参考 https://blog.csdn.net/weixin_44307065/article/details/106599207

根据实验说明，查看tsh.c,要求为实现下列函数

```
void eval(char *cmdline);
int builtin_cmd(char **argv);
void do_bgfg(char **argv);
void waitfg(pid_t pid);

void sigchld_handler(int sig);
void sigtstp_handler(int sig);
void sigint_handler(int sig);
```

相关知识

```
signal_handler();//截取信号
strcmp(str1,str2);//相等则返回0
parseline(char *buf,char **argv);//解析输入行并将其传输到argv中，返回bg，如果为1，在后台执行
pid_t waitpid(pid_t pid, int *status, int options);//等待进程切换状态
kill(pid,signal);//发送信号给进程或进程组，如果不指定信号，默认发送TERM
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);//上锁，解锁等
//how有3个参数
SIG_BLOCK
The set of blocked signals is the union of the current set and the set argument.

SIG_UNBLOCK
The  signals  in set are removed from the current set of blocked signals.  It is
permissible to attempt to unblock a signal which is not blocked.

SIG_SETMASK
The set of blocked signals is set to the argument set.
```

builtin_cmd

参考p525，判断是否为内置指令，是则立即执行，否则返回0，根据实验说明，内置命令如下：

- `tsh` should support the following built-in commands:
  - The `quit` command terminates the shell.
  - The `jobs` command lists all background jobs.
  - The `bg <job>` command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the background. The `<job>` argument can be either a PID or a JID.
  - The `fg <job>` command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the foreground. The `<job>` argument can be either a PID or a JID.

查询相应的工具函数，实现如下

```
int builtin_cmd(char** argv) {
    if (!strcmp(argv[0], "quit")) {
        exit(0);
    }
    if (!strcmp(argv[0], "&")) {
        return 1;
    }
    if (!strcmp(argv[0], "jobs")) {
        listjobs(jobs);//系统已定义的全局变量jobs
        return 1;
    }
    if (!strcmp(argv[0], "bg") || !strcmp(argv[0], "fg")) {
        do_bgfg(argv);
        return 1;
    }
    return 0;
}
```

sigint_handler

> The kernel sends a SIGINT to the shell whenver the user types ctrl-c at the keyboard. Catch it and send it along
> to the foreground job.

SIGINT信号默认行为为终止，是来自键盘的中断CTRL+C，在键盘上输入 CTRL+C 会导致一个SIGINT信号被发送到shell。shell捕获该信号，然后发送 SIGINT 信号到这个前台进程组中的每个进程。在默认情况下，结果是终止前台作业。

```
void sigint_handler(int sig) {
    int olderrno = errno;//根据p536 G2要求
    pid_t fpid;

    fpid = fgpid(jobs);//获取前台的pid

    if (fpid != 0) {
        kill(-fpid, SIGINT);    //给前台进程及其子进程发送
    }
```

```
    errno = olderrno;
    return;
}
```

sigtstp_handler

> The kernel sends a SIGTSTP to the shell whenever the user types ctrl-z at the keyboard. Catch it and suspend the foreground job by sending it a SIGTSTP.

SIGTSPT信号默认行为是停止直到下一个 SIGCONT，是来自终端的停止信号，在键盘上输入 CTRL+Z 会导致一个 SIGTSPT信号被发送到shell。shell捕获该信号，然后发送SIGTSPT信号到这个前台进程组中的每个进程。在默认情况下，结果是停止或挂起前台作业。

```
void sigtstp_handler(int sig)
{
    int olderrno = errno;//根据p536 G2要求
    pid_t fpid;
    fpid = fgpid(jobs);//获取前台的pid

    if (fpid != 0) {
        kill(-fpid, SIGTSTP);    //给前台进程及其子进程发送
    }
    errno = olderrno;
    return;
}
```

sigchld_handler

> - sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
> - a child job terminates (becomes a zombie), or stops because it
> - received a SIGSTOP or SIGTSTP signal. The handler reaps all
> - available zombie children, but doesn't wait for any other
> - currently running children to terminate.

当一个子进程终止或者停止时，内核会发送一个SIGCHLD信号给父进程。因此父进程必须回收子进程，以避免在系统中留下僵死进程。父进程捕获这个SIGCHLD

```
void sigchld_handler(int sig)
{
    int olderrno = errno;
    sigset_t mask, pre_mask;
```

```
    sigfillset(&mask);
    pid_t pid;
    int status;
    //仿照实验说明，给予一定的输出
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {//如果进入此循环，说明至少有一个子进程改变状态
        //WNOHANG 没有任何已结束的子进程 WUNTRACED 暂停情况
        //如果等待集合中没有任何子进程被停止或已终止，
        //那么返回值为 0,或者返回值等于那个被停止或者已终止的子进程的 PID。
        if (WIFEXITED(status)) {
            //正常退出
            sigprocmask(SIG_BLOCK, &mask, &pre_mask);
            deletejob(jobs, pid);
            sigprocmask(SIG_SETMASK, &pre_mask, NULL);//解锁
        }
        else if (WIFSIGNALED(status)) {
            //被信号杀死
            struct job_t* job = getjobpid(jobs, pid);
            sigprocmask(SIG_BLOCK, &mask, &pre_mask);
            printf("Job [%d] (%d) terminated by signal %d\n", job->jid, job->pid, WTERMSIG(status));
            deletejob(jobs, pid);
            sigprocmask(SIG_SETMASK, &pre_mask, NULL);//解锁
        }
        else if (WIFSTOPPED(status)) {
            //停止
            struct job_t* job = getjobpid(jobs, pid);
            sigprocmask(SIG_BLOCK, &mask, &pre_mask);
            printf("Job [%d] (%d) stopped by signal %d\n", job->jid, job->pid, WSTOPSIG(status));
            job->state = ST;//改变状态
            sigprocmask(SIG_SETMASK, &pre_mask, NULL);//解锁
        }
    }
    errno = olderrno;
    return;
}
```

waitfg

> Block until process pid is no longer the foreground process

阻塞一个前台进程直至其不为前台进程

```
void waitfg(pid_t pid)
{

    sigset_t mask_child, prev;
    sigemptyset(&mask_child);
    sigaddset(&mask_child, SIGCHLD);

    sigprocmask(SIG_BLOCK, &mask_child, &prev);
    while (getjobpid(jobs, pid) && getjobpid(jobs, pid)->state == FG) {
        sigsuspend(&prev);
        //每次调用sigsuspend()之前，要阻塞SIGCHLD，sigsuspend()取消阻塞SIGCHLD，然后休眠
        //直到捕获到SIGCHLD信号,p546
    }
    sigprocmask(SIG_SETMASK, &prev, NULL);

    return;
}
```

do_bgfg

> Each job can be identified by either a process ID (PID) or a job ID (JID), which is a positive integer
> assigned by tsh. JIDs should be denoted on the command line by the prefix ’%’. For example, “%5”
> denotes JID 5, and “5” denotes PID 5. (We have provided you with all of the routines you need for
> manipulating the job list.)

可以支持PID和JID

```
void do_bgfg(char** argv)
{

    struct job_t* job;
    char* para;
    sigset_t mask, pre_mask;
    int jid;

```

```
    sigfillset(&mask);
    para = argv[1];
    if (para == NULL) {    //参数不足
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        fflush(stdout);
        return;
    }
    if (para[0] == '%') {//JID
        jid = atoi(&(para[1]));//错误处理2，如果传入的参数不是规定的格式，报错返回
        if (jid == 0) {
            printf("%s:argument must be a PID or %%jobid\n", argv[0]);
            fflush(stdout);
            return;
        }
    }
    else {//PID
        jid = atoi(para);
        if (jid == 0) {
            printf("%s:argument must be a PID or %%jobid\n", argv[0]);
            fflush(stdout);
            return;
        }
        jid = pid2jid(jid);
    }
    job = getjobjid(jobs, jid);
    if (job == NULL) {
        if (para[0] == '%')  printf("(%s): No such job\n", para);
        else printf("(%s): No such process\n", para);
        fflush(stdout);
        return;
    }
    //区分bg还是fg
    if (!strcmp(argv[0], "bg"))///bg
    {
            job->state = BG;//修改状态为BG后台
            kill(-(job->pid), SIGCONT);//发送SIGCONTT信号给对应的子进程
            printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
        }
    }
    else//fg
```

```
    {
            job->state = FG;
            kill(-(job->pid), SIGCONT);
            waitfg(job->pid);
    }
    return;
}
```

eval

参考p525实现，书中这个进程不回收子进程，作相应的修改

如果用户请求了一个内置命令(quit、jobs、bg或fg)，立即执行。否则，fork一个子进程，在子进程的上下文中运行作业。如果工作进行在前台，等待它结束然后返回。注意:每个子进程必须有一个唯一的进程组ID，从而使输入ctrl-c (ctrl-z)时，后台子进程不接受到SIGINT (SIGTSTP)。

```
void eval(char* cmdline)
{
    char* argv[MAXARGS];  //execve()函数的参数
    char buf[MAXLINE];    //保存修改后的命令行
    sigset_t mask;
    int bg;               //判断作业在前台还是后台
    pid_t pid;            //进程id
    strcpy(buf, cmdline);  //拷贝命令行

    bg = parseline(buf, argv);  //解析命令行，返回给argv数组
    if (argv[0] == NULL) return;//忽略空行

    if (!builtin_cmd(argv))//如果是内置命令那么结束，否则创建新的子进程
    {
        sigemptyset(&mask);
        sigaddset(&mask, SIGCHLD);
        //在它派生子进程之前阻塞SIGCHLD信号，因为如果子进程运行快会直接终止，
        //发送SIGCHLD让父进程结束它，那么父进程会加一个空进程到列表
        sigprocmask(SIG_BLOCK, &mask, NULL);
        pid = fork();       //fork创建子进程
        if (pid == 0)         //在子进程的上下文中运行作业
        {
            sigprocmask(SIG_UNBLOCK, &mask, NULL);//解除阻塞
            setpgid(0, 0);//保证是一个独立的进程，防止把shell一起kill掉
            if (execve(argv[0], argv, environ) < 0)
```

```
            {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
        /*父进程等待前台作业终止*/
        //前台作业等待
        sigprocmask(SIG_BLOCK, &mask_all, NULL);//在修改jobs时，阻塞所有信号，结束后再开启
        addjob(jobs, pid, bg+1, cmdline);
        sigprocmask(SIG_SETMASK, &prev_child, NULL);

        if (!bg)//判断是在前台还是在后台运行
        {
            //前台运行
            waitfg(pid);
        }
        else
        {
            //后台运行
            printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
        }

    }
    return;
}
```

测试通过

```
gcc -Wall -O2    tsh.c   -o tsh
[root@localhost shlab-handout]# make test15
./sdriver.pl -t trace15.txt -s ./tsh -a "-p"
#
# trace15.txt - Putting it all together
#
tsh> ./bogus
./bogus: Command not found.
tsh> ./myspin 10
Job [1] (13942) terminated by signal 2
tsh> ./myspin 3 &
[1] (13944) ./myspin 3 &
tsh> ./myspin 4 &
[2] (13946) ./myspin 4 &
tsh> jobs
[1] (13944) Running ./myspin 3 &
[2] (13946) Running ./myspin 4 &
tsh> fg %1
Job [1] (13944) stopped by signal 20
tsh> jobs
[1] (13944) Stopped ./myspin 3 &
[2] (13946) Running ./myspin 4 &
tsh> bg %3
(%3): No such job
tsh> bg %1
[1] (13944) ./myspin 3 &
tsh> jobs
[1] (13944) Running ./myspin 3 &
[2] (13946) Running ./myspin 4 &
tsh> fg %1
tsh> quit
[root@localhost shlab-handout]# make rtest15
./sdriver.pl -t trace15.txt -s ./tshref -a "-p"
#
# trace15.txt - Putting it all together
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 10
Job [1] (13967) terminated by signal 2
tsh> ./myspin 3 &
[1] (13969) ./myspin 3 &
tsh> ./myspin 4 &
[2] (13971) ./myspin 4 &
tsh> jobs
[1] (13969) Running ./myspin 3 &
[2] (13971) Running ./myspin 4 &
tsh> fg %1
Job [1] (13969) stopped by signal 20
tsh> jobs
[1] (13969) Stopped ./myspin 3 &
[2] (13971) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (13969) ./myspin 3 &
tsh> jobs
[1] (13969) Running ./myspin 3 &
[2] (13971) Running ./myspin 4 &
tsh> fg %1
tsh> quit
[root@localhost shlab-handout]#
```