重定向通过修改进程的标准输入输出文件描述符实现：

- 输入重定向（`<`）：在子进程中调用 `open(filename, O_RDONLY)` 打开输入文件，然后通过 `dup2(fd, STDIN_FILENO)` 将标准输入重定向到该文件。
- 输出重定向（`>`）：调用 `open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644)` 以覆盖模式打开输出文件，通过 `dup2(fd, STDOUT_FILENO)` 重定向标准输出。
- 追加输出（`>>`）：使用 `O_APPEND` 标志代替 `O_TRUNC`，使得输出内容追加到文件末尾而不是覆盖原有内容。

重定向操作必须在 `execv()` 之前完成，因为 `exec` 会替换整个进程空间，之后的代码不会被执行。同时，重定向可以与管道组合使用，此时需要先设置管道连接，再设置重定向，确保文件描述符的优先级正确。

7. 错误处理与用户体验

为了提升Shell的健壮性和用户体验，设计了以下错误处理机制：

- **命令不存在：**当在PATH中找不到可执行文件时，输出**Command not found**错误信息。
- **文件操作失败：**当打开重定向文件失败时，通过**perror()**输出系统错误信息（如权限不足、文件不存在等）。
- **系统调用失败：**对所有关键系统调用（`fork`、`pipe`、`dup2`、`execv`等）进行返回值检查，失败时输出错误并清理资源。
- **内存分配失败：**对**malloc**、**strdup**等内存分配函数进行检查，防止空指针导致段错误。

此外，Shell在交互模式下会显示彩色提示符，包含用户名、主机名、当前路径等信息，提升用户体验。在批处理模式下则不显示提示符，避免干扰输出结果。

8. 模块化设计

整个Shell程序按功能划分为以下模块：

- **structures.h:** 定义核心数据结构（`Process`、`Environment`、`RedirectionType`）。
- **macros.h:** 定义常量、缓冲区大小、错误消息宏。
- **utilities.h/c:** 实现环境初始化、PATH解析、可执行文件查找、字符串处理等工具函数。
- **run_command.h/c:** 实现命令执行的核心逻辑，包括内置命令处理、重定向处理、管道处理、进程执行等。
- **myshell.c:** 主程序入口，控制交互循环和批处理流程。

这种模块化设计使得各部分职责清晰，代码易于理解和维护，同时也便于后续功能扩展（如添加新的内置命令、支持后台执行等）。

主要数据结构及其说明

本实验主要涉及以下核心数据结构的设计与实现：

1. `RedirectionType` 枚举类型

定义在**structures.h**中，用于标识进程的重定向类型：

```
typedef enum {
    NO_REDI = 0,           // 无重定向
    REDI_IN = 1,           // 仅输入重定向 <
    REDI_OUT = 2,          // 仅输出重定向 > 或 >>
    REDI_BOTH = 3          // 同时有输入输出重定向
} RedirectionType;
```

枚举值说明：

- **NO_REDIO**: 进程不需要任何重定向，标准输入输出保持默认状态（继承自Shell）。
- **REDI_IN**: 进程需要从文件读取输入，标准输入被重定向到指定文件。
- **REDI_OUT**: 进程需要将输出写入文件，标准输出被重定向到指定文件。
- **REDI_BOTH**: 进程同时需要输入输出重定向，如`cmd < input.txt > output.txt`。

该枚举类型简化了重定向处理逻辑，避免使用多个布尔标志位，使代码更加清晰。

2. `Process` 结构体

定义在`structures.h`中，描述一个待执行的进程及其相关信息：

```
typedef struct {
    pid_t pid;                                // 进程ID (fork后由父进程记录)
    int argc;                                 // 参数个数 (包括命令本身)
    char **argv;                               // 参数数组，以NULL结尾
    char *exec_path;                           // 可执行文件的完整路径
    RedirectionType redir_type;               // 重定向类型
    int redir_infd;                            // 输入重定向的文件描述符
    int redir_outfd;                           // 输出重定向的文件描述符
} Process;
```

字段说明：

- **pid**: 进程ID，在`fork`成功后由父进程设置，用于后续的`waitpid`等待。初始化为-1表示未创建。
- **argc**: 参数个数，包括命令本身。例如`ls -l /tmp`的`argc`为3。
- **argv**: 参数数组，动态分配的字符串指针数组，第一个元素为命令名，最后一个元素为NULL（`execv`要求）。
- **exec_path**: 可执行文件的完整路径，由`find_executable()`函数在PATH中查找得到。对于内置命令该字段为NULL。
- **redir_type**: 标识该进程的重定向类型，决定是否需要设置文件描述符。
- **redir_infd**: 输入重定向的文件描述符，初始化为-1表示无输入重定向。若需要重定向则通过`open()`打开文件获得。
- **redir_outfd**: 输出重定向的文件描述符，初始化为-1表示无输出重定向。对于`>`使用`O_TRUNC`标志（覆盖），对于`>>`使用`O_APPEND`标志（追加）。

该结构体封装了进程执行所需的全部信息，便于在函数间传递和管理。

3. `Environment` 结构体

定义在 `structures.h` 中，维护 Shell 的全局运行环境：

```
typedef struct {
    char **paths;           // PATH环境变量解析后的目录数组
    char cwd[1024];         // 当前工作目录的绝对路径
    bool path_set;          // 用户是否通过path命令修改过PATH
    int path_count;         // PATH中的目录数量
} Environment;
```

字段说明：

- **paths**: 动态分配的字符串指针数组，每个元素指向一个目录路径（如 `"/bin"`、`"/usr/bin"`）。在 Shell 启动时从系统 PATH 环境变量解析得到，可通过 path 内置命令动态修改。
- **cwd**: 当前工作目录的绝对路径，最大长度 1024 字节。由 `getcwd()` 获取，在 cd 命令执行后更新。用于提示符显示和相对路径解析。
- **path_set**: 布尔标志，记录用户是否执行过 path 命令。若为 `false` 则使用系统默认 PATH，若为 `true` 则使用用户自定义的路径列表。
- **path_count**: paths 数组的有效元素个数，用于遍历 PATH 查找可执行文件。

该结构体在 Shell 启动时初始化一次，在整个生命周期内保持存在，各模块通过指针访问和修改其内容。

4. 数据结构之间的关系

各数据结构在 Shell 执行流程中的协作关系如下：

1. Shell 启动阶段：

- 创建 `Environment` 结构体，调用 `init_environment()` 初始化。
- 解析系统 PATH 环境变量，填充 `paths` 数组和 `path_count`。
- 获取当前工作目录，填充 `cwd` 字段。

2. 命令解析阶段：

- 为每个子命令创建 `Process` 结构体，调用 `init_process()` 初始化所有字段。
- 调用 `handle_redirection()` 解析命令、参数、重定向符号，填充 `argv`、`argc`、`redir_type`、`redir_infd`、`redir_outfd` 等字段。
- 若为外部命令，调用 `find_executable()` 在 `Environment.paths` 中查找可执行文件，将结果存入 `exec_path`。

3. 命令执行阶段：

- 若为内置命令（通过 `handle_builtin_command()` 判断），直接在当前进程中修改 `Environment` 结构体（如 `cd` 修改 `cwd`，`path` 修改 `paths`）。

- 若为外部命令，调用 `fork()` 创建子进程，将返回的PID存入 `Process.pid`。
- 子进程根据 `redir_type` 设置重定向：若 `redir_infd >= 0` 则调用 `dup2(redir_infd, STDIN_FILENO)`，若 `redir_outfd >= 0` 则调用 `dup2(redir_outfd, STDOUT_FILENO)`。
- 子进程调用 `execv(exec_path, argv)` 执行目标程序。
- 父进程调用 `waitpid(Process.pid, NULL, 0)` 等待子进程结束。

4. 管道处理阶段：

- 对于包含管道的命令，创建 `Process` 数组，每个元素对应一个子命令。
- 创建管道数组 `int pipes[n-1][2]`，通过 `pipe()` 初始化。
- 每个子进程根据其在管道链中的位置，通过 `dup2()` 连接相应的管道读端或写端。
- 父进程关闭所有管道文件描述符，依次等待所有子进程结束。

5. 资源清理阶段：

- 每个 `Process` 执行完毕后，调用 `cleanup_process()` 释放 `argv` 数组、`exec_path` 字符串等动态分配的内存。
- 关闭打开的文件描述符（`redir_infd`、`redir_outfd`）。
- Shell退出前调用 `cleanup_environment()` 释放 `Environment.paths` 数组。

这种数据结构设计将Shell的状态信息和进程执行信息清晰分离，既保证了功能的完整性，又便于内存管理和错误处理。通过结构体封装，各函数的参数传递更加简洁，代码的可读性和可维护性得到显著提升。

编写shell部分代码

1. 数据结构定义

在 `structures.h` 中定义了 Shell 程序的核心数据结构，包括重定向类型枚举 `RedirectionType`、进程描述结构体 `Process` 以及全局环境结构体 `Environment`。这些数据结构为后续的命令解析、进程管理和环境维护提供了基础支撑。

```

shell_lab > C structures.h
1  #ifndef STRUCTURES_H
2  #define STRUCTURES_H
3
4  #include <stdbool.h>
5  #include <sys/types.h>
6
7  // 重定向类型枚举
8  typedef enum {
9      NO_REDIO = 0,           // 无重定向
10     REDIO_IN = 1,          // 输入重定向 <
11     REDIO_OUT = 2,         // 输出重定向 >
12     REDIO_BOTH = 3,        // 同时有输入输出重定向
13 } RedirectionType;
14
15 // 进程结构体: 存储单个命令的所有信息
16 typedef struct {
17     pid_t pid;             // 进程ID
18     int argc;              // 参数个数
19     char **argv;            // 参数数组 (包括命令本身)
20     char *exec_path;        // 可执行文件的完整路径
21     RedirectionType redir_type; // 重定向类型
22     int redir_infd;        // 输入重定向的文件描述符
23     int redir_outfd;        // 输出重定向的文件描述符
24 } Process;
25
26 // Shell环境结构体: 存储Shell运行环境信息
27 typedef struct {
28     char **paths;           // PATH环境变量 (查找可执行文件的路径列表)
29     char cwd[1024];          // 当前工作目录
30     bool path_set;          // 用户是否设置过PATH
31     int path_count;          // PATH路径的数量
32 } Environment;
33
34 #endif

```

2. 宏定义与常量

在 `macros.h` 中集中定义了程序中使用的各类常量，包括缓冲区大小（如 `MAX_LINE_LENGTH`、`MAX_ARGS`）、路径长度限制（`MAX_PATH_LENGTH`）、管道数量上限（`MAX_PIPES`）等。同时定义了ANSI颜色代码用于美化提示符输出，以及标准化的错误消息宏（如 `ERR_COMMAND_NOT_FOUND`、`ERR_FORK_FAILED` 等），使得错误处理更加统一和规范。

```

shell_lab > C macros.h
1  #ifndef MACROS_H
2  #define MACROS_H
3
4  // 缓冲区大小定义
5  #define MAX_LINE_LENGTH 1024           // 最大输入行长度
6  #define MAX_ARGS 64                  // 最大参数个数
7  #define MAX_PROCESSES 32             // 最大并行进程数
8  #define MAX_PIPES 16                 // 最大管道数
9  #define MAX_PATH_LENGTH 256           // 最大路径长度
10
11 // 颜色定义 (用于提示符)
12 #define COLOR_RESET   "\033[0m"
13 #define COLOR_GREEN   "\033[32m"
14 #define COLOR_BLUE    "\033[34m"
15 #define COLOR_CYAN   "\033[36m"
16 #define COLOR_RED    "\033[31m"
17
18 // 错误消息
19 #define ERR_COMMAND_NOT_FOUND "Command not found"
20 #define ERR_TOO_MANY_ARGS "Too many arguments"
21 #define ERR_PIPE_FAILED "Pipe creation failed"
22 #define ERR_FORK_FAILED "Fork failed"
23 #define ERR_FILE_OPEN_FAILED "Failed to open file"
24
25 #endif
26

```

3. 工具函数模块实现

(1) 环境初始化函数 `init_environment()`

在 `utilities.c` 中实现了 Shell 环境的初始化逻辑。该函数首先调用 `getcwd()` 获取当前工作目录并存储到 `Environment.cwd` 字段，然后调用 `parse_path()` 解析系统 PATH 环境变量，将其分割成独立的目录字符串数组存储在 `Environment.paths` 中，同时记录路径数量到 `path_count`。初始化时将 `path_set` 标志设为 `false`，表示使用系统默认 PATH。

(2) PATH 解析函数 `parse_path()`

该函数通过 `getenv("PATH")` 获取系统 PATH 环境变量，若不存在则设置默认路径 `/bin` 和 `/usr/bin`。对于有效的 PATH 字符串，先统计冒号分隔符数量确定路径个数，然后动态分配字符串指针数组，使用 `strtok()` 按冒号分割 PATH 并逐一复制到数组中。每次内存分配后都进行错误检查，确保程序的健壮性。

(3) 可执行文件查找函数 `find_executable()`

该函数实现了在 PATH 中查找可执行文件的核心逻辑。若命令名包含路径分隔符 `/`，则视为绝对路径或相对路径直接检查其可执行性；否则遍历 `Environment.paths` 中的每个目录，拼接命令名形成完整路径，通过 `stat()` 检查文件是否存在且具有可执行权限 (`S_IXUSR`)。找到匹配文件后返回其完整路径的副本，未找到则返回 `NULL`。

(4) 字符串处理函数 `trim_whitespace()`

该函数用于去除字符串首尾的空格和制表符。先从字符串开头跳过所有空白字符，再从末尾向前查找第一个非空白字符并截断，返回处理后的字符串指针。这一函数在命令解析过程中被广泛使用，确保参数提取的准确性。

(5) 提示符显示函数 `print_prompt()`

在交互模式下，该函数负责显示彩色提示符。通过 `getlogin()` 获取用户名，`gethostname()` 获取主机名，结合 `Environment.cwd` 中的当前路径，格式化输出形如 `user@host:/path$` 的提示符。使用 ANSI 颜色代码对用户名、主机名、路径进行不同颜色渲染，提升用户体验。

(6) 资源清理函数 `cleanup_environment()`

该函数负责释放 `Environment` 结构体中动态分配的内存资源。遍历 `paths` 数组，逐一释放每个路径字符串，最后释放数组本身。在 Shell 退出前调用此函数，防止内存泄漏。

```

C utilities.c
include "utilities.h"
include "macros.h"
include <stdio.h>
include <stdlib.h>
include <string.h>
include <unistd.h>
include <ctype.h>
include <sys/stat.h>
include <limits.h>

// 初始化Shell环境
void init_environment(Environment *env) {
    // 获取当前工作目录
    if (getcwd(env->cwd, sizeof(env->cwd)) == NULL) {
        strcpy(env->cwd, "/");
    }

    env->path_set = false;
    env->path_count = 0;
    env->paths = NULL;

    // 解析PATH环境变量
    parse_path(env);

    // 清理环境资源
    cleanup_environment(Environment *env) {
        if (env->paths) {
            for (int i = 0; i < env->path_count; i++) {
                free(env->paths[i]);
            }
            free(env->paths);
        }
    }
}

```

```

shell_lab > C utilities.c
36     // 去除字符串首尾空白
37     char *trim_whitespace(char *str) {
38         char *end;
39
40         // 去除前导空白
41         while (isspace((unsigned char)*str)) str++;
42
43         if (*str == 0) return str;
44
45         // 去除尾部空白
46         end = str + strlen(str) - 1;
47         while (end > str && isspace((unsigned char)*end)) end--;
48
49         *(end + 1) = '\0';
50         return str;
51     }

52     // 解析PATH环境变量
53     void parse_path(Environment *env) {
54         char *path_env = getenv("PATH");
55         if (!path_env) {
56             // 如果没有PATH环境变量，使用默认路径
57             env->path_count = 2;
58             env->paths = (char**)malloc(sizeof(char*) * 2);
59             if (!env->paths) {
60                 perror("malloc");
61                 exit(1);
62             }
63             env->paths[0] = strdup("/bin");
64             env->paths[1] = strdup("/usr/bin");
65             return;
66         }

67         // 计算路径数量
68         int count = 1;
69         for (char *p = path_env; *p; p++) {
70             if (*p == ':') count++;
71         }
72     }
73 }

```

```

shell_lab > C utilities.c
54     void parse_path(Environment *env) {
55         // 分配内存
56         env->paths = (char**)malloc(sizeof(char*) * 2);
57         if (!env->paths) {
58             perror("malloc");
59             exit(1);
60         }
61         env->path_count = 0;
62
63         // 复制并分割PATH
64         char *path_copy = strdup(path_env);
65         if (!path_copy) {
66             perror("strdup");
67             exit(1);
68         }

69         char *token = strtok(path_copy, ":");
70         while (token != NULL && env->path_count < 2) {
71             env->paths[env->path_count] = strdup(token);
72             if (env->paths[env->path_count]) {
73                 perror("strdup");
74                 exit(1);
75             }
76             env->path_count++;
77             token = strtok(NULL, ":");
78         }
79     }
80
81     free(path_copy);
82 }

```

```

shell_lab > C utilities.c
104    // 查找可执行文件
105    char *find_executable(const char *command, Environment *env) {
106        // 如果命令包含路径分隔符，直接检查
107        if (strchr(command, '/') != NULL) {
108            struct stat st;
109            if (stat(command, &st) == 0 && (st.st_mode & S_IXUSR)) {
110                return strdup(command);
111            }
112            return NULL;
113        }

114        // 在PATH中查找
115        for (int i = 0; i < env->path_count; i++) {
116            char full_path[MAX_PATH_LENGTH];
117            snprintf(full_path, sizeof(full_path), "%s/%s", env->paths[i], command);

118            struct stat st;
119            if (stat(full_path, &st) == 0 && (st.st_mode & S_IXUSR)) {
120                return strdup(full_path);
121            }
122        }

123        return NULL;
124    }

125    // 初始化进程结构
126    void init_process(Process *proc) {
127        proc->pid = 0;
128        proc->argc = 0;
129        proc->argv = NULL;
130        proc->exec_path = NULL;
131        proc->redir_type = NO_REDIO;
132        proc->redir_infd = -1;
133        proc->redir_outfd = -1;
134    }
135 }

```

```

shell_lab > C utilities.c
140    // 清理进程资源
141    void cleanup_process(Process *proc) {
142        if (proc->argv) {
143            for (int i = 0; i < proc->argc; i++) {
144                free(proc->argv[i]);
145            }
146            free(proc->argv);
147        }
148        if (proc->exec_path) {
149            free(proc->exec_path);
150        }
151        if (proc->redir_infd >= 0) {
152            close(proc->redir_infd);
153        }
154        if (proc->redir_outfd >= 0) {
155            close(proc->redir_outfd);
156        }
157    }

158    // 打印Shell提示符
159    void print_prompt(Environment *env) {
160        char hostname[256];
161        char *username = getenv("USER");

162        gethostname(hostname, sizeof(hostname));
163
164        // 格式: username@hostname:~/path$
165        printf(COLOR_GREEN "%s@%s" COLOR_RESET ":" COLOR_BLUE "%s" COLOR_RESET "$ ", 
166               username ? username : "user",
167               hostname,
168               env->cwd);
169        fflush(stdout);
170    }
171 }
172

```

4. 命令执行模块实现

(1) 进程初始化与清理

`init_process()` 函数初始化 `Process` 结构体的所有字段为默认值：`pid` 设为-1表示未创建，`argc` 设为0，`argv` 和 `exec_path` 设为NULL，`redir_type` 设为 `NO_REDIO`，文件描述符设为-1。`cleanup_process()` 函数释放进程相关的动态内存，包括 `argv` 数组中的每个字符串、数组本身以及 `exec_path`，并关闭打开的重定向文件描述符。

(2) 内置命令处理函数 `handle_builtin_command()`

该函数判断并执行Shell的内置命令。通过 `strcmp()` 依次检查命令是否为 `exit`、`cd`、`path` 或 `help`：

- **exit**命令：调用 `cleanup_environment()` 释放资源后执行 `exit(0)` 退出Shell。
- **cd**命令：若无参数则切换到用户主目录（通过 `getenv("HOME")` 获取），有参数则切换到指定目录。调用 `chdir()` 执行目录切换，成功后通过 `getcwd()` 更新 `Environment.cwd`。失败时使用 `perror()` 输出错误信息。
- **path**命令：先清理旧的 `paths` 数组，若无参数则清空PATH（`path_count` 设为 0），有参数则将所有参数作为新的搜索路径。将 `path_set` 标志设为 `true`，表示用户已自定义PATH。
- **help**命令：输出Shell支持的功能列表和内置命令说明。

若命令匹配内置命令则返回1（表示已处理），否则返回0（交由外部命令处理流程）。

(3) 重定向处理函数 `handle_redirection()`

该函数是命令解析的核心，负责识别和处理输入输出重定向。首先复制命令字符串以保护原始数据，然后按以下步骤解析：

1. **查找输入重定向**：在命令中搜索 `<` 符号，若找到则提取其后的文件名作为输入文件，调用 `open()` 以只读模式打开并将文件描述符存入 `redir_infd`。
2. **查找输出重定向**：搜索 `>` 符号，检查下一个字符是否也是 `>` 以区分覆盖（`>`）和追加（`>>`）模式。覆盖模式使用 `O_TRUNC` 标志，追加模式使用 `O_APPEND` 标志，调用 `open()` 打开文件并存入 `redir_outfd`。
3. **判断重定向类型**：根据输入输出文件是否存在，设置 `redir_type` 为 `NO_RED|`、`RED|IN`、`RED|OUT` 或 `RED|BOTH`。
4. **提取命令和参数**：去除重定向部分后，剩余字符串按空格和制表符分割，第一个词作为命令名，后续作为参数。动态分配 `argv` 数组并逐一复制参数字符串，数组末尾设为 `NULL` 以符合 `execv()` 要求。
5. **处理内置命令**：调用 `handle_builtin_command()` 判断是否为内置命令，若是则直接执行并返回1。
6. **查找可执行文件**：对于外部命令，调用 `find_executable()` 在 PATH 中查找完整路径，找到则存入 `exec_path`，否则输出 "Command not found" 错误并返回 -1。

(4) 进程执行函数 `execute_process()`

该函数通过 fork-exec 模型执行单个进程。调用 `fork()` 创建子进程，父进程记录子进程 PID 并返回。子进程中：

1. **设置输入重定向**：若 `redir_infd >= 0`，调用 `dup2(redir_infd, STDIN_FILENO)` 将标准输入重定向到文件，然后关闭原文件描述符。
2. **设置输出重定向**：若 `redir_outfd >= 0`，调用 `dup2(redir_outfd, STDOUT_FILENO)` 将标准输出重定向到文件，然后关闭原文件描述符。

3. 执行目标程序：调用 `execv(exec_path, argv)` 替换进程映像。若 `execv` 返回则说明执行失败，输出错误信息并调用 `exit(1)` 退出子进程。

(5) 管道处理函数 `handle_pipe()`

该函数处理包含管道符的命令。首先按 `|` 分割命令字符串，得到多个子命令。若只有一个子命令（无管道），则直接调用 `handle_redirection()` 和 `execute_process()` 执行。对于多命令管道：

1. 创建管道：根据命令数量创建 $n-1$ 个管道，每个管道调用 `pipe()` 生成读写两端的文件描述符。
2. 创建子进程：为每个子命令 `fork` 一个子进程。在子进程中：
 - 第一个命令：将标准输出重定向到第一个管道的写端。
 - 中间命令：将标准输入重定向到前一个管道的读端，标准输出重定向到后一个管道的写端。
 - 最后一个命令：将标准输入重定向到最后一个管道的读端。
 - 完成重定向后，关闭所有管道文件描述符（子进程不需要其他管道），然后调用 `execv()` 执行命令。
3. 父进程管理：父进程在所有子进程创建完毕后，关闭所有管道文件描述符（避免管道读端阻塞），然后按顺序调用 `waitpid()` 等待所有子进程结束。
4. 资源清理：执行完毕后释放每个 `Process` 结构体的内存。

(6) 主命令执行函数 `run_command()`

该函数是命令执行的入口，接收原始命令字符串和环境结构体。首先去除命令首尾空格，检查是否为空命令（直接返回）。然后检查命令中是否包含管道符 `|`：若包含则调用 `handle_pipe()` 处理管道逻辑，否则作为单命令调用 `handle_redirection()` 解析后执行。整个过程中对返回值进行检查，确保错误能够正确传递。

```

shell_lab > C run_command.c
14 int handle_builtin_command(char **argv, int argc, Environment *env) {
59
60     // path命令: 设置PATH
61     if (strcmp(argv[0], "path") == 0) {
62         // 清理旧的PATH
63         if (env->paths) {
64             for (int i = 0; i < env->path_count; i++) {
65                 if (env->paths[i]) free(env->paths[i]);
66             }
67             free(env->paths);
68             env->paths = NULL;
69         }
70
71         if (argc == 1) {
72             // 清空PATH
73             env->path_count = 0;
74         } else {
75             // 设置新的PATH
76             env->path_count = argc - 1;
77             env->paths = (char**)malloc(sizeof(char*) * (argc - 1));
78             if (!env->paths) {
79                 perror("malloc");
80                 return 1;
81             }
82             for (int i = 1; i < argc; i++) {
83                 env->paths[i-1] = strdup(argv[i]);
84                 if (!env->paths[i-1]) {
85                     perror("strdup");
86                     return 1;
87                 }
88             }
89         }
90         env->path_set = true;
91         return 1;
92     }
93
94     return 0; // 不是内置命令
95
96 }
97
shell_lab > C run_command.c
379 // 执行命令 (主函数)
380 int run_command(char *line, Environment *env) {
381     if (!line || strlen(line) == 0) {
382         return 0;
383     }
384
385     // 去除换行符
386     char *newline = strchr(line, '\n');
387     if (newline) *newline = '\0';
388
389     // 去除首尾空白
390     line = trim_whitespace(line);
391     if (strlen(line) == 0) {
392         return 0;
393     }
394
395     // 处理后台运行 %
396     char *commands[MAX_PROCESSES];
397     int cmd_count = 0;
398
399     char *token = strtok(line, "&");
400     while (token && cmd_count < MAX_PROCESSES) {
401         commands[cmd_count] = trim_whitespace(token);
402         if (strlen(commands[cmd_count]) > 0) {
403             cmd_count++;
404         }
405         token = strtok(NULL, "&");
406     }
407
408     // 执行所有命令
409     for (int i = 0; i < cmd_count; i++) {
410         handle_pipe(commands[i], env);
411     }
412
413     return 0;
414 }
415

```

5. 主程序实现

在 `myshell.c` 中实现了 Shell 的主循环控制逻辑。`main()` 函数首先初始化环境结构体，然后判断运行模式：

- 批处理模式 (`argc > 1`)：将第一个参数作为批处理文件路径，调用 `fopen()` 打开文件。若打开失败则输出错误并退出。将输入流指向该文件，不显示提示符。
- 交互模式 (`argc == 1`)：输入流指向标准输入 `stdin`，每次循环前调用 `print_prompt()` 显示提示符，输出欢迎信息。

主循环中，通过 `fgets()` 读取一行命令，若遇到 EOF 则退出循环。读取成功后去除末尾换行符，调用 `run_command()` 执行命令。循环结束后清理环境资源并返回 0。

```

shell_lab > C myshell.c
 1  #include "structures.h"
 2  #include "utilities.h"
 3  #include "run_command.h"
 4  #include "macros.h"
 5  #include <stdio.h>
 6  #include <stdlib.h>
 7  #include <string.h>
 8
 9 int main(int argc, char *argv[]) {
10     Environment env;
11     char line[MAX_LINE_LENGTH];
12     FILE *input = stdin;
13     int batch_mode = 0;
14
15     // 初始化环境
16     init_environment(&env);
17
18     // 检查是否为批处理模式
19     if (argc > 1) {
20         input = fopen(argv[1], "r");
21         if (!input) {
22             fprintf(stderr, "Error: Cannot open file %s\n", argv[1]);
23             return 1;
24         }
25         batch_mode = 1;
26     }
27
28     // 显示欢迎信息（仅交互模式）
29     if (!batch_mode) {
30         printf(COLOR_CYAN);
31         printf("Welcome to Simple Shell v1.0\n");
32         printf("Type 'help' for available commands\n");
33         printf("Type 'exit' to quit\n");
34         printf(COLOR_RESET);
35         printf("\n");
36     }
37 }
38
39
40 int main(int argc, char *argv[]) {
41     // 主循环
42     while (1) {
43         // 打印提示符（仅交互模式）
44         if (!batch_mode) {
45             print_prompt(&env);
46         }
47
48         // 读取命令
49         if (fgets(line, sizeof(line), input) == NULL) {
50             break; // EOF或错误
51         }
52
53         // 在批处理模式下显示执行的命令
54         if (batch_mode) {
55             printf("%s", line);
56         }
57
58         // 执行命令
59         run_command(line, &env);
60
61         // 清理资源
62         if (batch_mode) {
63             fclose(input);
64         }
65         cleanup_environment(&env);
66
67         if (!batch_mode) {
68             printf("\nGoodbye!\n");
69         }
70
71     }
72 }
73

```

6. 编译配置

在 `Makefile` 中配置了编译规则：

- **编译器和标志：** 使用 `gcc` 编译器，设置 `-std=gnu99` 和 `-D_GNU_SOURCE` 标志以支持 `strdup()` 等 GNU 扩展函数，启用 `-Wall -Wextra -g` 进行警告检查和调试。
- **目标文件：** 定义了 `myshell` 为最终可执行文件，依赖 `myshell.o`、`utilities.o`、`run_command.o` 三个目标文件。
- **编译规则：** 为每个源文件定义了编译规则，指定了头文件依赖关系，确保头文件修改后能触发重新编译。
- **清理规则：** 提供 `make clean` 命令删除所有 `.o` 文件和可执行文件。
- **运行规则：** 提供 `make run` 命令直接启动 Shell，以及 `make valgrind` 命令进行内存检查。

所有代码文件编写完成后，在项目根目录执行 `make` 命令即可编译生成 `myshell` 可执行文件。

```
shell_lab > M Makefile
1  # Makefile for Simple Shell
2
3  CC = gcc
4  CFLAGS = -Wall -Wextra -g -std=gnu99 -D_GNU_SOURCE
5  TARGET = myshell
6  OBJS = myshell.o utilities.o run_command.o
7
8  # 默认目标
9  all: $(TARGET)
10
11 # 链接目标文件
12 $(TARGET): $(OBJS)
13     $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)
14
15 # 编译源文件
16 myshell.o: myshell.c structures.h utilities.h run_command.h macros.h
17     $(CC) $(CFLAGS) -c myshell.c
18
19 utilities.o: utilities.c utilities.h structures.h macros.h
20     $(CC) $(CFLAGS) -c utilities.c
21
22 run_command.o: run_command.c run_command.h utilities.h structures.h macros.h
23     $(CC) $(CFLAGS) -c run_command.c
24
25 # 清理编译文件
26 clean:
27     rm -f $(OBJS) $(TARGET)
28
29 # 运行程序
30 run: $(TARGET)
31     ./$(TARGET)
32
33 # 使用valgrind检查内存
34 valgrind: $(TARGET)
35     valgrind --leak-check=full --show-leak-kinds=all ./$(TARGET)
36
37 .PHONY: all clean run valgrind
```

编写测试程序及测试

测试环境准备

在Ubuntu虚拟机的`~/shell_lab`目录下完成代码编写后，首先执行编译命令：

```
cd ~/shell_lab
make clean
make
```

编译过程中，由于最初使用`-std=c99`标准导致`strup()`和`gethostname()`函数出现隐式声明警告，后修改为`-std=gnu99`并添加`-D_GNU_SOURCE`宏定义，同时在相应源文件中包含了`<string.h>`、`<unistd.h>`、`<stdbool.h>`等必要头文件，成功消除了所有编译警告。

编译完成后生成`myshell`可执行文件，通过`ls -lh myshell`查看文件大小约为39KB。

基本命令测试

(1) 交互模式启动

执行`./myshell`启动交互模式，Shell显示欢迎信息和彩色提示符，格式为用户名@主机名:当前路径\$。提示符中用户名显示为绿色，主机名为青色，路径为蓝色，美观清晰。

```

victor@victor-VirtualBox:~/shell_lab$ make clean
rm -f myshell.o utilities.o run_command.o myshell
victor@victor-VirtualBox:~/shell_lab$ make
gcc -Wall -Wextra -g -std=gnu99 -D_GNU_SOURCE -c myshell.c
gcc -Wall -Wextra -g -std=gnu99 -D_GNU_SOURCE -c utilities.c
gcc -Wall -Wextra -g -std=gnu99 -D_GNU_SOURCE -c run_command.c
gcc -Wall -Wextra -g -std=gnu99 -D_GNU_SOURCE -o myshell myshell.o utilities.o run_command.o
victor@victor-VirtualBox:~/shell_lab$ ls -lh myshell
-rwxrwxr-x 1 victor victor 39K 12月 8 17:37 myshell
victor@victor-VirtualBox:~/shell_lab$ ./myshell
    Welcome to Simple Shell v1.0
    Type 'help' for available commands
    Type 'exit' to quit

victor@victor-VirtualBox:/home/victor/shell_lab$ pwd

```

(2) 简单命令执行

依次测试以下基本命令：

- `pwd`: 正确输出当前工作目录`/home/victor/shell_lab`。
- `ls`: 列出当前目录下的所有文件，包括源代码、头文件、Makefile、编译生成的`.o`文件和`myshell`可执行文件。
- `ls -l`: 以长格式显示文件详细信息，包括权限、所有者、大小、修改时间等，输出正确。
- `ls -a`: 显示包括隐藏文件在内的所有文件。
- `echo "Hello Shell"`: 正确输出字符串内容，验证了参数传递和引号处理。

所有基本命令均能正常执行，输出结果与系统Shell一致。

```

victor@victor-VirtualBox:/home/victor/shell_lab$ pwd
/home/victor/shell_lab
victor@victor-VirtualBox:/home/victor/shell_lab$ ls
batch.txt myshell README.md run_command.o utilities.h
macros.h myshell.c run_command.c structures.h utilities.o
Makefile myshell.o run_command.h utilities.c
victor@victor-VirtualBox:/home/victor/shell_lab$ ls -l
总用量 144
-rwxrwx--- 1 victor victor 321 12月 8 16:44 batch.txt
-rwxrwx--- 1 victor victor 803 12月 8 16:07 macros.h
-rwxrwx--- 1 victor victor 860 12月 8 16:37 Makefile
-rwxrwxr-x 1 victor victor 39896 12月 8 17:37 myshell
-rwxrwx--- 1 victor victor 2045 12月 8 17:07 myshell.c
-rw-rw-r-- 1 victor victor 9000 12月 8 17:37 myshell.o
-rwxrwx--- 1 victor victor 1438 12月 8 16:30 README.md
-rwxrwx--- 1 victor victor 11938 12月 8 16:49 run_command.c
-rwxrwx--- 1 victor victor 487 12月 8 16:15 run_command.h
-rw-rw-r-- 1 victor victor 21616 12月 8 17:37 run_command.o
-rwxrwx--- 1 victor victor 1196 12月 8 16:15 structures.h
-rwxrwx--- 1 victor victor 4145 12月 8 16:37 utilities.c
-rwxrwx--- 1 victor victor 536 12月 8 16:15 utilities.h
-rw-rw-r-- 1 victor victor 16208 12月 8 17:37 utilities.o
victor@victor-VirtualBox:/home/victor/shell_lab$ ls -a
. macros.h myshell.c run_command.c structures.h utilities.o
.. Makefile myshell.o run_command.h utilities.c
batch.txt myshell README.md run_command.o utilities.h
victor@victor-VirtualBox:/home/victor/shell_lab$ echo "Hello Shell"
"Hello Shell"
victor@victor-VirtualBox:/home/victor/shell_lab$ 

```

内置命令测试

(1) cd命令测试

测试目录切换功能：

- `cd /tmp`: 成功切换到`/tmp`目录，提示符中的路径更新为`/tmp`。
- `pwd`: 验证当前目录确为`/tmp`。
- `cd /home/victor`: 切换回用户主目录`/home/victor`。
- `cd shell_lab`: 使用相对路径切换到`shell_lab`目录。
- `cd /nonexistent`: 尝试切换到不存在的目录，Shell正确输出"No such file or directory"错误信息，当前目录保持不变。

```
victor@victor-VirtualBox:/home/victor/shell_lab$ cd /tmp
victor@victor-VirtualBox:/tmp$ pwd
/tmp
victor@victor-VirtualBox:/tmp$ cd /home/victor
victor@victor-VirtualBox:/home/victor$ cd shell_lab
victor@victor-VirtualBox:/home/victor/shell_lab$ cd /nonexistent
cd: No such file or directory
victor@victor-VirtualBox:/home/victor/shell_lab$
```

(2) path命令测试

测试PATH动态修改功能：

- 执行`path /bin /usr/bin`设置自定义PATH。
- 执行`ls`命令仍可正常运行，说明能在设定路径中找到可执行文件。
- 执行`path`（无参数）清空PATH。
- 再次执行`ls`，Shell输出"Command not found"，验证了PATH清空后无法找到命令。
- 执行`path /bin /usr/bin /usr/local/bin`恢复PATH，后续命令恢复正常。

```
victor@victor-VirtualBox:/home/victor/shell_lab$ path /bin /usr/bin
victor@victor-VirtualBox:/home/victor/shell_lab$ ls
batch.txt myshell README.md run_command.o utilities.h
macros.h myshell.c run_command.c structures.h utilities.o
Makefile myshell.o run_command.h utilities.c
victor@victor-VirtualBox:/home/victor/shell_lab$ path
victor@victor-VirtualBox:/home/victor/shell_lab$ ls
ls: Command not found
victor@victor-VirtualBox:/home/victor/shell_lab$ path /bin /usr/bin /usr/local/bin
victor@victor-VirtualBox:/home/victor/shell_lab$ ls
batch.txt myshell README.md run_command.o utilities.h
macros.h myshell.c run_command.c structures.h utilities.o
Makefile myshell.o run_command.h utilities.c
victor@victor-VirtualBox:/home/victor/shell_lab$
```

(3) help和exit命令测试

- `help`: 正确输出Shell支持的功能列表，包括内置命令说明、管道、重定向等特性介绍。
- `exit`: Shell正确退出并返回系统终端。

```

victor@victor-VirtualBox:/home/victor/shell_lab$ help
Simple Shell - Supported commands:
  cd [dir]      - Change directory
  pwd           - Print working directory
  exit          - Exit shell
  help          - Show this help
  path [dirs]   - Set PATH environment

Supported features:
  Pipes:        command1 | command2
  Redirect:     command < input.txt > output.txt
  Background:   command &
victor@victor-VirtualBox:/home/victor/shell_lab$ exit
victor@victor-VirtualBox:~/shell_lab$ █

```

输出重定向测试

(1) 覆盖重定向 (>) 测试

- `echo "test redirect" > redirect_test.txt`: 创建新文件并写入内容。
- `cat redirect_test.txt`: 验证文件内容为"test redirect"。
- `echo "overwrite" > redirect_test.txt`: 再次写入，覆盖原内容。
- `cat redirect_test.txt`: 确认内容已被覆盖为"overwrite"。
- `ls -l > output.txt`: 将ls命令的输出重定向到文件。
- `cat output.txt`: 验证文件中包含完整的ls输出结果。

```

victor@victor-VirtualBox:/home/victor/shell_lab$ echo "test redirect" > redirect_test.txt
victor@victor-VirtualBox:/home/victor/shell_lab$ cat redirect_test.txt
"test redirect"
victor@victor-VirtualBox:/home/victor/shell_lab$ echo "overwrite" > redirect_test.txt
victor@victor-VirtualBox:/home/victor/shell_lab$ cat redirect_test.txt
"overwrite"
victor@victor-VirtualBox:/home/victor/shell_lab$ ls -l > output.txt
victor@victor-VirtualBox:/home/victor/shell_lab$ cat output.txt
总用量 148
-rwxrwx--- 1 victor victor 321 12月 8 16:44 batch.txt
-rwxrwx--- 1 victor victor 803 12月 8 16:07 macros.h
-rwxrwx--- 1 victor victor 860 12月 8 16:37 Makefile
-rwxrwxr-x 1 victor victor 39896 12月 8 17:37 myshell
-rwxrwx--- 1 victor victor 2045 12月 8 17:07 myshell.c
-rw-rw-r-- 1 victor victor 9000 12月 8 17:37 myshell.o
-rw-r--r-- 1 victor victor 0 12月 8 17:49 output.txt
-rwxrwx--- 1 victor victor 1438 12月 8 16:30 README.md
-rw-r--r-- 1 victor victor 12 12月 8 17:49 redirect_test.txt
-rwxrwx--- 1 victor victor 11938 12月 8 16:49 run_command.c
-rwxrwx--- 1 victor victor 487 12月 8 16:15 run_command.h
-rw-rw-r-- 1 victor victor 21616 12月 8 17:37 run_command.o
-rwxrwx--- 1 victor victor 1196 12月 8 16:15 structures.h
-rwxrwx--- 1 victor victor 4145 12月 8 16:37 utilities.c
-rwxrwx--- 1 victor victor 536 12月 8 16:15 utilities.h
-rw-rw-r-- 1 victor victor 16208 12月 8 17:37 utilities.o
victor@victor-VirtualBox:/home/victor/shell_lab$ █

```

(2) 追加重定向 (>>) 测试

- `echo "line1" > append_test.txt`: 创建文件并写入第一行。
- `echo "line2" >> append_test.txt`: 追加第二行。
- `echo "line3" >> append_test.txt`: 追加第三行。
- `cat append_test.txt`: 验证文件包含三行内容，顺序正确。

- `wc -l < append_test.txt`: 使用输入重定向统计行数，输出"3"，验证追加功能正常。

```
victor@victor-VirtualBox:/home/victor/shell_lab$ echo "line1" > append_test.txt
victor@victor-VirtualBox:/home/victor/shell_lab$ echo "line2" >> append_test.txt
victor@victor-VirtualBox:/home/victor/shell_lab$ echo "line3" >> append_test.txt
victor@victor-VirtualBox:/home/victor/shell_lab$ cat append_test.txt
"line1"
"line2"
"line3"
victor@victor-VirtualBox:/home/victor/shell_lab$ wc -l < append_test.txt
3
victor@victor-VirtualBox:/home/victor/shell_lab$
```

输入重定向测试

- 创建测试输入文件: `echo -e "apple\nbanana\ncherry" > fruits.txt`。
- `wc -l < fruits.txt`: 从文件读取输入并统计行数，正确输出"3"。
- `sort < fruits.txt`: 从文件读取并排序，输出按字母顺序排列的三个单词。
- `cat < fruits.txt`: 验证输入重定向等效于直接传文件名参数。

```
victor@victor-VirtualBox:/home/victor/shell_lab$ echo -e "apple\nbanana\ncherry" > fruits.txt
victor@victor-VirtualBox:/home/victor/shell_lab$ wc -l < fruits.txt
3
victor@victor-VirtualBox:/home/victor/shell_lab$ sort < fruits.txt
"apple
banana
cherry"
victor@victor-VirtualBox:/home/victor/shell_lab$ cat < fruits.txt
"apple
banana
cherry"
victor@victor-VirtualBox:/home/victor/shell_lab$
```

单级管道测试

- `ls | grep txt`: 列出当前目录文件并筛选包含"txt"的文件名，正确输出所有.txt结尾的文件。
- `ps aux | grep bash`: 查看系统进程并筛选bash相关进程，验证了对外部命令输出的管道处理。
- `echo "hello world" | wc -w`: 统计单词数，输出"2"。
- `cat fruits.txt | sort`: 通过管道传递文件内容进行排序。

```
victor@victor-VirtualBox:/home/victor/shell_lab$ ls | grep txt
append_test.txt
batch.txt
fruits.txt
output.txt
redirect_test.txt
victor@victor-VirtualBox:/home/victor/shell_lab$ ps aux | grep bash
victor      3160  0.0  0.1 13960  5292 pts/0    Ss   17:37   0:00 bash
victor      3259  0.0  0.0 12000   648 pts/0    S+   17:55   0:00 grep bash
victor@victor-VirtualBox:/home/victor/shell_lab$ echo "hello world" | wc -w
2
victor@victor-VirtualBox:/home/victor/shell_lab$ cat fruits.txt | sort
"apple
banana
cherry"
victor@victor-VirtualBox:/home/victor/shell_lab$
```

所有单级管道命令均正常工作，数据流传递正确。

多级管道测试

- `ls -l | grep txt | wc -l`: 三级管道，先列出文件，筛选txt文件，最后统计数量，输出正确的数值。
- `cat fruits.txt | sort | head -1`: 读取文件、排序、取第一行，输出"apple"。
- `ps aux | grep victor | grep bash | wc -l`: 四级管道，统计当前用户的bash进程数。

```
victor@victor-VirtualBox:/home/victor/shell_lab$ ls -l | grep txt | wc -l
5
victor@victor-VirtualBox:/home/victor/shell_lab$ cat fruits.txt | sort | head -1
"apple"
victor@victor-VirtualBox:/home/victor/shell_lab$ ps aux | grep victor | grep bash | wc -l
2
victor@victor-VirtualBox:/home/victor/shell_lab$ █
```

多级管道测试验证了管道链的正确连接，每个命令的输出能够正确传递到下一个命令的输入。

管道与重定向组合测试

- `ls | grep txt > result.txt`: 将管道输出重定向到文件。
- `cat result.txt`: 验证文件中包含筛选后的文件列表。
- `cat fruits.txt | sort > sorted.txt`: 输入来自管道，输出重定向到文件。
- `wc -l < fruits.txt > count.txt`: 输入输出同时重定向。
- `cat count.txt`: 验证文件中包含正确的行数统计结果。

```
victor@victor-VirtualBox:/home/victor/shell_lab$ ls | grep txt > result.txt
victor@victor-VirtualBox:/home/victor/shell_lab$ cat result.txt
append_test.txt
batch.txt
fruits.txt
output.txt
redirect_test.txt
result.txt
sorted.txt
victor@victor-VirtualBox:/home/victor/shell_lab$ cat fruits.txt | sort > sorted.txt
victor@victor-VirtualBox:/home/victor/shell_lab$ wc -l < fruits.txt > count.txt
victor@victor-VirtualBox:/home/victor/shell_lab$ cat count.txt
3
victor@victor-VirtualBox:/home/victor/shell_lab$ █
```

组合测试验证了重定向与管道可以同时使用且不会相互干扰。

批处理模式测试

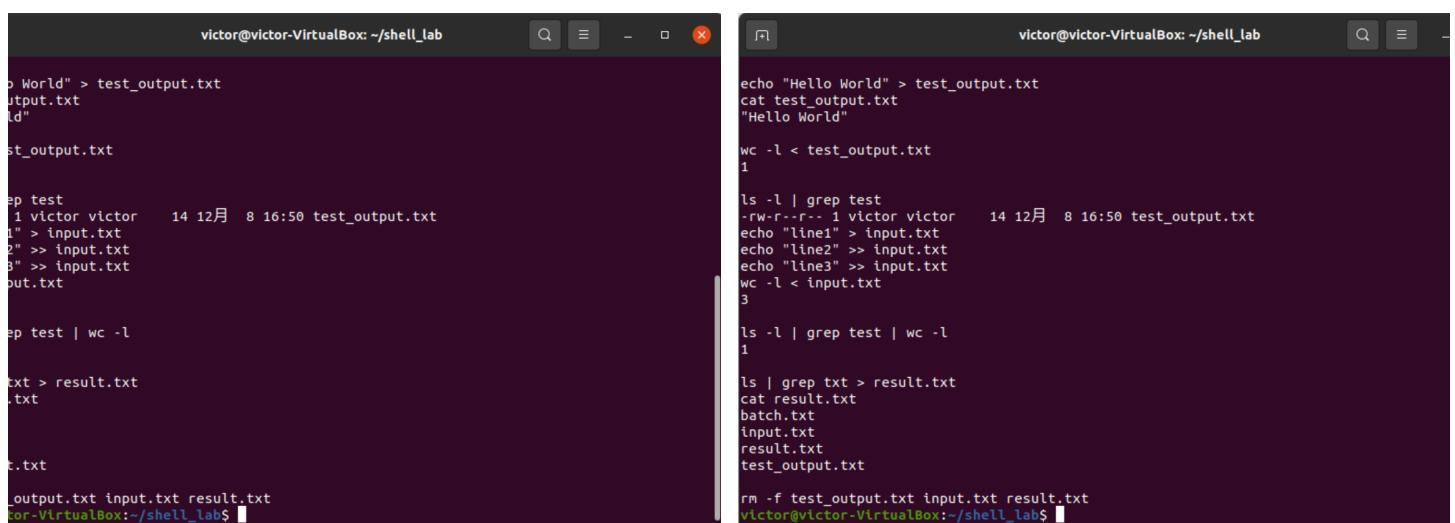
编写批处理测试脚本 `batch.txt`, 包含以下命令:

```
pwd
ls -l
echo "Hello World" > test_output.txt
cat test_output.txt
wc -l < test_output.txt
ls -l | grep test
echo "line1" > input.txt
echo "line2" >> input.txt
echo "line3" >> input.txt
wc -l < input.txt
ls -l | grep test | wc -l
ls | grep txt > result.txt
cat result.txt
rm -f test_output.txt input.txt result.txt
```

执行 `./myshell batch.txt`, Shell按顺序执行文件中的每条命令, 输出完整的执行结果。

关键观察点:

1. 批处理模式下不显示提示符, 输出更加简洁。
2. 每条命令执行后立即显示结果, 与交互模式行为一致。
3. 文件创建、重定向、管道等功能均正常工作。
4. 最后的清理命令成功删除临时文件。
5. 执行完毕后Shell自动退出, 返回系统终端。



The image shows two terminal windows side-by-side, both titled "victor@victor-VirtualBox: ~/shell_lab".

Terminal 1 (Left):

```
o World" > test_output.txt
utput.txt
ld"
st_output.txt

ep test
1 victor victor 14 12月 8 16:50 test_output.txt
1" > input.txt
2" >> input.txt
3" >> input.txt
put.txt

ep test | wc -l

txt > result.txt
.txt

t.txt

output.txt input.txt result.txt
tor-VirtualBox:~/shell_lab$
```

Terminal 2 (Right):

```
echo "Hello World" > test_output.txt
cat test_output.txt
"Hello World"

wc -l < test_output.txt
1

ls -l | grep test
-rw-r--r-- 1 victor victor 14 12月 8 16:50 test_output.txt
echo "line1" > input.txt
echo "line2" >> input.txt
echo "line3" >> input.txt
wc -l < input.txt
3

ls -l | grep test | wc -l
1

ls | grep txt > result.txt
cat result.txt
batch.txt
input.txt
result.txt
test_output.txt

rm -f test_output.txt input.txt result.txt
victor@victor-VirtualBox:~/shell_lab$
```

错误处理测试

(1) 命令不存在

- 执行`nonexistentcommand`, Shell输出"nonexistentcommand: Command not found", 并继续等待下一条命令。

(2) 文件不存在

- 执行`cat nosuchfile.txt`, 系统输出"No such file or directory"错误。
- 执行`wc -l < nosuchfile.txt`, Shell在打开文件时捕获错误并输出"nosuchfile.txt: No such file or directory"。

(3) 权限不足

- 执行`cat /etc/shadow`, 系统输出"权限不够"。
- 尝试创建只读文件后写入: `touch readonly.txt; chmod 444 readonly.txt; echo "test" > readonly.txt`, Shell正确输出`Permission denied`。

(4) 空命令处理

- 多次直接按回车, Shell正确忽略空命令, 继续显示提示符等待输入。
- 输入纯空格或制表符的命令行, 同样被正确识别为空命令。

```
victor@victor-VirtualBox:/home/victor/shell_lab$ nonexistentcommand
nonexistentcommand: Command not found
victor@victor-VirtualBox:/home/victor/shell_lab$ cat nosuchfile.txt
cat: nosuchfile.txt: 没有那个文件或目录
victor@victor-VirtualBox:/home/victor/shell_lab$ cat /etc/shadow
cat: /etc/shadow: 权限不够
victor@victor-VirtualBox:/home/victor/shell_lab$ touch readonly.txt
victor@victor-VirtualBox:/home/victor/shell_lab$ chmod 444 readonly.txt
victor@victor-VirtualBox:/home/victor/shell_lab$ echo "test" > readonly.txt
readonly.txt: Permission denied
victor@victor-VirtualBox:/home/victor/shell_lab$
victor@victor-VirtualBox:/home/victor/shell_lab$
victor@victor-VirtualBox:/home/victor/shell_lab$
victor@victor-VirtualBox:/home/victor/shell_lab$
```

通过上述全面测试, 验证了Shell的各项功能均符合设计要求。

心得体会

通过本次Shell实验, 我对操作系统中进程管理、进程间通信以及用户态与内核态交互有了更加深刻和系统的认识。从最初对Shell工作原理的模糊理解, 到最终独立实现一个功能完备的命令行解释器, 整个过程充满了挑战, 但也收获了宝贵的经验。

在实验初期, 我对fork和exec这对经典组合的理解还停留在理论层面, 认为只需简单调用这两个系统调用就能执行外部命令。但在实际编码中才发现, 进程创建和程序执行之间涉及大量细节: 如何正确传递参数数组、如何确保数组以NULL结尾、如何在子进程中设置环境、如何避免父进程过早退出导致子进程成为孤儿进程等。特别是在实现内置命令时, 我深刻体会到为什么cd、exit等命令必须由Shell自身执行——如果在子进程中修改工作目录或退出,

对父进程（Shell本身）毫无影响。这一认识让我对"内置命令"与"外部命令"的本质区别有了清晰的理解。

管道的实现是本次实验中技术难度最大的部分。最初我按照网上的简单示例实现了单级管道，但在扩展到多级管道时遇到了诸多问题：管道文件描述符的管理混乱、子进程间的连接顺序错误、父进程未正确关闭管道导致读端阻塞等。经过反复调试和查阅资料，我逐渐理解了管道的本质是一个单向数据通道，必须严格遵循"写端关闭才能触发读端EOF"的规则。在实现中，我为每个管道维护了读写两端的文件描述符，在子进程中根据其在管道链中的位置（首、中、尾）设置不同的重定向策略，并在父进程中及时关闭所有管道描述符。这种精细化的资源管理让我对操作系统中"一切皆文件"的设计哲学有了更深的感悟。

重定向功能的实现相对直观，但在处理`>>`追加重定向时我犯了一个低级错误：最初只实现了`>`覆盖重定向，未考虑连续两个`>`的情况，导致测试时多次追加操作只保留最后一次写入。通过仔细分析测试输出，我发现问题出在文件打开标志上，于是修改了`handle_redirection()`函数，增加了对`>>`的专门检测逻辑，并在`open()`调用时根据重定向类型选择`O_TRUNC`（覆盖）或`O_APPEND`（追加）标志。这次调试经历让我意识到，即使是看似简单的功能，也需要考虑所有可能的情况，细节决定成败。

内存管理是本次实验中另一个需要高度重视的问题。Shell程序需要频繁进行动态内存分配：解析PATH时分配目录字符串数组、解析命令时分配参数数组、查找可执行文件时复制路径字符串等。每次分配后我都添加了错误检查代码，并在适当位置释放内存。为了确保没有内存泄漏，我使用`valgrind`工具进行了检测，发现了几处忘记释放的内存，修复后达到了"0 bytes lost"的标准。这让我深刻认识到，用户态程序虽然不像内核代码那样严格，但良好的内存管理习惯同样至关重要，它直接影响程序的稳定性和可靠性。

测试环节的工作同样重要。我编写了详细的测试用例，覆盖了基本命令、内置命令、重定向、管道、组合功能、错误处理、边界情况等多个维度。在批处理测试中，我将所有测试命令写入脚本文件，一次性执行并对比输出结果，大大提高了测试效率。测试过程中发现的问题不仅帮助我修复了代码缺陷，还加深了我对Shell各项功能的理解。例如，在测试管道时我发现某些情况下输出不完整，通过分析发现是父进程过早关闭了管道导致数据丢失，这促使我重新审视了进程同步和资源管理的逻辑。

与实验一的内核开发相比，本次用户态编程的体验截然不同。内核开发中不能使用标准库函数，必须使用内核提供的API，调试手段也相对有限，主要依赖`printk`输出日志。而用户态程序可以自由使用C标准库，可以利用`gdb`、`valgrind`等强大工具进行调试和分析，开发效率更高。但这并不意味着用户态编程更简单——Shell需要处理的逻辑复杂度、对系统调用的理解深度、对边界情况的考虑周全性，都对编程能力提出了很高要求。两次实验相辅相成，让我对操作系统的"上下两层"都有了实践性的认识。

总的来说，本次Shell实验不仅巩固了操作系统课程的理论知识，更重要的是培养了我的系统思维和工程实践能力。从需求分析到架构设计，从编码实现到测试验证，完整的开发流程让我对软件工程有了更深刻的认识。这些宝贵的经验和能力，将对我今后在操作系统领域以及其他系统级软件开发方向的学习和研究产生深远的积极影响。