System Programming Shell Lab

2016-13692 Seulgi, Kim

1. 전체 FlowChart에 대한 설명

Main함수에서는 먼저 stderr을 stdout으로 redirect하여 출력을 모두 stdout으로 연결되게끔 dup2 함수를 구현하였다. 또한 option을 주었다. 따라서 우리는 이에 맞추어 help, verbose, prompt 에 대해 각각 다른 기능을 수행하게끔 하였다.

이후 signal handler를 설치하였다. 따라서 우리는 이에 맞추어 각 시그널에 따라 서로 다른 동작을 하는 시그널 핸들러를 구현하였다.

이후 eval 함수를 호출하여 명령어를 실행하도록 하였다.

Eval 함수 구현 내용을 살펴보자. 먼저 shell 에서 signal을 보낼 때 child process의 프로그램이 영향을 받지 않게끔 unique한 process group ID를 부여하였다.

```
//if SIGINT, SIGTSTP, use different process group id from shell
//so that background child process do not receive the signal
//alter pgid using its own pid
if(setpgid(0,0)<0) perror("cannot alter current process's pgid to its own pid");
```

Figure 1 eval 함수, set unique process group id

다음으로 CommandLine으로는 크게 세 가지 경우의 수가 들어올 수 있다.

첫째, 아무것도 들어오지 않을 경우(NULL), 예를 들어 사용자가 tsh shell에서 enter를 칠 경우, tiny shell은 즉시 리턴한다.

둘째, builtin Command가 실행될 경우 tsh shell은 해당 command를 즉시 실행한 후 리턴한다.

셋째, 다른 실행 파일 경로의 이름일 경우, fork()를 통해 child processor를 생성하고, child processor에서 execve에 해당하는 프로그램을 실행시킨다. 이 때, 맨 끝에 &가 들어올 경우 해당 Command는 Background에서 실행되며, 이외에는 Foreground에서 실행된다.

Commandline으로 들어온 명령어를 적절히 파싱하여 Redirection과 Pipe command('|') 또한 구현하였다.

2. 함수별 문제 상황 및 해결 방안

< race problem 문제 상황 >

Waitfg 함수는 인자로 들어온 process id가 foreground process가 더 이상 아닐 때까지 block해주는 역할을 하는 함수이다.

시그널을 보낼 때 parent process에서 addjob을 하기 전에 child processor에서 deletejob이 호출될 수 있다(race problem). 따라서 waitfg() 함수에서는 실행되고 있는 foreground job이 종료될 때까지 기다리는 역할을 한다.

<해결 방안 : waitfg(), eval()>

이를 while(j->pid == pid 이고 j->state가 FG인 동안) sleep하는 라인을 생성하여 해결하였다. 또한 eval함수에서는 SIGCHLD 시그널을 먼저 블록한 후, fork를 호출한 다음에 블록한 SIGCHLD 시그널을 해제하여 addjob후에 deletejob이 실행되도록 하였다. (이 때 fork에서 blocked된 시그널까지 이어받은 child processor는 fork 호출 후 execve 호출 전까지 SIGCHLD 시그널을 unblock해준다.)

```
/*
 * waitfg - Block until process pid is no longer the foreground process
 */
void waitfg(pid_t pid)
{
  struct job_t *j = getjobpid(jobs, pid);
  if(!j) return;
--
  while(j->pid==pid && j-> state==FG) sleep(1);
  if(verbose){
    printf("waitfg : Process (%d) no longer the fg process \n", pid);
  }
  return;
}
```

Figure 2 Waitfg 함수, child process가 foreground process인 한 멈춰 있는다.

```
  //block SIGCHLD handler
  if(sigemptyset(&set)!=0) perror("signal set not empty");
  if(sigaddset(&set, SIGCHLD) != 0) perror("SIGCHLD hasn't added");
  if(sigaddset(&set, SIGINT) != 0) perror("cannot add SIGINT");
  if(sigaddset(&set, SIGTSTP) != 0) perror("cannot add sigtstp");
  if(sigprocmask(SIG_BLOCK, &set, NULL) !=0) perror("cannot block SIGCHLD");
---
  if((pid[i]=fork()) == 0){
    //unblock SIGCHLD handler
    if(sigprocmask(SIG_UNBLOCK, &set, NULL)!=0) perror("cannot unblock SIGCHLD")
```

Figure 3 child process에서 부모에서 block한 SIGCHLD handler를 unblocking 해주었다.

```
      for (int i=0;i<pipec;i++){
      if(background){
        //add current process to job list
        addjob(jobs, pid[i], BG, *argv[i]);//pid[pipec-1]
      }
      else{//background
        //add current process to job list
        addjob(jobs, pid[i], FG, *argv[i]);//pid[pipec-1]
      }
      //unblock SIGCHLD handler
      if(sigprocmask(SIG_UNBLOCK, &set, NULL)!=0)
        perror("cannot unblock SIGCHLD");
------
```

Figure 4 parent process에서 addjob이 모두 수행되고 난후, SIGCHLD handler를 unblock한다.

<서로 다른 동작을 하는 핸들러 함수 구현에 대한 문제 상황>

SIGINT handler함수는 Ctrl+C를 보내면 핸들러로 처리해야 한다.

SIGTSTP handler 함수는 Ctral+Z를 보내면 핸들러로 처리해야 한다.

Eval 함수에서는 시그널들의 집합을 맨 처음에 add할 때, sigaddset함수를 이용하여 SIGTSTP과 SIGINT를 set에 추가해야 한다.

SIGCHLD handler 함수는 Child processor가 종료되었을 때 부모 프로세서에게 온 SIGCHLD 신호를 처리해야 한다.

<해결 방안: sigint_handler(), sigtstp_handler(), eval(), sigchld_handler()>

먼저 SIGINT handler 함수에서는 현재 실행되고 있는 프로세스의 정보를 fgpid를 통해 받아, 해당 프로세스 그룹의 모든 프로세스(pid<0, 자신 포함)에 kill로 SIGINT를 보내어 프로세스를 바로 종료시키도록 했다. 여기에서 fgpid를 사용하여 프로세스의 정보를 받은 이유는 SIGINT가 foreground에서만 작동하도록 하기 위함이다. 이는 다음 그림과 같다.

```
/*
 * sigint_handler - The kernel sends a SIGINT to the shell whenver the
 *    user types ctrl-c at the keyboard.  Catch it and send it along
 *    to the foreground job.
 */

void sigint_handler(int sig)
{
  pid_t pid;
  //return pid number of current foreground
  pid = fgpid(jobs);
  if(verbose) printf("sigint_handler : entering\n");
  if(pid>0){
    kill(-pid, SIGINT);//send sigint to every process within the process group
    if(verbose) printf("sigint handler : Job (%d) killed\n", pid);
  }
  if(verbose) printf("sigint_handler: exiting\n");
  return;
}
```

Figure 5 SIGINT handler, kill 대상의 pid가 음수임에 유의

SIGTSTP handler 함수에서는 Ctrl + Z 가 입력되었을 때 현재 실행되고 있는 프로세스의 정보를 fgpid를 통해 받아, 해당 프로세스 그룹의 모든 프로세스(자신 포함)에 kill 함수를 이용하여 시그

널을 보낸다. 여기서도 fgpid를 받은 이유는 foreground에서만 SIGTSTP가 동작하도록 하기 위함이다. (만약 해당하는 job을 fgpid에서 못 잡으면 0을 리턴하게 된다).

```
/*
 * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
 *     the user types ctrl-z at the keyboard. Catch it and suspend the
 *     foreground job by sending it a SIGTSTP.
 */
//
void sigtstp_handler(int sig)
{
  //deletejob
  pid_t pid;
  pid = fgpid(jobs);//current process value
  if(verbose) printf("sigtstp_hanlder: entering\n");
  if(pid>0){
    kill(-pid, SIGTSTP);//send sigtstp to itself
    if(verbose) printf("sigtstp_hanlder: Job[%d] (%d) stopped\n", pid2jid(pid), pid);
  }
  if(verbose) printf("sigtstp_handler: exiting\n");
  return;
}
```

Figure 6 SIGTSTP handler, kill 대상의 pid가 음수임에 유의

이렇게 구현한 SIGINT, SIGTSTP을 eval 함수에서 sigaddset을 사용하여 추가하였다.

```
if(sigemptyset(&set)!=0) perror("signal set not empty");
if(sigaddset(&set, SIGCHLD) != 0) perror("SIGCHLD hasn't added");
if(sigaddset(&set, SIGINT) != 0) perror("cannot add SIGINT");
if(sigaddset(&set, SIGTSTP) != 0) perror("cannot add sigtstp");
if(sigprocmask(SIG_BLOCK, &set, NULL) !=0) perror("cannot block SIGCHLD");
```

Figure 7 eval 함수의 signal add 부분 구현

다음으로 SIGCHLD handler 함수는 다음 그림 3과 같이 구현하였다.

```
/*
 * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
 *     a child job terminates (becomes a zombie), or stops because it
 *     received a SIGSTOP or SIGTSTP signal. The handler reaps all
 *     available zombie children, but doesn't wait for any other
 *     currently running children to terminate.
 */

void sigchld_handler(int sig)
{
  int status;
  pid_t pid;

  //wait child process terminate
  //return terminated process id
  while((pid = waitpid(-1, &status, WNOHANG|WUNTRACED))>0){
```

Figure 8 SIGCHLD handler – all signal received

먼저 while문을 쓴 이유는 펜딩 시그널이 있을 경우 이를 모두 수신하기 위함이다. 또한 waitpid의 인자로 WNOHANG|WUNTRACED를 사용하였는데, 그 이유는 대기 집합에 있는 모든 child

processes들이 정지하거나 종료했을 때 즉시 리턴하기 위함이다.

또한 프로세스가 Signal에 의해 Terminated 되었을 경우, 그 경우를 3가지로 나누었다. 먼저 child process가 exit되었을(WIFSIGNALED) 경우, 상황을 STDOUT에 알린 후 joblist에서 job을 삭제한다. 둘째, child process가 멈추었을 경우(SIGTSTP signal을 받고 정지했을 경우), job의 state를 ST로 바꾼 후 상황을 STDOUT에 알린다. 마지막으로 child process가 정상적으로 종료되었을 경우, joblist에서 job을 삭제한다.

이렇게 하면 SIGCHLD 핸들러에서 background에서 돌던 job을 종료시키는 기능까지 추가되게 된다. 즉, 프로세스가 정상적으로(WIFEXITED)/혹은 비정상적으로 종료되었을 때 해당하는 child process를 삭제하여 zombie process가 되지 않도록 구현한다.

```
    struct job_t *j = getjobpid(jobs, pid);
    //if child process terminated by signal
    if(WIFSIGNALED(status)!=0){//if child process exit
      fprintf(stdout, "Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid
          ,WTERMSIG(status));
      //delete job from joblist
      deletejob(jobs, pid);
    }
    //if child has stopped
    else if(WIFSTOPPED(status)){
      //change job state
      j->state = ST;
      fprintf(stdout, "Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(status));
    }
    //if process terminates normally,
    else if(WIFEXITED(status)){
      //delete that process so that not making zombie
      deletejob(jobs, pid);
    }
  }
  return;
}
```

Figure 9 SIGCHLD handler – changing job states

<Foreground/Background job 구분 및 처리 에 대한 문제상황>

Builtin command 에는 bg 로 시작하는 명령어, fg 로 시작하는 명령어가 들어올 수 있다. 이 둘의 명령어에 대한 처리는 다른데, 이때 bg 가 첫 명령어로 쓰이는지, fg 가 첫 명령어로 쓰이는지에 따라 do_bgfg 함수에서 처리해주어야 한다.

<해결 방안 : builtin_cmd(), do_bgfg()>

먼저 builtin_cmd 함수를 보자.

```
int builtin_cmd(char *(*argv)[MAXARGS] )
{
  //quit
  //printf("%c, %c", *argv[0][0], *argv[0][1]);
  if(!strcmp(*argv[0], "quit")) exit(0);
  else if(!strcmp(*argv[0], "&")) return 1;
  //bg -> call do_bgfg()
  else if(!strcmp(*argv[0], "bg") || !strcmp(*argv[0], "fg")){
    do_bgfg(argv);
    return 1;
  }
```

Figure 10 builtin_cmd 함수에서 첫번째 명령어가 bg 이거나 fg이면 do_bgfg를 실행한다.

먼저 백그라운드 job에 관련된 내용이다.

Do_bgfg 함수에서는 명령어로 bg + % + job id가 들어오는지, 명령어로 bg + pid가 들어오는지에 따라 다르게 argv vector를 파싱하여 명령어에 해당하는 job을 찾는다. Job ID 혹은 PID에 해당하는 job을 getjobjid or getjobpid 함수를 통해 찾는 기능을 각 경우마다 구현하였다.

```
//check atoi(argv[1]) to exclude jid, pid
if(argv[0][1][0]=='%'){
  char *p = &(argv[0][1][1]);
  jid = atoi(p);
  job = getjobjid(jobs, jid);
  if(job==NULL){
    printf("%s: No such job\n", argv[0][1]);
    return;
  }
}
```

Figure 11 명령어로 bg + % + job id 가 들어오는 경우

```
else if(isdigit(&(argv[0][1][0]))){
  pid = atoi(&(argv[0][1][1]));
  job = getjobpid(jobs, pid);
  if(job==NULL){
    printf("(%d): No such process\n", pid);
  }
}
else{
  printf("%s: argument must be a PID or %%jobid\n", flag? "fg": "bg");
  return;
}
```

Figure 12 명령어로 bg + pid가 들어오는 경우

```
kill(-(job->pid), SIGCONT);
```

이렇게 job을 찾았다면, kill을 이용하여 SIGCONT 시그널을 보낸 후 state를 바꾸게 되면 background에서 다시 실행되게 된다.

```
else{
  job->state = BG;
  printf("[%d](%d) %s", job->jid, job->pid, job->cmdline);
}
```

Figure 13 background process임을 알리기 위해 state를 바꾼다.

다음으로 Foreground process를 살펴보자. 먼저 fg + % + job id가 명령어로 들어왔는지, fg + pid가 명령어로 들어왔는지, 그 여부에 따라 명령어를 파싱하여 job을 찾는 과정은 그림 11, 12와 동일한 과정을 거친다. 이렇게 해당하는 job을 찾았으면, job의 state를 FG로 바꿔준다는 점과, foreground에서 실행이 되므로 waitfg함수를 이용하여 foreground에서 돌고 있었던 child process들이 종료되기 기다린다는 점이 background와 다르다.

```
kill(-(job->pid), SIGCONT);
if(flag){
  job->state =FG;
  waitfg(job->pid);
}
```

Figure 14 Foreground process의 루틴

<Multi Pipe 구현 시 문제 상황>

커맨드라인 하나가 여러 개의 파이프로 이루어진 커맨드일 수 있다. 따라서 child process 하나를 fork하는 것으로는 모자라며, pipe의 개수에 맞게 fork와 pipe making이 이루어져야 한다.

<해결 방안 : eval()>

```
for(int i=0;i<(pipec-1);i++){
  if(pipe(fileds[i])<0) perror("cannot make pipe");
}
```

Figure 15 먼저 파이프의 개수에 맞게 pipe를 install 하는 과정이다.

Child process가 n개이면, 이들을 잇는 파이프는 n-1개 만들어진다. 이에 맞게 파이프를 먼저 install해 주었다.

만약 첫번째로 들어온 child process라면 STDIN_FILENO로부터 input을 받는다. 또한 마지막으로 나갈 child process라면 STDOUT_FILENO로 redirect시켜주어야 한다. 중간에 있는 child process라면 파이프의 한 쪽 통로로부터 input을 받고, 파이프의 다른 쪽 통로로 output을 내보낸다. 이후 나머지 파이프 통로를 close해주는 과정이 수반되도록 구현하였다.

```
if(flag == 1 && pipec == 1) {
  for(int j = 0; j < (pipec-1); j++) {
    close(fileds[j][0]);
    close(fileds[j][1]);
  }
}
if(pipec > 1) {
  if(i == 0) {
    dup2(fileds[i][1], STDOUT_FILENO);// make output
  }
  else if(i == pipec-1) {
    dup2(fileds[i-1][0], STDIN_FILENO);//get input
  }
  else {
    dup2(fileds[i-1][0], STDIN_FILENO);
    dup2(fileds[i][1], STDOUT_FILENO);
  }
  for(int j = 0; j < (pipec-1); j++) {
    close(fileds[j][0]);
    close(fileds[j][1]);
  }
}
```

Figure 16 child process의 pipe 처리 과정

부모 프로세스 또한 모든 파이프의 통로를 닫아줌으로써 EOF를 알린다.

```
// Parent close all pipes
for(int i=0; i<(pipec-1); i++){
  close(fileds[i][0]);
  close(fileds[i][1]);
}
```

Figure 17 parent process의 pipe closing 과정, EOF를 child에게 알린다.

또한 부모 프로세스는 파이프로 연결된 모든 child들의 foreground job이 끝날 때까지 waitfg를 실행하게 된다.

```
//parent waits for foreground job to terminate
if(!background){
  //wait while pid[pipec-1] is no longer foreground process
  waitfg(pid[pipec-1]);
}
else{
  printf("[%d] (%d) %s", pid2jid(pid[pipec-1]),pid[pipec-1], cmdline);//*argv[i]
}
```

Figure 18 모든 child process가 돌아올 때까지 대기하는 parent process

<Redirection 구현 시 문제 상황>

">" command가 중간에 들어올 경우, 그 뒤에 오는 파일의 경로로 redirect를 시켜주어야 했다.

<해결 방안 : eval()>

Eval 함수에서 먼저 parseline 함수에서 파싱된 argv벡터를 iterate하며 > 부분이 있는지 여부를 확인하였다.

```
int ri, rj, flag=0, rfd, oldstdo;
for(ri=0;ri<pipec; ri++){
  for(rj=0; argv[ri][rj]!=NULL; rj++){
    if(!strcmp(argv[ri][rj], ">")){
      //ri = i;
      flag = 1;
      break;
    }
  }
  if(flag) break;
}
```

Figure 19 Redirection(">") 파싱 과정

이를 dup2 함수를 이용하여 redirect해준다. 이 때 STDOUT을 redirect하는 과정에서 STDOUT이 원래 가리키던 곳이 사라져버린다는 단점이 있다. 이를 oldstdo 라는 변수에다가 미리 저장해주었다.

```
if(flag){
  if(!argv[ri][rj+1]) return;
  else{
    if((rfd = open(argv[ri][rj+1], O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0) {
      perror("cannot open redirect file");
      return;
    }
    oldstdo = dup(1);
    dup2(rfd, 1);
    close(rfd);
  }
  argv[ri][rj] = 0x00;
}
```

Figure 20 Redirection 과정, STDOUT이 가리키는 곳을 미리 저장해준다.

```
if(flag == 1) {
  dup2(oldstdo, 1);
}
```

Figure 21 STDOUT 원상복구 과정

나중에 parent process 에서는 waitfg 함수를 호출하기 전 STDOUT 이 가리키는 곳을 원상복구 시켜준다.

<에러 처리>

1. Eval 함수
   A. If 명령어에 아무것도 들어오지 않았을 경우 즉시 리턴한다.
   B. > 뒤에 파일을 열지 못할 경우 perror 후 즉시 리턴한다.
   C. 파이프를 생성하지 못할 경우 perror 를 통해 알린다.
   D. 처음 Signal set 을 비우지 못할 경우 perror 를 통해 알린다.
   E. 처음 Signal set 에 signal 을 더하지 못할 경우 perror 를 통해 알린다.
   F. SIGCHLD 시그널을 막을 수 없을 경우 perror 를 통해 알린다.
   G. Setpgid 로 process group id 를 바꿀 수 없을 경우 perror 를 통해 알린다.
   H. Execve 를 할 수 없을 경우 perror 후 terminate 한다.

2. Do_bgfg() 함수
   A. Bg 나 fg 명령어 이후 아무런 명령어가 뒤따라 오지 않았을 경우 알림 후 리턴한다.
   B. Bg/fg + % 이후 아무런 숫자도 없을 경우 알림 후 리턴한다.
   C. 해당하는 job 이 없을 경우(Joblist 에서 찾을 수 없을 때) 알림 후 리턴한다.
   D. Bg/Fg 이후 %로 시작하지도 않고, 숫자로 시작하지도 않을 경우 알림 후 리턴한다.