Shlab Report

本次lab实现了一个简易的shell `tsh`，支持程序执行和作业控制。

下面迭代式地实现各个需求。首先实现builtin指令的处理：

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    int bg; /* Is the cmd backgroud? */

    bg = parseline(cmdline, argv);
    if (argv[0] == NULL || builtin_cmd(argv))
        return;
}

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

tsh支持四条builtin指令，分别是`quit`、`jobs`、`fg`、`bg`。`eval`先检查运行的指令是否是这四条之一，如果是就直接在当前进程中处理。

前面的代码已经支持quit和jobs两条指令。fg和bg的处理放在`do_bgfg`。主要代码如下：

```
void do_bgfg(char **argv)
{
```

```
    char *id; /* pid or jid */
    struct job_t *job;
    int jid;
    pid_t pid;

    /* get job from pid/jid */
    ...

    /* let the specified process group continue */
    if (kill(-pid, SIGCONT) < 0)
        unix_error("kill (cont) error");

    if (!strcmp("fg", argv[0])) {
        job->state = FG;
        waitfg(pid); /* block */
    }
    else {
        job->state = BG;
        printf("[%d] (%d) %s", jid, pid, job->cmdline);
    }
}
```

其中，程序从jobs中找到给定jid或pid对应的job，用kill向pid所在的进程组发送一个SIGCONT信号使它继续运行，再修改对应job的状态。如果job变成了前台任务需要调用`waitfg`阻塞。

waitfg简单地使用忙等待，循环查询等待的pid是否还是前台进程。

```
void waitfg(pid_t pid)
{
    struct job_t *job;

    if ((job = getjobpid(jobs, pid)) != NULL) {
        while (pid == fgpid(jobs)) ;
    }
}
```

waitfg通过`fgpid(jobs)`获得当前前台进程。前台进程暂停/终止时系统会向shell所在的进程发送一个SIGCHLD信号，这时`sigchld_handler`会修改前台job状态为暂停或者删除已终止的前台job，使得fgpid返回值改变，从而使waitfg结束等待。因此我们需要实现SIGCHLD的handler：

```
void sigchld_handler(int sig)
{
```

```
    int status;
    pid_t pid;

    while ((pid = waitpid(-1, &status, WNOHANG|WUNTRACED)) > 0) {
        if (WIFSTOPPED(status)) { /* a child stopped */
            printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid),
pid, WSTOPSIG(status));
            getjobpid(jobs, pid)->state = ST;
        }
        else if (WIFSIGNALED(status)) { /* a child terminated by signal
*/
            printf("Job [%d] (%d) terminated by signal %d\n",
pid2jid(pid), pid, WTERMSIG(status));
            deletejob(jobs, pid);
        }
        else if (WIFEXITED(status)) { /* a child terminated normally */
            deletejob(jobs, pid);
        }
    }
}
```

handler的核心是一个waitpid调用`waitpid(-1, &status, WNOHANG|WUNTRACED)`。选项WNOHANG会使waitpid不因为还不可用的child阻塞（此时返回0），WUNTRACED则使status包含被暂停的child的状态，进而可以通过`WIFSTOPPED(status)`和`WSTOPSIG(status)`得到子进程是否是被暂停以及被什么信号暂停。对于waitpid得到的子进程pid，分三种情况（暂停/异常终止/正常终止）进行处理。使用while循环的原因是可能同时有多个子进程完成，handler需要回收所有当前available的zombie children。

至此我们已经实现了所有的builtin指令。接下来是SIGINT和SIGTSTP信号的处理：

```
void sigint_handler(int sig)
{
    pid_t pid;

    if ((pid = fgpid(jobs))) {
        if (kill(-pid, sig) < 0)
            unix_error("kill (int) error");
    }
}

void sigtstp_handler(int sig)
{
    pid_t pid;
```

```
    if ((pid = fgpid(jobs))) {
        if (kill(-pid, sig) < 0)
            unix_error("kill (tstp) error");
    }
}
```

两个信号的handler是类似的：如果当前有前台job，用kill向前台job进程组发送一个对应信号，否则直接忽略信号。

然后我们修改`eval`以支持外部程序的执行。shell做法是fork出一个子进程，在其中exec运行。然而如hints所述，有一些需要注意的点：

1. 在shell所在的进程中，fork完需要把新的job加到jobs里。由于fork的子进程和父进程的运行先后是不确定的，可能存在子进程先结束，父进程先处理SIGCHLD的情况。这会导致父进程先在handler内deletejob，再在eval内addjob，产生错误结果。因此shell需要先阻塞SIGCHLD直到完成addjob，以避免竞争条件。
2. shell也是前台进程组中的一员。按照要求我们的handler需要发送SIGINT/SIGTSTP到整个前台进程组，普通的fork&exec产生的子进程与shell所在的父进程属于同一进程组，这种处理会使shell也被信号终止。因此在fork和exec之间可以调用`setpgid(0, 0)`更改子进程到一个新的进程组（new pgid = pid）。

这样最终的eval如下：

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    int bg; /* Is the cmd backgroud? */
    pid_t pid;
    sigset_t mask;

    bg = parseline(cmdline, argv);
    if (argv[0] == NULL || builtin_cmd(argv))
        return;

    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    if (sigprocmask(SIG_BLOCK, &mask, NULL) < 0) /* block sigchld */
        unix_error("sigprocmask error");

    if ((pid = fork()) < 0) {
        unix_error("fork error");
    }
    else if (pid == 0) { /* child process */
```

```
        if (sigprocmask(SIG_UNBLOCK, &mask, NULL) < 0) /* unblock
sigchld */
            unix_error("sigprocmask error");
        if (setpgid(0, 0) < 0) /* put the child in a new process group
*/
            unix_error("setpgid error");
        if (execve(argv[0], argv, environ) < 0) {
            printf("%s: Command not found\n", argv[0]);
            exit(1);
        }
    }
    else { /* parent process */
        addjob(jobs, pid, bg ? BG : FG, cmdline);
        if (sigprocmask(SIG_UNBLOCK, &mask, NULL) < 0) /* unblock
sigchld */
            unix_error("sigprocmask error");
        if (bg)
            printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
        else
            waitfg(pid);
    }
}
```