# 1 Writing a simple linux shell

**Solution** This is a simple linux shell named *tsh*. It has the following features:

- The command line typed by the user should consist of a name and zero or more arguments, all separated by one or more spaces. If name is a built-in command, then *tsh* handles it immediately and waits for the next command line. Otherwise, *tsh* assumes that name is the path of an executable file, which it loads and runs in the context of an initial child process.
- *tsh* does not support pipes (|) or I/O redirection (< and >).
- Typing ctrl-c (ctrl-z) causes a SIGINT (SIGTSTP) signal to be sent to the current foreground job, as well as any descendents of that job. If there is no foreground job, then the signal will have no effect.
- If the command line ends with an ampersand &, then *tsh* runs the job in the background. Otherwise, it runs the job in the foreground.
- 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 are denoted on the command line by the prefix %. For example, %5 denotes JID 5, and 5 denotes PID 5.
- *tsh* supports the following built-in commands:
  - quit command terminates the shell.
  - jobs command lists all background jobs.
  - 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.
  - 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.

The source code of the *tsh* is shown on listing 1.

```c
#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 */
```

Listing 1 (Cont.): *tsh*.c

```c
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);

/*
 * 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;
```

Listing 1 (Cont.): *tsh*.c

```
        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);
    }

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

/*
 * 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];
    int bg;
    bg = parseline(cmdline, argv);
    if (builtin_cmd(argv))
        return;

    sigset_t mask;
    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");

    pid_t pid;
```

Listing 1 (Cont.): *tsh.c*

```c
    if ((pid = fork()) == 0)
    {
        if (sigprocmask(SIG_UNBLOCK, &mask, NULL) < 0)
            unix_error("sigprocmask error");

        setpgid(0, 0);
        if (execve(argv[0], argv, environ) < 0)
        {
            printf("%s: Command not found.\n", argv[0]);
            exit(0);
        }
    }
    else if (pid < 0)
    {
        unix_error("fork error");
    }

    int state = bg ? BG : FG;
    addjob(jobs, pid, state, cmdline);
    if (sigprocmask(SIG_UNBLOCK, &mask, NULL) < 0)
        unix_error("sigprocmask error");

    if (!bg)
    {
        waitfg(pid);
    }
    else
    {
        printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
        fflush(stdout);
    }

    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 */
```

Listing 1 (Cont.): *tsh*.c

```c
            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"))
        exit(0);

    if (!strcmp(argv[0], "jobs"))
    {
        listjobs(jobs);
        return 1;
    }

    if (!strcmp(argv[0], "bg") || !strcmp(argv[0], "fg"))
    {
        do_bgfg(argv);
        return 1;
    }

    return 0; /* not a builtin command */
}

/*
 * do_bgfg - Execute the builtin bg and fg commands
 */
void do_bgfg(char **argv)
{
    if (argv[1] == NULL)
    {
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }

    int jid;
    int pid;
    struct job_t *job;
    if (sscanf(argv[1], "%%%d", &jid))
    {
        job = getjobjid(jobs, jid);
        if (job == NULL)
        {
            printf("%s: No such job\n", argv[1]);
            return;
        }
```

Listing 1 (Cont.): *tsh*.c

```c
    }
    else if (sscanf(argv[1], "%d", &pid))
    {
        job = getjobpid(jobs, pid);
        if (job == NULL)
        {
            printf("(%s) No such process\n", argv[1]);
            return;
        }
    }
    else
    {
        printf("%s: argument must be PID or %%jobid\n", argv[0]);
        return;
    }

    if (!strcmp(argv[0], "bg"))
    {
        job->state = BG;
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
        kill(-(job->pid), SIGCONT);
    }
    else
    {
        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)
{
    struct job_t *job;
    while ((job = getjobpid(jobs, pid)) != NULL)
        if (job->state == FG)
            sleep(5);
        else
            break;
}

/*****************
 * 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)
{
    pid_t pid;
    int status;
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)
    {
        if (WIFEXITED(status))
        {
            deletejob(jobs, pid);
            continue;
        }

        if (WIFSIGNALED(status))
        {
            printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid),
                pid, WTERMSIG(status));
            deletejob(jobs, pid);
```

Listing 1 (Cont.): *tsh.c*

```c
            continue;
        }

        if (WIFSTOPPED(status))
        {
            printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid,
                WSTOPSIG(status));
            struct job_t *job;
            if ((job = getjobpid(jobs, pid)) == NULL)
                app_error("job not found");

            job->state = ST;
            continue;
        }
    }

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

    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;
    if ((pid = fgpid(jobs)) > 0)
        kill(-pid, SIGINT);

    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;
    if ((pid = fgpid(jobs)) > 0)
        kill(-pid, SIGTSTP);

    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]);
```

Listing 1 (Cont.): *tsh*.c

```c
}

/* 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;
}

/* 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;
}
```

Listing 1 (Cont.): *tsh*.c

```c

/* 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
```

Listing 1 (Cont.): *tsh*.c

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

/***********************
 * 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);
}
```

Listing 1: *tsh*.c