CSAPP Shell Lab

该lab的目标是实现一个简单的支持任务调度的Unix shell程序tsh，tsh读取来自用户输入的一个命令行，解析命令行并代表用户运行相应的程序，整个lab代码量不大，难度主要集中在信号处理部分，需要仔细阅读csapp第八章的内容，书上对很多容易出错的地方都做了提示，并且提供了一部分基础代码。

shell 是一个交互式命令行解释器，它代表用户运行程序。 shell 读取并解析用户输入的命令行，然后按照命令行内容的指示执行一些操作。 命令行是由空格分隔的 ASCII 文本单词序列。命令行中的第一个单词要么是内置命令的名称，要么是可执行文件的路径名。剩下的词是命令行参数。如果第一个单词是内置命令，shell 会立即执行当前进程中的命令。否则，该词被假定为可执行程序的路径名。在这种情况下，shell 会fork一个子进程，然后在子进程的上下文中加载和运行程序。由于解释单个命令行而创建的子进程统称为作业(job)。一般来说，一个作业可以由多个通过 Unix 管道连接的子进程组成。如果命令行以 & 符号结尾，则作业在后台运行，这意味着 shell 在打印提示符并等待下一个命令行之前不会等待该作业终止。否则，作业在前台运行，这意味着 shell 在等待下一个命令行之前必须等待该作业终止。因此，在任何时候，最多只能有一个作业在前台运行。但是，可以有任意数量的作业在后台运行。

Unix shell 支持作业控制的概念，它允许用户在后台和前台之间来回移动作业，并更改作业中进程的进程状态（运行、停止或终止）。 键入 ctrl-c 会导致将 SIGINT 信号传递给前台作业进程组中的每个进程，SIGINT 信号的默认操作是终止进程。同样，键入 ctrl-z 会导致将SIGTSTP 信号传递给前台作业进程组中的每个进程，SIGTSTP 的默认操作是将进程置于停止状态，直到它被 SIGCONT 信号唤醒为止。 此外，Unix shell 还提供了各种支持作业控制的内置命令：

- `jobs`：列出处于运行状态和停止状态的后台作业
- `bg <job>`：将一个处于停止状态的后台作业切换至运行状态
- `fg <job>`：将一个处于停止或运行状态的后台作业切换至在前台运行
- `kill <job>`：结束一个作业

父进程创建一些子进程各自独立的运行，并且父进程能够在子进程运行时自由地去做其它的工作，当一个子进程终止或停止时，内核就会发送一个SIGCHLD信号给父进程，父进程捕获到这个SIGCHLD信号，调用处理程序回收子进程。然而信号处理程序与主程序是并发运行的，共享同样的全部变量，可能会出现一些难以调试的错误。因此在编写信号处理程序时需要遵循一些原则：

1. 处理程序尽可能简单
2. 处理程序中只调用异步信号安全的函数
3. 保存和恢复errno（只有在处理程序要返回时才有此必要）
4. 在访问共享全局数据结构时阻塞所有信号
5. 用`volatile`声明全局变量：`volatile`限定符告诉编译器不要缓存这个变量，强迫编译器每次都要从内存中读取变量的值（否则每次读取缓存在寄存器中的副本可能导致永远得不到处理程序更新过的值）
6. 用`sig_atomic_t`声明标志：C提供一种整型数据类型`sig_atomic_t`，对它的读和写保证会是原子的（注意这里对原子性的保证只适用于单个的读和写）

`void eval(char *cmdline)`：

`eval`函数解析和解释命令行，其首要任务是调用parseline函数，该函数解析以空格分隔的命令行参数，并构造最终会传递给execve的argv向量，argv是一个char指针数组，parseline函数将命令行参数按空格分隔后放入argv，如果命令行最后一个参数是一个“&”字符，那么parseline函数返回1，表示应该在后台执行该作业（shell不会等待它执行完成），返回0表示应该在前台执行该作业（shell会等待它完成）。在解析了命令行参数后，eval函数调用`builtin_cmd`函数，该函数检查第一个命令行参数是否是一个内置的shell命令。如果是，它就立即解释这个命令，并返回值1，否则返回0。如果`builtin_cmd`函数返回0，那么第一个命令行参数是可执行文件的路径名，此时shell会创建一个子进程，并在子进程

中执行所请求的程序，如果用户要求在后台运行该程序，那么shell返回到循环的顶部，等待下一个命令行。否则shell调用`waitfg`函数等待作业结束，当作业结束时，shell就开始下一轮循环。

通过在调用`fork`之前，阻塞SIGCHLD信号，然后在调用`addjob`之后取消阻塞这些信号，从而保证了在子进程被添加到作业列表中之后回收该子进程（子进程继承了父进程的被阻塞集合，因此必须在子进程中调用`execve`之前，解除子进程中阻塞的SIGCHLD信号）。此外在访问全局变量时也需要阻塞所有信号

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];
    int bg;
    pid_t pid;

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

    /* name is the path of an executable file */
    if (!builtin_cmd(argv)) {
        sigset_t mask_all, prev_mask, mask_chld;
        sigfillset(&mask_all);
        sigemptyset(&mask_chld);
        sigaddset(&mask_chld, SIGCHLD);
        sigprocmask(SIG_BLOCK, &mask_chld, &prev_mask);
        if ((pid = fork()) == 0){
            setpgid(0, 0);
            sigprocmask(SIG_SETMASK, &prev_mask, NULL);
            if ((execve(argv[0], argv, environ)) < 0) {
                printf("%s: command not found.\n", argv[0]);
                exit(0);
            }
        }
        sigprocmask(SIG_BLOCK, &mask_all, NULL);
        addjob(jobs, pid, bg?BG:FG, cmdline);
        sigprocmask(SIG_SETMASK, &prev_mask, NULL);
        /* parent waits for foreground job to terminate */
        sigprocmask(SIG_BLOCK, &mask_chld, NULL);
        if (!bg) {
            waitfg(pid);
        } else {
            sigprocmask(SIG_BLOCK, &mask_all, NULL);
            printf("[%d] (%d) %s", pid2jid(pid), pid ,cmdline);
        }
        sigprocmask(SIG_SETMASK, &prev_mask, NULL);
    }
    return;
}
```

`int builtin_cmd(char **argv)`：

`builtin_cmd`函数检查第一个命令行参数是不是一个内置的shell命令：

1. quit：退出shell程序
2. jobs：列出所有后台作业

3. bg/fg job：通过发送SIGCONT信号恢复指定作业在后台/前台运行
4. kill job：终止指定作业

```
int builtin_cmd(char **argv)
{
    if (!strcmp(argv[0], "quit")) /* quit command, terminates the shell */
        exit(0);
    if (!strcmp(argv[0], "jobs")) /* lists all background jobs */
    {
        sigset_t mask_all, prev_mask;
        sigfillset(&mask_all);
        sigprocmask(SIG_BLOCK, &mask_all, &prev_mask);
        listjobs(jobs);
        sigprocmask(SIG_SETMASK, &prev_mask, NULL);
        return 1;
    }
    /* bg/fg command restart job by sending a SIGCONT signal */
    if (!strcmp(argv[0], "bg") || !strcmp(argv[0], "fg"))
    {
        do_bgfg(argv);
        return 1;
    }
    if (!strcmp(argv[0], "kill")) {
        do_bgfg(argv);
        return 1;
    }
    if (!strcmp(argv[0], "&")) /* Ignore singleton & */
    {
        return 1;
    }
    return 0;     /* not a builtin command */
}
```

`void waitfg(pid_t pid)`:

因为至多只有一个前台作业，父进程调用`waitfg`函数等待前台作业结束，`sigsuspend`函数暂时用mask替换当前的阻塞集合（即解除对SIGCHLD信号的阻塞），然后挂起该进程，直到父进程捕获信号，当父进程捕获到前台作业的SIGCHLD信号时，会将全局变量`fgflag`的值置为1，即前台作业结束

```
volatile sig_atomic_t fgflag;

// sigsuspend函数等价于以下代码的原子（不可中断）版本
// sigprocmask(SIG_SETMASK, &mask, &prev);
// pause();
// sigprocmask(SIG_SETMASK, &prev, NULL);

void waitfg(pid_t pid)
{
    sigset_t mask_wait;
    sigemptyset(&mask_wait);
    fgflag = 0;
    while (!fgflag){
        sigsuspend(&mask_wait);
    }
    return;
}
```

`void sigint_handler(int sig)`：

SIGINT信号对应的处理程序，因为涉及到对全局变量的访问，因此需要调用`sigprocmask`函数阻塞所有的信号，通过`fgpid`函数获取前台作业的pid，并调用kill函数发送信号给其它进程，-pid表示发送信号给进程组|pid|（pid的绝对值）中的每个进程，最后保存和恢复错误代码即可

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

    sigfillset(&mask_all);
    sigprocmask(SIG_BLOCK, &mask_all, &prev_mask);
    pid = fgpid(jobs);
    kill(-pid, sig);
    sigprocmask(SIG_SETMASK, &prev_mask, NULL);
    errno = olderror;
    return;
}
```

`void sigtstp_handler(int sig)`：

SIGTSTP信号对应的处理程序

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

    sigfillset(&mask_all);
    sigprocmask(SIG_BLOCK, &mask_all, &prev_mask);
    pid = fgpid(jobs);
    kill(-pid, sig);
    sigprocmask(SIG_SETMASK, &prev_mask, NULL);
    errno = olderror;
    return;
}
```

`void sigchld_handler(int sig)`：

SIGCHLD信号对应的处理程序，`waitpid(pid, state, options)`函数会暂时停止目前进程的执行, 直到有信号来到或子进程结束（这里参数pid=-1表示等待任何子进程结束），捕获到SIGCHLD信号说明已经有子进程结束了，子进程的结束状态值会由参数status 返回, 而子进程的pid也会一并返回，参数options是WNOHANG（如果没有任何已经结束的子进程则马上返回, 不予以等待）和WUNTRACED（如果子进程进入暂停执行情况则马上返回）的组合，后续对返回的pid和状态值做一些判断即可：

- `job->state == FG`：结束的子进程是一个前台作业，将全局变量fgflag置为1
- `WIFEXITED(status)`：如果子进程正常结束则为非0值，则应该调用`deletejob`函数删除该作业
- `WIFSIGNALED(status)`：如果子进程是因为信号而结束则为非0值，打印相关信息，删除该作业
- `WIFSTOPPED(status)`：如果子进程处于暂停执行情况则为非0值，切换作业状态为停止

```
void sigchld_handler(int sig)
{
    int olderrno = errno;
```

```
    sigset_t mask_all, prev_mask;
    pid_t pid;
    int status;

    sigfillset(&mask_all);
    if ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
        sigprocmask(SIG_BLOCK, &mask_all, &prev_mask);
        struct job_t *job = getjobpid(jobs, pid);
        if (job->state == FG) {
            fgflag = 1;
        }
        if (WIFEXITED(status)) {
            deletejob(jobs, pid);
        }
        if (WIFSIGNALED(status)) {
            printf("Job [%d] (%d) terminated by signal %d\n", job->jid, pid,
WTERMSIG(status));
            deletejob(jobs, pid);
        }
        if (WIFSTOPPED(status)) {
            printf("Job [%d] (%d) stopped by singal %d\n", job->jid, pid,
WSTOPSIG(status));
            job->state = ST;
        }
        sigprocmask(SIG_SETMASK, &prev_mask, NULL);
    }
    errno = olderrno;
    return;
}
```

`void do_bgfg(char **argv)`：

shell内置命令`bg`、`fg`和`kill`的处理程序，每个作业都可以通过进程ID（PID）或作业ID（JID）来标识，JID在命令行中应该用前缀 '%' 表示（如“%5”表示JID 5，“5”表示PID 5）。因此首先需要判断标识作业的是JID还是PID，验证命令行第二个参数的第一个char是不是分号%即可，并通过JID/PID找到对应的job。再根据命令行第一个参数判断是bg、fg还是kill命令，bg和fg对指定进程发送SIGCONT信号即可，前台作业还需要等待其完成；kill命令发送SIGQUIT信号终止进程

```
sigset_t mask_all, prev_mask;
    pid_t pid;
    struct job_t *job;
    sigfillset(&mask_all);
    sigprocmask(SIG_BLOCK, &mask_all, &prev_mask);

    if(argv[1] == NULL) {
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    } else if (argv[1][0] == '%') {
        int jid = atoi(argv[1]+1);
        job = getjobjid(jobs, jid);
        if(job == NULL){
            printf("%s: No such job\n", argv[1]);
            return;
        }
        pid = job->pid;
    } else {
        int _pid = atoi(argv[1]);
```

```
        if (_pid == 0) {
            printf("%s: argument must be a PID or %%jobid\n", argv[0]);
            return;
        }
        job = getjobpid(jobs, _pid);
        if(job == NULL){
            printf("(%d): No such process\n", _pid);
            return;
        }
        pid = job->pid;
    }
    if(!strcmp(argv[0], "bg")) {
        kill(pid, SIGCONT);
        job->state = BG;
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
    } else if(!strcmp(argv[0], "fg")) {
        kill(-pid, SIGCONT);
        job->state = FG;
        waitfg(pid);
    } else {
        kill(-pid, SIGQUIT);
    }
    sigprocmask(SIG_SETMASK, &prev_mask, NULL);
    return;
}
```