Shell Lab Report

22302010017 包旭

1.错误包装函数

因为特意说明需要关注错误信息，而在使用一些 unix 进程相关的函数时需要显示的错误信息类似，所以把各个错误信息加上需要调用的函数本身都给包装起来了，方便直接调用，如下：

```
pid_t Fork(void){
    pid_t pid;
    if((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
void Sigprocmask(int how, const sigset_t *set, sigset_t *oldset){
    if(sigprocmask(how, set, oldset) < 0)
        unix_error("Sigprocmask error");
}
void Sigemptyset(sigset_t *set){
    if(sigemptyset(set) < 0)
        unix_error("Sigprocmask error");
}
void Sigfillset(sigset_t *set){
    if(sigfillset(set) < 0)
        unix_error("Sigfillset error");
}
void Sigaddset(sigset_t *set, int signum){
    if(sigaddset(set, signum) < 0)
        unix_error("Sigaddset error");
}
void Execve(const char *filename, char *const argv[], char *const envp[]){
    if(execve(filename, argv, envp) < 0){
        printf("%s: Command not found\n", argv[0]);
    }
}
void Setpgid(pid_t pid, pid_t pgid){
    if(setpgid(pid, pgid) < 0){
        unix_error("Setpid error");
    }
}
void Kill(pid_t pid, int sig){
    if(kill(pid, sig) < 0){
        unix_error("Kill error");
    }
}
```

2.eval()

eval 函数是整个 tsh 的核心，作用是获取用户输入并根据用户输入创建子进程并将其加到 jobs 中，这些子进程乃至子进程组的祖先都是 tsh 进程。值得注意的点就是在 addjobs 之前需要屏蔽 SIGCHLD ，避免在执行 addjobs 之前子进程已经完成并发送给父进程 SIGCHLD 信号，父进程处理并执行之后导致 addjobs 出错。

代码如下:

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    char buf[MAXLINE];
    int state;
    pid_t pid;

    sigset_t all_masks, temp_mask, prev_mask;

    strcpy(buf, cmdline);
    state = parseline(buf, argv) ? BG : FG;
    if(argv[0] == NULL){
        return;
    }
    Sigfillset(&all_masks);
    Sigemptyset(&temp_mask);
    Sigaddset(&temp_mask, SIGCHLD);
    if(!builtin_cmd(argv)){
        // fork 前阻塞 SIG_BLOCK，标准做法，避免后面 addjob 之前子进程已经跑完
        Sigprocmask(SIG_BLOCK, &temp_mask, &prev_mask);
        if((pid = fork()) == 0){
            // 子进程代码
            // 取消对 SIG_BLOCK 的屏蔽，使得子进程能正常接受 SIG_BLOCK 信号
            Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
            Setpgid(0, 0);
            Execve(argv[0], argv, environ);
            // Execve 有执行错误的风险，为了避免无法回收要加一个退出代码
            exit(0);
        }
        // 后台进程
        // 在 addjob 的时候作业列表属于临界区，要原子性地访问
        if(state == BG){
            Sigprocmask(SIG_BLOCK, &all_masks, &prev_mask);
            addjob(jobs, pid, state, cmdline);
            // 解除对 SIG_CHILD 的屏蔽，使父进程可以正常回收子进程
            Sigprocmask(SIG_SETMASK, &temp_mask, NULL);
            printf("[%d] (%d) %s",pid2jid(pid), pid, cmdline);
        }else{
            // 前台进程
            Sigprocmask(SIG_BLOCK, &all_masks, &prev_mask);
            addjob(jobs, pid, state, cmdline);
            // 解除对 SIG_CHILD 的屏蔽，使父进程可以正常回收子进程
            Sigprocmask(SIG_SETMASK, &temp_mask, NULL);
            waitfg(pid);
        }
        Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

```
    }
    return;
}
```

3.builtin_cmd()

判断用户输入的命令是否是内置命令，通过 strcmp 字符串比较即可实现

代码如下

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

4.do_bgfg()

这个函数用于将已经暂停的 job 以后台/前台的方式继续执行。通过 jid 或者 pid 可以找到相应的进程和进程组，利用 kill 命令可以发送 SIGCONT 信号以使得暂停的进程/进程组继续运行。给 kill 传 -pid 就可以给整个进程组发送信号。

代码如下：

```
void do_bgfg(char **argv)
{
    struct job_t *job = NULL;
    int state;
    int id;
    state = strcmp(argv[0], "bg") ? BG : FG;
    if(argv[1]==NULL){
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }
    // 通过 jid 查找
    if(argv[1][0]=='%'){
       if(sscanf(&argv[1][1], "%d", &id) > 0){
            job = getjobjid(jobs, id);
            if(job==NULL){
                printf("%%%d: No such job\n", id);
```

```
                return;
            }
        }
    }else if(!isdigit(argv[1][0])) {
        // 非法输入，报错
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }else{
        // 通过 pid 查找
        id = atoi(argv[1]);
        job = getjobpid(jobs, id);
        if(job==NULL){
            printf("(%d): No such process\n", id);
            return;
        }

    }
    // 向整个进程组发信号以恢复执行
    Kill(-(job->pid), SIGCONT);
    job->state = state;
    if(state==BG){
        printf("[%d] (%d) %s",job->jid, job->pid, job->cmdline);
    }else{
        waitfg(job->pid);
    }
    return;
}
```

5.waitfg()

这个函数用于等待前台进程完成，在前台进程结束之前不返回，以保证只有一个前台进程在运行。利用`sigsuspend(&empty_mask);`，可以恢复对"空信号集"的屏蔽，也就是允许所有信号进入，并挂起，等待信号到来。

代码如下：

```
void waitfg(pid_t pid)
{
    sigset_t empty_mask;
    Sigemptyset(&empty_mask);
    while (fgpid(jobs) != 0){
        // 恢复对"空信号集"的屏蔽，即允许所有信号进入，并挂起等待信号到来
        sigsuspend(&empty_mask);
    }
    return;
}
```

6.sigchld_handler()

用于处理 SIGCHLD 信号，当进程退出、停止或者终止的时候会发出这个信号(因为所有进程都是 tsh 的子进程)，tsh 将处理这个信号，例如将 job 从 jobs 中删除。需要注意的是 jobs 是全局变量，类似临界区资源，在访问之前需要做和 addjobs 之前类似的操作，即屏蔽信号，避免错误。另外，在所有处理信号的过程中都可能改变全局 errno 的值，因此在每个 handler 开始的时候都要保存现有的值，并在 handler 过程结束之后恢复原有的值。

代码如下：

```
void sigchld_handler(int sig)
{
    int olderrno = errno;
    int status;
    pid_t pid;
    struct job_t *job;
    sigset_t all_masks, prev_mask;
    sigfillset(&all_masks);
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0){
        // 即将访问临界区，屏蔽所有信号
        sigprocmask(SIG_BLOCK, &all_masks, &prev_mask);
        // 进程正常退出
        if (WIFEXITED(status)){
            deletejob(jobs, pid);
        }
        // 进程收到信号终止
        else if (WIFSIGNALED(status)){
            printf ("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid,
WTERMSIG(status));
            deletejob(jobs, pid);
        }
        // 进程收到信号停止
        else if (WIFSTOPPED(status)){
            printf ("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid,
WSTOPSIG(status));
            job = getjobpid(jobs, pid);
            job->state = ST;
        }
        // 退出临界区，取消信号屏蔽
        sigprocmask(SIG_SETMASK, &prev_mask, NULL);
    }
    errno = olderrno;
    return;
}
```

7. sigint_handler() && sigtstp_handler()

这两个的信号处理方式较为类似，都是利用 kill 系统调用给相应的进程/进程组发 SIGINT 信号或者 SIGTSTP 信号；同样需要做信号屏蔽处理。

代码如下：

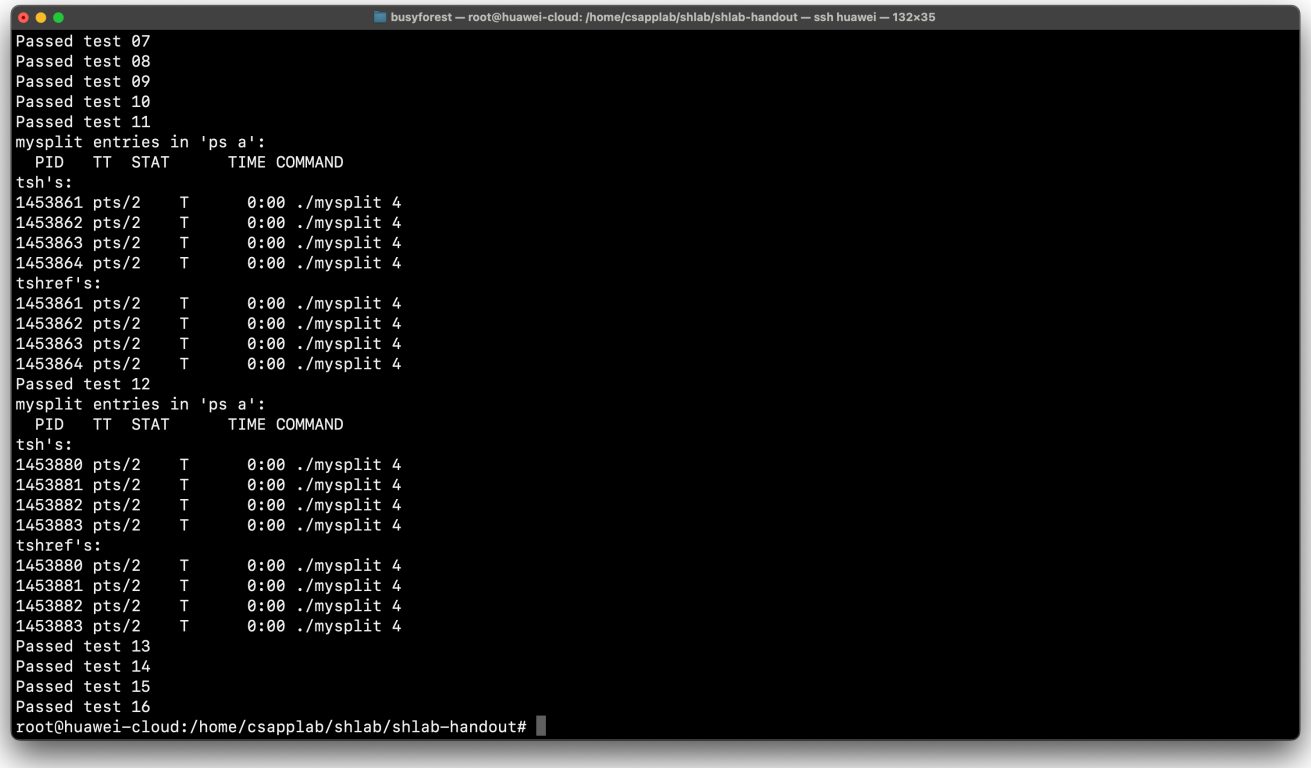
```
void sigint_handler(int sig)
{
```

```
    int olderrno = errno;
    int pid;
    sigset_t all_masks, prev_mask;
    Sigfillset(&all_masks);
    // jobs 属于临界区资源，加上屏蔽之后才能访问
    Sigprocmask(SIG_BLOCK, &all_masks, &prev_mask);
    if((pid = fgpid(jobs)) != 0){
        Kill(-pid, SIGINT);
    }
    Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
    errno = olderrno;
    return;
}

void sigtstp_handler(int sig)
{
    int olderrno = errno;
    int pid;
    sigset_t all_masks, prev_mask;
    Sigfillset(&all_masks);
    Sigprocmask(SIG_BLOCK, &all_masks, &prev_mask);
    if((pid = fgpid(jobs)) > 0){
        Kill(-pid, SIGTSTP);
    }
    Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
    errno = olderrno;
    return;
}
```

8.运行结果

运行 checker.py，执行结果如下：


busyforest — root@huawei-cloud: /home/csapplab/shlab/shlab-handout — ssh huawei — 132×35
Passed test 07
Passed test 08
Passed test 09
Passed test 10
Passed test 11
mysplit entries in 'ps a':
PID TT STAT TIME COMMAND
tsh's:
1453861 pts/2 T 0:00 ./mysplit 4
1453862 pts/2 T 0:00 ./mysplit 4
1453863 pts/2 T 0:00 ./mysplit 4
1453864 pts/2 T 0:00 ./mysplit 4
tshref's:
1453861 pts/2 T 0:00 ./mysplit 4
1453862 pts/2 T 0:00 ./mysplit 4
1453863 pts/2 T 0:00 ./mysplit 4
1453864 pts/2 T 0:00 ./mysplit 4
Passed test 12
mysplit entries in 'ps a':
PID TT STAT TIME COMMAND
tsh's:
1453880 pts/2 T 0:00 ./mysplit 4
1453881 pts/2 T 0:00 ./mysplit 4
1453882 pts/2 T 0:00 ./mysplit 4
1453883 pts/2 T 0:00 ./mysplit 4
tshref's:
1453880 pts/2 T 0:00 ./mysplit 4
1453881 pts/2 T 0:00 ./mysplit 4
1453882 pts/2 T 0:00 ./mysplit 4
1453883 pts/2 T 0:00 ./mysplit 4
Passed test 13
Passed test 14
Passed test 15
Passed test 16
root@huawei-cloud:/home/csapplab/shlab/shlab-handout#