Shell Lab

1、实验总览

- 本实验的代码（$tsh.c$）储存在$./code/tsh.c$，实验结果储存在$./code/ans.out$。
- 自己编写的shell先读入使用者输入的指令，然后调用$eval$函数对指令进行处理。先使用$parseline$函数对指令进行分析，调用$builtin_cmd$函数判断该指令是否为内置指令，如果是则执行，否则返回$eval$中处理。
- 在实验中，编写了$sigchld_handler$、$sigint_handler$和$sigtstp_handler$三个函数，分别对$SIGCHLD$、$SIGINT$和$SIGTSTP$三种信号进行自定义处理。

2、函数分析

- $void\ eval(char\ *cmdline)$

  该函数的作用是处理输入的指令。

  先对指令进行分析，然后判断输入的指令是否为内置指令，如果不是则创建一个子进程，在子进程中处理这条指令。对于父进程，需要判断子进程在前台或后台执行。如果在后台执行，则父进程无需等待子进程结束，只需输出子进程信息即可。如果在前台执行，则父进程需要等待子进程结束，等待方式在下文详细讲解。

  对于添加和删除任务，为了防止子进程与父进程之间的竞争导致先删除在添加的错误发生，需要在创建子进程前添加$SIGCHLD$的信号遮罩，使得进入父进程时可以先执行添加操作，然后解除遮罩后才接受$SIGCHLD$信号，保证了删除操作一定在添加操作之后。

```
void eval(char *cmdline) {
    char *argv[MAXARGS];
    int bg;
    sigset_t mask_all, mask_one, prev_one;
    sigfillset(&mask_all);
    sigemptyset(&mask_one);
    sigaddset(&mask_one, SIGCHLD); // 设置信号遮罩
    bg = parseline(cmdline, argv); // 分析指令
    if (argv[0] == NULL) return; // 跳过空指令行
    if (!builtin_cmd(argv)) {
        sigprocmask(SIG_BLOCK, &mask_one, &prev_one); // 设置信号遮罩
        pid_t pid = fork(); // 创建子进程
```

```
        if (pid == 0) {
            sigprocmask(SIG_SETMASK, &prev_one, NULL); // 解除信号遮罩
            setpgid(0, 0); // 将子进程的进程组识别号设置为自己，防止父进程组的信号的影响
            if (execve(argv[0], argv, environ) < 0) { // 执行指令
                printf ("%s: Command not found\n", argv[0]);
                fflush(stdout); // 输出错误信息
            }
            exit(0);
        }

        if (!bg) {
            sigprocmask(SIG_BLOCK, &mask_all, NULL);
            addjob(jobs, pid, FG, cmdline);
            sigprocmask(SIG_SETMASK, &prev_one, NULL);
            // 设置信号遮罩并添加任务，防止先删除后添加的情况发生

            waitfg(pid); // 等待前台任务完成
        }
        else {
            sigprocmask(SIG_BLOCK, &mask_all, NULL);
            addjob(jobs, pid, BG, cmdline);
            sigprocmask(SIG_SETMASK, &prev_one, NULL);

            printf ("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
            fflush(stdout); // 输出后台任务的信息
        }
    }
    memset(cmdline, 0, strlen(cmdline)); // 清空之前储存的指令
    return;
}
```

- $int\ builtin\_cmd(char\ **argv)$

  该函数的作用是判断一个指令是否为内置指令，如果是则处理，并且返回1。如果不是，则返回0，将指令交给$eval$函数处理。

  使用$if-else$对输入的指令进行判断，判断其是否为内置指令。

```
int builtin_cmd(char **argv)  {
    if (strlen(argv[0]) == 4 && strcmp(argv[0], "quit") == 0) {
        exit(0);
    } // 处理quit指令
```

```
    else if (strlen(argv[0]) == 4 && strcmp(argv[0], "jobs") ==
0) {
        listjobs(jobs);
        return 1;
    } // 处理jobs指令
    else if (strlen(argv[0]) == 2 &&
             (strcmp(argv[0], "bg") == 0 ||
              strcmp(argv[0], "fg") == 0)) {
        do_bgfg(argv);
        return 1;
    } // 处理bg和fg指令
    return 0;     /* not a builtin command */
}
```

- $void\ do\_bgfg(char\ **argv)$

  该函数的作用为处理bg和fg这两条内置指令。

  在该函数内部需要进行多个错误处理：未输入任务的jid或pid、输入的jid和pid错误、没有制定的任务等。这些错误依次判断，如果出现就输出错误信息并返回。

  该函数需要两个子函数辅助运行，分别是$check$函数和$nojob$函数。其中，$check$函数用于判断输入的jid或pid是否是正确的，判断方法为检查每个字符是否都是数据。$nojob$函数用于找不到任务时返回错误信息，需要对jid和pid分开处理。

```
bool check(char *id) {
    int pos = 0;
    if (id[0] == '%') pos++; // 判断jid
    for (; pos < strlen(id); ++pos)
        if (id[pos] < '0' || id[pos] > '9')
            return false; // 检测是否全部为数字
    return true;
}

void nojob(char **argv) {
    if (argv[1][0] == '%')
        printf ("%s: No such job\n", argv[1]); // 处理jid
    else
        printf ("(%s): No such process\n", argv[1]); // 处理pid
    fflush(stdout);
}

void do_bgfg(char **argv) {
    if (argv[1] == NULL) {
```

```
20          printf ("%s command requires PID or %%jobid argument\n",
    argv[0]);
21          fflush(stdout);
22          return;
23      } // 处理未输入任务的jid和pid
24
25      if (!check(argv[1])) {
26          printf ("%s argument must be a PID or %%jobid\n",
    argv[0]);
27          fflush(stdout);
28          return;
29      } // 处理输入的jid或pid错误
30
31      struct job_t *job;
32      if (argv[1][0] == '%') job = getjobjid(jobs, atoi(argv[1] +
    1));
33      else job = getjobpid(jobs, atoi(argv[1]));
34      // 分类讨论获取制定的任务
35      if (job == NULL) {
36          nojob(argv);
37          return;
38      } // 处理任务不存在的情况
39      kill(-(job->pid), SIGCONT); // 向该任务所在的组发信号
40
41      if (strcmp(argv[0], "fg") == 0) {
42          job->state = FG; // 设置为前台
43          waitfg(job->pid); // 等待进程结束
44      }
45      else {
46          job->state = BG; // 设置为后台
47          printf ("[%d] (%d) %s", job->jid, job->pid, job-
    >cmdline);
48          fflush(stdout);
49      }
50      return;
51  }
```

- *void waitfg(pid_t pid)*

  该函数的作用为当子进程在前台运行时，等待子进程结束。

在这个函数中，使用*sigsuspend*函数进行等待，可以降低程序对资源的消耗，防止了忙等待子进程的情况出现。在等待过程中，先屏蔽所有信号，然后判断前台进程与子进程是否一致，若一致则继续等待，否则可以结束等待并且取消对信号的屏蔽。

```
void waitfg(pid_t pid) {
    sigset_t mask_all, prev_one;
    sigfillset(&mask_all);
    sigprocmask(SIG_BLOCK, &mask_all, &prev_one); // 屏蔽信号
    while (pid == fgpid(jobs))
        sigsuspend(&prev_one); // 等待子进程结束
    sigprocmask(SIG_SETMASK, &prev_one, NULL); // 取消信号遮罩
    return;
}
```

- *void sigchld_handler(int sig)*

  该函数的用于处理*SIGCHLD*信号。

  该函数用于处理子进程的状态。需要判断是否有子进程处于终止或者暂停状态，如果有的话才需要进行处理。**需要注意的是，***waitpid*不能使用0作为第三个参数，因为这样会导致其等待所有程序全部终止才结束循环。收到信号后，对不同的信号进行不同的处理。在从任务列表中删除任务的时候，同样采取遮罩信号的方法以保证删除操作的原子性。

```
void sigchld_handler(int sig) {
    int status;
    pid_t pid;
    sigset_t mask_all, prev_one;
    sigfillset(&mask_all);
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) { // 判断信号
        if (WIFSIGNALED(status)) {
            printf ("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(status));
            fflush(stdout);
            sigprocmask(SIG_BLOCK, &mask_all, &prev_one);
            deletejob(jobs, pid);
            sigprocmask(SIG_SETMASK, &prev_one, NULL);
        } // 处理终止信号
        else if (WIFSTOPPED(status)) {
            printf ("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(status));
            fflush(stdout);
```

```
            struct job_t *job = getjobpid(jobs, pid);
            job->state = ST;
        } // 处理暂停信号
        else if (WIFEXITED(status)) {
            sigprocmask(SIG_BLOCK, &mask_all, &prev_one);
            deletejob(jobs, pid);
            sigprocmask(SIG_SETMASK, &prev_one, NULL);
        } // 处理子进程不是因为信号而退出的情况
    }
    return;
}
```

- *void sigint_handler(int sig)*

  该函数的用于处理$SIGINT$信号。

  $SIGINT$信号用于终止前台进程，所以收到信号后需要获取前台进程，如果没有前台进程，则无需进行任何操作，否则向子进程所在的进程组发送$SIGINT$信号以终止子进程。

```
void sigint_handler(int sig) {
    pid_t pid = fgpid(jobs); // 获取前台进程
    if (pid == 0) return; // 判断前台进程是否存在
    kill(-pid, SIGINT); // 发送信号
    return;
}
```

- *void sigtstp_handler(int sig)*

  该函数的用于处理$SIGTSTP$信号。

  $SIGTSTP$信号用于暂停前台进程，所以收到信号后需要获取前台进程，如果没有前台进程，则无需进行任何操作，否则向子进程所在的进程组发送$SIGTSTP$信号以暂停子进程。

```
void sigtstp_handler(int sig) {
    pid_t pid = fgpid(jobs); // 获取前台进程
    if (pid == 0) return; // 判断前台进程是否存在
    kill(-pid, SIGTSTP); // 发送信号
    return;
}
```

3、答案生成

- 编写sh脚本帮助进行测试，先对程序进行编译，然后对每一个测试的结果输入到*ans.out*文件中。脚本内容如下所示：

```
make
make test01 >  ans.out
make test02 >> ans.out
make test03 >> ans.out
make test04 >> ans.out
make test05 >> ans.out
make test06 >> ans.out
make test07 >> ans.out
make test08 >> ans.out
make test09 >> ans.out
make test10 >> ans.out
make test11 >> ans.out
make test12 >> ans.out
make test13 >> ans.out
make test14 >> ans.out
make test15 >> ans.out
make test16 >> ans.out
```

4、总结

- 在该实验中，需要注意子进程的进程组标号，防止因为一个信号而导致和父进程在一个进程组的所有子进程都出现问题。
- 父进程对于子进程信号的处理分类要完全，*waitpid*函数的使用要正确。
- 父进程等待子进程时要使用*sigsuspend*函数，其本质为*pause*指令，但是能够做到原子性，保证正确的同时能够降低程序对资源的消耗。