Assignment 3: *shlab*

CSE4009: System Programming

Goals

- Understanding how a shell process works
  - Interpreting commands
  - Sending a signals to the target processes
  - Handling received signals

Check out Your Assignment

- Clone it to the project directory
  - At your VM instance

```
$ cd Projects/{your directory}
$ git pull origin main
$ cd ./03_shlab
```

Check your files

- You have files
  - `README`
  - `Makefile`
  - `sdriver.pl`: testing program
  - **`tsh.c`**: your tiny shell (incomplete)
  - `tsh-ref`: the reference binary for tsh.c
  - `trace01.txt ~ trace16.txt`: tests to validate your tsh
  - `tshref.out`: example output for tshref
  - `myspin.c, mysplit.c, mystop.c, myint.c`

Instructions for *shlab*

- A ***shell*** is an application program that runs programs on behalf of the user
  - **`sh`** Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - **`csh/tcsh`** BSD Unix C shell
  - **`bash`** "Bourne-Again" Shell (default Linux shell)

- Your shell performs in a sequence of steps
  - Read – read a command line from the user
  - Evaluate – parses the command line and runs programs
- Your shell should perform with commands in trace01~16.txt
- And the results should be same as **`tshref`**

1. Handling Built-in Commands

- test01 seems to be done, and move on to test02
  - Your **`tsh`** should exit with `quit` command.
  - To do so you need to complete `eval()`
    - converts a command in the line `cmdline` to `char*[] argv`
    - check if it has a valid command with `builtin_cmd(char**)`

      `quit, jobs, bg, fg`

```
161 /*
162  * eval - Evaluate the command line that the user has just typed in
163  *
164  * If the user has requested a built-in command (quit, jobs, bg or fg)
165  * then execute it immediately. Otherwise, fork a child process and
166  * run the job in the context of the child. If the job is running in
167  * the foreground, wait for it to terminate and then return.  Note:
168  * each child process must have a unique process group ID so that our
169  * background children don't receive SIGINT (SIGTSTP) from the kernel
170  * when we type ctrl-c (ctrl-z) at the keyboard.
171  */
172 void eval(char *cmdline)
173 {
174   return;
175 }
```

Built-in Commands

- `quit` – terminate the shell itself
- `jobs` – list all jobs created by the current shell process
- `bg {pid|job_id}`
- `fg {pid|job_id}`

```
tsh> ./myspin 30
^ZJob [1] (9228) stopped by signal 20
tsh> ./myspin 30 &
[2] (9229)./myspin 30 &
tsh> ./myspin 30 &
[3] (9230)./myspin 30 &
tsh> jobs
[1] (9228) Stopped ./myspin 30
[2] (9229) Running ./myspin 30 &
[3] (9230) Running ./myspin 30 &
tsh> bg %1 (or 9228)
[1] (9228) ./myspin 30
tsh> jobs
[1] (9228) Running ./myspin 30
[2] (9229) Running ./myspin 30 &
[3] (9230) Running ./myspin 30 &
tsh> fg %3
```

2. Launch a New Program

- test03 requires execution of an external programs
- The new process runs in either foreground or background
- When the program exits, control returns to **`tsh`**
- To this end, you need `fork()` & `exec()`

```
173 void eval(char *cmdline)
174 {
175   pid_t pid;
176   char *argv[MAXARGS];
177   parseline(cmdline, argv);
178   if (!builtin_cmd(argv)) {
179     if ((pid = fork()) == 0) {
180       execve(argv[0], argv, environ);
181
182     }
183   }
184   return;
185 }
```

addjob()

- Job is not a Linux process
  - Process is Linux data structure created by `fork()`
  - Job is an own data structure for the shell
    - jobs is a global variable for the job list
    - Each job has own `jid` (and `pid` as well)
    - Each job has three states: `FG`, `BG`, `ST`

```
330 /* addjob - Add a job to the job list */
331 int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline)
332 {
333   int i;
334
335   if (pid < 1)
336     return 0;
337
338   for (i = 0; i < MAXJOBS; i++) {
339     if (jobs[i].pid == 0) {
340       jobs[i].pid = pid;
341       jobs[i].state = state;
342       jobs[i].jid = nextjid++;
343       if (nextjid > MAXJOBS)
344         nextjid = 1;
345       strcpy(jobs[i].cmdline, cmdline);
346       if(verbose){
347         printf("Added job [%d] %d %s\n", jobs[i].jid, jobs[i].pid, jobs[i].cmdline);
348       }
349       return 1;
350     }
351   }
352   printf("Tried to create too many jobs\n");
353   return 0;
354 }
```

```
29 /*
30  * Jobs states: FG (foreground), BG (background), ST (stopped)
31  * Job state transitions and enabling actions:
32  *     FG -> ST  : ctrl-z
33  *     ST -> FG  : fg command
34  *     ST -> BG  : bg command
35  *     BG -> FG  : fg command
36  * At most 1 job can be in the FG state.
37  */
```

deletejob()

- When the process exits, your **`tsh`** should reap it and remove it from the job list
  - `void waitfg(pid_t)`
  - `void deletejob(struct job_t*, pid_t)`

```
void eval(char *cmdline)
{
  pid_t pid;
  char *argv[MAXARGS];
  parseline(cmdline, argv);
  if (argv[0] && !builtin_cmd(argv)) {
    if ((pid = fork()) == 0) {
      if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found\n", argv[0]);
        exit(0);
      }
    } else if (pid > 0) {
      addjob(jobs, pid, FG, cmdline);
      waitfg(pid);
    }
  }
  return;
}
```

waitfg()

- The example considers a foreground process only
- Your `tsh` should handler for a backround process as well

```c
/*
 * waitfg - Block until process pid is no longer the foreground process
 */
void waitfg(pid_t pid)
{
  int status;
  waitpid(pid, &status, 0);
  deletejob(jobs, pid);
}
```

```c
void eval(char *cmdline)
{
  int bg;
  pid_t pid;
  char *argv[MAXARGS];
  bg = parseline(cmdline, argv);
  if (argv[0] && !builtin_cmd(argv)) {
    if ((pid = fork()) == 0) {
      if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found\n", argv[0]);
        exit(0);
      }
    } else if (pid > 0) {
      addjob(jobs, pid, bg ? BG : FG, cmdline);
      if (!bg)
        waitfg(pid);
      else
        fprintf(stderr, "[%d] (%d) %s", pid2jid(pid), pid, cmdline);
    }
  }
  return;
}
```

listjob()

- & makes the process run in the background
- **jobs** shows all jobs created by **tsh**

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

3. Signals from Background Processes

- The example cannot reap child processes in the background

```
void sigchld_handler(int sig)
{
  pid_t pid;
  int status;
  while ((pid = waitpid(-1, &status, 0)) > 0) {
    if (WIFEXITED(status)) {
      deletejob(jobs, pid);
    }
  }
  return;
}
```

```
void waitfg(pid_t pid)
{
  struct job_t* job = getjobpid(jobs, pid);
  if (job) {
    while (job->pid == pid && job->state == FG) {
      sleep(1);
    }
  }
}
```

4. Signals from Keyboard

- `Ctrl-c/z` sends `SIGINT/SIGTSTP`
  - Kernel sends signals to all processes in the same group with `tsh`
    - `setpgid(0, 0);`
  - Shell sends signals to a foreground process
    - `void sigtstp_handler(int);`
  - Shell receives SIGCHLD signals when the child process stopped

```c
void sigtstp_handler(int sig)
{
  pid_t pid;
  if ((pid = fgpid(jobs)) > 0) {
    kill(-pid, SIGTSTP);
  }
  return;
}
```

```c
void sigchld_handler(int sig)
{
  pid_t pid;
  int status;
  struct job_t* job;
  while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
    if (WIFSTOPPED(status)) {
      job = getjobpid(jobs, pid);
      if (!job)
        continue;
      job->state = ST;
      fprintf(stderr, "Job [%d] (%d) stopped by signal %d\n",
          pid2jid(pid), pid, WSTOPSIG(status));
    } else if (WIFEXITED(status)) {
      deletejob(jobs, pid);
    }
  }
}
```

Self-test for Your `tsh`

- `make rtest01 ~ rtest16`
  - Check how your tsh behaves with each trace file
- `make test01 ~ test16`
  - self-test your tsh with each trace file
- `make {rtests|tests}`
  - self-test with all trace files
- Make sure each output is identical to the reference, except for PID

Submission Guideline

- Push your `tsh.c`

```
$ git add tsh.c
$ git commit -m "submission of assignment 03"
$ git push origin main
```