《计算机系统》

ShellLab 实验报告

班级: GitHub

学号: 78268851

姓名: AnicoderAndy

目录

1 实验项目

1.1 项目名称

Shell Lab——实现简易 Shell 程序

1.2 实验目的

- 熟悉 Linux Shell 的基本概念。
- 熟悉 Linux 的异常控制流。
- 熟悉 Linux 的进程控制、进程间通信的基本方法。

1.3 实验资源

- 实验环境：Linux Ubuntu 22.04 LTS
- 实验工具：Visual Studio Code、GNU Make 4.3、GCC 11.4.0
- 其他资源：实验手册、16 个测试文件以及 4 个测试代码、测试脚本、`tsh.c`。

2 实验任务

本实验要求实现一个建简易的 Shell 程序，要求支持执行程序、执行 quit、jobs、fg、bg 四个内置命令。

本实验提供的代码实现了一些如解析命令行 parseline、sigquit 等简单的函数，需要实现：

```
void eval(char* cmdline);
int builtin_cmd(char** argv);
void do_bgfg(char** argv);
void waitfg(pid_t pid);

void sigchld_handler(int sig);
void sigtstp_handler(int sig);
void sigint_handler(int sig);
```

2.1 便利工具函数

为了后续代码方便，实现 verbose_printf 方便进行开启-v 情况下的输出：

```
inline int verbose_printf(const char* fmt, ...) {
    if (verbose) {
        va_list args;
        va_start(args, fmt);
        int ret = vprintf(fmt, args);
        va_end(args);
        return ret;
    }
    return -1;
}
```

由于 SIGCHLD 信号的处理函数和 waitfg 函数中都需要进行工作回收，因此实现了一个二者都可调用的工作回收函数：

```
void job_reaping(pid_t pid, int status) {
    if (WIFEXITED(status)) {
        deletejob(jobs, pid);
        verbose_printf("job_reaping: Job [%d] (%d) deleted\n",
                       pid2jid(pid), pid);
    } else if (WIFSIGNALED(status)) {
        int jid = pid2jid(pid);
        int sig = WTERMSIG(status);
        deletejob(jobs, pid);
        verbose_printf("job_reaping: Job [%d] (%d) deleted\n", jid,
                       pid);
        printf("Job [%d] (%d) terminated by signal %d\n", jid, pid,
               sig);
```

```
    } else if (WIFSTOPPED(status)) {
        int jid = pid2jid(pid);
        int sig = WSTOPSIG(status);
        struct job_t* job = getjobjid(jobs, jid);
        job->state = ST;
        printf("Job [%d] (%d) stopped by signal %d\n", jid, pid, sig);
    }
}
```

2.2 eval 解析函数

eval 函数的功能是解析命令行，分析输入的命令是否合法，判断输入的是否是内置命令，执行指定输入的命令。

```
void eval(char* cmdline) {
    if (cmdline == NULL) {
        return;
    }
    // Local copy of argument list
    char* argv[MAXARGS];
    // Parse the command line and specify if it is a background job
    int bg = parseline(cmdline, argv);
    if (argv[0] == NULL)
        return;
    int state = bg ? BG : FG;
    // Judge if a built-in command is requested
    if (builtin_cmd(argv)) {
        return;
    }

    // Prepare to block SIGCHLD
    sigset_t mask_all, prev_mask;
    sigemptyset(&mask_all);
    sigaddset(&mask_all, SIGCHLD);

    // Fork a child process
    sigprocmask(SIG_BLOCK, &mask_all, &prev_mask);
    pid_t pid = fork();
    if (pid == 0) {
        // Child process
        sigprocmask(SIG_SETMASK, &prev_mask, NULL);
        setpgid(0, 0); // Set the process group ID to the child PID
        if (execve(argv[0], argv, environ) < 0) {
            printf("%s: Command not found\n", argv[0]);
            exit(1);
        }
    } else if (pid > 0) {
        // Parent process

        // Add the job to the job list
```

```
        addjob(jobs, pid, state, cmdline);
        sigprocmask(SIG_SETMASK, &prev_mask, NULL);

        if (state == FG) {
            // Wait for the foreground job to terminate
            waitfg(pid);
        } else {
            printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
        }
    } else {
        // Fork error
    }
    return;
}
```

这段代码首先试图以内置指令的方式执行输入命令；如果不是内置命令，则会执行 fork 函数创建子进程，子进程会执行 execve 函数来执行输入的命令，父进程会将子进程添加到作业列表中，并根据输入的命令是否是前台命令来决定是否等待子进程结束。

在 fork 之前还要注意使用 sigprocmask 函数来阻塞 SIGCHLD 信号，以避免在子进程创建时父进程接收到信号导致的竞态错误。

2.3 执行内置函数

builtin_cmd 函数的功能是判断输入的命令是否是内置命令，如果是 quit 指令，直接执行 exit(0)；如果是 jobs 指令，调用已经实现好的 listjobs 函数；如果是 fg 或 bg，调用待实现的 do_bgfg 函数；如果不是则返回 0：

```
int builtin_cmd(char** argv) {
    if (strcmp(argv[0], "quit") == 0) {
        exit(0);
    } else if (strcmp(argv[0], "jobs") == 0) {
        listjobs(jobs);
        return 1;
    } else if (strcmp(argv[0], "bg") == 0 ||
               strcmp(argv[0], "fg") == 0) {
        do_bgfg(argv);
        return 1;
    }
    return 0; /* not a builtin command */
}
```

下面还需要实现 do_bgfg 函数来处理 bg 和 fg 命令：

```
void do_bgfg(char** argv) {
    int jid = 0;
    pid_t pid = 0;
```

```
4       struct job_t* job;
5
6       // Parse the argument of fg/bg
7       if (argv[1] == NULL) {
8           printf("%s command requires PID or %%jobid argument\n",
9                   argv[0]);
10          return;
11      } else if (argv[1][0] == '%') {
12          jid = atoi(&argv[1][1]);
13          job = getjobjid(jobs, jid);
14          if (job == NULL) {
15              printf("%s: No such job\n", argv[1]);
16              return;
17          }
18          pid = job->pid;
19      } else if (isdigit(argv[1][0])) {
20          pid = atoi(argv[1]);
21          job = getjobpid(jobs, pid);
22          if (job == NULL) {
23              printf("(%d): No such process\n", pid);
24              return;
25          }
26          jid = job->jid;
27      } else {
28          printf("%s: argument must be a PID or %%jobid\n", argv[0]);
29          return;
30      }
31
32      // Execute fg/bg
33      if (strcmp(argv[0], "bg") == 0) {
34          printf("[%d] (%d) %s", jid, pid, job->cmdline);
35          job->state = BG;
36          kill(-pid, SIGCONT);
37      } else if (strcmp(argv[0], "fg") == 0) {
38          // printf("[%d] (%d) %s", jid, pid, job->cmdline);
39          job->state = FG;
40          kill(-pid, SIGCONT);
41          waitfg(pid);
42      }
43      return;
44  }
```

这段代码首先检查了输入指令的合法性，如果不合法会输出相应提示。对于合法的输入，会使用 `kill` 函数向命令中的指定进程发送 `SIGCONT` 信号来继续执行该进程并且将其状态设置为前台或后台。

2.4 waitfg 函数和信号处理函数

waitfg 函数会等待前台进程结束，并对其回收；sigchld_handler 函数会在接收到子进程发送的 SIGCHLD 信号时进行工作回收，二者逻辑相似，使用刚刚实现的 job_reaping 函数即可完成对应逻辑：

```
void waitfg(pid_t pid) {
    int status;
    if (waitpid(pid, &status, WUNTRACED) == pid) {
        job_reaping(pid, status);
    }
    verbose_printf("waitfg: Process (%d) no longer the fg process\n",
                   pid);
}

void sigchld_handler(int sig) {
    verbose_printf("sigchld_handler: entering\n");

    int status;
    pid_t pid;
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
        job_reaping(pid, status);
    }

    verbose_printf("sigchld_handler: exiting\n");
}
```

读者可能会有疑问，前台进程退出时，两个函数可能都会被调用，是否有可能因为二者都调用了 job_reaping 函数而导致错误？答案是不会的，因为在一个进程中只会有一个 waitpid 函数会获取到子进程的退出状态，无法获取的 waitpid 函数不会返回进程号，因此只有一个函数会回收前台进程。

实验还要求实现 SIGINT 和 SIGTSTP 信号的处理函数，可以使用 kill 函数向前台进程所在的进程组发送相应信号：

```
void sigint_handler(int sig) {
    verbose_printf("sigint_handler: entering\n");
    pid_t pid = fgpid(jobs);
    if (pid > 0) {
        kill(-pid, SIGINT);
        verbose_printf("sigint_handler: Job (%d) killed\n", pid);
    }
    verbose_printf("sigint_handler: exiting\n");
    return;
}

void sigtstp_handler(int sig) {
    verbose_printf("sigtstp_handler: entering\n");
```

```
    pid_t pid = fgpid(jobs);
    if (pid > 0) {
        verbose_printf("sigtstp_handler: Job [%d] (%d) stopped\n",
                       pid2jid(pid), pid);
        kill(-pid, SIGTSTP);
    }
    verbose_printf("sigtstp_handler: exiting\n");
    return;
}
```

2.5 测试

经测试，实验提供的 tsh 程序可以正常运行，并且运行结果与实验资源中提供的测试样例一致。

由于实验中实现进程回收的方式与 tshref 实现的方式可能存在不同，后者的进程回收统一在 sigchld_handler 中进行，所以在开启-v 选项时，输出的内容会存在差异。由于部分老师要求实现的程序需要与 tshref 具有一样的行为，在这一要求下，注释掉 sigchld_handler 中的 verbose_printf 函数，将 job_reaping 中输出的日志伪造成 sigchld_handler 的输出即可。

```
void job_reaping(pid_t pid, int status) {
    verbose_printf("sigchld_handler: entering\n");
    if (WIFEXITED(status)) {
        deletejob(jobs, pid);
        verbose_printf("sigchld_handler: Job [%d] (%d) deleted\n",
                       pid2jid(pid), pid);
    } else if (WIFSIGNALED(status)) {
        int jid = pid2jid(pid);
        int sig = WTERMSIG(status);
        deletejob(jobs, pid);
        verbose_printf("sigchld_handler: Job [%d] (%d) deleted\n", jid,
                       pid);
        printf("Job [%d] (%d) terminated by signal %d\n", jid, pid,
               sig);
    } else if (WIFSTOPPED(status)) {
        int jid = pid2jid(pid);
        int sig = WSTOPSIG(status);
        struct job_t* job = getjobjid(jobs, jid);
        job->state = ST;
        printf("Job [%d] (%d) stopped by signal %d\n", jid, pid, sig);
    }
    verbose_printf("sigchld_handler: exiting\n");
}
```

3 总结

3.1 实验中出现的问题

- 实验过程中 Visual Studio Code 刚开始一直对实验资源提供的代码提示错误，但又可以正常通过 `make` 命令构建。后来发现按照代码规范，要启用 GNU 提供的扩展功能才能使用 `sigset_t` 等信号相关的类型，在预处理部分添加 `#define _GNU_SOURCE` 解决了这一问题。

- 在2.4节中，实现 `sigchld_handler` 函数时，调用 `waitpid` 函数时一开始并没有开启 `WNOHANG` 选项，导致 `sigchld_handler` 可能会被阻塞，在查阅 `waitpid` 的手册后了解了这一 API 的基本使用方法，添加了所需选项实现了所需功能。

3.2 心得体会

通过本次 ShellLab 实验，我深入理解了 Linux 系统中进程控制和信号处理的机制。实验要求实现一个简易的 Shell，支持前后台作业管理、信号处理以及内置命令的执行，这使我对操作系统的核心概念有了更直观的认识。

在实验过程中，我学会了如何使用 `fork` 创建子进程，利用 `execve` 执行新程序，以及通过 `waitpid` 等待子进程的结束。同时，掌握了如何设置进程组 ID，以便正确地向前台作业发送信号。此外，实验中对 `SIGCHLD`、`SIGINT` 和 `SIGTSTP` 等信号的处理，使我理解了信号在进程间通信中的重要作用。

通过调试和测试，我体会到编写健壮的系统程序需要细致的思考和严谨的编码习惯。例如，在处理信号时，必须注意阻塞和解除阻塞的时机，以防止竞态条件的发生。实验还让我认识到，良好的代码结构和清晰的逻辑对于实现复杂功能至关重要。

总的来说，ShellLab 实验不仅提升了我的编程能力，更加深了我对操作系统原理的理解，为后续学习和实践打下了坚实的基础。