Linklab实验报告

张睿恒 2024302182001

一、摘要

本总结共分为四个部分：摘要、实验概览、实验内容与总结。本实验的任务是实现一个简单的Unix Shell程序。通过此次实验，会对CSAPP的第八章异常控制流的内容以及并发式编程有着更深的理解和思考。

二、实验概览

本次实验需要我们实现如下几个函数：

- eval：解析命令行
- buildin_cmd：检测是否为内置命令quit，fg，bg，jobs
- do_bgfg：实现内置命令bg，fg
- waitfg：等待前台作业执行完成
- sigchld_handler：处理sigchld信号，即子进程停止或终止
- sigint_handler：处理sigint信号，即来自键盘键入的Ctrl-C
- sigtstp_handler：处理sigstp信号，即来自终端的停止信号

要求：

- Shell的提示符应当是"tsh>"
- 用户键入的命令行应当包括命令参数和若干个参数，并以一个或多个空格分隔
- 不需要支持管道或IO重定向
- 如果命令行以&结束，则后台运行；否则，前台运行
- tsh应当支持以下内置命令：quit，jobs，bg，fg
- tsh必须回收所有的僵尸进程

三、实验内容

本部分将阐述每个函数的功能与实现。

1 . eval

这个函数是用来解析命令行，并判断是否为内置命令还是程序路径并分别执行。如果是前台作业，那么要等待其完成；否则，输出后台作业的相应信息。代码如下：

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];          //存放解析的参数
    char buf[MAXLINE];            //解析cmdline
    int bg;                       //判断程序是前台还是后台执行
    int state;                    //指示前台还是后台运行状态
    pid_t pid;                    //执行程序的子进程的pid

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);  //解析参数
    state = bg? BG:FG;
    if(argv[0] == NULL)           //空行，直接返回
        return;
    sigset_t mask_all, mask_one, prev_one;
    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);
    if(!builtin_cmd(argv)) {                                //判断是否为内置命令
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_one);       //fork前阻塞SIGCHLD信号
        if((pid = Fork()) == 0) {                           //创建子进程
            Sigprocmask(SIG_SETMASK, &prev_one, NULL);      //解除子进程的阻塞
            Setpgid(0, 0);                                  //创建新进程组，ID设置为进
程PID
            Execve(argv[0], argv, environ);                 //执行
            exit(0);                                        //子线程执行完毕后一定要退出
        }
        if(state==FG){
            Sigprocmask(SIG_BLOCK, &mask_all, NULL);            //添加工作前阻塞所有信
号
            addjob(jobs, pid, state, cmdline);                  //添加至作业列表
            Sigprocmask(SIG_SETMASK, &mask_one, NULL);
            waitfg(pid);                                        //等待前台进程执行完毕
        }
        else{
            Sigprocmask(SIG_BLOCK, &mask_all, NULL);            //添加工作前阻塞所有信
号
            addjob(jobs, pid, state, cmdline);                  //添加至作业列表
            Sigprocmask(SIG_SETMASK, &mask_one, NULL);
            printf("[%d] (%d) %s",pid2jid(pid), pid, cmdline);  //打印后台进程信息
        }
        Sigprocmask(SIG_SETMASK, &prev_one, NULL);          //解除阻塞
    }
    return;
}
```

思路注释中已经写的比较详细了。值得注意的是，调用printf的时候也会产生阻塞信号：因为printf是线程不安全的，这里还打印了全局变量，所以可能出现读内存的同时另一个线程修改他的情况。

2.buildin_cmd

这个函数就很简单，判断是否为内置命令就结束了。代码如下：

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

3.do_bgfg

这个函数要实现内置命令bg和fg，功能如下

- bg：通过向对应的作业发送SIGCONT信号并前台运行
- fg：通过向对应的作业发送SIGCONT信号并后台运行

输入的参数有%则代表jid，没有则是pid。代码如下：

```
void do_bgfg(char **argv)
{
    struct job_t *job = NULL;         //要处理的job
    int state;                        //输入的命令
    int id;                           //存储jid或pid
    if(!strcmp(argv[0], "bg")) state = BG;
        else state = FG;
    if(argv[1]==NULL){                //没带参数
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }
    if(argv[1][0]=='%'){              //说明是jid
       if(sscanf(&argv[1][1], "%d", &id) > 0){
            job = getjobjid(jobs, id);  //获得job
            if(job==NULL){
                printf("%%%d: No such job\n", id);
                return;
            }
        }
    }
    else if(!isdigit(argv[1][0])) {  //其它符号，非法输入
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }
    else{                         //pid
        id = atoi(argv[1]);
        job = getjobpid(jobs, id);
        if(job==NULL){
            printf("(%d): No such process\n", id);
            return;
        }

    }
    Kill(-(job->pid), SIGCONT);       //重启进程，这里发送到进程组
    job->state = state;
    if(state==BG)
        printf("[%d] (%d) %s",job->jid, job->pid, job->cmdline);
    else
        waitfg(job->pid);
    return;
}
```

4. waitfg

这个函数要求实现阻塞父进程，直到当前的前台进程不再处于前台。这里要显示的等待信号。考虑以下代码：

```
while(fgpid(jobs) != 0){
    pause();
}
```

这个代码会有产生一个竞争，同时有可能导致致命的错误。考虑以下事件：父进程调用fgpid函数，此时有一个子进程仍然在前台运行，所以进入循环；但假定父进程进入循环之后但在执行pause之前进行了一个上下文切换，这是之前的前台子进程停止了；父进程在接收到SIGCHLD并处理完毕之后才会调用pause。由于pause只在接收到信号才会返回，这就导致父进程将永久休眠。解决的方法就是用sigsuspend函数，他相当于以下代码的原子版本：

```
sigprocmask(SIG_SETMASK,&mask,&prev);
pause();
sigprocmask(SIG_SETMASK,&prev,null);
```

因为是原子版本，所以不会产生刚才的上下文切换的问题。代码如下：

```
void waitfg(pid_t pid)
{
    sigset_t mask;
    Sigemptyset(&mask);
    while (fgpid(jobs) != 0){
        sigsuspend(&mask);       //暂停时取消阻塞
    }
    return;
}
```

5.sigchld_handler

这个函数是一个SIGCHLD信号处理函数，用来回收所有的僵尸进程。

```
void sigchld_handler(int sig)
{
    int olderrno = errno;   //由于errno是全局变量,注意保存和恢复errno
    int status;
    pid_t pid;
    struct job_t *job;
    sigset_t mask, prev;
    sigfillset(&mask);
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0){      //立即返回该子
进程的pid
        sigprocmask(SIG_BLOCK, &mask, &prev);   //阻塞所有信号
        if (WIFEXITED(status)){                 //正常终止
            deletejob(jobs, pid);
        }
        else if (WIFSIGNALED(status)){          //因为信号而终止，打印
            printf ("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid,
WTERMSIG(status));
            deletejob(jobs, pid);
        }
        else if (WIFSTOPPED(status)){           //因为信号而停止，打印
            printf ("Job [%d] (%d) stoped by signal %d\n", pid2jid(pid), pid,
WSTOPSIG(status));
            job = getjobpid(jobs, pid);
            job->state = ST;
        }
        sigprocmask(SIG_SETMASK, &prev, NULL);
    }
    errno = olderrno;
    return;
}
```

6.sigint_handler

这是一个SIGINT信号处理函数，原理跟上面的差不多。

```
void sigint_handler(int sig)
{
    int olderrno = errno;
    int pid;
    sigset_t mask_all, prev;
    Sigfillset(&mask_all);
    Sigprocmask(SIG_BLOCK, &mask_all, &prev);    //jobs为全局变量
    if((pid = fgpid(jobs)) != 0){
        Sigprocmask(SIG_SETMASK, &prev, NULL);
        Kill(-pid, SIGINT);
    }
    errno = olderrno;
    return;
}
```

7.sigstp_handler

这是一个SIGSTOP信号处理函数，将信号传递给前台的进程。

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

四、实验总结

本实验过程中还是蛮有意思的：我们每天都在用shell，但其实一个最基本的作业都要考虑很多问题：进程回收、避免竞争等等。当时学习第八章的时候，其实对好多地方都不甚理解，所以还是需要自己动手写一个简单的shell，才能更深体会异常和控制流的过程。同时也是我第一次接触并发式编成，确实思考的东西会很多。