CSAPPLab7 ShellLab

本实验的目的是通过编写一个简单地支持作业控制（*job control*）的 *shell* 程序，以熟悉进程控制和信号的概念。

文件 *tsh.c* 中实现了一个简单地 *Unix shell* 功能框架，我们的任务是完成下面的几个函数：

- ***eval*** : Main routine that parses and interprets the command line. [70 lines]
- ***builtin_cmd*** : Recognizes and interprets the built-in commands: quit, fg, bg, and jobs. [25 lines]
- ***do_bgfg*** : Implements the bg and fg built-in commands. [50 lines]
- ***waitfg*** : Waits for a foreground job to complete. [20 lines]
- ***sigchld_handler*** : Catches SIGCHILD signals. 80 lines]
- ***sigint_handler*** : Catches SIGINT (ctrl-c) signals. [15 lines]
- ***sigtstp_handler*** : Catches SIGTSTP (ctrl-z) signals. [15 lines]

---

eval function

```
/*
 * 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];
    char buf[MAXLINE];
    int bg;
    sigset_t mask_all, mask_one, prev_all;
    pid_t pid;

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;
```

```
    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);

    if (!builtin_cmd(argv)){
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_all);
        if ((pid = Fork()) == 0){
            Sigprocmask(SIG_SETMASK, &prev_all, NULL);
            setpgid(0, 0);
            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, buf);
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);

        if (!bg){
            //printf("Not bg\n");
            waitfg(pid);
        }else{
            //printf("Is bg\n");
            printf("[%d] (%d) %s", pid2jid(pid), pid, buf);
        }
    }

    return;
}
```

该函数代码与书籍《深入理解计算机系统》第525页基本一致。因程序中增加了 *job* 控制，为消除该函数中调用 *addjob* 和处理程序中调用 *deletejob* 之间存在的竞争，需在调用 *fork* 之前阻塞 *SIGCHLD* 信号，然后在调用 *addjob* 之后取消阻塞这些信号，保证在子进程被添加到作业列表之后回收该子进程。应注意子进程继承了它们父进程的被阻塞集合，所以必须在调用 *execve* 之前解除子进程中阻塞的 *SIGCHLD* 信号（见书籍第542，543页）。

如果该 *job* 在 *foreground* 运行，*shell* 必须等待当前 *job* 运行终止之后才能继续取下一条命令；如果该 *job* 在 *background* 运行，不必等待当前 *job* 运行终止。

---

builtin_cmd function

```
/*
 * builtin_cmd - If the user has typed a built-in command then execute
 *    it immediately.
 */
int builtin_cmd(char **argv)
{
    if (!strcmp(argv[0], "quit"))   /* quit command */
        exit(0);
    if (!strcmp(argv[0], "fg") || !strcmp(argv[0], "bg")){    /* fg or bg
command */
        do_bgfg(argv);
        return 1;
    }
```

```
    if (!strcmp(argv[0], "jobs")){    /* jobs command */
        listjobs(jobs);
        return 1;
    }
    return 0;     /* not a builtin command */
}
```

实验要求包含的 *built-in commands* 如下：

- The ***quit*** command terminates the shell.
- The ***jobs*** command lists all background jobs.
- The ***bg*** command restarts by sending it a SIGCONT signal, and then runs it in the background. The argument can be either a PID or a JID.
- The ***fg*** command restarts by sending it a SIGCONT signal, and then runs it in the foreground. The argument can be either a PID or a JID.

---

do_bgfg function

```
/*
 * do_bgfg - Execute the builtin bg and fg commands
 */
void do_bgfg(char **argv)
{
    pid_t pid;
    int jid;
    struct job_t *job = NULL;
    char *id = argv[1];

    if (id == NULL){
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }
    if (id[0] == '%'){
        jid = atoi(&id[1]);
        job = getjobjid(jobs, jid);
        if (job == NULL){
            printf("%%%d: No such job\n", jid);
            return;
        }
    } else{
        pid = atoi(id);
        if (pid == 0){  /* function atoi return 0 if id is not a digit */
            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;
        }
    }
/*
    if (id[0] == '%'){
        jid = atoi(&id[1]);
        job = getjobjid(jobs, jid);
        if (job == NULL){
```

```
            printf("(%d): No such process\n", jid);
            return;
        }
    } else if (isdigit(id[0])){
        pid = atoi(id);
        job = getjobpid(jobs, pid);
        if (job == NULL){
            printf("(%d): No such process\n", pid);
            return;
        }
    } else {
        pid = atoi(id);
        printf("%d\n", pid);
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }
*/
    //Kill(-(job->pid), SIGCONT);

    if (!strcmp(argv[0], "bg")){
        //printf("Do bg\n");
        job->state = BG;
        Kill(-(job->pid), SIGCONT);
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
    }else {
        //printf("Do fg\n");
        job->state = FG;
        Kill(-(job->pid), SIGCONT);
        waitfg(job->pid);
    }

    return;
}
```

命令 *bg* 表示在 *background* 继续进程，命令 *fg* 表示在 *foreground* 继续进程。两者均需通过 *kill* 函数发送 *SIGCONT* 信号，且改变相应的 *job* 状态。若在 *foreground* 继续进程，则应调用 *waitfg* 函数等待当前 *job* 运行终止。

函数 *int kill(pid_t pid, int sig)* 的使用详见书籍《深入理解计算机系统》第530页。

---

waitfg function

```
/*
 * waitfg - Block until process pid is no longer the foreground process
 */
void waitfg(pid_t pid)
{
    while (pid == fgpid(jobs)){
        sleep(0);
    }
    return;
}
```

---

sigchld_handler function

```c
/*
 * 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.
 */
void sigchld_handler(int sig)
{
    int olderrno = errno;
    int status;
    sigset_t mask_all, prev_all;
    pid_t pid;

    Sigfillset(&mask_all);
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0){ /* Reap
a zombie child */
        int jid = pid2jid(pid);
        if (WIFEXITED(status)){
            Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
            deletejob(jobs, pid);
            Sigprocmask(SIG_SETMASK, &prev_all, NULL);
            //printf("Job [%d] (%d) terminated normally with exit
status=%d\n",jid, pid, WEXITSTATUS(status));
            return;
        }
        if (WIFSIGNALED(status)){
            Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
            deletejob(jobs, pid);
            Sigprocmask(SIG_SETMASK, &prev_all, NULL);
            printf("Job [%d] (%d) terminated by signal %d\n", jid, pid,
WTERMSIG(status));
            return;
        }
        if (WIFSTOPPED(status)){
            Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
            struct job_t *job = getjobpid(jobs, pid);
            if (job != NULL)
            job->state = ST;
            Sigprocmask(SIG_SETMASK, &prev_all, NULL);
            printf("Job [%d] (%d) stopped by signal %d\n", jid, pid,
WSTOPSIG(status));
            return;
        }
    }
/*
    if (errno != ECHILD)
        unix_error("waitpid error");
*/
    errno = olderrno;

    return;
}
```

函数 *pid_t waitpid(pid_t pid, int *statusp, int options)* 的使用详见书籍《深入理解计算机系统》第517页。其已回收子进程的退出状态为以下 *3* 种情况：

- *WIFEXITED(status)* : 如果子进程通过调用 *exit* 或者一个返回（*return*）正常终止，就返回真；

- *WIFSIGNALED(status)* : 如果子进程是因为一个未捕获的信号终止的，就返回真；
- *WIFSTOPPED(status)* : 如果引起返回的子进程当前是停止的，就返回真。

程序中的 *if* 分支判断正对应以上三种情况。当子进程终止时，需调用 *deletejob* 删除相应 *job* ； 子进程停止时，需改变相应 *job* 的状态为 *ST* 。

注意：程序最后不应加入 *if (errno != ECHILD)* 判断，因函数 *waitpid* 可能被一个信号中断，返回 *-1* ，设置 *errno* 为 *EINTR* ，或者函数 *waitpid* 因等待集合的子进程都没有停止或终止，返回值为 *0* , 设置 *errno* 为另外一个值，两种均属合理情况。所以，增加判断函数出错时的 *errno* 应为 *ECHILD* 为不合理操作。

---

sigint_handler function

```
/*
 * 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)
{
    pid_t pid = fgpid(jobs);
    if (pid != 0)
        Kill(-pid, sig);
    return;
}
```

在键盘上输入 *Ctrl + c* 会导致内核发送一个 *SIGINT* 信号到前台进程组中的每个进程。默认情况下，结果是终止前台作业。本 *shell* 通过函数 *Signal(SIGINT, sigint_handler)* ，改变默认行为，调用信号处理程序 *sigint_handler* ，在该程序中通过 *kill* 函数发送信号 *SIGINT* 给前台进程组中的每个进程，终止前台作业，促使内核发送 *SIGCHLD* 信号，调用信号处理程序 *sigchld_handler* 。

函数 *sighandler_t signal(int signum, sighandler_t handler)* 的使用详见书籍《深入理解计算机系统》第531页。

---

sigtstp_handler function

```
/*
 * 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)
{
/*
    pid_t pid = fgpid(jobs);
    if (pid != 0){
        struct job_t *job = getjobpid(jobs, pid);
        job->state = ST;
        printf("Job [%d] (%d) stopped by signal %d\n", job->jid, pid, sig);
    }
*/
    int olderrno = errno;
    pid_t pid = fgpid(jobs);
    if (pid != 0){
        Kill(-pid, sig);
    }
```

```
    errno = olderrno;

    return;
}
```

在键盘上输入 *Ctrl + z* 会导致内核发送一个 *SIGTSTP* 信号到前台进程组中的每个进程。默认情况下，结果是停止（挂起）前台作业。本 *shell* 通过函数 *Signal(SIGTSTP, sigtstp_handler)* ，改变默认行为，调用信号处理程序 *sigtstp_handler* ，在该程序中通过 *kill* 函数发送信号 *SIGTSTP* 给前台进程组中的每个进程，停止前台作业，促使内核发送 *SIGCHLD* 信号，调用信号处理程序 *sigchld_handler* 。

---

other functions

```
pid_t Fork(void);
int Sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int Sigemptyset(sigset_t *set);
int Sigfillset(sigset_t *set);
int Sigaddset(sigset_t *set, int signum);
int Sigdelset(sigset_t *set, int signum);
int Sigismember(const sigset_t *set, int signum);
int Kill(pid_t pid, int sig);

pid_t Fork(void){
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
int Sigprocmask(int how, const sigset_t *set, sigset_t *oldset){
    int fg;
    if ((fg = sigprocmask(how, set, oldset)) < 0)
        unix_error("sigprocmask error");
    return fg;
}
int Sigemptyset(sigset_t *set){
    int fg;
    if ((fg = sigemptyset(set)) < 0)
        unix_error("sigemptyset error");
    return fg;
}
int Sigfillset(sigset_t *set){
    int fg;
    if ((fg = sigfillset(set)) < 0)
        unix_error("sigfillset error");
    return fg;
}
int Sigaddset(sigset_t *set, int fgnum){
    int fg;
    if ((fg = sigaddset(set, fgnum)) < 0)
        unix_error("sigaddset error");
    return fg;
}
int Sigdelset(sigset_t *set, int fgnum){
    int fg;
    if ((fg = sigdelset(set, fgnum)) < 0)
        unix_error("sigdelset error");
```

```
    return fg;
}
int Sigismember(const sigset_t *set, int fgnum){
    int fg;
    if ((fg = sigismember(set, fgnum)) < 0)
        unix_error("sigismember error");
    return fg;
}
int Kill(pid_t pid, int sig){
    int fg;
    if ((fg = kill(pid, sig)) < 0)
        unix_error("kill error");
    return fg;
}
```

附：完整代码

```
/*
 * tsh - A tiny shell program with job control
 *
 * <Put your name and login ID here>
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

/* Misc manifest constants */
#define MAXLINE    1024   /* max line size */
#define MAXARGS     128   /* max args on a command line */
#define MAXJOBS      16   /* max jobs at any point in time */
#define MAXJID    1<<16   /* max job ID */

/* Job states */
#define UNDEF 0 /* undefined */
#define FG 1    /* running in foreground */
#define BG 2    /* running in background */
#define ST 3    /* stopped */

/*
 * Jobs states: FG (foreground), BG (background), ST (stopped)
 * Job state transitions and enabling actions:
 *     FG -> ST  : ctrl-z
 *     ST -> FG  : fg command
 *     ST -> BG  : bg command
 *     BG -> FG  : fg command
 * At most 1 job can be in the FG state.
 */

/* Global variables */
extern char **environ;      /* defined in libc */
```

```
char prompt[] = "tsh> ";    /* command line prompt (DO NOT CHANGE) */
int verbose = 0;            /* if true, print additional output */
int nextjid = 1;            /* next job ID to allocate */
char sbuf[MAXLINE];         /* for composing sprintf messages */

struct job_t {              /* The job struct */
    pid_t pid;              /* job PID */
    int jid;                /* job ID [1, 2, ...] */
    int state;              /* UNDEF, BG, FG, or ST */
    char cmdline[MAXLINE];  /* command line */
};
struct job_t jobs[MAXJOBS]; /* The job list */
/* End global variables */


/* Function prototypes */

/* Here are the functions that you will implement */
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);

/* Here are helper routines that we've provided for you */
int parseline(const char *cmdline, char **argv);
void sigquit_handler(int sig);

void clearjob(struct job_t *job);
void initjobs(struct job_t *jobs);
int maxjid(struct job_t *jobs);
int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline);
int deletejob(struct job_t *jobs, pid_t pid);
pid_t fgpid(struct job_t *jobs);
struct job_t *getjobpid(struct job_t *jobs, pid_t pid);
struct job_t *getjobjid(struct job_t *jobs, int jid);
int pid2jid(pid_t pid);
void listjobs(struct job_t *jobs);

void usage(void);
void unix_error(char *msg);
void app_error(char *msg);
typedef void handler_t(int);
handler_t *Signal(int signum, handler_t *handler);

pid_t Fork(void);
int Sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int Sigemptyset(sigset_t *set);
int Sigfillset(sigset_t *set);
int Sigaddset(sigset_t *set, int signum);
int Sigdelset(sigset_t *set, int signum);
int Sigismember(const sigset_t *set, int signum);
int Kill(pid_t pid, int sig);

/*
```

```c
 * main - The shell's main routine
 */
int main(int argc, char **argv)
{
    char c;
    char cmdline[MAXLINE];
    int emit_prompt = 1; /* emit prompt (default) */

    /* Redirect stderr to stdout (so that driver will get all output
     * on the pipe connected to stdout) */
    dup2(1, 2);

    /* Parse the command line */
    while ((c = getopt(argc, argv, "hvp")) != EOF) {
        switch (c) {
            case 'h':             /* print help message */
                usage();
                break;
            case 'v':             /* emit additional diagnostic info */
                verbose = 1;
                break;
            case 'p':             /* don't print a prompt */
                emit_prompt = 0;  /* handy for automatic testing */
                break;
            default:
                usage();
        }
    }

    /* Install the signal handlers */

    /* These are the ones you will need to implement */
    Signal(SIGINT,  sigint_handler);   /* ctrl-c */
    Signal(SIGTSTP, sigtstp_handler);  /* ctrl-z */
    Signal(SIGCHLD, sigchld_handler);  /* Terminated or stopped child */

    /* This one provides a clean way to kill the shell */
    Signal(SIGQUIT, sigquit_handler);

    /* Initialize the job list */
    initjobs(jobs);

    /* Execute the shell's read/eval loop */
    while (1) {

        /* Read command line */
        if (emit_prompt) {
            printf("%s", prompt);
            fflush(stdout);
        }
        if ((fgets(cmdline, MAXLINE, stdin) == NULL) && ferror(stdin))
            app_error("fgets error");
        if (feof(stdin)) { /* End of file (ctrl-d) */
            fflush(stdout);
            exit(0);
        }

        /* Evaluate the command line */
```

```
        eval(cmdline);
        fflush(stdout);
        fflush(stdout);
    }

    exit(0); /* control never reaches here */
}

pid_t Fork(void){
    pid_t pid;

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

/*
 * 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];
    char buf[MAXLINE];
    int bg;
    sigset_t mask_all, mask_one, prev_all;
    pid_t pid;

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;

    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);

    if (!builtin_cmd(argv)){
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_all);
        if ((pid = Fork()) == 0){
            Sigprocmask(SIG_SETMASK, &prev_all, NULL);
            setpgid(0, 0);
            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, buf);
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
```

```
        if (!bg){
            //printf("Not bg\n");
            waitfg(pid);
        }else{
            //printf("Is bg\n");
            printf("[%d] (%d) %s", pid2jid(pid), pid, buf);
        }
    }

    return;
}

/*
 * parseline - Parse the command line and build the argv array.
 *
 * Characters enclosed in single quotes are treated as a single
 * argument.  Return true if the user has requested a BG job, false if
 * the user has requested a FG job.
 */
int parseline(const char *cmdline, char **argv)
{
    static char array[MAXLINE]; /* holds local copy of command line */
    char *buf = array;          /* ptr that traverses command line */
    char *delim;                /* points to first space delimiter */
    int argc;                   /* number of args */
    int bg;                     /* background job? */

    strcpy(buf, cmdline);
    buf[strlen(buf)-1] = ' ';  /* replace trailing '\n' with space */
    while (*buf && (*buf == ' ')) /* ignore leading spaces */
        buf++;

    /* Build the argv list */
    argc = 0;
    if (*buf == '\'') {
        buf++;
        delim = strchr(buf, '\'');
    }
    else {
        delim = strchr(buf, ' ');
    }

    while (delim) {
        argv[argc++] = buf;
        *delim = '\0';
        buf = delim + 1;
        while (*buf && (*buf == ' ')) /* ignore spaces */
            buf++;

        if (*buf == '\'') {
            buf++;
            delim = strchr(buf, '\'');
        }
        else {
            delim = strchr(buf, ' ');
        }
    }
```

```
    argv[argc] = NULL;

    if (argc == 0)  /* ignore blank line */
        return 1;

    /* should the job run in the background? */
    if ((bg = (*argv[argc-1] == '&')) != 0) {
        argv[--argc] = NULL;
    }
    return bg;
}

/*
 * builtin_cmd - If the user has typed a built-in command then execute
 *    it immediately.
 */
int builtin_cmd(char **argv)
{
    if (!strcmp(argv[0], "quit"))   /* quit command */
        exit(0);
    if (!strcmp(argv[0], "fg") || !strcmp(argv[0], "bg")){    /* fg or bg
command */
        do_bgfg(argv);
        return 1;
    }
    if (!strcmp(argv[0], "jobs")){    /* jobs command */
        listjobs(jobs);
        return 1;
    }
    return 0;     /* not a builtin command */
}

/*
 * do_bgfg - Execute the builtin bg and fg commands
 */
void do_bgfg(char **argv)
{
    pid_t pid;
    int jid;
    struct job_t *job = NULL;
    char *id = argv[1];

    if (id == NULL){
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }
    if (id[0] == '%'){
        jid = atoi(&id[1]);
        job = getjobjid(jobs, jid);
        if (job == NULL){
            printf("%%%d: No such job\n", jid);
            return;
        }
    } else{
        pid = atoi(id);
        if (pid == 0){  /* function atoi return 0 if id is not a digit */
            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;
        }
    }
/*
    if (id[0] == '%'){
        jid = atoi(&id[1]);
        job = getjobjid(jobs, jid);
        if (job == NULL){
            printf("(%d): No such process\n", jid);
            return;
        }
    } else if (isdigit(id[0])){
        pid = atoi(id);
        job = getjobpid(jobs, pid);
        if (job == NULL){
            printf("(%d): No such process\n", pid);
            return;
        }
    } else {
        pid = atoi(id);
        printf("%d\n", pid);
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }
*/
    //Kill(-(job->pid), SIGCONT);

    if (!strcmp(argv[0], "bg")){
        //printf("Do bg\n");
        job->state = BG;
        Kill(-(job->pid), SIGCONT);
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
    }else {
        //printf("Do fg\n");
        job->state = FG;
        Kill(-(job->pid), SIGCONT);
        waitfg(job->pid);
    }

    return;
}

/*
 * waitfg - Block until process pid is no longer the foreground process
 */
void waitfg(pid_t pid)
{
    while (pid == fgpid(jobs)){
        sleep(0);
    }
    return;
}

/*****************
```

```
 * Signal handlers
 *****************/

/*
 * 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.
 */
void sigchld_handler(int sig)
{
    int olderrno = errno;
    int status;
    sigset_t mask_all, prev_all;
    pid_t pid;

    Sigfillset(&mask_all);
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0){ /* Reap a
zombie child */
        int jid = pid2jid(pid);
        if (WIFEXITED(status)){
            Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
            deletejob(jobs, pid);
            Sigprocmask(SIG_SETMASK, &prev_all, NULL);
            //printf("Job [%d] (%d) terminated normally with exit
status=%d\n",jid, pid, WEXITSTATUS(status));
            return;
        }
        if (WIFSIGNALED(status)){
            Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
            deletejob(jobs, pid);
            Sigprocmask(SIG_SETMASK, &prev_all, NULL);
            printf("Job [%d] (%d) terminated by signal %d\n", jid, pid,
WTERMSIG(status));
            return;
        }
        if (WIFSTOPPED(status)){
            Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
            struct job_t *job = getjobpid(jobs, pid);
            if (job != NULL)
                job->state = ST;
            Sigprocmask(SIG_SETMASK, &prev_all, NULL);
            printf("Job [%d] (%d) stopped by signal %d\n", jid, pid,
WSTOPSIG(status));
            return;
        }
    }
/*
    if (errno != ECHILD)
    unix_error("waitpid error");
*/
    errno = olderrno;

    return;
}

/*
```

```
 * 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)
{
    pid_t pid = fgpid(jobs);
    if (pid != 0)
        Kill(-pid, sig);
    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.
 */
void sigtstp_handler(int sig)
{
/*
    pid_t pid = fgpid(jobs);
    if (pid != 0){
        struct job_t *job = getjobpid(jobs, pid);
        job->state = ST;
        printf("Job [%d] (%d) stopped by signal %d\n", job->jid, pid, sig);
    }
*/
    int olderrno = errno;
    pid_t pid = fgpid(jobs);
    if (pid != 0){
        Kill(-pid, sig);
    }
    errno = olderrno;

    return;
}

/*********************
 * End signal handlers
 *********************/

/***********************************************
 * Helper routines that manipulate the job list
 **********************************************/

/* clearjob - Clear the entries in a job struct */
void clearjob(struct job_t *job) {
    job->pid = 0;
    job->jid = 0;
    job->state = UNDEF;
    job->cmdline[0] = '\0';
}

/* initjobs - Initialize the job list */
void initjobs(struct job_t *jobs) {
    int i;

    for (i = 0; i < MAXJOBS; i++)
```

```
        clearjob(&jobs[i]);
}

/* maxjid - Returns largest allocated job ID */
int maxjid(struct job_t *jobs)
{
    int i, max=0;

    for (i = 0; i < MAXJOBS; i++)
        if (jobs[i].jid > max)
            max = jobs[i].jid;
    return max;
}

/* addjob - Add a job to the job list */
int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline)
{
    int i;

    if (pid < 1)
    return 0;

    for (i = 0; i < MAXJOBS; i++) {
        if (jobs[i].pid == 0) {
            jobs[i].pid = pid;
            jobs[i].state = state;
            jobs[i].jid = nextjid++;
            if (nextjid > MAXJOBS)
                nextjid = 1;
            strcpy(jobs[i].cmdline, cmdline);
            if(verbose){
                printf("Added job [%d] %d %s\n", jobs[i].jid, jobs[i].pid,
jobs[i].cmdline);
            }
            return 1;
        }
    }
    printf("Tried to create too many jobs\n");
    return 0;
}

/* deletejob - Delete a job whose PID=pid from the job list */
int deletejob(struct job_t *jobs, pid_t pid)
{
    int i;

    if (pid < 1)
        return 0;

    for (i = 0; i < MAXJOBS; i++) {
        if (jobs[i].pid == pid) {
            clearjob(&jobs[i]);
            nextjid = maxjid(jobs)+1;
            return 1;
        }
    }
    return 0;
}
```

```c
/* fgpid - Return PID of current foreground job, 0 if no such job */
pid_t fgpid(struct job_t *jobs) {
    int i;

    for (i = 0; i < MAXJOBS; i++)
        if (jobs[i].state == FG)
            return jobs[i].pid;
    return 0;
}

/* getjobpid  - Find a job (by PID) on the job list */
struct job_t *getjobpid(struct job_t *jobs, pid_t pid) {
    int i;

    if (pid < 1)
        return NULL;
    for (i = 0; i < MAXJOBS; i++)
        if (jobs[i].pid == pid)
            return &jobs[i];
    return NULL;
}

/* getjobjid  - Find a job (by JID) on the job list */
struct job_t *getjobjid(struct job_t *jobs, int jid)
{
    int i;

    if (jid < 1)
        return NULL;
    for (i = 0; i < MAXJOBS; i++)
        if (jobs[i].jid == jid)
            return &jobs[i];
    return NULL;
}

/* pid2jid - Map process ID to job ID */
int pid2jid(pid_t pid)
{
    int i;

    if (pid < 1)
        return 0;
    for (i = 0; i < MAXJOBS; i++)
        if (jobs[i].pid == pid) {
            return jobs[i].jid;
        }
    return 0;
}

/* listjobs - Print the job list */
void listjobs(struct job_t *jobs)
{
    int i;

    for (i = 0; i < MAXJOBS; i++) {
        if (jobs[i].pid != 0) {
            printf("[%d] (%d) ", jobs[i].jid, jobs[i].pid);
```

```
            switch (jobs[i].state) {
            case BG:
                printf("Running ");
                break;
            case FG:
                printf("Foreground ");
                break;
            case ST:
                printf("Stopped ");
                break;
            default:
                printf("listjobs: Internal error: job[%d].state=%d ",
                   i, jobs[i].state);
            }
            printf("%s", jobs[i].cmdline);
        }
    }
}
/******************************
 * end job list helper routines
 ******************************/


/***********************
 * Other helper routines
 ***********************/

/*
 * usage - print a help message
 */
void usage(void)
{
    printf("Usage: shell [-hvp]\n");
    printf("   -h   print this message\n");
    printf("   -v   print additional diagnostic information\n");
    printf("   -p   do not emit a command prompt\n");
    exit(1);
}

/*
 * unix_error - unix-style error routine
 */
void unix_error(char *msg)
{
    fprintf(stdout, "%s: %s\n", msg, strerror(errno));
    exit(1);
}

/*
 * app_error - application-style error routine
 */
void app_error(char *msg)
{
    fprintf(stdout, "%s\n", msg);
    exit(1);
}

/*
```

```
 * Signal - wrapper for the sigaction function
 */
handler_t *Signal(int signum, handler_t *handler)
{
    struct sigaction action, old_action;

    action.sa_handler = handler;
    sigemptyset(&action.sa_mask); /* block sigs of type being handled */
    action.sa_flags = SA_RESTART; /* restart syscalls if possible */

    if (sigaction(signum, &action, &old_action) < 0)
        unix_error("Signal error");
    return (old_action.sa_handler);
}

/*
 * sigquit_handler - The driver program can gracefully terminate the
 *    child shell by sending it a SIGQUIT signal.
 */
void sigquit_handler(int sig)
{
    printf("Terminating after receipt of SIGQUIT signal\n");
    exit(1);
}


pid_t Fork(void){
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
int Sigprocmask(int how, const sigset_t *set, sigset_t *oldset){
    int fg;
    if ((fg = sigprocmask(how, set, oldset)) < 0)
        unix_error("sigprocmask error");
    return fg;
}
int Sigemptyset(sigset_t *set){
    int fg;
    if ((fg = sigemptyset(set)) < 0)
        unix_error("sigemptyset error");
    return fg;
}
int Sigfillset(sigset_t *set){
    int fg;
    if ((fg = sigfillset(set)) < 0)
        unix_error("sigfillset error");
    return fg;
}
int Sigaddset(sigset_t *set, int fgnum){
    int fg;
    if ((fg = sigaddset(set, fgnum)) < 0)
        unix_error("sigaddset error");
    return fg;
}
int Sigdelset(sigset_t *set, int fgnum){
```

```
    int fg;
    if ((fg = sigdelset(set, fgnum)) < 0)
        unix_error("sigdelset error");
    return fg;
}
int Sigismember(const sigset_t *set, int fgnum){
    int fg;
    if ((fg = sigismember(set, fgnum)) < 0)
        unix_error("sigismember error");
    return fg;
}
int Kill(pid_t pid, int sig){
    int fg;
    if ((fg = kill(pid, sig)) < 0)
        unix_error("kill error");
    return fg;
}
```