

A Tiny Unix Shell (tsh) – Part I

Introduction

This is the first part of a bigger assignment on process control and signaling. To expand your understanding on these topics, you will write a simple Unix shell program that supports job control.

This first assignment asks you to write a few helper routines for job manipulation. In doing so, you will also learn how to manipulate pointers and structures in C. This assignment must be completed *individually*.

General Overview of Unix Shells

A *shell* is an interactive command-line interpreter that runs programs on behalf of the user. A shell repeatedly prints a prompt, waits for a *command line* on `stdin`, and then carries out some action, as directed by the contents of the command line.

The command line is a sequence of ASCII text words delimited by whitespace. The first word in the command line is either the name of a built-in command or the pathname of an executable file. The remaining words are command-line arguments. If the first word is a built-in command, the shell immediately executes the command in the current process. Otherwise, the word is assumed to be the *pathname* of an executable program. In this case, the shell forks a child process, then loads and runs the program in the context of the child. The child processes created as a result of interpreting a single command line are known collectively as a *job*.

If the command line ends with an ampersand "&", then the job runs in the *background*, which means that the shell does not wait for the job to terminate before printing the prompt and awaiting the next command line. Otherwise, the job runs in the *foreground*, which means that the shell waits for the job to terminate before awaiting the next command line. Thus, at any point in time, at most one job can be running in the foreground. However, an arbitrary number of jobs can run in the background.

For example, typing the command line

```
tsh> jobs
```

causes the shell to execute the built-in `jobs` command. Typing the command line

```
tsh> /bin/ls -l -d
```

runs the `ls` program in the foreground. By convention, the shell ensures that when the program begins executing its main routine

```
int main(int argc, char *argv[])
```

the `argc` and `argv` arguments have the following values:

- `argc == 3,`
- `argv[0] == ``/bin/ls'',`
- `argv[1]== ``-l'',`
- `argv[2]== ``-d''.`

Alternatively, typing the command line

```
tsh> /bin/ls -l -d &
```

runs the `ls` program in the background.

Unix shells support the notion of *job control*, which allows users to move jobs back and forth between background and foreground, and to change the process state (running, stopped, or terminated) of the processes in a job. Unix shells provide various built-in commands that support job control. For example:

- `jobs`: List the running and stopped background jobs.
- `bg <job>`: Change a stopped background job to a running background job.
- `fg <job>`: Change a stopped or running background job to a running in the foreground.
- `kill <job>`: Terminate a job.

Manipulating Jobs

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.

In this assignment you will write a set of routines for manipulating jobs stored in a list. Start with the following definitions:

```

#include <unistd.h>

#define MAXLINE      1024    /* max line size */
#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 */
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 */
/* Recall that jobs is a pointer to the first job in the list */

int nextjid = 1;      /* Next job ID to allocate */
/* End global variables */

```

Implement the following functions:

1. `void clearjob(struct job_t *pjob);`
 Clears the entries in the job passed as an argument (sets `pid` and `jid` to 0, `state` to `UNDEF` and `cmdline` to the `NULL` string).
2. `void initjobs(struct job_t *jobs);`
 Initializes the job list (by clearing all entries).
3. `int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline);`
 Adds to `jobs` a new job with identifier `pid`, `state` and `cmdline` as in the argument list. The job identifier of the new job is the next available one (the value of `nextjid`). If `nextjid` exceeds `MAXJID`, reset it back to 1. Make sure to not exceed the maximum number of jobs, `MAXJOBS`.

4. `int maxjid(struct job_t *jobs);`
Returns the largest allocated job ID.
5. `int deletejob(struct job_t *jobs, pid_t pid);`
Deletes from `jobs` that job whose process id equals `pid`. Update `nextjid`, if necessary (i.e, if the job removed had maximum `jid`, then `nextjid` should become `maxjid(jobs)+1`).
6. `pid_t fgpid(struct job_t *jobs);`
Returns the identifier `pid` of current foreground job from the `jobs` list, 0 if no such job.
7. `struct job_t *getjobpid(struct job_t *jobs, pid_t pid);`
Finds a job (by searching for `pid`) in `jobs`.
8. `struct job_t *getjobjid(struct job_t *jobs, int jid);`
Finds a job (by searching for `jid`) in the job list `jobs`.
9. `int pid2jid(pid_t pid);`
Returns the job ID of the process with identifier `pid` from `jobs`.
10. `void listjobs(struct job_t *jobs);`
Prints out all jobs in `jobs` (one per line).

Thoroughly test each function by creating sample calls in the main function. Here is an example:

```
int main()

    initjobs(jobs);
    listjobs(jobs);
    addjob(jobs, 111, FG, "ls -l");
    listjobs(jobs);
    /* etc. */
```

Hint: To find out the name of a header containing the declaration of a particular function, type in

```
man function_name
```

at the shell prompt. For instance, typing in

```
man strcpy
```

will list information about the `strcpy` function.

Evaluation

Your score will be computed out of a maximum of 50 points based on the following distribution:

40 Correctness.

10 Style points. We expect you to have good comments and to check for errors (eg., deleting from an empty list, or checking the validity of an argument (`jid < 0`, for instance), or checking for NULL pointers, etc.). Do not allow me to break your code by trying something like

```
addjob(jobs, 100, FG, 0);  
listjobs(jobs);
```

Hand In Instructions

Hand in a printed copy of your source code (`tsh.c`) and 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. I will test your code on felix. Good luck!