Lab 4: Shell Lab Report

2018-13947 Woorim Shin

0. Overview

이번 shell lab 은 아래와 같은 기능을 제공한다.

- Background & Foreground
  작업이 다 terminate 할 때까지 기다리는 foreground 작업과 기다리지 않는 background 작업을 모두 수행한다. 둘의 구별은 comman 마지막에 '&' 표시로 이뤄지고, 스켈레톤 코드로 제공되는 parse_line 에서 그 둘을 구별해준다. Background 작업은 한번에 여러개일 수 있고, foreground 작업은 한번에 최대 1 개만 running 할 수 있다.
- Signal handling
  SIGINT 와 SIGSTP, 그리고 SIGCHLD singal 을 handling 할 수 있어야 한다. 이를 위해 handler 함수를 별도로 구현한다.
- I/O redirection
  '>' 기호를 통해 output redirection 을 지원한다.
- Pipe
  'ls | sort' 처럼 pipe 를 통해 다수의 명령어를 한 command 로 처리할 수 있어야 한다.
- 4 built-in commands
  4 개의 빌트인 command 를 지원해야 한다. Quit 을 입력하면 쉘을 종료하고, jobs 를 입력하면 현재 모든 background 작업을 나열하고, bg 는 SIGCONT 를 통해 현재 정지된 프로세스를 백그라운드에서 다시 돌린다. Fg 는 foregroung 로 돌린다.

1. Brief descriptions of the implementation

이번 구현을 위해 총 7 개의 function 을 구현하였다. 각각에 대한 간단한 설명은 아래와 같다.

```
int builtin_cmd(char *(*argv)[MAXARGS] )
{
  if(strcmp(argv[0][0], "quit") == 0) {

    exit(0);
    return 1;
  } else if (strcmp(argv[0][0], "bg") == 0) {
    do_bgfg(argv);
    return 1;
  } else if (strcmp(argv[0][0], "fg") == 0) {
    do_bgfg(argv);
    return 1;
  } else if (strcmp(argv[0][0], "jobs") == 0) {
    listjobs(jobs);
    fflush(stdout);
    return 1;
  } else return 0;
}
```

builtin_cmd 는 4 개의 빌트인 커맨드인 경우 1 을 반환하고, 아닐 경우 0 을 반환하며, 4 개의 빌트인 커맨드를 실행할 수 있도록 한다. 이 리턴값은 eval 에서 fork 를 할지 말지를 결정하는 데 사용된다.

```
// for block sigchild
Sigemptyset(&mask_sigchld);
Sigaddset(&mask_sigchld, SIGCHLD);


// mask SIGCHLD
Sigprocmask(SIG_BLOCK, &mask_sigchld, &prev_one);
// send SIGCONT signal
Kill(-(job->pid), SIGCONT);

// bg : Stopped background job -> Running background job
// fg : Backgroung job -> Running foreground job
if (is_bg){
  printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
  job->state = BG;
} else {
  job->state = FG;
}
Sigprocmask(SIG_SETMASK, &prev_one, NULL);

if(!is_bg){
  waitfg(job->pid);
}
```

do_bgfg 는 이어서 bg 와 fg 를 실행할 수 있게 한다. 미리 구현된 함수를 통해 jobid 로 job 을 얻고, 그 job 에 SIGCONT signal 을 보낸다. 그 후 명령어의 종류에 맞게 작업의 state 를 변경해준다. Kill 을 통해 Signal 을 보낼 땐 pid 에 (-)기호를 붙여 그 그룹 전체에 보내도록 하고, sigprocmask 를 이용해 도중에 SIGCHLD signal 을 받는 것을 방지한다.

```
void waitfg(pid_t pid)
{
  while(1){
    sleep(1);
    if (fgpid(jobs) != pid) break;
  }
  return;
}
```

```
Sigfillset(&mask_all);
while((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {

  // with status, we can check the exit condition of reaped child
  // If exited nomally by exit call or return
  if (WIFEXITED(status)){
    // block all signal and delete job
    Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
    deletejob(jobs, pid);
```

아래는 handout 에 제공된 hint 중 하나이다. 이 hint 를 참고하여 waitfg 와 sigchld_handler 를 구현하였다. Sigchld_handler 는 waitpid 를 while 문 안에 넣어 signal 을 놓치는 일이 없도록 하였다. 기타 구현에 있어서 상세한 부분은 교과서 8 장의 내용을 참고하였다.

One of the tricky parts of the assignment is deciding on the allocation of work between the waitfg and sigchld handler functions. We recommend the following approach:

- In waitfg, use a busy loop around the sleep function.
- In sigchld handler, use exactly one call to waitpid.

While other solutions are possible, such as calling waitpid in both waitfg and sigchld handler,

these can be very confusing. It is simpler to do all reaping in the handler.
The WUNTRACED and WNOHANG options to waitpid will also be useful.

```c
void sigint_handler(int sig)
{
  // save errno for it not to be modified.
  int olderrno = errno;

  // find foreground job and send SIGINT by kill function
  pid_t pid = fgpid(jobs);
  if (pid == 0) {
    return;
  }
  Kill(-pid, SIGINT);

  //restore it
  errno = olderrno;

  return;
}
```

```c
void sigtstp_handler(int sig)
{
  // save errno
  int olderrno = errno;

  // find foreground job
  pid_t pid = fgpid(jobs);
  if (pid == 0){
    return;
  }

  Kill(-pid, SIGTSTP);

  // for not to be interrupted by all signal
  sigset_t mask_all, prev_all;
  Sigfillset(&mask_all);
  Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);

  // make job state ST
  struct job_t *fgjob = getjobpid(jobs, pid);
  fgjob->state = ST;

  // restore block bit vector and errno
  Sigprocmask(SIG_SETMASK, &prev_all, NULL);
  errno = olderrno;
  return;
}
```

Sigint_handler 는 ctl+C 로 주게 되는 SIGINT signal 의 handler 함수이고, Sigtstp_handler 는 ctrl+Z 로 주게 되는 SIGTSTP 의 handler 함수이다. Stop signald 의 경우 stop 시킨 job 을 찾아 그 state 를 ST 로 변경해주어야 한다.

```c
redic = strchr(cmdline, '>');
// if command has redirection mark(<), we redirect output with dup2.
// now redic is ptr for file name, cmdline only contain command.
if ((redic != NULL) && (*(redic-1)==' ') && (*(redic+1)==' ') ){
*redic = '\0'; redic++;
while( (*redic) && (*redic ==' ')) redic++;
temp = strchr(redic, '\n');
*temp = '\0';
is_redirect = 1;
}

if(is_redirect == 1){
// open file and redirect
filedescriptor = open(redic, O_RDWR|O_CREAT|O_TRUNC, S_IRUSR | S_IRGRP | S_IWGRP | S_IWUSR |S_IROTH | S_IWOTH);
if(filedescriptor < 0) unix_error("failed to open file\n");
if(dup2(filedescriptor, STDOUT_FILENO) < 0) unix_error("We failed to redirect file\n");
close(filedescriptor);
}
```

Eval 의 I/O redirection 구현은 다음과 같이 이루어진다. Strchr 를 통해 꺽쇠 기호의 pointer 를 얻게 된다면 그 pointer 가 가리키는 주소값을 기준으로 이전 command 와 이후 file name 을 구분할 수 있게 되며, dup2 를 통해 표준 출력의 파일식별자에 file name 의 파일식별자를 덮어씌운다.

```
 //pipe issue
// give output
 if(i!=rpipec-1){
   if(dup2(pipefd[i][WRITE],STDOUT_FILENO) < 0){
     printf ("i = %d\n", i);
     unix_error("failed to give output");
   }
 }

 //take input
 if(i!=0){
   if(dup2(pipefd[i-1][READ],STDIN_FILENO) < 0) {
     printf ("i = %d\n", i);
     unix_error("failed to take input");
   }
 }

for(int j = 0; j<rpipec; j++){
 close(pipefd[j][WRITE]);
 close(pipefd[j][READ]);
}
```

파이프의 구현은 이와 같다. Fork() 함수는 for 문을 통해 파이프 개수만큼 반복된다. For 문 안에서 파이프의 WRITE 와 READ 를 줄줄이 이어 파이프로 명령어들이 연결되게 한다. 이후 execvp 를 통해 path search 도 필요없이 명령어를 수행할 수 있다. 여기서 close 를 적절히 하는 것이 중요한다. Parent 부분에 경우 다음 i 에서 사용되지 않을 예정인 end 를 닫아두니 정상적으로 실행이 되었다.

```
// parent
Sigprocmask(SIG_BLOCK, &mask_all, NULL);
addjob(jobs, pid, is_bg+1, cmdline);
Sigprocmask(SIG_SETMASK, &prev_one, NULL);
```

아래 또한 handout 이 제공하는 hint 중 하나이다.
In eval, the parent must use sigprocmask to block SIGCHLD signals before it forks the child, and then unblock these signals, again using sigprocmask after it adds the child to the job list by calling addjob. Since children inherit the blocked vectors of their parents, the child must be sure to then unblock SIGCHLD signals before it execs the new program.
The parent needs to block the SIGCHLD signals in this way in order to avoid the race condition where the child is reaped by sigchld handler (and thus removed from the job list) before the parent calls addjob.
이 방법을 신실하게 구현함으로써 race condition 의 위험에서 벗어나게 되었다.

실제로 trace file 을 통해 정상적인 작동이 되는 것을 확인하였고, 다른 더 복잡한 case 들도 잘 작동되는 것을 확인할 수 있었다. 아래는 그 예시이다.

2. Difficulties during the implementation

1. 포인터 계산
   오랜만에 C 프로그래밍을 하는 것이다 보니, 포인터를 다루는 것이 무척 까다로웠다. 특히 이차원 문자열 배열을 처리하면서 포인터 연산자를 어떻게 사용해야 하는 것인지가 약간 헷갈렸던 것 같다.

2. 파이프 close end
   fork()의 과정이 너무 복잡해서, 어떻게 해야 모든 사용하지 않는 파이프를 닫을 수 있을지를 찾아내는 과정에서 시간이 많이 소요되었다. Parents 에서는 앞으로 (i 가 증가하면) 사용하게 되지 않을 파이프들을 close 했고, child 에서는 전부 닫았다.

3. 다른 프로세서에서 수신하는 signal handling (trace16)
   터미널에서 보내는 시그널은 제대로 캐치를 하는데, 다른 프로세스에서 kill 함수로 보내는 시그널은 제대로 잡고 있지 못하는 문제가 있었다. 디버깅을 해본 결과 SIGTSTP 를 발신했을 때, 핸들러 함수가 아예 실행이 안되고 있다. man exeve 에 따르면, exeve 를 실행하면 signal 관련된 것들이 모두 default 로 바뀌게 되는데 아마 이게 원인일 것 같다. SIGINT 에 경우는 어짜피 핸들링 함수가 하는 일이 kill 함수로 시그널을 보내는 것이었기 때문에 별 문제가 되지 않았지만, SIGTSTP 의 경우 job 을 가져와서 그 state 를 ST 로 바꾸는 작업이 handling 함수에서 일어났었기 때문에, 이걸 따로 해줄 필요가

있었다. 그래서 sig_chld handling 에서 그 작업을 할 수 있도록 수정하니 정상적으로 작동되었다.

```
devel@csapvm ~/work/lab-4-shell-lab $ make test16
./sdriver.pl -t trace/trace16.txt -s ./tsh -a "-p"
#
# trace16.txt - Tests whether the shell can handle SIGTSTP and SIGINT
#     signals that come from other processes instead of the terminal.
#
tsh> ./mystop 2
Job [1] (6368) stopped by signal 20
tsh> jobs
[1] (6368) Stopped ./mystop 2
tsh> ./myint 2
Job [2] (6371) terminated by signal 2
devel@csapvm ~/work/lab-4-shell-lab $
```

감격스러운 Test 성공 순간이다.
아래는 수정된 sigchld_handling 함수의 일부이다.

```
// if the child is stopped
if (WIFSTOPPED(status)) {
  printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(status));
  fflush(stdout);

  // if signal is sent by other process, not terminal, we need additional works for editing job list
  struct job_t *fgjob = getjobpid(jobs, pid);
  Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
  if(fgjob->state != ST){
    fgjob->state = ST;
  }
  Sigprocmask(SIG_SETMASK, &prev_all, NULL);
}
```