[读书笔记]CSAPP：ShellLab

README：http://csapp.cs.cmu.edu/3e/README-shlab

说明：http://csapp.cs.cmu.edu/3e/shlab.pdf

代码：http://csapp.cs.cmu.edu/3e/shlab-handout.tar

复习：http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/recitations/rec09.pdf

---

该实验主要自己实现一个简易的shell，有一些难度，而且有些错误很难找，做完后感觉对shell有了更好的理解。

该实验在`tsh.c`文件中实现了大部分的框架，需要自己完成以下函数内容：

- `eval`：解析和解释命令行的主例程，大约70行。
- `builtin_cmd`：识别并解释内置命令：`quit`、`fg`、`bg`和`job`，大约25行。
-
  - `quit`：退出当前shell
  - `fg`：通过发送`SIGCONT`信号来重启，并在前台运行。其中可以是作业也可以是进程，`%1`表示作业号为`1`的作业，`1`表示进程号为`1`的进程。
  - `bg`：通过发送`SIGCONT`信号来重启``，并在后台运行。
  - `job`：列出所有后台作业。
- `waitfg`：等待前台作业完成
- `sigchld_handler`：`SIGCHLD`信号的处理函数
- `sigint_handler`：`SIGINT`信号的处理函数
- `sigtstp_handler`：`SIGTSTP`信号的处理函数

我们希望实现的shell具有以下功能：

- 提示应为字符串`tsh>`
- 用户键入的命令行应包含一个名称和零个或多个参数，所有参数均由一个或多个空格分隔。 如果名称是内置命令，则shell应该立即处理它并等待下一个命令行。 否则，shell应该假定名称是可执行文件的路径，它在初始子进程的上下文中加载并运行。
- shell不需要支持管道`|`或I/O重定向`<`和`>`
- 键入`ctrl-c`（`ctrl-z`）应该会导致`SIGINT`（`SIGTSTP`）信号发送到当前前台作业以及该作业的任何后代，如果没有前台作业，那么信号应该没有效果。
- 如果命令行以`&`结束，则shell应该在后台运行作业，否则它将在前台运行该作业。
- 每个作业都可以通过进程ID（PID）或作业ID（JID）进行标识，该ID是tsh分配的正整数。
- shell需要支持以下内置命令：`quit`、`jobs`、`bg`和`fg`。
- shell应该回收所有僵死子进程，如果任何作业由于接收到未捕获到的信号而终止，则shell应该识别此事件并打印一条消息，其中包含该作业的PID和有问题的信号的描述。

通过`make`来得到我们shell的可执行目标文件，然后这里给出了一系列的验证文件，比如`trace01.txt`，其中包含了一些命令，我们可以通过`make test01`来得到我们shell的输出结果，可以和`make rtest01`输出的结果对比，或`tshref.out`比较，判断我们shell是否正确。

接下来将通过以下各个函数来介绍shell的行为

```
int main(int argc, char **argv)
{
    char c;
    char cmdline[MAXLINE];
    int emit_prompt = 1; /* emit prompt (default) */
```

```
    /* Redirect stderr to stdout (so that driver will get all output
     * on the pipe connected to stdout) */
    dup2(1, 2);

    /* Parse the command line */
    while ((c = getopt(argc, argv, "hvp")) != EOF) {
        switch (c) {
            case 'h':             /* print help message */
                usage();
                break;
            case 'v':             /* emit additional diagnostic info */
                verbose = 1;
                break;
            case 'p':             /* don't print a prompt */
                emit_prompt = 0;  /* handy for automatic testing */
                break;
            default:
                usage();
        }
    }

    /* Install the signal handlers */

    /* These are the ones you will need to implement */
    Signal(SIGINT,  sigint_handler);   /* ctrl-c */
    Signal(SIGTSTP, sigtstp_handler);  /* ctrl-z */
    Signal(SIGCHLD, sigchld_handler);  /* Terminated or stopped child */

    /* This one provides a clean way to kill the shell */
    Signal(SIGQUIT, sigquit_handler);

    /* Initialize the job list */
    initjobs(jobs);

    /* Execute the shell's read/eval loop */
    while (1) {
        /* Read command line */
        if (emit_prompt) {
            printf("%s", prompt);
            fflush(stdout);
        }
        if ((fgets(cmdline, MAXLINE, stdin) == NULL) && ferror(stdin))
            app_error("fgets error");
        if (feof(stdin)) { /* End of file (ctrl-d) */
            fflush(stdout);
            exit(0);
        }

        /* Evaluate the command line */
        eval(cmdline);
        fflush(stdout);
        fflush(stdout);
    }

    exit(0); /* control never reaches here */
}
```

在`main`函数中，我们为了能捕获`Ctrl+C`、`Ctrl+Z`、命令`quit`和子进程发出的`SIGINT`、`SIGTSTP`、`SITGQUIT`和`SIGCHLD`信号，需要通过`Signal`将信号和对应的处理函数绑定起来。然后通过死循环通过`eval`函数来重复解析输入的命令行。

```
void eval(char *cmdline){
    int bg;
    static char array[MAXLINE]; /* holds local copy of command line */
        char *buf = array;          /* ptr that traverses command line */
    char *argv[MAXARGS];    //命令行参数
    pid_t pid;  //子进程PID
    sigset_t mask_one, prev, mask_all;

    strcpy(buf, cmdline);   //缓存命令行
    bg = parseline(buf, argv);

    if(argv[0] == NULL)
        return;

    if(!builtin_cmd(argv)){
        sigemptyset(&mask_one);
        sigaddset(&mask_one, SIGCHLD);
        sigfillset(&mask_all);
        //防止addjob和deletejob竞争，需要先阻塞SIGCHLD信号
        sigprocmask(SIG_BLOCK, &mask_one, &prev);
        //如果不是内置命令，则fork一个子进程，并execve程序
        if((pid = fork()) == 0){    //子进程中
            //printf("in process:%d\n",pid);
            fflush(stdout);
            setpgid(0, 0);  //将子进程放入新的进程组，防止和shell冲突
            sigprocmask(SIG_SETMASK, &prev, NULL);
            if(execve(argv[0], argv, environ) < 0){
                printf("%s: Command not found\n", argv[0]);
                exit(0);
            }
        }
        //printf("parent:%d\n",getpid());
        //对全局数据结构jobs进行访问时，要阻塞所有信号
        sigprocmask(SIG_BLOCK, &mask_all, NULL);
        addjob(jobs, pid, bg?BG:FG, buf);
        //listjobs(jobs);
        sigprocmask(SIG_SETMASK, &prev, NULL);  //解阻塞

        if(bg){ //后台作业
            printf("[%d] (%d) %s", pid2jid(pid), pid, buf);
        }else{  //前台作业
            waitfg(pid);    //需要等待前台作业完成
        }
    }
}
```

在`eval`函数中主要解析刚刚输入的命令行，首先通过`parseline`函数来将输入的命令行解析为命令和若干个参数的集合`argv`，其形式为`{命令, arg1, arg2, ..., NULL}`，并且会判断最后的字符是否为`&`来判断该命令是前台的还是后台的。

然后会将命令`argv[0]`输入到`build_cmd`来判断是否为内置命令，如果是内置命令，就直接在`build_cmd`函数中执行了，否则需要在shell中通过`fork`新建一个子进程，并在该子进程中用`execve`来执行该命令，然后在shell中通过`addjob`来添加该作业，如果是前台作业，就等待前台作业运行完毕，如果是后台作业，就执行解析下一条命令。

注意：

- **内置命令和非内置命令的区别：**内置命令会直接运行，而非内置命令需要shell通过`fork`新建一个子进程，并在子进程中通过`execve`执行该命令，并且需要添加该作业，并等待前台作业。

- 我们通过 `fork` 函数新建一个子进程后，该进程可能在任意时刻终止或停止，使得shell跳转去执行对应的信号处理程序，并在该信号处理程序中对该作业进行修改，如果在 `addjob` 函数之前跳转，则由于未保存该作业而导致错误，所以需要在 `fork` 函数之前将 `SIGCHLD` 信号阻塞。并且由于子进程共享 `fork` 函数之前的设置，所以子进程也阻塞了 `SIGCHLD` 信号，所以子进程中需要恢复接收该信号。由于 `execve` 函数除非出错，否则不会再返回，所以需要在执行 `execve` 之前恢复该信号。
- 这里可以看到在子进程中，我们通过 `setpgid(0, 0)` 将当前进程的进程组ID设置为自己的PID。这要从shell的角度来看，shell本身为一个进程，我们通过 `fork` 函数创建的子进程与shell本身都处在相同的进程组中，当我们键入 `Ctrl+C` 或 `Ctrl+Z` 想要终止或停止前台作业时，会发送 `SIGINT` 或 `SIGTSTP` 信号给shell，此时我们会通过 `kill` 函数将该信号发送到前台作业所在的进程组中的所有进程，来终止或停止这些进程，由于shell也在该进程组中，所以shell也会受到影响。所以需要将子进程和shell独立开来，通过将子进程设置自己的进程组ID，使得对子进程所在进程组发送信号时，不会影响到shell。
- 我们这里以看到一种shell视图，shell本身是一个进程，我们通过 `fork` 创建一个子进程，在该子进程中执行作业，该子进程具有自己的PID以及JID，而该子进程中执行命令会自己再创建一系列进程，这些进程都属于相同的进程组中，对于shell而言，它可以通过PID和JID获得该子进程或该作业的入口。
- 这里将保存作业的数组定义为

```
struct job_t {              /* The job struct */
    pid_t pid;              /* job PID */
    int jid;                /* job ID [1, 2, ...] */
    int state;              /* UNDEF, BG, FG, or ST */
    char cmdline[MAXLINE];  /* command line */
};
struct job_t jobs[MAXJOBS]; /* The job list */
```

则在 `addjob` 函数中，会对全局共享结构体 `jobs` 进行修改，从安全信号处理的角度来看，需要在执行 `addjob` 函数之前阻塞所有信号，然后在 `addjob` 之后恢复对这些信号的接收，防止 `addjob` 函数在对 `jobs` 进行修改时，由于信号中断而使得 `jobs` 中各部分的状态不同。

- **前台作业和后台作业的区别**：前台作业shell需要等待该作业执行完毕，而后台作业shel无需等待。

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

该内置命令只要执行一些简单的命令，如果输入的命令 `argv` 是内置命令，则返回1，否则返回0。其中 `do_bgfg` 函数主要用来执行 `bg` 和 `fg` 内置命令。

```
 void do_bgfg(char **argv){
    int jid;
    struct job_t *job;
    pid_t pid;
    sigset_t mask, prev;
```

```
    if(argv[1] == NULL){
        printf("%s command requires PID or %%jobid argument\n",argv[0]);
        return;
    }

    //首先确定是pid还是jid，然后将其转化为kill的参数
    if(sscanf(argv[1],"%%%d",&jid) > 0){    //jid
        job = getjobjid(jobs, jid); //需要获得job，因为要修改job信息
        if(job == NULL || job->state == UNDEF){
            printf("%s: No such job\n", argv[1]);
            return;
        }
    }else if(sscanf(argv[1],"%d",&pid) > 0){    //pid
        job = getjobpid(jobs, pid);
        if(job == NULL || job->state == UNDEF){
            printf("(%s): No such process\n", argv[1]);
            return;
        }
    }else{
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }
    //修改job信息
    sigfillset(&mask);
    sigprocmask(SIG_BLOCK, &mask, &prev);
    if(!strcmp(argv[0], "fg")){
        job->state = FG;
    }else{
        job->state = BG;
    }
    sigprocmask(SIG_SETMASK, &prev, NULL);

    pid = job->pid;
    //发送SIGCONT重启
    kill(-pid, SIGCONT);
    if(!strcmp(argv[0], "fg")){
        waitfg(pid);
    }else{
        printf("[%d] (%d) %s", job->jid, pid, job->cmdline);
    }
}
```

我们通过`waitfg`函数来等待前台作业的完成

```
void waitfg(pid_t pid){
    //这里就用简单的sleep来等待
    while(pid == fgpid(jobs)){
        sleep(1);
    }
}
```

这里根据实验要求，就用简单的`sleep`函数等待一段时间，反复检测当前子进程的PID是否为前台进程的PID。还有更好的方式来实现等待，可以通过`sigsuspend`函数。

接下来是比较复杂的信号处理程序，先从简单的来看

```
void sigint_handler(int sig){
    int old_errno = errno;  //首先需要保存原始的errno
    pid_t pid = fgpid(jobs);
    if(pid!=0){
```

```
        kill(-pid,sig);
    }
    errno = old_errno;
}

void sigtstp_handler(int sig){
    int old_errno = errno;  //首先需要保存原始的errno
    pid_t pid = fgpid(jobs);
    if(pid!=0){
        kill(-pid,sig);
    }
    errno = old_errno;
}
```

当我们键入`Ctrl+C`时，内核会发送`SIGINT`信号给shell，而shell只需要通过`kill`函数将其转发给子进程，当我们键入`Ctrl+Z`时也同理。

注意：

- 我们需要保存`errno`，并在返回时重新赋值，防止它被改变。
- 我写这段代码时，一直有个疑惑，我们在最外侧通过`Signal`来捕获了这两个信号，则`fork`新建的子进程应该也会通过这两个信号处理程序来处理该信号。所以当shell发送`SIGINT`或`SIGTSTP`信号给前台进程时，前台进程不也会跑到这两个信号处理程序中吗？但是我发现会跑到`SIGCHLD`的信号处理程序中。**这里是由于：**
- - 当一个进程调用`fork`时，因为子进程在开始时复制父进程的存储映像，信号处理程序的地址在子进程中是有意义的，所以子进程继承父进程定义的信号处理程序。
  - 但是当子进程调用`execve`后，因为`execve`运行新的程序后会覆盖从父进程继承来的存储映像，那么信号处理程序在新程序中已无意义，所以`execve`会将原先设置为要捕捉的信号都更改为默认动作。所以当shell发送`SIGINT`和`SIGTSTP`信号给子进程时，他们会执行默认操作，即终止进程，所以当子进程终止时，内核会发送`SIGCHLD`信号给父进程，所以可以发送跳转到shell的`SIGCHLD`的信号处理程序中了。

```
void sigchld_handler(int sig){
    int old_errno = errno;  //首先需要保存原始的errno
    pid_t pid;
    sigset_t mask, prev;
    int state;  //保存waitpid的状态，用来判断子进程是终止还是停止
    struct job_t *job;

    sigfillset(&mask);
    //由于信号不存在队列，而waitpid一次只会回收一个子进程，所以用whild
    while((pid = waitpid(-1, &state, WNOHANG | WUNTRACED)) > 0){    //要检查停止和终止的，
并且不要卡在这个循环中
        //对全局结构变量jobs进行修改时，要阻塞所有信号
        sigprocmask(SIG_BLOCK, &mask, &prev);
        if(WIFEXITED(state)){   //子进程通过调用exit或return正常终止，需要从jobs中删除该作业
            deletejob(jobs, pid);
        }else if(WIFSIGNALED(state)){   //子进程因为一个未捕获的信号终止
            printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid,
WTERMSIG(state));
            deletejob(jobs, pid);
        }else if(WIFSTOPPED(state)){    //如果子进程是停止的，需要修改改作业的状态
            job = getjobpid(jobs, pid);
            job->state = ST;
            printf("Job [%d] (%d) stopped by signal %d\n", job->jid, pid,
WSTOPSIG(state));
        }
        sigprocmask(SIG_SETMASK, &prev, NULL);  //恢复信号接收
    }
```

```
    errno = old_errno;
}
```

`SIGCHLD`信号的处理程序比较复杂。首先，由于信号不存在队列，所以我们需要假设一个未处理信号表明至少有一个信号到达，所以我们需要使用`while`循环，而不能用`if`。其次，在使用`waitpid`函数回收子进程时，我们需要设置选项为`WNOHANG | WUNTRACED`，`WNOHANG`表示当没有子进程终止时，父进程不会被挂起，而是`waitpid`函数返回0，这样防止当shell中还存在子进程时，由于`while`的存在，而卡在这个循环中；`WUNTRACED`保证能返回被终止和停止的子进程PID，因为子进程收到`SIGINT`和`SIGTSTP`信号时会采取默认默认行为而终止和停止，则内核会发送`SIGCHLD`信号给shell，如果没有`WUNTRACED`，则当子进程是被停止的，则会卡在这个循环中。

这里可以通过不同的信息来决定如何修改`jobs`的状态。

此外我们还需注意：

- 执行信号处理程序和主程序处于相同的进程中
- 信号是内核发送给父进程的，比如键入`Ctrl+C`或`Ctrl+Z`时，内核会发送`SIGINT`或`SIGTSTP`信号给父进程，当子进程停止或终止时，内核会发送`SIGCHLD`信号给父进程，然后在父进程中执行对应的信号处理程序。需要时刻注意当前的执行的进程是什么。

简单总结以下shell的行为。

shell本身作为一个进程，接收到命令行后，会先判断是否为内置命令

- 如果该命令为内置命令，则shell直接执行该命令
- 如果不是内置命令，则shell会通过`fork`新建一个子进程，并未该子进程分配一个独立的进程组ID，与shell进程独立开来，然后通过`execve`函数来执行可执行目标文件。如果是前台作业，则shell会等待该作业执行完毕，如果是后台作业，则shell会直接读取下一条命令。**注意：**`execve`函数会消除我们定义的信号处理程序。

当我们键入`Ctrl+C`或`Ctrl+Z`时，由于我们是在执行shell进程，所以内核会发送`SIGINT`或`SIGTSTP`信号给shell，此时shell就需要将该信号通过`kill`函数发送给对应的前台进程。由于该前台进程执行了`execve`函数，所以会采用默认行为，要么终止或停止该子进程，然后内核会发送`SIGCHLD`信号给shell，表示有子进程被终止或停止了，然后shell再通过对应的信号处理程序对该子进程进行回收或修改作业状态。