shell lab实验报告

实验目标

在理解进程控制，信号及其屏蔽等知识的基础上， 设计开发一个tiny shell，用于模拟shell的部分功能，并实现几个简单的内建指令。

基础函数框架

实验的框架内容基本上已经完成了，命令语句字符串解析的工作也已经完成了，整个程序里边各个函数的功能也已经提示得很明确了：

main（）主函数，注册几个信号处理函数，使用一个while（1）死循环持续不断地从stdin之中读取指令，将全部的命令都给eval传参

parseline（）字符串解析函数：代码已经写好了，用途是把命令语句里边的各个参数都提取出来放到argv里边以供eval使用。

eval（）主体函数，对于main提供的每一条指令，首先利用parseline进行解析，将解析的结果放入到argv里边。首先判断该指令是不是内部指令，如果是的话就直接将参数列表argv传递给builtin-cmd函数执行即可。否则的话，eval要创建一个新的子进程，并利用execve将新的子进程换壳，用子进程来执行命令行。在这种情况下，需要分类讨论该命令是否是后台运行。这两者的区别是，如果程序后台执行，只需要输出该进程的相关信息即可继续结束返回主函数，等待下一条指令。如果是前台执行的话，需要使用waitfg函数等待前台进程执行完毕再返回。

builtin-cmd（）内建命令执行函数。对于quit和jobs两个内建指令，直接执行：quit意味着结束所有的进程，因此向进程列表之中的所有进程都发送SIGKILL信号；jobs命令要求罗列所有的信号，需要遍历一遍joblist列表，输出所有工作的相关信息。如果是fg和bg两个指令，调用专门的函数do_bgfg来执行

do-bgfg（）专门用来处理bg指令和fg指令。首先要从第二个参数argv[1]之中分别解析是pid还是jid，如果是bg，只需要向他们发送一个SIGCONT的信号就可以，如果是fg的话，由bg到fg最主要的区别就是eval主函数需要等待任务完成之后才可以返回。因此要从background 转换成为foreground，在do-bgfg内部调用wait-fg函数等待任务完成再返回就满足要求。

waitfg是用于等待前台任务的函数机制，这里主要使用sleep实现延时等待。但是，回收已经结束的前台任务进程这项工作是由sigchld handler来完成的，所以当sigchld handler完成了这项工作之后，需要设置某种标志，来告诉waitfg前台任务子进程已经被回收。具体实现细节如下所示。首先通过函数getjobpid使用进程号来获得一个指向joblist之中一个具体位置的指针。如果这个指针为空，说明前台任务在此时已经被完成，直接返回即可。while内部的循环条件是前台任务进程的pid是否发生变化。因为一旦被回收，在sigchld handler之中这项工作会从joblist之中删除，跟据delete job的实现原理，被清除的工作jobid和pid会被置为0。因此跟据whlie的条件可以判断出来是否前台任务已经被回收。

```
void waitfg(pid_t pid)
{
    struct job_t *p=getjobpid(jobs,pid);
    if(p==NULL)
        return;
    while(fgpid(jobs)==pid && fgpid(jobs)!=0){
        sleep(1);
        printf("sleep %d\n",pid);
    }
    return;
}
```

job组织管理

下面几个函数是已经是事先被写好的，有关joblist的一些操作函数。这里的joblist是我们的tiny shell管理所有工作进程的一套体系。每一个进程都有一个pid，但是这个pid是系统来分配的，用户没有办法自己设置。而jid是另一套编号体系，joblist通过这两种编号将所有的任务都组织管理起来，以便于可以查询所需要的工作并向他们发送信号。

- addjob添加一个工作到列表之中
- deletejob 删除一个工作
- fgpid获取前台工作的pid
- getjobpid通过pid获得一个指向任务列表之中相应任务的指针
- getjobjid通过jobid获得一个指向任务列表之中相应任务的指针
- pid2jid两种编号形式的转变
- listjobs打印所有的工作

joblist是通过数组来实现的，并且所有的查询类的操作都是通过遍历一遍来实现的。

特殊信号处理

几个信号处理的方式如下

sigchld-handler（）当有子进程终结时，父进程向内核发送的信号。尤其要特别注意waitpid的实现方式。因为出现了sigchld信号，证明一定"至少"会出现一个等待回收的进程。而之所以是"至少"，是因为信号是没有等待机制的，如果在处理信号的过程之中出现了新的信号，是不会递归地调用handler的；如果在信号被接受处理之前出现了新的信号，也无法重复计数，handler只会被调用一次。所以要使用while回收多个进程。与此同时，如果使用默认的option，会导致waitpid始终在等待，与sigchld-handler要达到的目的是相悖的。所以要使用WHOHANG|WUNREACED，表示非阻塞态，并且同时监察所有被停止的进程。非阻塞态的好处是，跟据waitpid的返回值，就可以知道有没有仍未被回收的子进程并且回收所有的现有进程。

```
void sigchld_handler(int sig)
{
    int status;
    int olderrno=errno;
    pid_t reapid;
    struct job_t* p;
    while((reapid=waitpid(-1,&status,WNOHANG|WUNTRACED))>0){
        if(WIFSIGNALED(status) && WTERMSIG(status)==2){
            printf("terminate%d\n",reapid);

            printf("Job [%d] (%d) ternimated by signal
2\n",pid2jid(reapid),reapid);
            deletejob(jobs,reapid);
        }
        if(WIFSTOPPED(status) && WSTOPSIG(status)==SIGTSTP){
            printf("stop%d\n",reapid);
            getjobpid(jobs,reapid)->state=ST;
            printf("Job [%d] (%d) stoped by signal
20\n",pid2jid(reapid),reapid);
        }
        if(WIFEXITED(status)){
            printf("reap%d\n",reapid);
            deletejob(jobs,reapid);
        }
    }
    if(reapid==-1 && errno!=ECHILD)
```

```
        unix_error("waitpid error");
    errno=olderrno;
    return;
```

sigint_handler（）和sigstp_handler()负责处理意外中断，分别向所有的前台进程通过kill发送SIGINT和SIGSTEP信号使其停止或终止。

```
void sigtstp_handler(int sig)
{
    printf("receive sigstp\n");
    struct job_t *p;
    pid_t pid=fgpid(jobs);
    if(pid){
        kill(-pid,SIGTSTP);
        //p=getjobpid(jobs,pid);
        //p->state=ST;
    }
    return;
}
void sigint_handler(int sig)
{
    printf("receive sigint\n");
    struct job_t *p;
    pid_t pid=fgpid(jobs);
    if(pid){
        kill(-pid,SIGINT);
        //p=getjobpid(jobs,pid);
        //p->state=ST;
        //deletejob(jobs,pid);
    }
    return;

}
```

疑难与特殊处理

1. 通过execve让子进程运行任务之前，必须通过setpgid（0，0）让这个进程单独成组。否则的话，它就会和tiny shell处于同一个进程组之中。这就会导致sigint和sigtstp等信号的处理出现很大的问题：终结一个job，会使得tiny shell同时也终结。
2. 需要设置信号屏蔽。因为使用了fork之后，要由子进程负责execve执行新的任务，有父进程把相关的任务信息和pid添加到joblist之中，而这个addjob的过程必须是在deletejob之前。deletejob是由sigchld-handler来完成的，所以在主进程addjob之前，必须使用信号屏蔽来忽略sigchld信号
3. 除了sigchld信号之外，还有两个信号需要设置屏蔽，sigint和sigtstp，因为这两个信号的处理方式，父进程和子进程是不一样的，我们的tiny shell应当拦截这些信号，将他们转发给所有的子进程，但是子进程应当按照默认行为处理。所以在创建子进程之后，在使用signal将sigint和sigtstp注销之前，需要将这两个信号屏蔽。
4. execve默认情况下调用一次，永不返回，但是在调用失败的时候，是会返回-1的，所以，像这种在一个函数之内使用了fork，处理完相应任务之后，应当加一句exit（），防止通过子进程返回这种意外的bug
5. 在写的过程之中还遇到的一个bug，就是使用waitpid函数的返回值status的时候，WIFSIGNALED（）是只针对被终止的程序，不处理被停止的程序！停止的程序如果被捕捉到了，有另外的接口。