实验报告

实验目的

本实验目的是实现一个自定义shell，实现quit，jobs，bg，pg等内置命令和进程间信号通信机制并在自定义shell中实现了管道和文件重定向功能。

实现具体实现

quit内置命令实现

实现思路

我们将quit设置为shell的内置命令，当执行eval时首先解析传进来的命令行参数，之后调用builtin_cmd函数，并在builtin_cmd函数中直接终止这个进程。

代码

在builtin_cmd实现quit内置命令

```
if (!strcmp(argv[0], "quit"))
    exit(0);
```

jobs内置命令实现

实现思路

我们将jobs和设定后台程序的‘&’符号都设定为内置命令。当输入仅为'&'字符时，直接忽略。当输入为jobs时，调用listjobs函数，列出当前所有任务的状态。同时，在eval函数中判断命令是否为内置命令。当不是内置命令时，fork()一个子进程，并将其单独设置为一个进程组。当为任务是前台任务时，在主进程中等待其完成；当任务为后台任务时，在主进程中打印其信息后，直接进入下一轮循环。

代码

```
// eval函数中
if (jobState == FG) {
    waitfg(pid);
}
else
    printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);

// builtin_cmd函数中
if (!strcmp(argv[0], "&"))
        return 1;
if (!strcmp(argv[0], "jobs")) {
        listjobs(jobs);
        return 1;
    }
// waitfg函数中
void waitfg(pid_t pid)
{
    while (pid == fgpid(jobs))
        ;
    return ;
}
```

实现进程间发送信号的机制

实现思路

这一部分主要是实现3个函数与信号屏蔽，sigchld_handler， sigint_handler， sigtstp_handler。其中实现信号SIGINT，SIGTSTP之间的传递并在sigchld_handler处理。

sigint_handler， sigtstp_handlers函数实现

首先，判断传入信号是否是信号处理函数的信号，之后获取前台任务的pid，并对前台应用所在的进程组发送对应信号。

sigchld_handler函数实现

在函数体中，设置循环不断收集前台应用给shell发送的SIGCHLD信号，当进程正确退出时，将进程从任务列表中删除；当进程因收到SIGINT信号而中断时，删除任务并打印信息；当进程因收到SIGTSTP信号暂停时，将其任务状态改成STOP。

信号屏蔽

首先我们在fork子进程前将SIGCHLD信号屏蔽，之后在addjobs之前恢复，避免在子进程还未加入任务队列时删除出错。

代码

```
//在eval函数中处理信号屏蔽
if (!builtin_cmd(argvs)) {

        sigfillset(&allMask);
        sigemptyset(&sigchldMask);
        sigaddset(&sigchldMask, SIGCHLD);
        sigprocmask(SIG_BLOCK, &sigchldMask, &prev);

        if ((pid = fork()) == 0) {
            sigprocmask(SIG_SETMASK, &prev, NULL);
            if (setpgid(0, 0)) {
                printf("Set pgid Error!\n");
                exit(0);
            }
            if (execve(argvs[0], argvs, environ) < 0) {
                printf("%s: Command not found!\n", argvs[0]);
                exit(0);
            }
        } else {
            addjob(jobs, pid, jobState, cmdline);
            sigprocmask(SIG_SETMASK, &prev, NULL);
            if (jobState == FG) {
                waitfg(pid);
            }
            else
                printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
        }
}
```

```c
void sigchld_handler(int sig)
{
    if (sig != SIGCHLD)
        printf("Send SIGCHLD Error!\n");
    int status = 0;
    pid_t pid;

    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
        if (WIFEXITED(status)) {
            deletejob(jobs, pid);
        } else if (WIFSIGNALED(status)) {
            int jid = pid2jid(pid);
            deletejob(jobs, pid);
            printf("Job [%d] (%d) terminated by signal 2\n", jid, pid);
        } else if (WIFSTOPPED(status)) {
            int jid = pid2jid(pid);
            struct job_t* fgJob = getjobpid(jobs, pid);
            if (!fgJob) return ;
            printf("Job [%d] (%d) stopped by signal 20\n", jid, pid);
            fgJob->state = ST;
        }
    }

}

void sigint_handler(int sig)
{
    if (sig != SIGINT)
        printf("Send SIGINT Error!\n");
    pid_t pid = fgpid(jobs);
    if (pid > 0)
        kill(-pid, SIGINT);
}

void sigtstp_handler(int sig)
{
    if (sig != SIGTSTP)
        printf("Send SIGTSTP Error!\n");
    pid_t pid = fgpid(jobs);
    if (pid > 0)
        kill(-pid, SIGTSTP);
}
```

bg， pg内置命令实现

实现思路

我们收现将bg， pg加入builtin_cmd杉树函数中，之后调用do_bgpg函数。在do_bgpg函数中，首先排除argv没有第二个参数的情况，之后将情况分为输入参数为pid与jid两种情况，当输入参数为jid时，首先将字符串转化为整数，之后通过jid找到对应的job。如果job为NULL，表示没有此任务直接返回；当输入参数为pid时，同样将字符串转化为整数，之后找到对应的job，若无job直接返回。通过以上，我们找到了对应job的pid值，之后若任务处于阻塞态，则发送SIGCONT信号将其将其唤醒，根据命令是bg还是pg将任务状态设置为FG与BG。若为FG则等待任务结束，否则直接返回。

代码

```
void do_bgfg(char **argv)
{
    if (!argv[1]) {
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return ;
    }
    pid_t pid = 0;
    int len = strlen(argv[1]), i;
    if (argv[1][0] == '%') {
        int jid = 0;
        for (i = 1; i < len; i ++ ) {
            jid *= 10;
            jid += argv[1][i] - '0';
            if (!isdigit(argv[1][i])) {
                printf("%s: argument must be a PID or %%jobid\n", argv[0]);
                return ;
            }
        }
        struct job_t* job = getjobjid(jobs, jid);
        if (!job) {
            printf("%%%d: No such job\n", jid);
            return ;
        }
        pid = job->pid;
    } else {
        for (i = 0; i < len; i ++ ) {
            pid *= 10;
            pid += argv[1][i] - '0';
            if (!isdigit(argv[1][i])) {
                printf("%s: argument must be a PID or %%jobid\n", argv[0]);
                return ;
            }
        }
        struct job_t* job = getjobpid(jobs, pid);
        if (!job) {
            printf("(%d): No such process\n", pid);
            return ;
        }
    }

    struct job_t* job = getjobpid(jobs, pid);
    if (job->state == ST)
        kill(-pid, SIGCONT);
```

```
    job->state = (strcmp(argv[0], "bg")? FG : BG);
    if (job->state == FG)
        waitfg(pid);
    else
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
    return ;
}
```

文件重定向实现

实现思路

在执行可执行文件之前，判断命令行参数中是否含有字符'>'。若含有，首先排除未正确输入文件名的情况，之后利用dup2函数将STDOUT输出重定向到文件中。

代码

```
// redirection
for (i = 0; argvs[i]; i ++ ) {
    if (!strcmp(argvs[i], ">")) {
        argvs[i] = NULL;
        if (!i || !argvs[i + 1]) {
            printf("redirection need file name!\n");
            exit(1);
        }
        int fd = open(argvs[i + 1], O_WRONLY|O_CREAT|O_TRUNC, 0644);
        dup2(fd, STDOUT_FILENO);
        argvs[i + 1] = NULL;
        close(fd);
    }
}
```

管道功能实现

在执行可执行文件之前，首判断命令行参数中是否含有字符'|',若含有将两个需要执行的可执行文件命令行参数分离后作为参数加入do_pipe函数中，并结束子进程。在do_pipe函数中，我们先创建子进程。在管道中执行多个命令时，使用fork()函数创建一个新的子进程来执行每个命令。之后使用管道进行进程间通信，我们使用pipe()函数创建一个管道，用于在两个进程之间传递数据。在父进程中，将输出发送到管道中，然后在子进程中从管道中读取输入。最后我们使用execve()函数来执行可执行文件，并使用dup2()函数将管道的输入或输出重定向到子进程中。

代码

```
// 在eval函数中
// pipe
int i, j;
for (i = 0; argvs[i]; i ++ ) {
    if (!strcmp(argvs[i], "|")) {
        if (!i || !argvs[i + 1]) {
            printf("pipe need another file!\n");
            exit(1);
        }

        argvs[i] = NULL;
        char* prog [MAXARGS] = {NULL};
        for (j = 0; argvs[i + j + 1]; j ++ ) {
            prog[j] = (char*) malloc(MAXARGS * sizeof(char));
            strcpy(prog[j], argvs[i + j + 1]);
            argvs[i + j + 1] = NULL;
        }
        do_pipe(argvs, prog);
        if (jobState == BG)
            printf("tsh> ");
        exit(0);
    }
}

//do_pipe函数
void do_pipe(char** prog1, char** prog2) {
    int pipefd[2];
    int pid1, pid2;
    pipe(pipefd);
    if ((pid1 = fork()) == 0) {
        close(pipefd[0]);
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[1]);
        if (execve(prog1[0], prog1, environ) < 0)
            exit(1);
    } else {
        int status;
        waitpid(pid1, &status, WUNTRACED);
        if (!WEXITSTATUS(status)) {
            if ((pid2 = fork()) == 0) {
                close(pipefd[1]);
                dup2(pipefd[0], STDIN_FILENO);
                close(pipefd[0]);
                if (execve(prog2[0], prog2, environ) < 0) {
                    printf("%s: Command not found\n", prog2[0]);
                    exit(1);
                }
            } else {
                close(pipefd[0]);
                close(pipefd[1]);
                waitpid(pid2, NULL, 0);
            }
```

```
        } else {
            printf("%s: Command not found\n", prog1[0]);
            exit(1);
        }
    }
}
```

正确性检测