

# 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.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <ctype.h>
6  #include <signal.h>
7  #include <sys/types.h>
8  #include <sys/wait.h>
9  #include <errno.h>
10
11  /* Misc manifest constants */
12  #define MAXLINE 1024 /* max line size */
13  #define MAXARGS 128 /* max args on a command line */
14  #define MAXJOBS 16 /* max jobs at any point in time */
15  #define MAXJID 1 << 16 /* max job ID */
16
17  /* Job states */
18  #define UNDEF 0 /* undefined */
19  #define FG 1 /* running in foreground */
20  #define BG 2 /* running in background */
21  #define ST 3 /* stopped */
22
23  /*
24   * Jobs states: FG (foreground), BG (background), ST (stopped)
25   * Job state transitions and enabling actions:
26   *   FG -> ST : ctrl-z
27   *   ST -> FG : fg command
28   *   ST -> BG : bg command
29   *   BG -> FG : fg command
30   * At most 1 job can be in the FG state.
31   */
32
33  /* Global variables */
34  extern char **environ; /* defined in libc */
35  char prompt[] = "tsh>"; /* command line prompt (DO NOT CHANGE) */
36  int verbose = 0; /* if true, print additional output */
37  int nextjid = 1; /* next job ID to allocate */
```

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

```

38  char sbuf[MAXLINE];          /* for composing sprintf messages */
39
40  struct job_t
41  {
42      pid_t pid;                /* The job struct */
43      int jid;                  /* job PID */
44      int state;                /* job ID [1, 2, ...] */
45      char cmdline[MAXLINE];    /* UNDEF, BG, FG, or ST */
46  };
47
48  struct job_t jobs[MAXJOBS]; /* The job list */
49  /* End global variables */
50
51  /* Function prototypes */
52
53  /* Here are the functions that you will implement */
54  void eval(char *cmdline);
55  int builtin_cmd(char **argv);
56  void do_bgfg(char **argv);
57  void waitfg(pid_t pid);
58
59  void sigchld_handler(int sig);
60  void sigtstp_handler(int sig);
61  void sigint_handler(int sig);
62
63  /* Here are helper routines that we've provided for you */
64  int parseline(const char *cmdline, char **argv);
65  void sigquit_handler(int sig);
66
67  void clearjob(struct job_t *job);
68  void initjobs(struct job_t *jobs);
69  int maxjid(struct job_t *jobs);
70  int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline);
71  int deletejob(struct job_t *jobs, pid_t pid);
72  pid_t fgpid(struct job_t *jobs);
73  struct job_t *getjobpid(struct job_t *jobs, pid_t pid);
74  struct job_t *getjobjid(struct job_t *jobs, int jid);
75  int pid2jid(pid_t pid);
76  void listjobs(struct job_t *jobs);
77
78  void usage(void);
79  void unix_error(char *msg);
80  void app_error(char *msg);
81  typedef void handler_t(int);
82  handler_t *Signal(int signum, handler_t *handler);
83
84  /*
85   * main - The shell's main routine
86   */
87  int main(int argc, char **argv)
88  {
89      char c;
90      char cmdline[MAXLINE];
91      int emit_prompt = 1; /* emit prompt (default) */
92
93      /* Redirect stderr to stdout (so that driver will get all output
94       * on the pipe connected to stdout) */
95      dup2(1, 2);
96
97      /* Parse the command line */
98      while ((c = getopt(argc, argv, "hvp")) != EOF)
99      {
100          switch (c)
101          {
102              case 'h': /* print help message */
103                  usage();
104                  break;
105              case 'v': /* emit additional diagnostic info */
106                  verbose = 1;
107                  break;
108              case 'p': /* don't print a prompt */
109                  emit_prompt = 0; /* handy for automatic testing */
110                  break;

```

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

```

111         default:
112             usage();
113     }
114 }
115
116 /* Install the signal handlers */
117
118 /* These are the ones you will need to implement */
119 Signal(SIGINT, sigint_handler); /* ctrl-c */
120 Signal(SIGTSTP, sigtstp_handler); /* ctrl-z */
121 Signal(SIGCHLD, sigchld_handler); /* Terminated or stopped child */
122
123 /* This one provides a clean way to kill the shell */
124 Signal(SIGQUIT, sigquit_handler);
125
126 /* Initialize the job list */
127 initjobs(jobs);
128
129 /* Execute the shell's read/eval loop */
130 while (1)
131 {
132     /* Read command line */
133     if (emit_prompt)
134     {
135         printf("%s", prompt);
136         fflush(stdout);
137     }
138     if ((fgets(cmdline, MAXLINE, stdin) == NULL) && ferror(stdin))
139         app_error("fgets error");
140     if (feof(stdin))
141     { /* End of file (ctrl-d) */
142         fflush(stdout);
143         exit(0);
144     }
145
146     /* Evaluate the command line */
147     eval(cmdline);
148     fflush(stdout);
149 }
150
151 exit(0); /* control never reaches here */
152 }
153
154 /*
155 * eval - Evaluate the command line that the user has just typed in
156 *
157 * If the user has requested a built-in command (quit, jobs, bg or fg)
158 * then execute it immediately. Otherwise, fork a child process and
159 * run the job in the context of the child. If the job is running in
160 * the foreground, wait for it to terminate and then return. Note:
161 * each child process must have a unique process group ID so that our
162 * background children don't receive SIGINT (SIGTSTP) from the kernel
163 * when we type ctrl-c (ctrl-z) at the keyboard.
164 */
165 void eval(char *cmdline)
166 {
167     char *argv[MAXARGS];
168     int bg;
169     bg = parseline(cmdline, argv);
170     if (builtin_cmd(argv))
171         return;
172
173     sigset_t mask;
174     if (sigemptyset(&mask) < 0)
175         unix_error("sigemptyset error");
176
177     if (sigaddset(&mask, SIGCHLD) < 0)
178         unix_error("sigaddset error");
179
180     if (sigprocmask(SIG_BLOCK, &mask, NULL) < 0)
181         unix_error("sigprocmask error");
182
183     pid_t pid;

```

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

```

184     if ((pid = fork()) == 0)
185     {
186         if (sigprocmask(SIG_UNBLOCK, &mask, NULL) < 0)
187             unix_error("sigprocmask error");
188
189         setpgid(0, 0);
190         if (execve(argv[0], argv, environ) < 0)
191         {
192             printf("%s: Command not found.\n", argv[0]);
193             exit(0);
194         }
195     }
196     else if (pid < 0)
197     {
198         unix_error("fork error");
199     }
200
201     int state = bg ? BG : FG;
202     addjob(jobs, pid, state, cmdline);
203     if (sigprocmask(SIG_UNBLOCK, &mask, NULL) < 0)
204         unix_error("sigprocmask error");
205
206     if (!bg)
207     {
208         waitfg(pid);
209     }
210     else
211     {
212         printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
213         fflush(stdout);
214     }
215
216     return;
217 }
218
219 /*
220  * parseline - Parse the command line and build the argv array.
221  *
222  * Characters enclosed in single quotes are treated as a single
223  * argument. Return true if the user has requested a BG job, false if
224  * the user has requested a FG job.
225  */
226 int parseline(const char *cmdline, char **argv)
227 {
228     static char array[MAXLINE]; /* holds local copy of command line */
229     char *buf = array;          /* ptr that traverses command line */
230     char *delim;                 /* points to first space delimiter */
231     int argc;                    /* number of args */
232     int bg;                      /* background job? */
233
234     strcpy(buf, cmdline);
235     buf[strlen(buf) - 1] = ' '; /* replace trailing '\n' with space */
236     while (*buf && (*buf == ' ')) /* ignore leading spaces */
237         buf++;
238
239     /* Build the argv list */
240     argc = 0;
241     if (*buf == '\'', '\'')
242     {
243         buf++;
244         delim = strchr(buf, '\'', '\'');
245     }
246     else
247     {
248         delim = strchr(buf, ' ');
249     }
250
251     while (delim)
252     {
253         argv[argc++] = buf;
254         *delim = '\0';
255         buf = delim + 1;
256         while (*buf && (*buf == ' ')) /* ignore spaces */

```

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

```

257         buf++;
258
259         if (*buf == '\\')
260         {
261             buf++;
262             delim = strchr(buf, '\\');
263         }
264         else
265         {
266             delim = strchr(buf, ' ');
267         }
268     }
269
270     argv[argc] = NULL;
271
272     if (argc == 0) /* ignore blank line */
273         return 1;
274
275     /* should the job run in the background? */
276     if ((bg = (*argv[argc - 1] == '&')) != 0)
277     {
278         argv[--argc] = NULL;
279     }
280
281     return bg;
282 }
283
284 /*
285  * builtin_cmd - If the user has typed a built-in command then execute
286  * it immediately.
287  */
288 int builtin_cmd(char **argv)
289 {
290     if (!strcmp(argv[0], "quit"))
291         exit(0);
292
293     if (!strcmp(argv[0], "jobs"))
294     {
295         listjobs(jobs);
296         return 1;
297     }
298
299     if (!strcmp(argv[0], "bg") || !strcmp(argv[0], "fg"))
300     {
301         do_bgfg(argv);
302         return 1;
303     }
304
305     return 0; /* not a builtin command */
306 }
307
308 /*
309  * do_bgfg - Execute the builtin bg and fg commands
310  */
311 void do_bgfg(char **argv)
312 {
313     if (argv[1] == NULL)
314     {
315         printf("%s command requires PID or %%jobid argument\n", argv[0]);
316         return;
317     }
318
319     int jid;
320     int pid;
321     struct job_t *job;
322     if (sscanf(argv[1], "%%%d", &jid))
323     {
324         job = getjobjid(jobs, jid);
325         if (job == NULL)
326         {
327             printf("%s: No such job\n", argv[1]);
328             return;
329         }
330     }

```

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

```

330     }
331     else if (sscanf(argv[1], "%d", &pid))
332     {
333         job = getjobpid(jobs, pid);
334         if (job == NULL)
335         {
336             printf("(%s) No such process\n", argv[1]);
337             return;
338         }
339     }
340     else
341     {
342         printf("%s: argument must be PID or %%jobid\n", argv[0]);
343         return;
344     }
345
346     if (!strcmp(argv[0], "bg"))
347     {
348         job->state = BG;
349         printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
350         kill(-(job->pid), SIGCONT);
351     }
352     else
353     {
354         job->state = FG;
355         kill(-(job->pid), SIGCONT);
356         waitfg(job->pid);
357     }
358
359     return;
360 }
361
362 /*
363  * waitfg - Block until process pid is no longer the foreground process
364  */
365 void waitfg(pid_t pid)
366 {
367     struct job_t *job;
368     while ((job = getjobpid(jobs, pid)) != NULL)
369         if (job->state == FG)
370             sleep(5);
371     else
372         break;
373 }
374
375 /******
376  * Signal handlers
377  *****/
378
379 /*
380  * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
381  * a child job terminates (becomes a zombie), or stops because it
382  * received a SIGSTOP or SIGTSTP signal. The handler reaps all
383  * available zombie children, but doesn't wait for any other
384  * currently running children to terminate.
385  */
386 void sigchld_handler(int sig)
387 {
388     pid_t pid;
389     int status;
390     while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)
391     {
392         if (WIFEXITED(status))
393         {
394             deletejob(jobs, pid);
395             continue;
396         }
397
398         if (WIFSIGNALED(status))
399         {
400             printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid),
401                 pid, WTERMSIG(status));
402             deletejob(jobs, pid);

```

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

```

402         continue;
403     }
404
405     if (WIFSTOPPED(status))
406     {
407         printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid,
408             WSTOPSIG(status));
409         struct job_t *job;
410         if ((job = getjobpid(jobs, pid)) == NULL)
411             app_error("job not found");
412
413         job->state = ST;
414         continue;
415     }
416
417     if (pid < 0 && errno != ECHILD)
418         unix_error("waitpid error");
419
420     return;
421 }
422
423 /*
424  * sigint_handler - The kernel sends a SIGINT to the shell whenever the
425  * user types ctrl-c at the keyboard. Catch it and send it along
426  * to the foreground job.
427  */
428 void sigint_handler(int sig)
429 {
430     pid_t pid;
431     if ((pid = fgpid(jobs)) > 0)
432         kill(-pid, SIGINT);
433
434     return;
435 }
436
437 /*
438  * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
439  * the user types ctrl-z at the keyboard. Catch it and suspend the
440  * foreground job by sending it a SIGTSTP.
441  */
442 void sigtstp_handler(int sig)
443 {
444     pid_t pid;
445     if ((pid = fgpid(jobs)) > 0)
446         kill(-pid, SIGTSTP);
447
448     return;
449 }
450
451 /* End signal handlers
452  */
453
454 /* Helper routines that manipulate the job list
455  */
456
457 /* clearjob - Clear the entries in a job struct */
458 void clearjob(struct job_t *job)
459 {
460     job->pid = 0;
461     job->jid = 0;
462     job->state = UNDEF;
463     job->cmdline[0] = '\0';
464 }
465
466 /* initjobs - Initialize the job list */
467 void initjobs(struct job_t *jobs)
468 {
469     int i;
470
471     for (i = 0; i < MAXJOBS; i++)
472         clearjob(&jobs[i]);
473 }

```

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

```

474 }
475
476 /* maxjid - Returns largest allocated job ID */
477 int maxjid(struct job_t *jobs)
478 {
479     int i, max = 0;
480
481     for (i = 0; i < MAXJOBS; i++)
482         if (jobs[i].jid > max)
483             max = jobs[i].jid;
484     return max;
485 }
486
487 /* addjob - Add a job to the job list */
488 int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline)
489 {
490     int i;
491
492     if (pid < 1)
493         return 0;
494
495     for (i = 0; i < MAXJOBS; i++)
496     {
497         if (jobs[i].pid == 0)
498         {
499             jobs[i].pid = pid;
500             jobs[i].state = state;
501             jobs[i].jid = nextjid++;
502             if (nextjid > MAXJOBS)
503                 nextjid = 1;
504             strcpy(jobs[i].cmdline, cmdline);
505             if (verbose)
506             {
507                 printf("Added job [%d] %d %s\n", jobs[i].jid, jobs[i].pid,
508                     jobs[i].cmdline);
509             }
510             return 1;
511         }
512     }
513     printf("Tried to create too many jobs\n");
514     return 0;
515 }
516
517 /* deletejob - Delete a job whose PID=pid from the job list */
518 int deletejob(struct job_t *jobs, pid_t pid)
519 {
520     int i;
521
522     if (pid < 1)
523         return 0;
524
525     for (i = 0; i < MAXJOBS; i++)
526     {
527         if (jobs[i].pid == pid)
528         {
529             clearjob(&jobs[i]);
530             nextjid = maxjid(jobs) + 1;
531             return 1;
532         }
533     }
534     return 0;
535 }
536
537 /* fgpid - Return PID of current foreground job, 0 if no such job */
538 pid_t fgpid(struct job_t *jobs)
539 {
540     int i;
541
542     for (i = 0; i < MAXJOBS; i++)
543         if (jobs[i].state == FG)
544             return jobs[i].pid;
545     return 0;
546 }

```



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

```

546
547 /* getjobpid - Find a job (by PID) on the job list */
548 struct job_t *getjobpid(struct job_t *jobs, pid_t pid)
549 {
550     int i;
551
552     if (pid < 1)
553         return NULL;
554     for (i = 0; i < MAXJOBS; i++)
555         if (jobs[i].pid == pid)
556             return &jobs[i];
557     return NULL;
558 }
559
560 /* getjobjid - Find a job (by JID) on the job list */
561 struct job_t *getjobjid(struct job_t *jobs, int jid)
562 {
563     int i;
564
565     if (jid < 1)
566         return NULL;
567     for (i = 0; i < MAXJOBS; i++)
568         if (jobs[i].jid == jid)
569             return &jobs[i];
570     return NULL;
571 }
572
573 /* pid2jid - Map process ID to job ID */
574 int pid2jid(pid_t pid)
575 {
576     int i;
577
578     if (pid < 1)
579         return 0;
580     for (i = 0; i < MAXJOBS; i++)
581         if (jobs[i].pid == pid)
582         {
583             return jobs[i].jid;
584         }
585     return 0;
586 }
587
588 /* listjobs - Print the job list */
589 void listjobs(struct job_t *jobs)
590 {
591     int i;
592
593     for (i = 0; i < MAXJOBS; i++)
594     {
595         if (jobs[i].pid != 0)
596         {
597             printf("[%d] (%d) ", jobs[i].jid, jobs[i].pid);
598             switch (jobs[i].state)
599             {
600             case BG:
601                 printf("Running ");
602                 break;
603             case FG:
604                 printf("Foreground ");
605                 break;
606             case ST:
607                 printf("Stopped ");
608                 break;
609             default:
610                 printf("listjobs: Internal error: job[%d].state=%d ",
611                     i, jobs[i].state);
612             }
613             printf("%s", jobs[i].cmdline);
614         }
615     }
616 }
617 /******
618 * end job list helper routines

```

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

```

619  *****/
620
621  *****
622  * Other helper routines
623  *****
624
625  /*
626  * usage - print a help message
627  */
628  void usage(void)
629  {
630      printf("Usage: shell [-hvp]\n");
631      printf("  -h   print this message\n");
632      printf("  -v   print additional diagnostic information\n");
633      printf("  -p   do not emit a command prompt\n");
634      exit(1);
635  }
636
637  /*
638  * unix_error - unix-style error routine
639  */
640  void unix_error(char *msg)
641  {
642      fprintf(stdout, "%s: %s\n", msg, strerror(errno));
643      exit(1);
644  }
645
646  /*
647  * app_error - application-style error routine
648  */
649  void app_error(char *msg)
650  {
651      fprintf(stdout, "%s\n", msg);
652      exit(1);
653  }
654
655  /*
656  * Signal - wrapper for the sigaction function
657  */
658  handler_t *Signal(int signum, handler_t *handler)
659  {
660      struct sigaction action, old_action;
661
662      action.sa_handler = handler;
663      sigemptyset(&action.sa_mask); /* block sigs of type being handled */
664      action.sa_flags = SA_RESTART; /* restart syscalls if possible */
665
666      if (sigaction(signum, &action, &old_action) < 0)
667          unix_error("Signal error");
668      return (old_action.sa_handler);
669  }
670
671  /*
672  * sigquit_handler - The driver program can gracefully terminate the
673  *   child shell by sending it a SIGQUIT signal.
674  */
675  void sigquit_handler(int sig)
676  {
677      printf("Terminating after receipt of SIGQUIT signal\n");
678      exit(1);
679  }

```

Listing 1: *tsh.c*