C/UNIX 程序设计大作业报告

Yujin Wang

2024

目录

1 总体完成情况

功能	实现情况
用户输入命令与参数，能正常执行命令	Yes
输入、输出重定向到文件	Yes
管道	Yes
后台执行程序	Yes
作业控制 (jobs,fg,bg)	Yes
历史命令 (history)	Yes
文件名 tab 补全，各种快捷键	Yes
环境变量、简单脚本	TODO

表 1: 任务完成情况

2 代码设计

2.1 输入命令与参数

本项目将命令分为两种：Shell 的内建命令与外部命令。调用逻辑如 1所示。

Listing 1: 外部命令与内部命令的处理

```
// do internal logic
int internal = 1;
if (strcmp(args[0], "cd") == 0)
{
    do_cd(arg_count, args);
}
else if (strcmp(args[0], "pwd") == 0)
{
    do_pwd();
}
// ... other internal logic commands
else
{
    internal = 0;
}

pid_t pid = fork();
if (pid == 0)
{ // child process
    // do external logic
    if (!internal)
    {
        if (is_executable(args[0]))
        {
            execv(args[0], args);
        }
```

```
        else
        {
            execvp(args[0], args);
        }
        perror("execvp");
        exit(1);
    }
    else
    {
        exit(0);
    }
}
else if (pid > 0)
{ // Parent process
    int status;
    if (background)
    {
        add_job(pid, cmd_copy, BACKGROUND);
        waitpid(pid, &status, WNOHANG);
        printf("[%d] %d\n", *job_count, pid);
    }
    else
    {
        add_job(pid, cmd_copy, FOREGROUND);
        waitpid(pid, &status, WUNTRACED);
    }
}
else
{
    perror("fork");
}
```

1. 内建命令：由 Shell 本身直接提供并支持的命令。这些命令在 Shell 的代码中实现，无需调用外部程序即可在父进程中执行。本项目目前支持的内建命令如表 2所示。
2. 外部命令：存储在文件系统中的独立可执行文件，Shell 通过路径搜索机制调用这些命令。具体而言，Shell 使用系统调用 `fork()` 创建子进程，并在子进程中加载并运行外部命令的可执行文件。

2.2 分割，重定向，管道

本项目支持若干分隔符：`;`（指令并列），`<`（重定向输入），`>`（重定向输出），`>`（重定向追加输出），`|`（管道），`&`（后台运行）。具体解析方式如下：

1. 使用 `strncpy` 函数，以`;`作为标识符，将命令分割为若干部分，否则执行第 2 步。
2. 检测命令末尾是否有 `&`，若有则将命令置为后台执行，否则执行第 3 步。
3. 以 `|` 作为标识符，检测命令中是否存在管道命令，若存在则创建管道将前一个命令的输出作为后一个命令的输入，否则执行第 4 步。
4. 以 `<`，`>`，`<<` 作为标识符，若存在则读取/写入相应的文件，否则执行第 5 步。

命令名称及参数	功能
cd [dir]	切换工作目录
pwd	显示当前工作目录
clear	清除屏幕信息
echo [string]	在屏幕上打印信息
history	打印历史输入的命令信息
about	输出关于 yaush 的提示信息
jobs	查看后台工作的情况
fg [job_id]	将某个后台运行或后台挂起的程序调至前台
bg [job_id]	将某个后台挂起的程序调至后台运行
exit	退出程序

表 2: 已实现的内建指令

5. 执行没有分隔符的指令。

```c
void process_commands(char *input)
{
    char *and_cmds[MAX_CMDS];
    int and_count;

    // Split by ;
    split_string(input, ";", and_cmds, &
        and_count);

    for (int i = 0; i < and_count; i++)
    {
        char *pipe_cmds[MAX_CMDS];
        int pipe_count;
        int background = 0;

        // Check for background execution
        if (and_cmds[i][strlen(and_cmds[i]) - 1]
            == '&')
        {
            background = 1;
            and_cmds[i][strlen(and_cmds[i]) - 1]
                = '\0';
        }

        // Split by |
        split_string(and_cmds[i], "|", pipe_cmds
            , &pipe_count);

        if (pipe_count == 1)
        {
            execute_command(pipe_cmds[0],
                background);
        }
        else
        {
            execute_pipeline(pipe_cmds,
                pipe_count, background);
        }

        // If the last command in the pipeline
            fails, stop executing further
        if (i < and_count - 1 && WEXITSTATUS(0)
            != 0)
        {
            break;
        }
    }
}
```

```c
void execute_pipeline(char **cmds, int cmd_count
    , int background)
{
    int pipe_fds[2];
    int prev_fd = -1;

    for (int i = 0; i < cmd_count; i++)
    {
        pipe(pipe_fds);

        pid_t pid = fork();
        if (pid == 0)
        { // Child process
            if (prev_fd != -1)
            {
                dup2(prev_fd, STDIN_FILENO);
                close(prev_fd);
            }
            if (i < cmd_count - 1)
            {
                dup2(pipe_fds[1], STDOUT_FILENO)
                    ;
            }
            close(pipe_fds[0]);
            close(pipe_fds[1]);

            execute_command(cmds[i], 0);
            exit(0);
        }
        else if (pid > 0)
        { // Parent process
            if (!background || i == cmd_count -
                1)
            {
                waitpid(pid, NULL, 0);
            }
            close(pipe_fds[1]);
            if (prev_fd != -1)
            {
                close(prev_fd);
            }
            prev_fd = pipe_fds[0];
        }
        else
        {
            perror("fork");
        }
    }
}
```

```c
void execute_command(char *cmd, int background)
{
    char *args[MAX_ARGS];
    int arg_count;
    char *input_file = NULL;
    char *output_file = NULL;
    int append_mode = 0;

    // copy the command to history
    char cmd_copy[COMMAND_SIZE];
    strncpy(cmd_copy, cmd, COMMAND_SIZE - 1);

    // Split the command into arguments
    split_string(cmd, "␣", args, &arg_count);

    // Check for redirection
    for (int i = 0; i < arg_count; i++)
    {
        if (strcmp(args[i], "<") == 0)
        {
            input_file = args[i + 1];
            args[i] = NULL;
            break;
        }
        else if (strcmp(args[i], ">>") == 0)
        {
            output_file = args[i + 1];
            append_mode = 1;
            args[i] = NULL;
            break;
        }
        else if (strcmp(args[i], ">") == 0)
        {
            output_file = args[i + 1];
            append_mode = 0;
            args[i] = NULL;
            break;
        }
    }
    ....
    // internal/external dispatch logic
}
```

2.3 后台执行程序

前文提到使用 & 符号来标识一个程序是否在后台进行执行。若检测到 &，则将 background 信号置为 1。该型号用于判定在执行子进程时，父进程是否需要等待子进程结束，这一点通过 WNOHANG 和 WUNTRACED 两个不同的配置字来实现，如 2所示。

Listing 2: 后台执行与前台执行

```
if (background)
{
    // ... other logic
    waitpid(pid, &status, WNOHANG);
}
else
{
    // ... other logic
    waitpid(pid, &status, WUNTRACED);
}
```

2.4 作业控制

为了记录 Shell 中不同进程的信息，需要建立一个结构体 Jobs，具体信息如 3。该结构体列表和所有的结构体数目需要在所有进程间可见，因此使用进程间共享内存进行存储，如 4。

Listing 3: Jobs 结构体

```
typedef struct
{
    pid_t pid;                    // pid number of the process
    char command[COMMAND_SIZE]; // command content
    int status;                   // RUNNING, SUSPEND, DONE
    int type;                     // FOREGROUND, BACKGROUND
    int fg_bg_flag;               // whether the command is controlled after a fg/bg command.
} Job;
```

Listing 4: Jobs 初始化

```
void init_jobs()
{
    // use shm
    int shm_id = shmget(IPC_CREAT, sizeof(Job) * MAX_JOBS + sizeof(int) * 2, IPC_CREAT |
        0666);
    if (shm_id == -1)
    {
        perror("shmget");
        exit(1);
    }
    void *shm = shmat(shm_id, 0, 0);
    jobs = (Job *)shm;
    job_count = (int *)((char *)shm + sizeof(jobs) * MAX_JOBS);

    *job_count = 0;
}
```

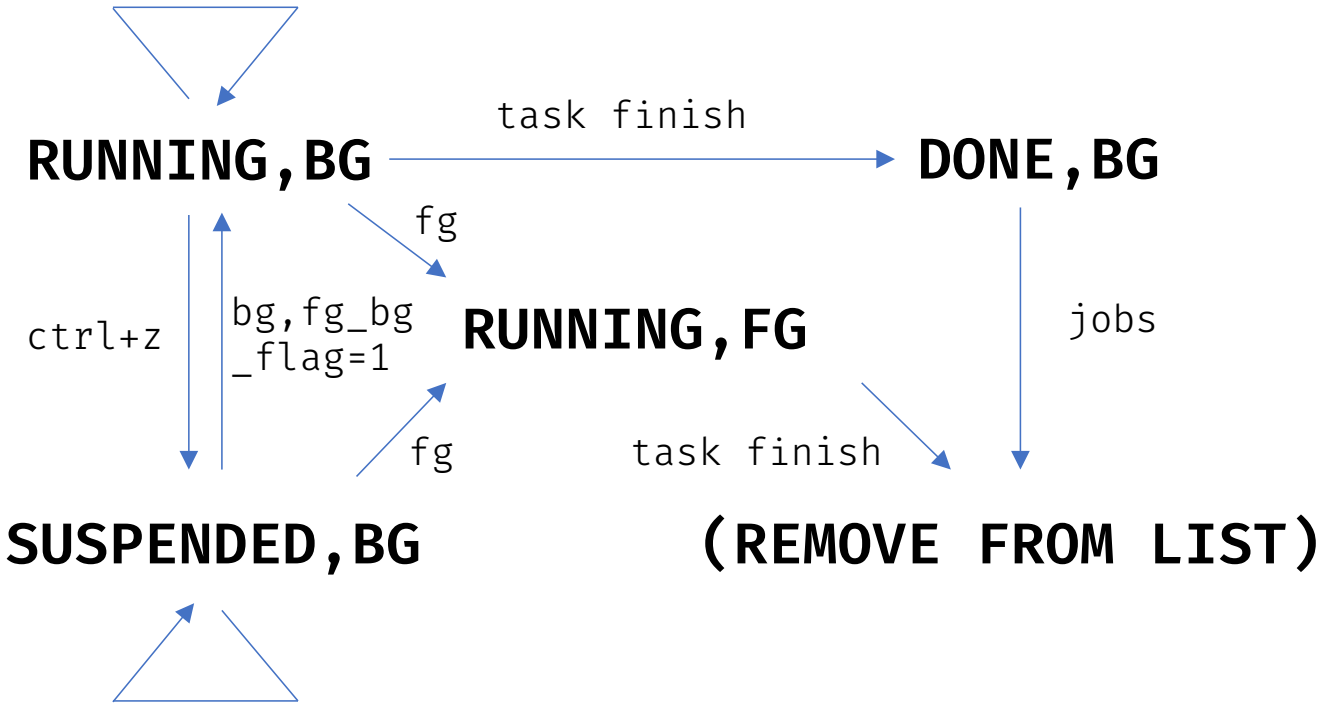

图 1: 进程状态和转移模式状态机

图 1列举了所有的进程状态和转移模式,据此可以在 do_bg,do_fg,do_job,handle_sigchld,handle_tstp 等函数中设置对应的状态转移机制。

2.5 历史命令

历史命令可以借助 readline 库进行实现，每次输入一个命令时将命令写入特定文件，并在输入 history 指令时进行打印。

```
void do_history()
{
    HIST_ENTRY **h = history_list();
    if (h)
    {
        int i = 0;
        while (h[i])
        {
            printf("%d: %s\n", i, h[i]->line);
            i++;
        }
    }
}
```

2.6 tab 补全和快捷键

借助 readline 库进行实现。默认支持 tab 补全、光标移动、上下方向键查看历史命令等功能。

3 运行方式

整体文件架构如下所示：

```
.
    Makefile // makefile
    handle.c // 进程接收到信号后的处理逻辑
    handle.h
    header.h // 所有用到的头文件
    internal_command.c // 内部命令
    internal_command.h
    jobs.h // 记录进程工作状态的结构体
    marcos.h // 所有定义的宏
    utils.c // 杂项函数
    utils.h
    yaush.c // 主函数
    yaush.h
```

运行命令：

```
$ make
$ ./yaush
yaush: Yet Another Unix Shell
Type 'exit' to quit
[ther@Ther /home/ther/c-unix-programming/lab] (2024-12-16 17:03:22)
    ls
Makefile   handle.c  handle.o  internal_command.c  internal_command.o  marcos.h  utils.h
    yaush    yaush.h
README.md  handle.h  header.h  internal_command.h  jobs.h              utils.c   utils.o
    yaush.c  yaush.o
```