

# A Tiny Unix Shell (tsh) – Part IV

## Introduction

This is the fourth piece of the project on process control and signaling. In this assignment you will extend your shell to process signals received from the keyboard. This assignment must be completed *individually*.

## Keyboard Signals

Recall that a *signal* is a small message that notifies a process that an event of some type has occurred in the system. Typing `ctrl-c` causes a `SIGINT` signal to be delivered to each process in the foreground job. The default action for `SIGINT` is to terminate the process.

Similarly, typing `ctrl-z` causes a `SIGTSTP` signal to be delivered to each process in the foreground job. The default action for `SIGTSTP` is to place a process in the stopped state, where it remains until it is awakened by the receipt of a `SIGCONT` signal.

## What to do

1. Implement signal handlers for the `SIGINT` and `SIGTSTP` signals. Typing `ctrl-c` (`ctrl-z`) should cause a `SIGINT` (`SIGTSTP`) signal to be sent to the current foreground job, as well as any descendents of that job (e.g., any child processes that it forked). If there is no foreground job, then the signal should have no effect.

Your shell should catch these signals and send them to the *entire* foreground process group, using “-pid” instead of “pid” in the argument to the `kill` function, i.e.

```
if (kill(-pid, SIGINT) < 0)
    printf("kill (sigint) error");
```

*Here is one problem:* When you run your shell from the standard Unix shell, your shell is running in the foreground process group. If your shell then creates a child process, by default that child will also be a member of the foreground process group. Since typing `ctrl-c` sends a `SIGINT` to *every* process in the foreground group, typing `ctrl-c` will send a `SIGINT` to your shell, as well as to every process that your shell created, which obviously is not correct.

*Here is the workaround:* After the `fork`, but before the `execvp`, the child process should call `setpgid(0, 0)`, which puts the child in a new process group whose group ID is identical to the child's PID. This ensures that there will be only one process, your shell, in the foreground process group. When you type `ctrl-c`, the shell should catch the resulting `SIGINT` and then forward it to the appropriate foreground job (or more precisely, the process group that contains the foreground job).

**Important Note.** Do not forget to add `SIGINT` and `SIGTSTP` to the mask set of blocked signals before the child gets forked. This eliminates race conditions where the child receives one of these signals *before* the parent calls `addjob`.

2. Thoroughly test your code for `ctrl-c` before moving on to step 3. It should properly terminate processes.

3. Once you have completed step 1, you will notice that ctrl-z stops a running process, however the parent is stuck in a loop waiting for the child to finish execution. The child is simply in a blocked state and can resume execution upon receiving a SIGCONT signal. Because the child did not finish, the parent will never receive the SIGCHLD signal and therefore it will be stuck in the waitpid call forever.

To fix this problem, you will need to replace the waitpid call by a call to the following waitfg function, which blocks the parent for as long as a FG child exists:

```
/*
 * waitfg - Block until process pid is no longer the FG process
 */
void waitfg(pid_t pid)
{
    struct job_t *j = getjobpid(jobs, pid);

    /* The FG job has completed and been reaped by the handler */
    if (!j)
        return;

    /* Busy waiting while there is a FG process */
    while (j->pid == pid && j->state == FG)
        sleep(1);

    printf("waitfg: Process (%d) no longer the FG process", pid);

    return;
}
```

## Evaluation

Your score will be computed out of a maximum of 60 points based on the following distribution:

**50** Correctness.

**10** Style points. We expect you to have good comments and to check for errors.

## Hand In Instructions

Hand in a printed copy of the following:

1. Source code `tsh.c`, `eval.c`, `siglib.c`, `eval.h`, `siglib.h`.
2. A sample output of your code.
3. A short README file explaining any bugs and deviations from the assignment specifications.

Make sure you have included your name and your Unix login name in the README file. Leave your source code in your Unix account. Good luck!