ShellLab 实验报告

涂奕腾 2020201018

抱歉第一次交的时候没有附上代码，在实验截至后一周才发现。实验报告内容和代码相较于之前都没有更改。

1. eval()

eval 函数对用户输入的命令参数进行解析和计算，对于内建命令行通过 builtin_cmd 函数直接执行；否则 fork 新的进程在该进程所创立的新进程组中运行。对于前端任务在父进程中应该等待其运行结束(waitfg)。

主要参照课本 P525 的代码，主要在书上代码的基础上增加了信号处理部分(参照 P543)

(1)在 folk 子进程前阻塞 SIG_CHLD 信号；在 execve 前取消对 SIG_CHLD 的阻塞；在 addjob 之后取消阻塞

(2)对于创造的子进程重新分配进程组 ID，通过 setpgid(0, 0)创建新的进程组

```
void eval(char *cmdline) {
    int bg;
    pid_t pid;
    char *argv[MAXARGS], buf[MAXLINE];
    sigset_t mask_one, prev_one, mask_all;
    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    //argv[0]为 shell 内置命令名 或 可执行目标文件
    //bg = 1：后台 bg = 0： 前台
    if(argv[0] == NULL) return;
    if(!builtin_cmd(argv)){//检查第一个命令行参数是否是一个 shell 内置命令
    //若不是，创造一个子进程，在子进程中执行所请求的程序
        sigemptyset(&mask_one);
        sigaddset(&mask_one, SIGCHLD);
        sigprocmask(SIG_BLOCK, &mask_one, &prev_one);//folk 前阻塞 SIG_CHLD 信号
        if((pid = fork()) == 0){
            sigprocmask(SIG_SETMASK, &prev_one, NULL);//execve 前解除阻塞的 SIG_CHLD 信号
            setpgid(0, 0);//重新分配进程组 ID
            if(execve(argv[0], argv, environ) < 0){
                printf("%s: Command not found.\n",argv[0]);
                exit(0);
            }
        }
        sigprocmask(SIG_BLOCK, &mask_all, NULL);
        if(!bg){//前台等待作业终止，终止后开始下一轮迭代
            addjob(jobs, pid, FG, cmdline);
            sigprocmask(SIG_SETMASK, &prev_one, NULL);//addjob 后取消阻塞信号
            waitfg(pid);
        }
        else {//后台执行(bg = 1)，shell 返回循环顶部，等待下一个命令行
```

```
            addjob(jobs, pid, BG, cmdline);
            sigprocmask(SIG_SETMASK, &prev_one, NULL);//addjob 后取消阻塞信号
            printf("[%d] (%d) %s",pid2jid(pid), pid, cmdline);
        }
    }
    return;
}
```

2. builtin_cmd()

判断是否为内建指令。根据题目要求，我们只需要对 quit，bg/fg，jobs 分别按照要求判断、处理即可；注意内建命令位于 agrv[0]中

```
int builtin_cmd(char **argv) {//如果是内置的 shell 命令，解释命令并返回 1
    char *cmd = argv[0];
    if(!strcmp(cmd,"quit")) exit(0);
    if(!strcmp(cmd,"jobs")){
        listjobs(jobs);
        return 1;
    }
    if(!strcmp(cmd,"bg") || !strcmp(cmd,"fg")){
        do_bgfg(argv);
        return 1;
    }
    return 0;     /* not a builtin command */
}
```

3. do_bgfg()

处理前端后端任务，一是判断是否出现命令错误；二是发送 SIG_CONT 重启 job，因为在 eval()中创建进程时重新分配了进程组 ID，所以在 kill 的 pid 参数设置为-(job->pid)以向整个进程组发送；另外还要修改 job 的状态为前端/后端。注意 job 的 jid 或 pid 存放于 argv[1]中

```
void do_bgfg(char **argv){
    //argv[1]为<job>的 pid 或 jid
    struct job_t *job;
    int id;
    if(argv[1] == NULL){
        printf("%s command requires PID or %%jobid argument\n",argv[0]);
        return;
    }
    if(argv[1][0]=='%'){//jid
        if(argv[1][1] >='0'&& argv[1][1] <='9'){
            sscanf(argv[1], "%%%d", &id);
            job = getjobjid(jobs, id);
```

```
            if(job == NULL){
                printf("%%%d: No such job\n", id);
            return;
            }
        }
        else{
            printf("%s: argument must be a PID or %%jobid\n",argv[0]);
            return;
        }
    }
    else{//pid
        if(argv[1][0] >='0'&& argv[1][0] <='9'){
            sscanf(argv[1], "%d", &id);
            job = getjobpid(jobs, id);
            if(job == NULL){
                printf("(%s): No such process\n",argv[1]);
                return;
            }
        }
        else{
            printf("%s: argument must be a PID or %%jobid\n",argv[0]);
            return;
        }
    }
    kill(-(job->pid), SIGCONT);//发送 SIG_CONT 重启<job>
    if(!strcmp(argv[0], "bg")){
        job->state = BG;
        printf("[%d] (%d) %s",job->jid, job->pid, job->cmdline);
    }
    else {
        job->state = FG;
        waitfg(job->pid);
    }
    return;
}
```

4. waitfg()

等待前端子进程完成。根据提示，我们应使用 sleep()或类似的函数，在这里我参考课本 P545-546 使用的是 sigsuspend()函数实现

```
void waitfg(pid_t pid){
    sigset_t mask;
    sigemptyset(&mask);
    while(fgpid(jobs) > 0) sigsuspend(&mask);
    return;
```

```
}
```

5. sigchld_handler()

根据提示，sigchld 有三种成因，于是我们在 waitpid 中设置 WNOHANG | WUNTRACED 参数判断是否存在已经停止或者终止的进程，如果存在则返回 pid，不存在则立即返回 0；另一方面，通过解析 status 参数判断是子进程正常中止还是接受 sigint 或 sigtstp 信号，对于 sigstp 应改变 job 的状态，其他二者应 deletejob

```
void sigchld_handler(int sig) {
    pid_t pid;
    int status, olderrno = errno, jid;
    sigset_t mask_all, prev_all;
    sigfillset(&mask_all);
    while((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0){
        if(WIFEXITED(status)){//zombie
            sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
            deletejob(jobs, pid);
            sigprocmask(SIG_SETMASK, &prev_all, NULL);
        }
        if(WIFSTOPPED(status)){//SIG_TSTP
            jid = pid2jid(pid);
            printf("Job [%d] (%d) stopped by signal %d\n",jid, pid, WSTOPSIG(status));
            getjobpid(jobs, pid)->state = ST;
        }
        if(WIFSIGNALED(status)){//SIG_INT
            jid = pid2jid(pid);
            printf("Job [%d] (%d) terminated by signal %d\n",jid, pid, WTERMSIG(status));
            sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
            deletejob(jobs, pid);
            sigprocmask(SIG_SETMASK, &prev_all, NULL);
        }
    }
    errno = olderrno;
    return;
}
```

6. sigint_handler() & sigtstp_handler()

这两个函数的内容基本相同，捕获到信号后通过 kill(-pid, sig)将信号发送至前端所有进程即可

```
void sigint_handler(int sig) {
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    sigfillset(&mask_all);
    sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
```

```
    pid_t pid = fgpid(jobs);
    sigprocmask(SIG_SETMASK, &prev_all, NULL);
    if(pid) kill(-pid, sig);//发送给整个前台进程组
    errno = olderrno;
    return;
}

void sigtstp_handler(int sig) {
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    sigfillset(&mask_all);
    sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
    pid_t pid = fgpid(jobs);
    sigprocmask(SIG_SETMASK, &prev_all, NULL);
    if(pid) kill(-pid,sig);
    errno = olderrno;
    return;
}
```