Shell Lab 보고서

2018-17329 최성혁

`Waitpid` `Fork` `Sigemptyset` 등 함수의 첫 글자가 대문자 처리되어 있는 함수들은 각각 원래 함수들의 wrapper function 으로, 원래 함수 (예를 들어, `Waitpid`라면 `waitpid` 함수)를 실행하고 에러가 발생했는지 확인하는 역할을 합니다.
`sio_puts` `sio_putl` 등 함수는 주어진 argument를 async-signal-safe 하게 출력하는 wrapper function 입니다.

eval

```
void eval(char *cmdline)
{
    int bg;
    int pid;
    sigset_t mask;
    char *argv[MAXARGS];

    bg = parseline(cmdline, argv);

    if (argv[0] == NULL) { //terminate on EOF
        return;
    }

    if (builtin_cmd(argv)) { //if it is a built-in-command: execute it and return
1. else return 0.
        return;
    }

    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);

    Sigprocmask(SIG_BLOCK, &mask, NULL); //mask SIGCHLD

    if ((pid = Fork()) == 0) {
        // Child's behavior
        setpgid(0, 0);
        Sigprocmask(SIG_UNBLOCK, &mask, NULL);

        if (Execve(argv[0], argv, environ) < 0) {
            printf("%s: Command not found.\n", argv[0]);
            exit(0);
        }
    }

    // Parent's behavior
    if (!bg) {
        addjob(jobs, pid, FG, cmdline);
        Sigprocmask(SIG_UNBLOCK, &mask, NULL);
```

```
        waitfg(pid); /* wait for the foreground job to finish */
    } else {
        addjob(jobs, pid, BG, cmdline);
        Sigprocmask(SIG_UNBLOCK, &mask, NULL);
        printf("[%d] (%d) %s", pid2jid(pid), (int)pid, cmdline); /* print out log
and execute in background */
    }

    return;
}
```

Shell의 메인이 되는 함수입니다. 이 함수에서 built-in command를 실행하거나 child를 `fork` 해서 `execve`로 다른 프로그램을 실행합니다.
Parent가 `addjob` 을 실행하기 전에 child의 작동이 끝나서 `SIGCHLD` 신호를 받으면 안 되기 때문에 신호를 막아주었습니다. `parseline` 함수의 결과에 따라 background 작업인지 foreground 작업인지 나누어 처리했습니다.

builtin_cmd

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

    if (!strcmp("quit", command)) {
        exit(0);
    } else if (!strcmp("jobs", command)) {
        listjobs(jobs);
    } else if (!strcmp("bg", command) || !strcmp("fg", command)) {
        do_bgfg(argv);
    } else {
        return 0;     /* not a builtin command */
    }

    return 1;
}
```

Built-in command를 실행하는 함수입니다.

do_bgfg

```
void do_bgfg(char **argv)
{
    struct job_t *job;
    int is_bg = !strcmp("bg", argv[0]);
    int is_pid;

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

```
    }

    if (argv[1][0] == '%') { /* if it's jid */
        is_pid = 1;
        job = getjobjid(jobs, my_atoi(&argv[1][1]));
    } else { /* if it's pid */
        is_pid = 0;
        job = getjobpid(jobs, my_atoi(argv[1]));
    }

    if (errno == EINVAL) { //my_atoi error (wrong argument)
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }

    if (job == NULL) { //can't find job
        if (is_pid) {
            printf("%s: No such job\n", argv[1]);
        } else {
            printf("(%s): No such process\n", argv[1]);
        }
    } else {
        Kill(-job->pid, SIGCONT); /* send SIGCONT signal to every process under
process group */

        if (is_bg) { // command: bg
            job->state = BG;
            printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
        } else { // command: fg
            job->state = FG;
            waitfg(job->pid);
        }
    }

    return;
}
```

Built-in command 중 하나인 `bg`와 `fg`를 처리하는 함수입니다.
스펙에 있는대로 job id나 process id를 받아서 멈춰있던 함수에 `SIGCONT` 신호를 보내줍니다. `bg`가 호출되었다면 background에서 마저 실행하고, `fg`가 호출되었다면 foreground에서 실행하면 됩니다.
이때, 해당 job들의 state도 변경해줘야 합니다.
예외 처리에 `my_atoi`라는 함수가 사용된 것을 보실 수 있는데, string을 int로 파싱해주기 위해 제가 만든 함수입니다. 만약 argument를 int로 바꾸는 게 불가능하다면 global variable인 `errno`의 값을 `EINVAL`로 바꿔주고 0을 반환합니다.
`my_atoi` 구현은 아래와 같습니다.

```
int my_atoi(char* start) {
    if (start == NULL) {
        errno = EINVAL;
        return 0;
```

```
    }

    int result = 0;
    char curr;

    while ((curr = *start) != '\0') {
        if ('0' <= curr && curr <= '9') {
            result = result*10 + curr - '0';
            ++start;
        } else {
            errno = EINVAL;
            return 0;
        }
    }

    return result;
}
```

waitfg

```
void waitfg(pid_t pid)
{
    /* according to the Hint on assignment handout, use budy loop around sleep
function
     * and do all reaping in the signal handler
     */

    while(1) {
        if (fgpid(jobs) != pid) { /* when given foreground job terminated */
            break;
        } else {
            Sleep(1);
        }
    }

    return;
}
```

handout에 적혀있듯, `sleep` 함수를 활용한 루프를 사용해 구현했습니다. `wait`하라고 주어진 pid가 foreground에서 수행이 끝나면 루프를 빠져나옵니다.

sigchld_handler

```
void sigchld_handler(int sig)
{
    pid_t pid;
    int status;
    char log_message_buff[1024];
    struct job_t *job;
```

```
    int prev_errno = errno;

    /* WNOHANG: return immediately if none of the child processes in the wait set
has terminated yet.
       WUNTRACED: return pid of the terminated or "stopped" child */
    while((pid = Waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
        job = getjobpid(jobs, pid);

        if (WIFSIGNALED(status)) {
            /* when the process terminated */
            snprintf(log_message_buff, 1024, "Job [%d] (%d) terminated by signal
%d\n", job->jid, (int)pid, WTERMSIG(status));
            sio_puts(log_message_buff);
            deletejob(jobs, pid);
        } else if (WIFSTOPPED(status)) {
            /* when the process stopped */
            snprintf(log_message_buff, 1024, "Job [%d] (%d) stopped by signal
%d\n", job->jid, (int)pid, WSTOPSIG(status));
            sio_puts(log_message_buff);
            job->state = ST;
        } else if (WIFEXITED(status)) {
            /* delete finished job from the job list
               without this, you get to send signal to wrong pid at sigint/sigtstp
handler. */
            deletejob(jobs, pid);
        }
    }

    errno = prev_errno;

    return;
}
```

`SIGCHLD` 신호를 받았을 때 작동할 signal handler 입니다.
현재 종료된 child 들을 모두 reap 해야 하기 때문에 while 문을 사용하고 `waitpid` 함수의 첫번째 parameter로 -1을 넘겨주었습니다. 또, 모든 child들이 끝나길 기다려서는 안 되고(`WNOHANG`), terminate 된 상태뿐 아니라 stop된 상태의 child도 다뤄야하기 때문에(`WUNTRACED`), 세번째 parameter로는 `WNOHANG | WUNTRACED`를 넘겨주었습니다.
`printf`는 async-signal-safe 하지 못합니다. 프로그램의 여러 곳에서 `printf`를 호출해주고 있기 때문에, `printf`가 호출되고 있는 상황에 신호가 들어와서 signal-handler가 호출되고 `printf`가 그 안에서 호출되면 데드락 상황이 발생합니다.
따라서 `printf`를 사용하는 대신 넘칠 일 없을만큼 충분히 큰 (여기서 출력하고자 하는 것은 로그 메시지이므로 길이의 상한을 이미 알고 있습니다) 버퍼 `log_message_buff`를 정적으로 할당해주고, `snprintf` 함수와 `sio_puts` 함수로 출력해주었습니다.

sigint_handler

```
void sigint_handler(int sig)
{
    /* preserve previous errno */
```

```
    int prev_errno = errno;

    pid_t foreground_pid = fgpid(jobs);

    if (foreground_pid != 0) {
        /* if there is a foreground job, send SIGINT */
        Kill(-foreground_pid, sig);
    }

    errno = prev_errno;
    return;
}
```

ctrl-c가 입력되었을 때 Shell은 여전히 작동하면서 foreground의 process는 중지시키도록 해주는 signal handler입니다.
현재 foreground에 실행되고 있는 process의 process id를 `fgpid` 함수로 얻어오고 그 process group에 `SIGINT` 신호를 보냅니다. `eval` 함수에서 `setpgid(0, 0);` 식으로 child의 process group id를 설정해줬기 때문에 `kill`의 첫번째 parameter로 `-foreground_pid`를 제공해주면 그 process group 전체를 terminate 할 수 있게 됩니다.
앞의 `my_atoi` 함수에서 `errno`를 활용해서 로직을 처리하기 때문에 이 signal handler가 `errno`를 임의로 변경하면 안 됩니다. 따라서 함수를 시작하면서 기존 `errno`를 저장해주고 함수가 끝날 때 다시 복원합니다.

sigtstp_handler

```
void sigtstp_handler(int sig)
{
    int prev_errno = errno;

    pid_t foreground_pid = fgpid(jobs);

    if (foreground_pid != 0) {
        /* if there is a foreground job, send SIGTSTP */
        Kill(-foreground_pid, sig);
    }

    errno = prev_errno;

    return;
}
```

`sigint_handler` 와 같습니다.

어려웠던 점

`sigchld_handler` 함수의 구현이 어려웠습니다. if - else if 문의 마지막 분기인

```
else if (WIFEXITED(status)) {
    deletejob(jobs, pid);
```

```
}
```

해당 부분에서 디버깅에 많은 시간을 투자하였습니다.
자꾸 `kill`에서 해당 process group을 찾을 수 없다는 에러가 발생했습니다. 디버깅 결과 `WIFSIGNALED(status)`는 catch하지 못한 signal로 종료된 작업들만 잡아낸다는 게 문제임을 알게 되었습니다.
정상적으로 종료된 child 들의 경우도 `deletejob`을 통해 job list에서 제거해줘야 하는걸 간과했습니다.
위와 같이 `WIFEXITED(status)`를 써서 해당 child process를 job list에서 지울 수 있었습니다.

새롭게 배운 점

구글링을 하지 않고 man 페이지를 참고하는 법을 익힐 수 있었습니다. 때로는 부정확한 정보를 얻게 되는 구글링에 비해 훨씬 신뢰할 수 있고 자세한 정보를 얻을 수 있었습니다.
또, signal handling과 fork, child process와 parent process에 대해 더 잘 이해할 수 있게 되었습니다.