시스템 프로그래밍 과제 2 shlab 레포트

Shell Lab: Writing Your Own Unix Shell

2019-13674
양현서
바이오시스템소재학부

Contents

1 Introduction

수업 시간에 signal과 프로세스 제어 등을 배웠다. 프로세스의 실행 중에는 강제 종료나 일시 정지 등 여러 가지 특별한 상황이 발생할 수 있다. 그리고 시그널을 통해 프로그램의 실행 중 발생한 이러한 상황들을 처리할 수 있다. 이번 랩에서는 실제 Unix 셸처럼 하위 프로세스들을 실행하거나, bg, fg, job 명령과 `^Z`, `^I` 등 시그널 입력으로 그러한 프로세스들을 관리할 수 있고, 발생하는 시그널들에 제대로 반응하는 미니 셸을 만든다.

2 Step 1: 기본 뼈대 완성

Part 1에서는 명령어 입력을 처리하고 quit, bg, fg, job 등의 기초 커맨드를 처리하는 부분을 작성한다.

```
293 int builtin_cmd(char **argv)
294 {
295     char * cmd = argv[0];
296     if(strcmp(cmd, "quit") ==0) {
297         exit(0);
298     } else if (strcmp(cmd, "jobs") == 0) {
299         listjobs(jobs);
300     } else if (strcmp(cmd, "bg") == 0 || strcmp(cmd, "fg") == 0) {
301        do_bgfg(argv);
302     } else {
303        return 0;  /* not a builtin command */
304     }
305     return 1;    /* builtin command is processed */
306 }
```

`argv[0]`을 기준으로 명령을 판별하여 해당하는 기능들을 실행한다. builtin command의 경우는 `eval`에 커맨드가 처리되었음을 알리기 위해 1을 리턴, 아닌 경우에는 0을 리턴한다.

그다음은 이 부분의 핵심인 `eval`을 설명한다.

```
167 void eval(char *cmdline)
168 {
169     char * argv[MAXARGS];
170     int bg = parseline(cmdline, argv);
171     if(argv[0]==NULL) {
172        // printf("Empty\n");
173        return;
174     }
```

`parseline`을 호출하여 argv들을 얻어오고, 해당 명령이 background로 실행해야 하는지를 알아낸다. 인자로 아무 것도 들어오지 않으면 아무 것도 하지 않는다.

```
175     // process quit, jobs, bg or fg primarily
176     if(builtin_cmd(argv)) {
177         return;
178     }
```

명령이 builtin command라면 처리하고 바로 종료한다.

```
179     struct sigaction intsig;
180     // prepare signal mask
181     if(sigemptyset(&intsig.sa_mask) != 0 ) {
182         unix_error("sigemptyset failed");
183     }
184     if(sigaddset(&intsig.sa_mask, SIGCHLD) != 0) {
185         unix_error("sigaddset(SIGCHLD) failed");
186     }
187     if(sigaddset(&intsig.sa_mask, SIGINT) !=0) {
188         unix_error("sigaddset(SIGINT) failed");
189     }
190     if(sigaddset(&intsig.sa_mask, SIGTSTP) !=0) {
191         unix_error("sigaddset(SIGTSTP) failed");
192     }
193     // block signals before calling fork()
194     if(sigprocmask(SIG_BLOCK, &intsig.sa_mask, NULL) !=0) {
195         unix_error("sigprocmask(SIG_BLOCK) failed");
196     }
```

fork 도중 문제를 막기 위해 SIGCHLD, SIGINT, SIGTSTP을 블록한다. sigemptyset을 통해 구조체를 초기화하고, sigaddset을 통해 시그널 집합에 SIGCHLD, SIGINT, SIGTSTP을 등록한다. 그다음 sigprocmask를 이용하여 잠시 시그널들을 블록한 것이다.

```
197     int pid = fork();
198     if(pid <0) {
199         unix_error("fork() failed");
```

fork가 실패하면 오류를 출력하고 종료한다.

```
200     } else if(pid == 0) { // child
201         if(sigprocmask(SIG_UNBLOCK, &intsig.sa_mask, NULL)!=0) { // Unblock signal
202             unix_error("sigprocmask(SIG_UNBLOCK) failed");
203         }
204         // Put the child in a new process group whose group ID is identical to the child's PID.
205         // This ensures that there will be only one process, this shell,
206         // in the foreground process group.
207         if(setpgid(0,0) < 0) {
208             unix_error("setpgid(0,0) failed");
209         }
210         // Exceute the child program
211         if (execve(argv[0], argv, environ) < 0) { //returns -1 on error
212             printf("%s: Command not found.\n", argv[0]);
213             exit(0);
214         }
215         //should not reach here
```

fork의 리턴 값이 0인 경우 child process에서 실행중인 것이다. 우선 fork전에 블록했던 시그널을 언블록한 후, execve를 이용하여 대상을 실행시킨다. 성공적으로 수행하였다면 execve는 리턴하지 않는다. 그러므로 리턴한다면 에러를 출력하고 종료한다.

```
216     } else { // parent
217         if(bg) { // Child should run in background.
218             addjob(jobs, pid, BG, cmdline);
219             sigprocmask(SIG_UNBLOCK, &intsig.sa_mask, NULL); // Unblock signal
220             printf("[%d] (%d) %s",pid2jid(pid),pid,cmdline);
```

```
221             } else {
222                 addjob(jobs, pid, FG, cmdline);
223                 sigprocmask(SIG_UNBLOCK, &intsig.sa_mask, NULL); // Unblock signal
224                 waitfg(pid); // Wait for the child to finish
225                 return;
226             }
227         }
228         //    printf("%s", argv[0]);
229         return;
230     }
```

fork의 리턴 값이 양수인 경우 parent process에서 실행중인 것이다. 새로운 job을 리스트에 등록하고, fork전에 블록했던 시그널을 언블록한 후, bg 여부에 따라 waitfg를 호출하여 이번 job을 기다리거나, 표준 출력에 새로 생긴 background job에 대한 정보를 출력한 후 함수를 종료한다.

3 Step 2: bg와 fg 명령 처리 실제 구현

셸에서, bg/fg [인자] 를 입력하면 해당 인자(pid 또는 jobid)에 해당하는 job을 찾아 background/foreground 작업으로 변경해 재생시킨다.

```
311 void do_bgfg(char **argv)
312 {
313     struct job_t * job;
314     if(argv[1] == NULL) {
315        printf("%s command requires PID or %%jobid argument\n", argv[0]);
316        return;
317     }
318     // Handle arguments
319     // Case 1 : PID provided
320     if(isdigit(argv[1][0])) {
321         long pid = strtol(&argv[1][0], NULL, 10);
322         if(pid < 0) {
323                 printf("%s: argument must be a PID or %%jobid\n", argv[0]);
324                 return;
325         }
326         job = getjobpid(jobs, pid);
327         if(job == NULL) {
328                 printf("(%s): No such process\n", argv[1]);
329                 return;
330         }
331     // Case 2: Job ID provided
332     } else if(argv[1][0] == '%') {
333         long jid = strtol(&argv[1][1], NULL, 10);
334         if(jid <=0) {
335                 printf("%s: No such job\n", argv[1]);
336                 return;
337         }
338         job = getjobjid(jobs, jid);
339         if(job == NULL) {
340                 printf("%s: No such job\n", argv[1]);
341                 return;
342         }
343     } else {
344         printf("%s: argument must be a PID or %%jobid\n", argv[0]);
345         return;
346     }
```

bg/fg 명령의 인자의 첫 글자가 '%'인지에 따라 %라면 job id, %가 아니라면 pid로 해석하여 strtol 함수로 정수형으로 변환한 뒤, 이것을 키로 이용하여 해당하는 struct job_t * 포인터를 얻어온다. 실패할 경우 그에 해당하는 오류 메시지를 출력한다.

```
347     if(strcmp(argv[0], "bg") == 0) {
348         int pid = job -> pid;
349         if(kill(-pid, SIGCONT)<0) { // Continue the child process in background
350             unix_error("kill failed\n");
351         }
352         job -> state = BG;
353         printf("[%d] (%d) %s", job->jid, pid, job->cmdline);
354         return;
355     } else if(strcmp(argv[0], "fg") == 0) {
356         int pid = job ->pid;
357         if(kill(-pid, SIGCONT)<0) { // Continue the child process
358             unix_error("kill failed\n");
359         }
360         job -> state = FG;
361         waitfg(pid); // Wait for the child process to finish
362         return;
363     } else {
364         printf("%s: Command not found\n", argv[0]);
365         return;
366     }
367     return;
368 }
```

이어 명령이 `bg` 라면 job의 상태를 `BG`, `fg`라면 `FG`로 바꾼다. 그리고 bg, fg 명령이 들어왔다는 것은 이미 그 job이 background로 돌고 있거나 멈춰 있다는 것이므로 작업을 재개하기 위해 `kill` 함수를 통해 해당 job의 pid 그룹에 대해 `SIGCONT` 시그널을 보낸다. 그 다음 명령이 bg라면 화면에 background job에 대한 정보를 출력하고 fg라면 `waitfg` 함수를 호출하여 foreground job의 종료를 기다린다.

4 Step 3: 시그널 처리와 waitfg 구현

이번에는 셸에서 생성한 자식 프로세스에게 시그널을 제대로 전달하거나 자식 프로세스가 종료될 때까지 대기하는 매커니즘을 작성한다. 과제 handout에 힌트가 제시되어 있어서 도움이 되었다. 자식 프로세스가 종료되면 부모 프로세스에게 `SIGCHLD` 시그널이 전달된다. 이것을 이용하여 `sigchld_handler`를 만들고, `waitfg`를 구현할 수 있다.

```
388 /*
389  * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
390  *     a child job terminates (becomes a zombie), or stops because it
391  *     received a SIGSTOP or SIGTSTP signal. The handler reaps all
392  *     available zombie children, but doesn't wait for any other
393  *     currently running children to terminate.
394  */
395 void sigchld_handler(int sig)
396 {
397     int status;
398     int pid;
399     struct job_t * job;
400     while((pid=waitpid(-1,&status,WNOHANG|WUNTRACED))>0) { //returns pid of child
        ↪ if OK,0 or -1 on error
401         job = getjobpid(jobs,pid);
402         if(WIFEXITED(status)) {
403             deletejob(jobs,pid); // delete the job if the job exited normally with
                ↪ exit()
404         }
405         if(WIFSIGNALED(status)) { // returns nonzero if the child terminated
            ↪ because it received a signal which was not handled
406             printf("Job [%d] (%d) terminated by signal 2\n",job->jid,job->pid);
407             deletejob(jobs,pid); //delete job terminated by SIGINT
408         }
409         if(WIFSTOPPED(status)) { // the child is stopped
```

```
410             printf("Job [%d] (%d) stopped by signal 20\n",job->jid, job->pid);
411             job->state = ST; // change the state to STOP
412         }
413     }
414     return;
415 }
```

SIGCHLD 시그널을 받았을 때 실행되는 함수 sigchld_handler에서는 모든 자식 프로세스들에 대해 waitpid를 수행하는데, WNOHANG 옵션을 이용하여 당장 처리할 자식 프로세스가 더 없으면 0을 리턴받아 루프를 종료하게 하였다. 또한 waitpid 함수의 설명에 따르면 이 함수로 자식 프로세스들을 reap 하여 좀비 프로세스들을 처리할 수도 있다고 하니 좀비 자식 프로세스들을 처리할 수 있다. WUNTRACED를 이용하여 해당 자식 프로세스의 상태가 stopped인지 판단할 수 있는 매크로 WIFSTOPPED(status)을 이용할 수 있었다. 위와 같이 sigchld_handler에서 foreground job의 state를 ST(정지) 또는 UNDEF(deletejob -> clearjob)으로 바꿔주게 설계하고, 아래와 같이 waitfg를 구현하였다.

```
370 /*
371  * waitfg - Block until process pid is no longer the foreground process
372  */
373 void waitfg(pid_t pid)
374 {
375     struct job_t * j = getjobpid(pid);
376     if(!j) // Invalid pid
377         return;
378     while(j -> state == FG) { // The job's state flag will be modified in signal
      ↪  handlers
379         sleep(1); // Wait 1 sec
380     }
381     return;
382 }
```

pid에서 해당하는 job의 구조체를 얻어오고, 해당 job의 state가 FG인 동안 1초씩 대기한다. 이렇게 하면 해당 job이 kill되거나 fg가 아닌 경우가 될 때까지 대기하는 것이 가능해진다.

다음은 나머지 시그널들의 핸들러이다. foregrund에 있는 프로세스 그룹들에 해당 시그널을 전달해 준다.

```
417 /*
418  * sigint_handler - The kernel sends a SIGINT to the shell whenver the
419  *    user types ctrl-c at the keyboard.  Catch it and send it along
420  *    to the foreground job.
421  */
422 void sigint_handler(int sig)
423 {
424     pid_t pid = fgpid(jobs); // get the pid of the fg job
425     if(pid > 0) {
426         if(kill(-pid, SIGINT) < 0) { //send SIGINT to the fg job group
427             //printf("kill(SIGINT failed");
428         }
429     }
430     return;
431 }
432
433 /*
434  * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
435  *     the user types ctrl-z at the keyboard. Catch it and suspend the
436  *     foreground job by sending it a SIGTSTP.
437  */
438 void sigtstp_handler(int sig)
439 {
440     pid_t pid = fgpid(jobs); // get the pid of fg job
441     if(pid > 0) {
442         kill(-pid, SIGTSTP); //send SIGSTP signal to the fg job group
443     }
```

```
444     return;
445 }
```

SIGINT와 SIGTSTP 핸들러는 현재 foreground job으로 관리하고 있는 job의 프로세스 그룹에게만 해당 시그널을 전달해 준다. 앞의 setpgid를 이용하여 이 셸에게만 시그널이 전달되게 하고, 그것을 내부적으로 관리한 foreground process에만 전달하는 것이다. kill에 pid를 주면 그 프로세스에만 전달되므로 -pid를 주어 그 프로세스와 그 프로세스의 자식 프로세스들에까지 전달되게 하였다.

5 Conclusion

5.1 어려웠던 점

개발 방향 탐색

지난번 linklab에서는 step 1, 2, 3을 단계적으로 해결하면 결과물이 완성되는 것이었지만 이번 shlab은 그런 단계 구분은 없고 뼈대 코드를 이용하여 알아서 설계하고 완성하는 것이라 조금 더 어려웠다.

5.2 놀라웠던 점

간단함

생각해 주어야 할 부분만 잘 처리해 주니 생각보다 구현할 내용이 많지 않아서 놀라웠다.

SIGSTP과 SIGTSTP의 차이

SIGSTP은 프로그램에서 발생시키는 것이고, SIGTSTP은 터미널에서 발생시키는 것이다. 또한 SIGTSTP은 무시할 수 있지만, SIGSTP은 무시할 수 없다고 한다.