CSED211: Shell Lab report / 20230784 jaewonyu 유재원

1. 개요

   본 과제는 간단한 쉘 프로그램을 직접 만들어 보는 과정에서 signal의 개념 및 처리법 등을 자세히 이해하는데 그 의의가 있다.

2. 구현

   간단한 쉘 프로그램을 만드는 함수이다. 작은 크기의 함수부터 보면 좋을 것 같다.

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

첫 인자로 pid를 받아, 해당 pid가 foreground 상태가 아니게 될 때 까지 sleep 함수를 이용해 대기하도록 하는 함수이다. writeup에는 busy loop를 이용하는 것을 추천하지만 프로그래밍 실습 서버는 여러 사람이 동시에 이용하기에, CPU 사용량이 높아질 우려가 있는 busy loop를 사용하는 것이 꺼려져 sleep 을 이용해 구현하였다. 만약 foreground process의 상태가 변한다면 그것에 대한 처리는 SIGCHLD handler에서 handling하도록 구현하였기에, waitfg 함수는 이정도로만 작성하면 충분할 것 같다.

```c
int builtin_cmd(char **argv) {
  char *cmd = argv[0];

  if (!strcmp(cmd, "quit")) {
    exit(0);

  } else if (!strcmp(cmd, "jobs")) {
    listjobs(jobs);
    return 1;

  } else if (!strcmp(cmd, "bg") || !strcmp(cmd, "fg")) {
    do_bgfg(argv);
    return 1;
  }

  return 0; /* not a builtin command */
}
```

주어진 입력이 built-in command인지 여부를 반환하는 함수이다. stdbool을 사용하여 bool을 반환할까라는 고민을 잠깐 하였지만, 주어진 templete내에서 수행하는 것이 좋을 것 같아 수정하지는 않았다. 주어진 입력을 후술할 eval 함수에서 파싱한 후 argv로 넘겨주기에, 편히 argv[0] 을 이용하여 built-in command 여부를 확인하였다. 단 `quit awesome`과 같은 특이한 입력 케이스에 대한 예외 처리는 따로 구현하지 않았다. built-in command의 경우 1을, 그렇지 않다면 0을 반환한다. (개인적으로 is_builtin_cmd 정도의 이름으로, built-in 이라면 true를 반환하는 것이 취향이다.)

```
void eval(char *cmdline) {
  char *argv[MAXARGS];
  int bg = parseline(cmdline, argv);
  sigset_t mask, prev_mask;
  pid_t pid;

  if (argv[0] == NULL) {
    return;
  }
```

실제 사용자의 입력을 처리하는 함수이다. cmdline을 먼저 파싱해 주어야 하는데, 그것은 주어진 코드로 수월히 진행할 수 있다. 이 때 사용자의 명령이 `&`에 의해 background 작업이라면 parseline 함수가 1을 반환한다. 그렇지 않다면 0을 반환한다. 만약 파싱한 결과가 null이라면, 수행할 작업이 없으므로 바로 함수를 종료한다.

```
  if (!builtin_cmd(argv)) {
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    sigprocmask(SIG_BLOCK, &mask, &prev_mask);

    if ((pid = fork()) == 0) {
      sigprocmask(SIG_SETMASK, &prev_mask, NULL);
      setpgid(0, 0);
      if (execve(argv[0], argv, environ) == -1) {
        printf("%s: Command not found\n", argv[0]);
        exit(0);
      }

    }
```

이후 파싱한 결과를 인자로 builtin_cmd 함수를 호출한다. 그 결과 built-in command였다면 eval 함수에서 수행할 동작은 없으므로, if문을 통과하여 함수를 종료한다.

그렇지 않다면 유효한 명령이므로 해당 명령을 실행한다. fork를 통해서 child를 만든 후, child는 execve를 통해 명령을 수행, parent인 shell은 addjob함수를 이용해 job을 추가할 것인데, addjob 함수 호출이 끝나기 전에 child의 상태가 변해버려 SIGCHILD signal을 핸들링하게 되면, 예상치 못한 동작을 유발한다. 따라서 적어도 fork를 하는 시점부터 addjob이 끝나기 전까지 parent process는 SIGCHILD signal을 block해 주어야 한다. fork된 child의 입장에서는 SIGCHLD signal을 더 이상 block할 이유가 없으므로 MASK 정보를 되돌린 후 process group id를 설정한다. 이는 shell을 통해 실행된 프로그램은 shell과 다른 group에 속하도록 하는 것이 관리하기 좋기 때문이다. 이후 SIGINT 등 signal handler에 대해 설명할 때 명확해질 것이다.

execve가 성공한다면 해당 child의 context는 execve의 결과 완전히 넘어가기에 execve 함수는 반환하지 않는다. 실패한다면 -1을 반환하고 errno를 설정한다. 만약 execve가 실패했다면 적당한 오류 메시지를 출력하고 해당 child를 exit으로 종료한다.

사실 execve의 실패 원인은 command not found 외에도 다양하다. 이는 `man execve`를 통해 확인할 수 있다. 하지만 본 과제에는 딱히 관련 handling을 요구하지 않으므로 일괄적으로 처리하였다만, 실제 상황이라면 설정된 errno에 따라 적절히 handling을 해 주는 것이 좋아보인다.

```
else {
      addjob(jobs, pid, bg ? BG : FG, cmdline);
      sigprocmask(SIG_SETMASK, &prev_mask, NULL);

      if (bg) {
        printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);

      } else {
        waitfg(pid);
      }
    }
  }
```

parent인 shell의 입장에서는 앞서 얘기했듯 addjob 함수를 이용해 job을 추가하고, 이제 MASK 상태를 되돌려 준다. 보고서를 작성하며 생각해 보니 bg 작업의 경우 printf를 이용해 문구 출력 후 MASK를 되돌려 주는 것이 안전해 보인다. 만약 addjob 함수와 if(bg)의 printf 사이에 해당 job이 SIGINT 등의 signal에 의해 종료된다면 후술할 signal handler에 의한 출력이 먼저 출력될 것인데, 그것은 좋은 상황이 아니다. 하지만 이 또한 주어진 test case에서는 나올 수 있는 상황이 아니므로 수정하지 않겠다.

이후 background 작업이라면 적당한 문구 출력, 그렇지 않다면(foreground 작업이라면) 해당 process가 종료(혹은 정지)될 때 까지 기다린다.

```
void sigint_handler(int sig) {
  pid_t pid;
  if ((pid = fgpid(jobs))) {
    kill(-pid, sig);
  }
  return;
}
```

```
void sigtstp_handler(int sig) {
  pid_t pid;
  if ((pid = fgpid(jobs))) {
    kill(-pid, sig);
  }
  return;
}
```

위 두 handler는 각각 SIGINT, SIGSTOP signal을 handling한다. fgpid 함수의 반환값이 0이 아니라면, 즉 foreground 상태인 job이 있다면 해당 job의 group에 signal을 전달, 그렇지 않다면 아무 작업도 하지 않는다.

이 때 SIGINT, SIGSTOP signal은 일반적으로 모든 child process에게 보낸다. 따라서 parent인 shell 입장에서 child와, 그것들의 child가 같은 group 이어야지만 signal 전달이 원활하다. 또한 만약 setgpid 함수를 쓰지 않았더라면 현 시점에서 BG에 존재하는 모든 process들에게도 signal을 전달하여야 한다. 이는 분명히 우리가 원하는 동작이 아니므로, eval 함수에서 fork된 child의 group id를 따로 설정해 주어야 하는 것이다.

```
void sigchld_handler(int sig) {
  pid_t pid;
  int wstatus;

  while ((pid = waitpid(-1, &wstatus, WNOHANG | WUNTRACED)) > 0) {
    if (WIFEXITED(wstatus)) {
      deletejob(jobs, pid);

    } else if (WIFSIGNALED(wstatus)) {
      printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid,
             WTERMSIG(wstatus));
      deletejob(jobs, pid);

    } else if (WIFSTOPPED(wstatus)) {
      printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid,
             WSTOPSIG(wstatus));
      getjobpid(jobs, pid)->state = ST;
    }
  }

  return;
}
```

다음은 중요한 SIGCHLD signal의 handler이다. 이 signal은 child 프로세스의 상태가 변했을 때 커널이 전달해 주며, parent인 shell은 이 handler을 이용해 좀비가 된 자식 프로세스의 reap등을 처리해 주어야 한다. waitpid 함수를 통해 상태가 변한 process의 id를 임의로 하나 가져온다. 이 때 각 인자의 의미는 `man waitpid`를 통해 확인하는 것이 좋다.

코드에 적어둔 인자의 의미는 `아무 자식 프로세스 중`, `그 결과는 wstatus에 저장`, `상태가 변환 프로세스가 없다면 즉시 0을 반환 + STOP 된 상태도 반환`이다. 기본적으로 waitpid 함수는 `terminated`로 상태가 변하는 프로세스만을 기다리기에 `WUNTRACED` 옵션이 필요하다.

프로그램이 정상 종료되었다면 WIFEEXITED(wstatus) 가 true로 되어 if문 속 내용이 실행되는 식이다. 각 if문의 조건에 대한 상세한 내용은 역시 `man waitpid`에 자세히 나와있다. 이 함수에서 일반적인 종료, SIGNAL에 의한 종료, STOP의 3가지 상황을 처리한다. 일반적인 종료시에는 job을 삭제만 해 주면 되고, SIGNAL에 의한 종료시에는 해당 문구 출력 후 job삭제, STOP되었다면 이 역시 문구를 출력 후 상태를 ST(STOP)으로 바꾸는 동작을 수행한다.

상태가 변한 모든 job에 대하여 handling을 해 주어야 한다. signal은 queue 되는 것이 아니기에, 여러 job이 동시에 죽더라도 SIGCHLD handler는 한 번만 수행될 수도 있기 때문이다. 따라서 waitpid 에 의해 잡히는 process가 없을 때까지 처리를 해 주어야 하기에, while 문을 이용한다.

```
void do_bgfg(char **argv) {
  if (argv[1] == NULL) {
    printf("%s command requires PID or %%jobid argument\n", argv[0]);
    return;
  }
```

do_bgfg 함수는 bg, fg 내장 명령에 대한 처리를 진행하는 함수이다. 과제에서 bg 혹은 fg 명령의 argv[1]이 없다면 적절한 오류 메시지를 출력하는 것을 요구하기에 그것에 대한 예외 처리를 먼저 진행해 주었다.

```
  if (argv[1][0] == '%') {
    struct job_t *job = getjobjid(jobs, atoi(argv[1] + 1));
    if (job == NULL) {
      printf("%s: No such job\n", argv[1]);
      return;
    }
    if (!strcmp(argv[0], "bg")) {
      kill(-job->pid, SIGCONT);
      job->state = BG;
      printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
    } else {
      kill(-job->pid, SIGCONT);
      job->state = FG;
      waitfg(job->pid);
    }
  }
```

만약 주어진 argv[1][0]이 `%`라면 그것은 jid를 받는 것을 의미한다. 이 경우 해당 jid의 job을 먼저 찾아온다. 그러한 jid를 가진 job이 없다면 예외 처리를 해준다.

만약 명령이 `bg`라면 해당 job의 process group에게 SIGCONT를 보내고 해당 job의 state를 BG로 바꿔준 후 적절한 문구를 출력한다. 그렇지 않다면(fg 라면) 마찬가지로 SIGCONT를 보내준 후, state를 FG로 바꾸고, 이제는 해당 process의 상태가 변할 때 까지 기다려 주어야 하므로(해당 작업이 foreground에서 돌아가고 있으므로) waitfg 함수를 이용해 대기한다.

```
else {
    if (!isdigit(argv[1][0])) {
      printf("%s: argument must be a PID or %%jobid\n", argv[0]);
      return;
    }
    struct job_t *job = getjobpid(jobs, atoi(argv[1]));
    if (job == NULL) {
      printf("(%s): No such process\n", argv[1]);
      return;
    }
    if (!strcmp(argv[0], "bg")) {
      kill(-job->pid, SIGCONT);
      job->state = BG;
      printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
    } else {
      kill(-job->pid, SIGCONT);
      job->state = FG;
      waitfg(job->pid);
    }
  }
```

jid를 받는 상황이 아니라면, 먼저 입력받은 pid가 유효한 숫자인지 확인한다. 위의 상황을 함께 고려해보면 어색한 예외 처리 위치이지만, 레퍼런스 tsh가 정확히 이렇게 예외를 처리함을 직접 실행해 확인할 수 있다. 나머지 부분은 위의 부분과 정확히 같은 로직으로 구성되어 있다.

3. 결론

본 과제를 통하여 쉘의 동작 원리 및 signal에 대한 처리법, 더 나아가 execve의 자세한 동작 등을 이해할 수 있었다.