

实验：Shell Lab

周炎亮 2018202196 信息学院

一、实验目标

完成一个自己的Unix Shell程序，支持一般的工作调度，进程信号控制和前后台程序切换等。

二、实验思路

根据要求，我们要完成的函数有：

- `eval` Shell命令执行的主路线。
- `builtin_cmd` 执行Shell程序中的内建命令。
- `do_bgfg` 执行前台后台任务切换及调度相关的内建命令。
- `waitfg` 等待一个前台任务结束。
- `sigchld_handler` 捕获并处理SIGCHLD信号。
- `sigint_handler` 捕获并处理SIGINT (ctrl-c) 信号。
- `sigtstp_handler` 捕获并处理SIGTSTP (ctrl-z) 信号。

首先从 `eval` 和 `builtin_cmd` 入手（因为课本中最先提到的也是这两个函数），课本中的 `eval` 代码如下：

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];    /* Holds modified command line */
    int bg;               /* Should the job run in bg or fg? */
    pid_t pid;            /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return; /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) { /* Child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}
```

`builtin_cmd` 代码如下:

```
int builtin_command(char **argv)
{
    if (!strcmp(argv[0], "quit"))    /* quit command */
        exit(0);
    if (!strcmp(argv[0], "&"))        /* Ignore singleton */
        return 1;
    return 0;                        /* Not a builtin command */
}
```

在阅读代码和下发的资料后可知, 在 `eval` 收到一条命令后, 先调用 `parsetline` 函数判断该命令是前台还是后台, 并将结果保存在 `bg` 中, 之后再调用 `builtin_command` 函数判断该命令是否是内建命令, 若是, 则 `builtin_command` 函数会直接执行他它, 否则由 `eval` 创建子进程来执行该条命令。在创建子进程后, 如果该命令是前台命令, 则等待前台任务结束后执行该命令; 否则在后台执行, 并输出对应任务。

builtin_command

因为 `builtin_command` 函数代码量较小, 因此先分析该函数。本实验中的内建命令有:

- `quit` 退出tsh (退出并回收所有任务)。
- `jobs` 列出所有的后台任务。
- `bg <job>` 通过向 `<job>` 发送一个SIGCONT信号使其在后台继续执行, `<job>` 可以是PID或JID, 此时应打印出这个任务的相关信息和命令。
- `fg <job>` 通过向 `<job>` 发送一个SIGCONT信号使其在前台继续执行, `<job>` 可以是PID或JID。

当收到匹配到`quit`时, 直接调用 `exit` 函数退出该进程; 当匹配到`jobs`时, 调用 `listjobs` 函数显示当前任务; 当匹配到`bg`和`fg`时, 因为这两个指令由同一个函数 `do_bgfg` 来实现, 因此可以将两个匹配的判定合并, 并调用 `do_bgfg` 函数。

代码如下:

```
int builtin_cmd(char **argv)
{
    sigset_t mask_all, mask_prev;
    sigfillset(&mask_all);
    if(strcmp(argv[0], "quit") == 0)
        exit(0);
    if(strcmp(argv[0], "jobs") == 0)
    {
        sigprocmask(SIG_BLOCK, &mask_all, &mask_prev);
        listjobs(jobs);
        sigprocmask(SIG_SETMASK, &mask_prev, NULL);
        return 1;
    }
    if(strcmp(argv[0], "bg")==0 || strcmp(argv[0], "fg")==0)
    {
        do_bgfg(argv);
        return 1;
    }
    return 0;    /* Not a builtin command */
}
```

可以注意到在调用 `listjobs` 函数前，程序先**将所有信号阻塞**，再调用完后才释放。这是因为函数调用时使用了**全局变量**`jobs`，而这么做是为了**保护对共享全局数据结构的访问**。在之后其他函数中，对`jobs`的访问也会用此方法。

G3. 阻塞所有的信号，保护对共享全局数据结构的访问。如果处理程序和主程序或其他处理程序共享一个全局数据结构，那么在访问(读或者写)该数据结构时，你的处理程序和主程序应该暂时阻塞所有的信号。这条规则的原因是从主程序访问一个数据结构 d 通常需要一系列的指令，如果指令序列被访问 d 的处理程序中断，那么处理程序可能会发现 d 的状态不一致，得到不可预知的结果。在访问 d 时暂时阻塞信号保证了处理程序不会中断该指令序列。

eval

书中给到的 `eval` 函数只是一个简单的样例，远不能符合本次实验的要求：

1. 没有在 `fork` 前预先屏蔽SIGCHLD信号以避免子进程过早执行完导致父进程无法正确完成如 `addjob` 之类的工作。
2. 每生成一个子进程，应该对其使用 `setpgid(0, 0)`，防止Ctrl-C等指令对父进程产生影响。
3. 为了有效配合 `listjobs` 函数的运行，每次新建完子进程后都应该使用 `addjob` 函数将该进程放入进程列表中。

根据上述思路，每次 `fork` 前应屏蔽SIGCHLD信号，并在最后解除。

每个子进程被建立后也要解除该信号，并调用 `setpgid(0, 0)` 重新分配进程组。

之后根据bg的不同调用 `addjob`，且因为 `addjob` 也要访问`jobs`，因此这之前也要将所有信号屏蔽，之后恢复到原来只屏蔽SIGCHLD的状态。

并且根据bg的不同决定是否调用 `waitfg` 函数，且在测试时与样例程序比对**trace04**文件后发现，如果是后台程序，还应输出该进程的相关信息（任务号、进程号以及命令）。

代码如下：

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    char buf[MAXLINE];
    int bg;
    pid_t pid;
    sigset_t mask, prev_mask, all_mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    sigfillset(&all_mask);

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if(argv[0] == NULL)
        return;
    if(!builtin_cmd(argv))
    {
        sigprocmask(SIG_BLOCK, &mask, &prev_mask);
        if((pid = fork()) == 0)
        {
            sigprocmask(SIG_SETMASK, &prev_mask, NULL); //子进程解除屏蔽
            setpgid(0, 0); //重新分配进程组号
            if(execve(argv[0], argv, environ) < 0)
            {

```

```

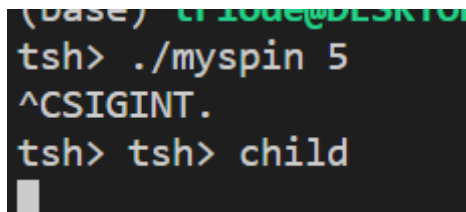
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
    }
}

sigprocmask(SIG_BLOCK, &all_mask, NULL);           //访问全局变量前屏蔽所有信号
if(bg)
    addjob(jobs,pid,BG,cmdline);
else
    addjob(jobs,pid,FG,cmdline);
sigprocmask(SIG_SETMASK, &mask, NULL);

if(bg)
    printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline); //输出后台进程的信
息
else
    waitfg(pid);
sigprocmask(SIG_SETMASK, &prev_mask, NULL);         //最后解除屏蔽
}
return;
}

```

在一开始测试时没有在父进程屏蔽SIGCHLD信号，就会发生以下情况：



```

(base) t10de@DESKTOP
tsh> ./myspin 5
^CSIGINT.
tsh> tsh> child
(base) t10de@DESKTOP

```

其中child是之后 `sigchld_handler` 处理信号时测试用的输出，可以看到子进程与父进程发生了竞争导致第三行出现了两个“tsh>”而第四行则没有“tsh>”。

do_bgfg

此函数是用以处理以bg/fg开头的命令，观察样例程序测试**trace14**时的输出：

```

./sdriver.pl -t trace14.txt -s ./tshref -a "-p"
#
# trace14.txt - Simple error handling
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 4 &
[1] (638) ./myspin 4 &
tsh> fg
fg command requires PID or %jobid argument
tsh> bg
bg command requires PID or %jobid argument
tsh> fg a
fg: argument must be a PID or %jobid
tsh> bg a
bg: argument must be a PID or %jobid
tsh> fg 9999999
(9999999): No such process
tsh> bg 9999999
(9999999): No such process
tsh> fg %2
%2: No such job
tsh> fg %1
Job [1] (638) stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (638) ./myspin 4 &
tsh> jobs
[1] (638) Running ./myspin 4 &

```

可以分为以下几类：

1. fg/bg后没有输入（报错）
2. fg/bg后有输入，但并不是合法任务号/进程号（报错）
3. fg/bg后有合法的任务号/进程号，但该任务号/进程号不存在（报错）
4. fg/bg后有正确且存在的任务号/进程号（执行）

且可以发现，在调用bg后也需要输出对应的任务信息（任务号、进程号以及命令）。

因此，可以首先判断bg/fg后是否有输入，如果没有，则直接输出错误信息并返回。

之后，再根据输入的字符将其分为任务号和进程号分开处理。并判断是否为合法的数字，若不是则输出错误信息并返回。之后调用 `getjobjid` / `getjobpid` 函数根据任务号/进程号获取进程信息，如果无法取得信息，则说明该任务号/进程号不存在，输出错误信息并返回。

最后，根据bg/fg的不同更改对应任务状态，并向对应任务发送SIGCONT信号。且类似 `eval` 函数的处理过程：若为fg，则调用 `waitfg` 函数；若为bg，则输出对应任务信息（任务号、进程号以及命令）。

代码如下：

```

void do_bgfg(char **argv)
{
    sigset_t mask_all, mask_prev;
    sigfillset(&mask_all);
    struct job_t *job;

```

```

char *id = argv[1];
if(id == NULL){
    printf("%s command requires PID or %%jobid argument\n",argv[0]);
    return;
}
if(id[0] == '%')
{
    int jid = atoi(id + 1);
    if(jid == 0)
    {
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }
    sigprocmask(SIG_BLOCK, &mask_all, &mask_prev);
    job = getjobjid(jobs, jid);
    sigprocmask(SIG_SETMASK, &mask_prev, NULL);
    if(job == NULL)
    {
        printf("%%d: No such job\n", jid);
        return;
    }
}
else
{
    pid_t pid = atoi(id);
    if(pid == 0)
    {
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }
    sigprocmask(SIG_BLOCK, &mask_all, &mask_prev);
    job = getjobpid(jobs, pid);
    sigprocmask(SIG_SETMASK, &mask_prev, NULL);
    if(job == NULL)
    {
        printf("(%d): No such process\n", pid);
        return;
    }
}
if(strcmp(argv[0], "bg") == 0)
{
    job->state = BG;
    kill(-(job->pid), SIGCONT);
    printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
}
else
{
    job->state = FG;
    kill(-(job->pid), SIGCONT);
    waitfg(job->pid);
}
return;
}

```

因为 getjobjid / getjobpid 也要访问 jobs，因此在调用前也要屏蔽所有信号。

waitfg

该函数是用来等待前台任务结束。如果使用循环调用 `pause()` 的方法，则可能因为竞争导致永远睡眠；而如果使用 `sleep()`，则又会导致效率过低。因此使用 `sigsuspend` 函数。

如果输入的 `pid` 并不是前台进程的 `pid`，那就说明该进程（已）不是前台进程，那么就没有等待的必要，因此循环的条件为 `pid == fgpid(jobs)`。

因为该函数中对 `jobs` 的访问是在 `while` 的循环条件中，且 `sigsuspend` 函数也会对信号屏蔽做处理，因此本代码中在访问 `jobs` 前屏蔽所有信号的操作就无从下手（担心会破坏函数的原子性）。

代码如下：

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

之后便是三个处理不同信号的函数，先分析两个较简单的 `sigint_handler` 和 `sigstsp_handler`

sigint_handler

因为 `Ctrl-C`（以及 `sigstsp_handler` 函数对应的 `Ctrl-Z`）都是针对前台进程，因此函数只需先获得前台进程的进程号，然后再利用 `kill` 函数对进程组发送对应信号（`SIGINT`）即可。

代码如下：

```
void sigint_handler(int sig)
{
    int olderrno = errno;
    sigset_t mask_all, mask_prev;
    pid_t pid;

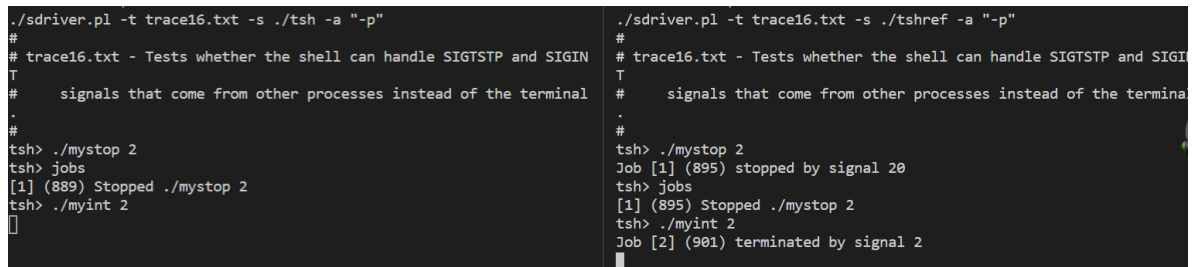
    sigfillset(&mask_all);
    sigprocmask(SIG_BLOCK, &mask_all, &mask_prev);
    pid = fgpid(jobs);
    sigprocmask(SIG_SETMASK, &mask_prev, NULL);

    if(pid != 0){
        kill(-pid, SIGINT);
    }
    //printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, sig);
    errno = olderrno;
    return;
}
```

可以看到代码首位多了 `int olderrno = errno;` 和 `errno = olderrno;` 这两句，这么做是为了防止类似的异步信号处理函数出错更改了 `errno` 导致其他可能依赖于 `errno` 的函数发生错误，因此在一开始使用一个局部变量将 `errno` “保护”起来（之后两个信号处理函数同样如此）。

G2. 保存和恢复 `errno`。许多 Linux 异步信号安全的函数都会在出错返回时设置 `errno`。在处理程序中调用这样的函数可能会干扰主程序中其他依赖于 `errno` 的部分。解决方法是在进入处理程序时把 `errno` 保存在一个局部变量中，在处理程序返回前恢复它。注意，只有在处理程序要返回时才有此必要。如果处理程序调用 `_exit` 终止该进程，那么就不需要这样做了。

此外，在测试 `trace06` 与样例程序对比时发现，如果函数因为 `SIGINT`（以及 `SIGTSTP`）信号中断那么也需要在屏幕上有相应输出。原本该输出是在本函数中实现的，但在 `trace16` 测试由其他进程发出信号导致当前进程中断的操作时发现这样的输出方式就不会输出对应信息（如下图），因此最后选择在 `sigchld_handler` 函数里面进行相应输出。



```
./sdriver.pl -t trace16.txt -s ./tsh -a "-p"
#
# trace16.txt - Tests whether the shell can handle SIGTSTP and SIGINT
#
# signals that come from other processes instead of the terminal
#
tsh> ./mystop 2
tsh> jobs
[1] (889) Stopped ./mystop 2
tsh> ./myint 2
[1]

./sdriver.pl -t trace16.txt -s ./tshref -a "-p"
#
# trace16.txt - Tests whether the shell can handle SIGTSTP and SIGINT
#
# signals that come from other processes instead of the terminal
#
tsh> ./mystop 2
Job [1] (895) stopped by signal 20
tsh> jobs
[1] (895) Stopped ./mystop 2
tsh> ./myint 2
Job [2] (901) terminated by signal 2
```

sigtstp_handler

该函数大体上同 `sigint_handler` 函数，只需将要发送的信号改为 `SIGTSTP` 即可。

代码如下：

```
void sigtstp_handler(int sig)
{
    int olderrno = errno;
    sigset_t mask_all, mask_prev;
    pid_t pid;

    sigfillset(&mask_all);
    sigprocmask(SIG_BLOCK, &mask_all, &mask_prev);
    pid = fgpid(jobs);
    sigprocmask(SIG_SETMASK, &mask_prev, NULL);

    if(pid != 0){
        kill(-pid, SIGTSTP);
    }
    //printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, sig);
    errno = olderrno;
    return;
}
```

sigchld_handler

对于子进程的回收，课件中使用的方法是用循环：`while ((pid = waitpid(-1, NULL, 0)) > 0)` 的方式尽可能多的回收子进程，虽然上课时说要把 `options` 改为 `WNOHANG`，但是保险起见在本代码中使用 `WNOHANG` | `WUNTRACED`。

子进程中断有以下原因：

1. 正常退出
2. 因为 `SIGINT` 信号而退出
3. 因为 `SIGTSTP` 信号而暂停

对于前两种情况，因为进程已经执行完毕，因此需要将其从jobs中删除；对于第三种情况，则需要将相应状态改为ST (stopped) 。

对于后两种情况，如之前所述，需要输出相应信息。

不同状态的甄别以及输出中信号的获取可以使用 `wait.h` 中定义的几个宏实现。

`deletejob` 和 `getjobpid` 都访问了jobs，因此需要在调用前阻塞所有信号。

代码如下：

```
void sigchld_handler(int sig)
{
    int olderrno = errno;
    sigset_t mask_all, mask_prev;
    pid_t pid;
    int status;
    struct job_t *job;

    sigfillset(&mask_all);
    while((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)
    {
        //printf("child\n");
        sigprocmask(SIG_BLOCK, &mask_all, &mask_prev);
        job = getjobpid(jobs, pid);
        if(WIFEXITED(status))
        {
            //printf("WIFEXITED\n");
            deletejob(jobs, pid);
        }
        else if(WIFSIGNALED(status))
        {
            //printf("WIFSIGNALED\n");
            printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid,
WTERMSIG(status));
            deletejob(jobs, pid);
        }
        else if(WIFSTOPPED(status))
        {
            //printf("WIFSTOPPED\n");
            printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid,
WSTOPSIG(status));
            if(job != NULL)
                job->state = ST;
        }

        sigprocmask(SIG_SETMASK, &mask_prev, NULL);
    }
    errno = olderrno;
    return;
}
```

三、其他

重难点：

- 在本次试验中，我认为较难的点为 `eval` 以及 `do_bgfg` 函数。前者是因为万事开头难，且涉及到父子进程的操作，特别是子进程进程组重新编号（如果材料中不提到应该也不会想到这点）。而后者

是因为需要考虑的情况较多（4类情况），需要额外小心（还好有测试程序的帮助）。

至于实验中碰到的问题，在上述实验思路中也基本提到了，在此做一下总结和补充：

- 在 `eval` 函数中，创建子进程之前需要先将 `SIGCHLD` 信号屏蔽，否则会造成竞争的情况。
- 对于 `SIGINT` 和 `SIGTSTP` 信号处理的输出应在 `sigchld_handler` 函数中实现，以便能处理由其他进程发送的相同信号。
- 在前一点提到的输出中，在输出对应信号的序号时不能直接输出传入的参数 `sig`，而是要调用 `WTERMSIG` 和 `WSTOPSIG` 获取导致进程停止的信号序号，否则会出现以下情况（17的地方应为20和2）：

```
./sdriver.pl -t trace16.txt -s ./tsh -a "-p"
#
# trace16.txt - Tests whether the shell can handle SIGT
T
#      signals that come from other processes instead of
.
#
tsh> ./mystop 2
Job [1] (941) stopped by signal 17
tsh> jobs
[1] (941) Stopped ./mystop 2
tsh> ./myint 2
Job [2] (944) terminated by signal 17
```

- 在 `sigchld_handler` 函数中误将循环写成了 `while((pid == waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)` 导致进程无法正确处理命令。

优秀的地方：

- 上述提到的在访问全局变量前阻塞所有信号以及在信号处理函数使用 `int olderrno = errno;` 和 `errno = olderrno;` 语句是在实验过程中查看课本时发现而中途加上的，原本的代码，特别是 `sigint_handler` 和 `sigtstp_handler` 函数十分简洁，比如：

```
void sigtstp_handler(int sig)
{

    pid_t pid;
    pid = fgpid(jobs);
    if(pid != 0){
        kill(-pid, SIGTSTP);
    }
    //printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, sig);
    return;
}
```

- 改进后的代码大大提高了严谨性。
- 且不断地与样例程序对比，使得代码最后的输出也基本符合样例程序。

总结反思：

- 不足之处是没有找到很好的办法在 `waitfg` 函数中，在访问 `jobs` 之前有效地阻塞所有信号以确保严谨性。