SHELL LAB

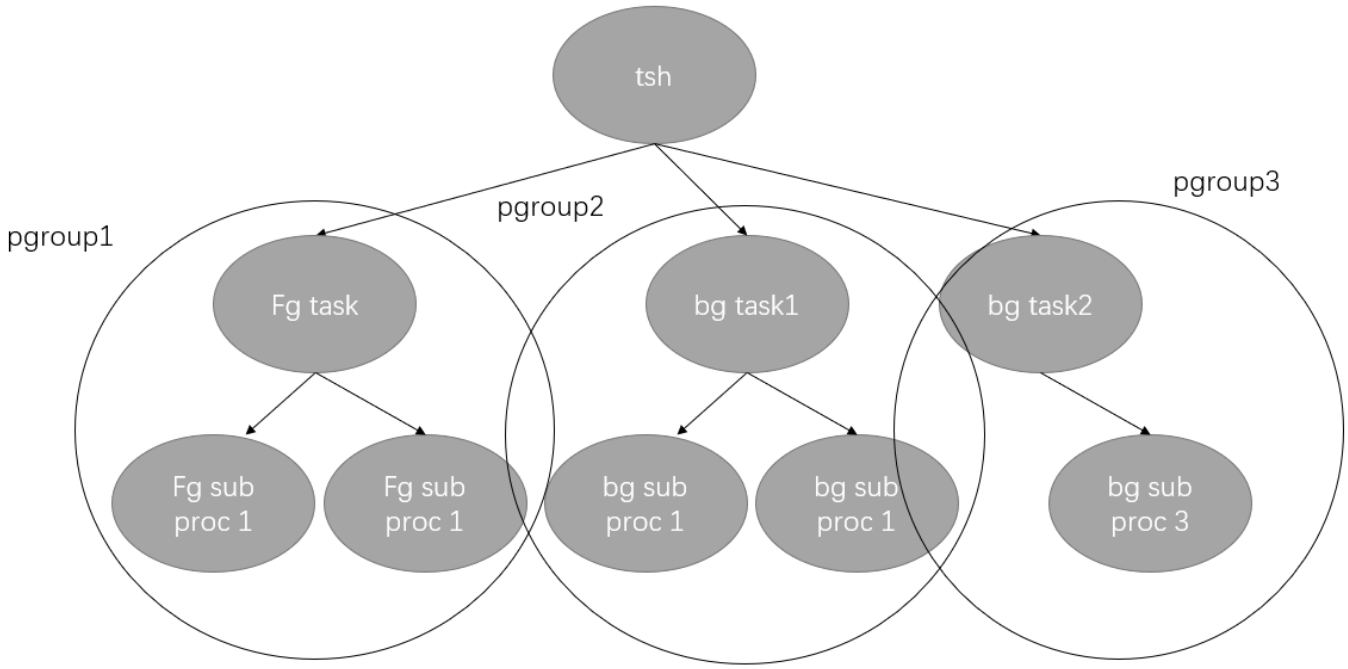
本实验的主要目的是熟悉 linux 系统编程，以及相关的进程调度，系统调用，进程通信概念。
理解了实验目的后，按照文档上的 hints 操作即可。特别需要注意的有几点。

子任务的状态

任何 tsh 派生的任务都有三种状态，FG,BG 和 ST。这些状态之间在某些操作下可以相互转换
FG -> ST : ctrl+z
ST -> FG : fg command
BG -> BG : bg command
ST -> BG : bg command
BG -> FG : fg command
这里并没有包括死亡(zombie)态，一个执行完毕正常退出的进程，或者被 ctrl+c 终止的进程都会变成 zombie。因为每个死掉的进程马上就会被 tsh 发现然后销毁，这是使用 sigchld_handler 实现的，后面会详细介绍 sigchld_handler 该怎么写。
本实验的要实现的主要功能就是安全的进程创建和销毁，以及基于上述操作实现的状态转换。

进程的衍生关系

理解进程间的派生和同步关系是理解程序流程的难点。



tsh 进程，即我们编写的 shell 软件是整个项目的主程。其他从 tsh 控制台输入的非 builtin_cmd 命令所产生的进程都是 tsh 的子进程，子进程还可以继续派生子进程。每个从 tsh fork 出来的进程自成一个进程组。

无论是前台还是后台任务，一旦它被 tsh fork 出来，它就是和 tsh 以及其他任务并发执行的新进程(尽管前台任务需要 tsh 等待它执行完毕，但这并不意味着它们是串行的)。为每个任务成立一个独立的进程组的原因是，本项目需要 SIGTSTP,SIGCONT,SIGINT 三种信号控制进程的执行与阻塞，而执行与阻塞从 tsh 的角度来看，必然是以任务为单位进行的，所以我们需要以进程组为单位进行三种信号的发送。

```
setpgid(0, 0); /* put child process into a individual pgroup */
```

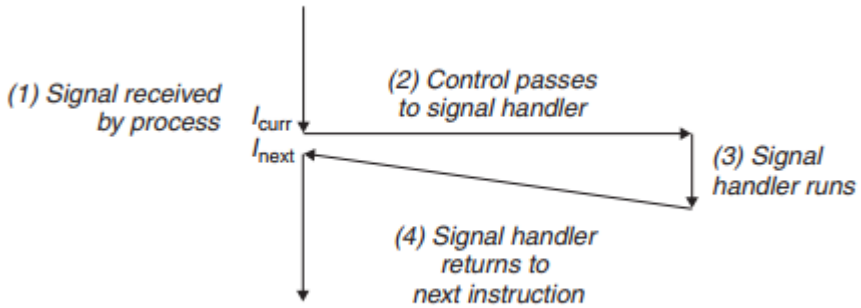
tsh 在 fork 子任务时，使用 setpgid 将子任务对应的子进程划分到另外的进程组内。

3 个 handler 的使用场景

理解 handler，需要理解它的执行原理，为此我们先回答两个问题。

1. 谁执行 handler?

handler 是信号处理程序，进程在从内核态返回时(系统调用结束时)，如果它有待处理信号，则系统会执行默认的信号处理例程。如果特别为该信号定义了对应的 handler，则执行该 handler 函数内的例程。



本实验中，键盘输入的 CTRL+Z 和 CTRL+C 发给 stdin 进程的 SIGTSTP 和 SIGINT，子进程结束时发给父亲进程的 SIGCHLD，它们的去向都是 tsh 进程。这告诉了我们一个重要事实，handler 的编写是面向 tsh 的，handler 能访问的是 tsh 所维护的数据结构，包括 jobs 以及相关请求(getjobpid 等)。

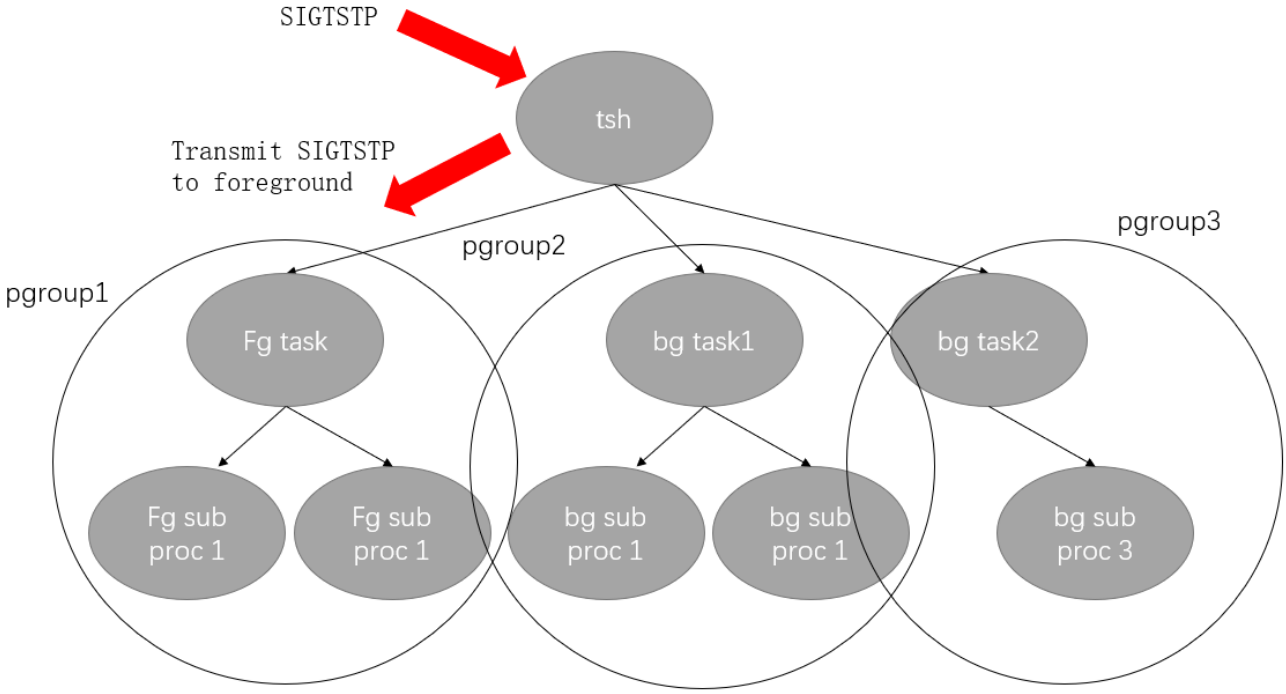
2. 什么时候执行 handler?

信号的到来不可预期，因此任意系统调用的结束都可能触发任意的 handler 的执行。handler 在内核态执行。执行完毕后，返回到下一条指令处。因此，tsh 执行的任意一行代码都应该做好被 handler 打断的准备。如果某些地方绝对不允许打断，则需要使用 sigprocmask 调用把某些信号阻塞掉。

这样，handler 的编写逻辑就变得清晰。

sigint_handler、sigtstp_handler

一旦键盘输入了 CTRL+Z 或 CTRL+C，则 tsh 就会收到这两种信号，然后触发两个 handler。虽然两者的实际效果不同，但是 tsh 处理它们的逻辑是相同的。两类信号的原本目的是打断 fg 任务的执行，但是 tsh 把 fg 进程划分到了其他的进程组内，这样 fg 不会接收到这个信号。为此，tsh 所需要完成的是简单的转发工作，借助 kill 函数。




```
/*
 * 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.
 */
void sigint_handler(int sig)
{
    /* keyboard interupt send a signal to shell process, shell should
    transmit this singal to fg process */
    pid_t pid = fgpid(jobs);

    if (pid)
        if (kill(-pid, SIGINT) < 0)
            unix_error("kill error");
}

/*
 * 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.
 */
void sigtstp_handler(int sig)
{
    /* keyboard interupt send a signal to shell process, shell should
    transmit this singal to fg process */
    pid_t pid = fgpid(jobs);

    if (pid)
        if (kill(-pid, SIGSTOP) < 0)
```

```
            unix_error("kill error");
}
```

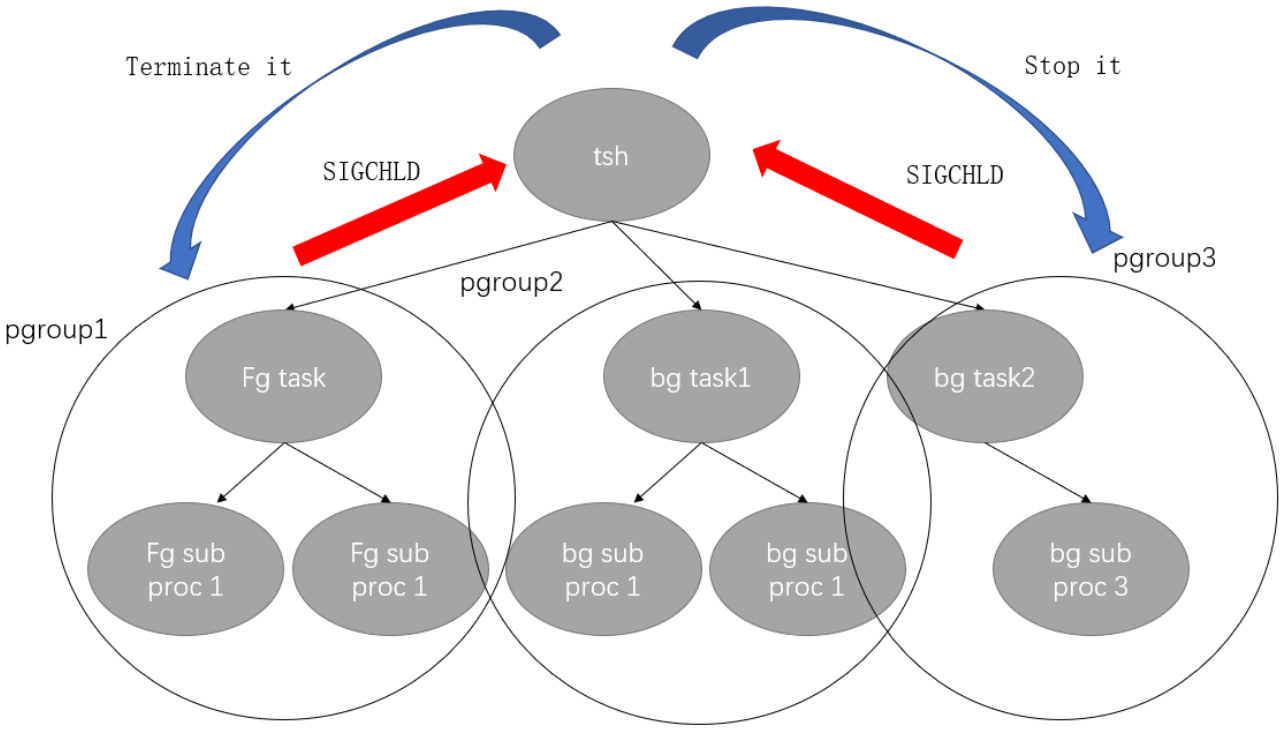
sigchld_handler

tsh 的子进程被 STOP 或者 TERMINATE 都会使这个子进程发送一个 SIGCHLD 信号给 tsh，然后 tsh 触发 sigchld_handler 的执行。因此，这个 handler 的实际作用应该是

1）如果子进程被 STOP 了，就修改 tsh 内部的数据结构，把这个子进程标记为 STOP 状态。

2）如果子进程被 TERMINATE 了，就回收它，并修改 tsh 内部的数据结构，把这个子进程从数据结构中移除。

因为 SIGCHLD 信号队列最多只能留下一个待处理信号，所以每次 handler 都要考虑所有的子进程。这里的实现使用了一个 while 循环，条件使用 waitpid 和 options= WUNTRACED|WNOHANG；这样的设置允许 tsh 循环收集每个被 STOP 或 TERMINATE 的子进程的 pid，并加以处理。一旦没有了这样的子进程，则立即跳出循环。



waitpid 获取到的 status 将帮助我们识别进程是 STOP 还是 TERMINATE。

```
void sigchld_handler(int sig)
{
    /* this function will be executed under kernel state of shell process, when a fg or bg process
    stops or is terminated. so it can use global variables like jobs */
    pid_t pid;
    int status;

    /* options = WUNTRACED | WNOHANG allows waitpid return directly if no process is stopped or terminated.
*/
    while ((pid = waitpid(-1, &status, WUNTRACED | WNOHANG)) > 0) {

        /* If a child is terminated by a signal, print the offending signal */
        if (WIFSIGNALED(status))
            printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, SIGINT);
```

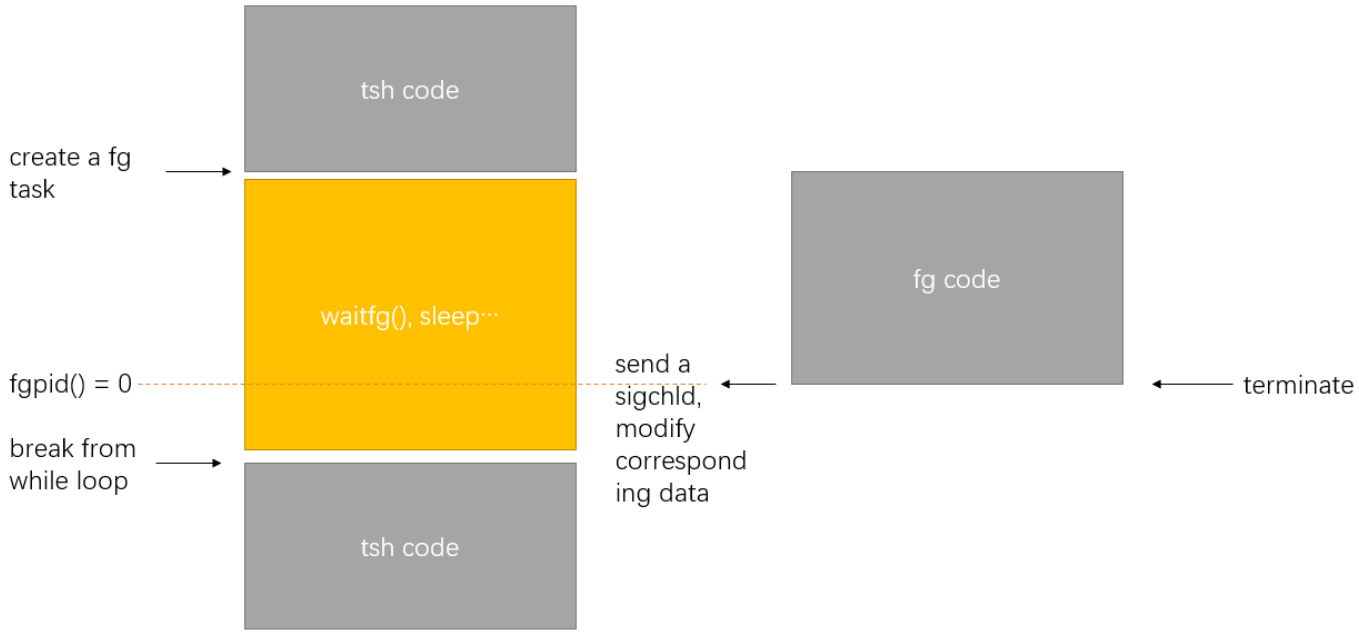
```
        /* If a child is stopped by a signal, modify its state and print a message */
        if (WIFSTOPPED(status)) {
            getjobpid(jobs, pid)->state = ST;
            printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, SIGTSTP);
        }
        else { /* A child is terminated, delete the corresponding job */
            deletejob(jobs, pid);
        }
    }

    if (pid < 0 && errno != ECHILD)
        unix_error("waitpid error");
}
```

异步与同步

实验涉及到多进程的并发执行，程序的执行逻辑是相当 confusing 的。bg 的子进程一经创建，它和 tsh 就是完全并行的关系，直到它被 STOP 或者 TERMINATE，才会通过传送一个 SIGCHLD 信号来告知 tsh 应该做些扫尾工作。
上面提到过，tsh 并不是什么时候都能打断的，创建一个新任务的整个过程(包括 fork 和 addjob)都是不允许 SIGCHLD 打断的，因为这时的 jid 分配，pgid 分配等任务都会受到 SIGCHLD 打断的影响，造成很坏的后果。为此，使用一个 sigprocmask 暂时阻塞掉 SIGCHLD。
因为 fork 出的子进程和 tsh 父进程拥有相同的 mask，所以要在子进程中 unblock 一次，父进程中 unblock 一次。
fg 任务和 tsh 是并发关系，尽管我们可以使用 waitpid 实现 tsh 和 fg 的同步，但是这将让代码变得相当难以实现(考虑到 SIGTSTP 和 SIGINT 都会让 fg 进程不再是 fg，这样 waitpid 将不可避免的报错)。一种相当巧妙的方法是用一个带有 sleep 的 busy while loop 实现 tsh 进程与 fg 的伪同步。之所以叫伪同步，是因为它们本质上仍然是异步执行的；

```
/*
 * waitfg - Block until process pid is no longer the foreground process
 */
void waitfg(pid_t pid)
{
    /* use a busy loop around the sleep function. */
    while (fgpid(jobs) == pid)
        sleep(1);
}
```

每次 tsh 创建好了一个 fg 进程，就调用 waitfg；一旦 fg 正常返回，或者被 SIGTSTP 和 SIGINT 终止或者停止，都会触发 SIGCHLD 改变 jobs 数据结构，从而 fgpid(jobs) == pid 的条件不再满足，循环解除。

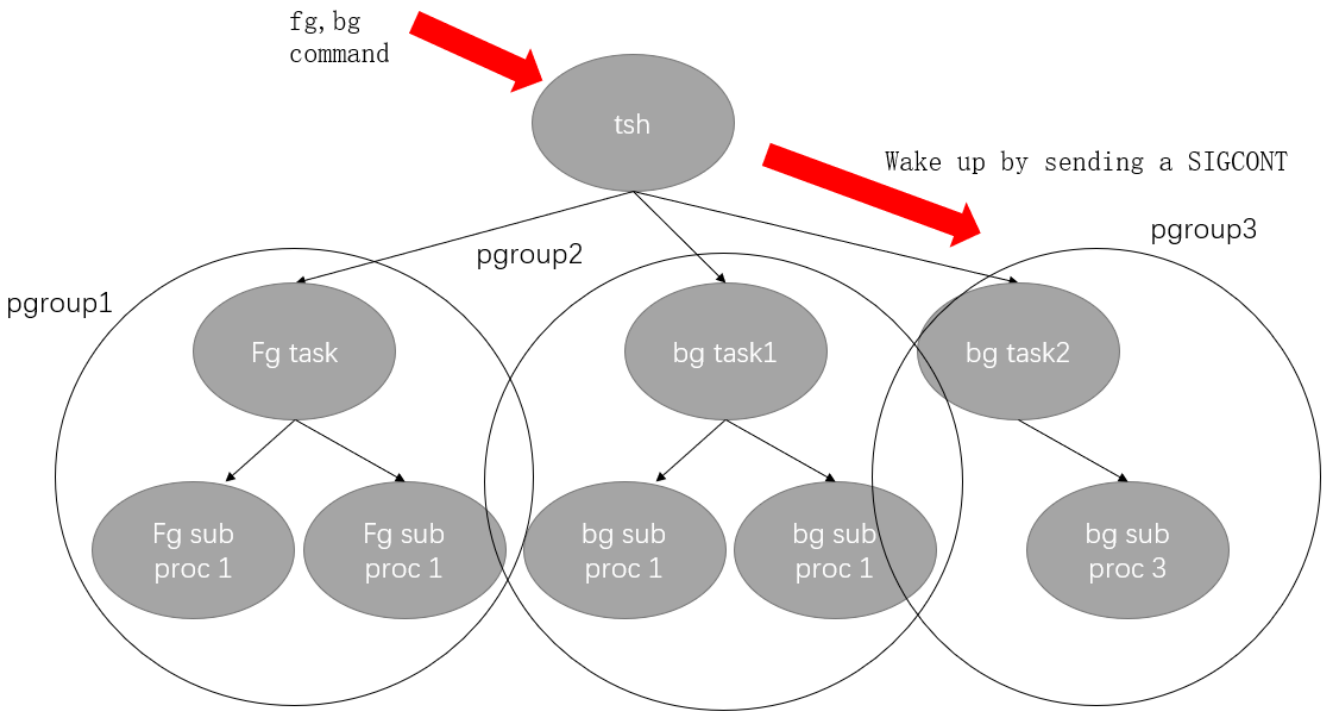



```c
/*
 * eval - Evaluate the command line that the user has just typed in
 *
 * If the user has requested a built-in command (quit, jobs, bg or fg)
 * then execute it immediately. Otherwise, fork a child process and
 * run the job in the context of the child. If the job is running in
 * the foreground, wait for it to terminate and then return.  Note:
 * each child process must have a unique process group ID so that our
 * background children don't receive SIGINT (SIGTSTP) from the kernel
 * when we type ctrl-c (ctrl-z) at the keyboard.
*/
void eval(char *cmdline)
{
    char *argv[MAXARGS];  /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in background or foreground */
    pid_t pid;           /* Process id */
    sigset_t mask; /* mask variable to block signals */

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

    if (!builtin_cmd(argv)) {
        /* parent blocks SIGCHLD signals */
        if (sigemptyset(&mask) < 0)
            unix_error("sigemptyset error");
        if (sigaddset(&mask, SIGCHLD) < 0)
            unix_error("sigaddset error");
```

```
        if (sigprocmask(SIG_BLOCK, &mask, NULL) < 0)
            unix_error("sigprocmask error");

        if ((pid = fork()) < 0)
            unix_error("fork error");

        if (pid == 0) { /* child process runs user job */
            setpgid(0, 0); /* put child process into a individual pgroup */

            if (sigprocmask(SIG_UNBLOCK, &mask, NULL) < 0) /* child unblocks signals */
                unix_error("sigprocmask error");

           if (execve(argv[0],argv,environ) < 0) { /* execute */
               printf("%s: Command not found.\n", argv[0]);
                exit(0);
           }
        }

        /* add new job infomation into global variable "jobs" */
        addjob(jobs, pid, bg ? BG : FG, cmdline);

        /* parent unblocks signals */
        if (sigprocmask(SIG_UNBLOCK, &mask, NULL) < 0)
            unix_error("sigprocmask error");
        /* if fg, parent waits for foreground job to terminate */
        if (!bg) {
            waitfg(pid);
        }
        /* if bg, break without waiting, sigchld_handler will do any dirty things */
        else {
            printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
        }
    }
}
```

进程重启

被 SIGTSTP 停止的 bg 进程可以被 SIGCONT 唤醒，因此唤醒进程的 fg 和 bg 指令很简单，只需要改变 jobs 数据结构状态，然后让 tsh 发送一个 SIGCONT 到子进程即可。如果是 fg，还需要执行 waitfg 等待该进程结束。



这个发送和 sigint_handler、sigtstp_handler 的转发类似，都是把信号发送给整个进程组。所以，使用的参数是-pid 而非 pid。

除了这些简单的逻辑，还需要自行进行 parse。要注意的异常情形有：argv 参数不足(用户只输入了 fg、bg 而没输入 jid 或 pid)，jid 或 pid 无法转换为整数(输入不标准)，jid 或 pid 不存在于当前任务列表中。

```
/*
 * do_bgfg - Execute the builtin bg and fg commands
 */
void do_bgfg(char **argv)
{
    /*
     *     ST -> FG  : fg command
     *     ST -> BG  : bg command
     *     BG -> FG  : fg command
     */
    pid_t pid, jid;
    struct job_t * job;

    if (!argv[1]) { /* Missing pid or jid */
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }

    /* parse argv[1], it could be a pid(p) or a jid(%j)  */
    if (*argv[1] == '%') {
        jid = atoi(argv[1] + 1);
        /* cannot convert */
        if (strcmp(argv[1] + 1, "0") && !jid) {
            printf("argument must be a PID or %%jobid\n");
            return;
```

```
        }
        job = getjobjid(jobs, jid);
        /* cannot find target job */
        if (!job) {
            printf("No such job\n");
            return;
        }
        pid = job->pid;
    }
    else {
        /* cannot convert */
        pid = atoi(argv[1]);
        if (strcmp(argv[1] + 1, "0") && !pid) {
            printf("argument must be a PID or %%jobid\n");
            return;
        }
        job = getjobpid(jobs, pid);
        /* cannot find target process */
        if (!job) {
            printf("No such process\n");
            return;
        }
    }


    if (!strcmp(argv[0], "fg")) {
        job->state = FG;
        if (kill(-pid, SIGCONT) < 0)
            unix_error("kill error");
        waitfg(pid);
    }
    else {
        job->state = BG;
        if (kill(-pid, SIGCONT) < 0)
            unix_error("kill error");
        printf("[%d] (%d) %s", job->jid, pid, job->cmdline);
    }
}
```