유닉스 프로그래밍 개인 프로젝트 보고서

12191621 안용모 - 2020년 10월 30일 (1차)

과제 #1

첫번째 과제의 요구 사항은 다음과 같다.

1. ‘cd’ 명령어가 제대로 작동하지 않는 버그 수정
2. ‘exit’ 명령어 구현
3. 백그라운드 실행 (&) 구현

‘cd’ 명령어 버그 수정

'cd’ 명령어는 현재 쉘의 current working directory를 변경하는 명령어이다.

```
myshell> ls -al
합계 44
drwxr-xr-x 4 aym506 aym506  4096 10월  30 00:08 .
drwxr-xr-x 4 aym506 aym506  4096 10월  30 00:06 ..
drwxr-xr-x 2 aym506 aym506  4096 10월  30 00:08 .vscode
drwxr-xr-x 2 aym506 aym506  4096 10월  30 00:08 mydir
-rwxr-xr-x 1 aym506 aym506 20632 10월  30 00:08 simple_myshell
-rw-r--r-- 1 aym506 aym506  1215  8월   4 08:35 simple_myshell.c
myshell> cd .
main(): No such file or directory
myshell> cd ..
main(): No such file or directory
myshell> cd mydir
main(): No such file or directory
```

원본 코드에서 'cd' 명령어를 실행하는 경우, 올바른 폴더명으로 이동을 시도해도 “main(): No such file or directory”이라는 오류가 발생하게 된다. 이는 PATH에서 'cd’에 대한 실행 파일이 존재하지 않는다는 의미로, 쉘에서 다른 명령어(ls, vim 등등)와는 달리 ‘cd’ 명령어는 쉘에서 새로운 프로세스 형태로 실행되지 않음을 알 수 있다. 즉, 쉘 내부에서 자체적으로 'cd’ 명령어를 구현해야 'cd'가 제대로 작동할 것이다.

'cd’ 명령어는 쉘의 current working directory를 바꾸는 명령어이므로, 'cd' 명령어를 실행하는 경우 chdir()으로 구현해야 하는데, chdir()은 **현재 프로세스**의 current working directory를 변경하므로 fork()하지 않고 쉘 프로세스에서 직접 chdir()를 호출하는 방식으로 구현할 것이다.

fork() 이전에 쉘에 입력한 문자열을 확인해야 하므로, 입력한 argument들을 cmdvector에 저장하는 makelist 함수를 fork() 이전에 실행되도록 main()의 while문 내부, switch문 이전의 내용을 다음과 같이 수정해야 한다.

의사 코드

```
print Prompt // myshell>
get user's raw input
reorganize user's input to vector // makelist

IF first argument is 'cd'
      change shell's current working directory to second argument
      continue (next loop)

switch statement for fork()
```

실제 코드 (while문)

```c
while (1)
{
   fputs(prompt, stdout);
   fgets(cmdline, BUFSIZ, stdin);
   cmdline[strlen(cmdline) - 1] = '\0';
   i = makelist(cmdline, " \t", cmdvector, MAX_CMD_ARG);

   // cd 명령어는 fork()하지 않음
   if (strcmp(cmdvector[0], "cd") == 0)
   {
     cd();
     continue;
   }

   switch (pid = fork())
   {
   case 0:
     execvp(cmdvector[0], cmdvector);
     fatal("main()");
   case -1:
     fatal("main()");
   default:
     wait(NULL);
   }
}
```

실제 코드 (cd 함수)

```c
void cd()
{
 if (chdir(cmdvector[1]) == -1)
   perror("cd"); // 에러 발생
}
```

실행 결과

```
myshell> pwd
/home/aym506/문서/수업/유닉스프로그래밍/프로젝트
myshell> cd /home/aym506
myshell> pwd
/home/aym506
myshell> mkdir dirA
myshell> cd dirA
myshell> pwd
/home/aym506/dirA
myshell> cd ..
myshell> pwd
/home/aym506
```

bash에서 실행하는 경우 테스트 결과에 알맞게 제대로 실행됨을 확인할 수 있다.

그러나, 현재 구현한 'cd'는 실제 쉘에서 구현되는 ‘cd’와는 달리 빠른 home directory로의 이동('~')과 바로 전 directory로의 이동('-')을 구현하지 않았다. 이는 각각 getenv(HOME)과 stack을 이용한 history로 구현할 수 있을 것이다. 이 두가지의 구현은 필요한지 명시되지 않아 생략하였다.

‘exit’ 명령어 구현

‘exit’ 명령어는 현재 쉘 환경을 종료하라는 의미이다. 직접 만든 쉘을 실행하고 'exit' 명령어를 입력할 시 직접 만든 쉘이 종료되고 원래 쉘(bash) 환경으로 돌아갈 것이다.

```
myshell> exit
main(): No such file or directory
```

그러나 myshell에서 ‘exit’ 명령어를 입력하는 경우 ‘cd’ 명령어가 실행되지 않는 오류와 마찬가지로 “main(): No such file or directory”이라는 오류가 발생한다. 이는 exit 역시 쉘 내부에서 직접 구현해야 함을 의미한다.

exit 명령어는 첫 argument가 'exit'인지 확인하고, 같다면 while문에서 빠져나오는 방식으로 구현할 수 있다. 따라서 main() 함수 while문 내부의 switch문 앞에 다음과 같은 내용의 코드를 작성하면 된다.

의사 코드

```
IF first argument is exit: break (terminate loop)
```

실제 코드

```
if (strcmp(cmdvector[0], "exit") == 0)
     break;
```

실행 결과

```
[aym506@aymarch 프로젝트]$ ./simple_myshell
myshell> exit
[aym506@aymarch 프로젝트]$ ps -u 1000 ¦ grep simple
[aym506@aymarch 프로젝트]$
```

bash에서 테스트하는 경우 명령어가 제대로 실행되어 프로세스가 종료되었음을 확인할 수 있다.

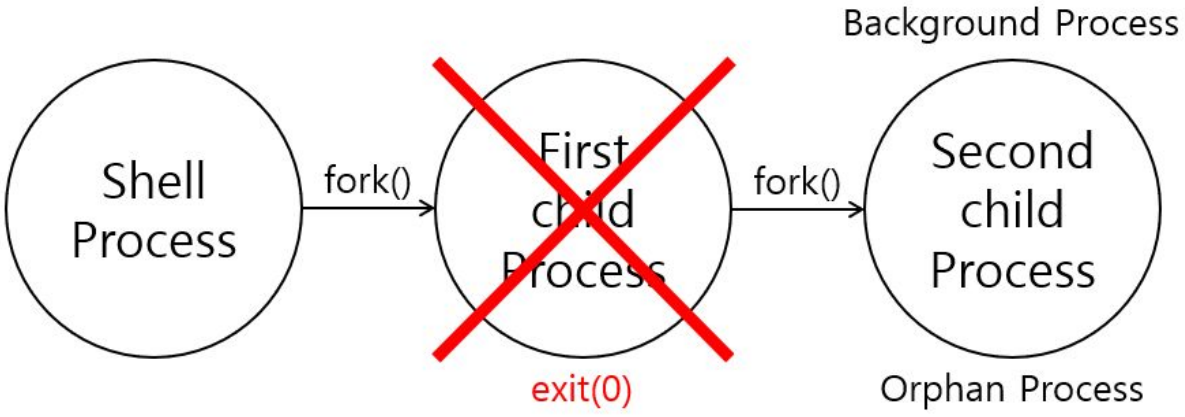
백그라운드 실행 구현

백그라운드 실행은 특정 명령어 뒤에 '&' 매개변수를 붙이는 경우 그 명령어는 백그라운드에서 실행되도록 (작업 도중에도 쉘에 입력이 가능하도록) 구현하면 된다.

```
myshell> sleep 10 &
sleep: invalid time interval `&'
Try 'sleep --help' for more information.
```

원본 코드에서는 백그라운드 실행이 구현되어 있지 않아 '&' 옵션이 매개변수로 인식되는 상태이다.

구현을 위해서 먼저 마지막 argument가 '&'인지 확인하고, 맞다면 vector에서 '&'를 제외하고 주어진 명령어를 백그라운드로 실행할 것이다. 이때 백그라운드 실행은 두번의 fork()로 고아 프로세스를 만들어 그 프로세스를 백그라운드 프로세스로 사용하는 방식으로 구현할 것이다. 이를 그림으로 표현하면 다음과 같다.



이러한 내용은 switch의 case 0 내부의 내용, 즉 fork()된 프로세스에서 또다시 fork()하여 부모 프로세스인 경우 waitpid()를 WNOHANG 옵션으로 호출한 뒤 종료하면 된다. 자식 프로세스이면 일반적인 경우와 동일하게 exec를 수행하면 된다.

의사 코드

```
IF process is child of shell process
     IF last argument is '&'
          another fork()
          exclude '&' from argument
          IF process is parent of second process
               wait with WNOHANG option (don't wait)
               terminate this process (exit)
     try running external process (execvp)
```

실제 코드

```
switch (pid = fork())
{
   case 0:
     if (strcmp(cmdvector[i - 1], "&") == 0) // Background Process
     {
       cmdvector[i-- - 1] = '\0';
       child_pid = fork();
       if (child_pid != 0)
       {
         waitpid(-1, NULL, WNOHANG);
         exit(0);
       }
     }

     execvp(cmdvector[0], cmdvector);
     fatal("main()");
   case -1:
     fatal("main()");
   default:
     wait(NULL);
}
```

실행 결과

첫번째 테스트를 수행한다. ‘sleep 10 &’과 ‘sleep 20 &’을 명령어에 입력한 뒤 ps로 현재 사용중인 tty에서 실행중인 프로세스들을 표시한다.

```
myshell> sleep 10 &
myshell> sleep 20 &
myshell> ps
    PID TTY          TIME CMD
  16112 pts/3    00:00:00 bash
  16169 pts/3    00:00:00 simple_myshell
  17975 pts/3    00:00:00 sleep
  17982 pts/3    00:00:00 sleep
  17984 pts/3    00:00:00 ps
```

이 테스트에서 2개의 백그라운드 프로세스 (17975, 17982)가 제대로 작동함을 확인할 수 있다. 현재 구현 방식에서는 이러한 테스트 방식에서 오류가 발생하지 않았다.

이제 두번째 테스트를 수행한다. myshell에서 다음과 같이 두 명령어를 실행한다. 첫번째 두번째 ‘sleep 120’에서 myshell은 입력할 수 없는 상태가 되게 된다.

```
myshell> sleep 120 &
myshell> sleep 120
```

입력할 수 없는 상태인 동안 **또다른 쉘**에 ‘ps -ef | grep sleep’을 입력한다. 이는 현재 실행중인 프로세스 중 이름에 ‘sleep’이 들어간 프로세스들을 표시하게 된다.

```
[aym506@aymarch ~]$ ps -ef | grep sleep
aym506     16382       1  0 19:14 pts/3    00:00:00 sleep 120
aym506     16386   16169  0 19:14 pts/3    00:00:00 sleep 120
aym506     16393   16194  0 19:14 pts/4    00:00:00 grep sleep
```

여기서 3번째 열의 숫자는 프로세스의 ppid인데, ppid가 1인 프로세스가 myshell에서 백그라운드로 실행한 sleep 120이며, ppid가 16169인 sleep 120은 myshell에서 포어그라운드로 실행한 프로세스임을 확인할 수 있다.

마지막으로 좀비 프로세스가 발생하는지 테스트를 시행한다. ‘ps -u 1000 &’을 세번 시행하고 ‘ps -u 1000’ 을 실행한 결과는 ps 출력 결과.txt에 적어놓았다. 이때 이름이 <defunct>인 좀비 프로세스가 발생하지 않음을 확인할 수 있다.