System Programming Lab #2 Report

2019-18499 김준혁

함수 별 구현 방법

void eval(char *cmdline)

argument를 받을 char 2차원 배열을 MAXARGS * MAXLINE 크기로 선언하고, 가장 첫 위치를 char **argv에 저장한다. background에서 실행하는지 저장할 int bg, Child의 pid를 저장할 pid_t pid, signal masking을 위한 sigset_t sigmask를 선언하고 시작한다. 파싱을 하기 전, 공백만이 들어오는 경우에 아무 행동도 하지 않고 다음 명령을 기다리기 위해서 '₩n'이 나올 때까지 isspace()를 통해 검사하여 공백만 있는 경우에 return을 하도록 처리했다. 이후 sigemptyset, sigaddset을 통해 signal block을 위한 mask를 만들었다.

parseline을 돌려서 나온 리턴값은 bg에 저장, 파싱값은 argv에 저장한다. builtin_cmd함수의 리턴값이 0, 즉 builtin command가 아닐 경우, SIGCHLD를 block한 후, fork를 한다. child의 경우, signal unblock하고, pgid를 자기 자신으로 바꾸어준 후(에러 발생 시 프로세스 종료), execve로 명령을 실행한다. parent의 경우, bg에 따라 행동을 나눈다. foreground 프로그램의 경우, jobs에 해당 job을 add하고(에러시 kill), fg group의 프로그램들을 한 group에 모으기 위해 setpgid(pid, fgpid(jobs))를 해준 후, signal unblock하고 foreground 프로그램이 끝날 때까지 기다린다(waitfg). background의 경우, setpgid, waitfg를 제외한 addjob, sigprocmask를 실행한 후, 해당 프로그램의 정보(jid, pid, cmdline)을 출력한다.

int builtin_cmd(char **argv)

eval을 통해 받은 argv를 이용하여, argv[][]의 비교를 통해 quit, jobs, bg, fg, kill(spec pdf에 정의만 되어있고, 상세한 스펙이 없지만 간단하게 구현)을 구분하여 이들은 즉시 실행하고(jobs->listjobs, bg/fg->do_bgfg) 1을 반환한다(quit의 경우 바로 exit(0)으로 종료). 나머지의 경우 0을 반환한다.

void do_bgfg(char **argv)

bg, fg의 경우 모두 이 함수를 호출하는데, 일단은 argv[1]가 NULL인, 즉 명령어만 존재하는 경우, pid나 jid가 잘못 입력된 경우, 해당하는 pid나 jid에 대해 프로세스가 없는 경우, job이 없는 경우에 대해 에러를 출력하고 리턴한다. 그 외의 경우, 즉 올바르게 수행할 수 있는 값이 들어온 경우 작업을 실행한다.

bg의 경우, ST state의 job만 골라내어 state를 BG로 바꾸고, 정보 출력, 그리고 kill(pid,

SIGCONT)를 통해 프로그램을 재시작한다. job의 기존 상태가 ST가 아닌 경우, 해당하는 job이 없다는 에러를 띄운다.

fg의 경우, ST, BG state의 job에 대해서 state를 FG로 바꾸고, SIGCONT signal을 kill로 전달한 후, waitfg를 호출한다. 이때, kill에서 pid를 -(job->pid)로 하여, Foreground group의 모든 프로세스에 SIGCONT를 보낸다. state가 ST, BG가 아닌 경우 에러 메시지를 출력한다.

void waitfg(pid_t pid)

해당하는 job이 없는 경우, 즉 foreground가 없는 특이한 경우 발생시, 에러 메시지를 띄우고 반환한다. 일반적인 경우, 새로운 signal set을 만들어 foreground의 job이 FG인동안 sigsuspend를 통해 signal이 오기를 기다린다. signal이 올 경우, 해당하는 handler를 돌고 되돌아올텐데, 이때 while문에서 FG였던 job의 상태가 바뀌었는지 확인, 바뀌면 waitpid를 하여 안전하게 종료된(child_statue == 0) 프로그램을 reap한다. 안전하게 종료되지 않은 경우(ex: SIGINT, SIGTSTP, 멈춘 경우 포함), signal handler에서 처리될 것이기 때문에 고려할 필요가 없을 것이라 판단하였다.

void sigchld_handler(int sig)

waitpid(-1, &child_status, WNOHANG | WUNTRACED)를 통해 멈추거나 종료된 child가 없으면 바로 return하며, 해당하는 child가 존재할 경우 아래의 행동을 따른다.

child가 SIGINT로 종료된 경우(child_status == SIGINT), terminated 메시지를 띄우며, jobs에서 해당 pid의 job을 제거한다.

child가 SIGTSTP로 멈춘 경우(WSTOPSIG(child_status) == SIGTSTP), stopped 메시지를 띄우며, 해당하는 job의 state를 ST로 바꾸어준다.

child가 정상종료(child_status == 0)된 경우, 추가 행동 없이 jobs에서 해당 job을 제거한다.

이외의 signal을 받아 종료/멈춘 경우에는 pass한다. (발생하지 않는 case다)

void sigint_handler(int sig)

SIGINT로 종료할 foreground job이 없으면, 특별한 행동 없이 바로 return한다.

foreground job이 존재할 경우, kill(-pid, SIGINT)를 통해 해당하는 pgid를 가진 모든 foreground jobs(group의 모든 jobs)에 SIGINT signal을 보낸다. 이 signal을 받은 프로세스는 종료하면서 SIGCHLD를 parent에 보내며, 이는 sigchld_handler에서 처리된다. 이후, reap되지 않은 프로세스를 reap해주고, jobs에서 해당 job을 제거한다. 이 reaping은 sigchld_handler에서 대부분 처리가 될 것이지만, 남겨두었다.

void sigtstp_handler(int sig)

SIGTSTP로 멈출 foreground job이 없으면, 특별한 행동 없이 바로 return한다.

foreground job이 존재할 경우, job state를 ST로 바꾸고, kill(-pid, SIGTSTP)로 foreground group의 모든 jobs에 SIGTSTP signal을 보낸다. 이 signal을 받은 프로세스는 멈추면서 SIGCHLD를 parent에 보내며, sigchld_handler에서 처리된다. 이때 이 함수에서 state를 바꾸는 것은 sigchld_handler에서 처리될 것이며, 한 foreground job에 대해서만 변경이 이루어져 쓸모없는 행동이지만, kill이 정상적으로 모든 job에 대해 행동하는지 확인하기 위한 개인적인 용도로 남겨두었다.

어려웠던 점

signal masking set 및 block 관련 함수, setpgid, waitpid 의 WUNTRACED, child로의 SIGINT/SIGTSTP 이후 child가 전송하는 SIGCHLD 및 child_status에 대한 정보 등 알아야 할 것이 너무 많았으며, 과제를 진행하면서 이에 대해 존재를 모르거나 사용법을 제대로 알지 못할 경우에 중간에 막히는 시간이 너무 많았어서 피로함이 너무 컸다. 특히 WNOHANG과 WUNTRACED가 동시에 사용 가능한 것을 몰라 특히 SIGTSTP를 어떻게 처리해야하는지 몰랐을 때 가장 많은 헛짓거리를 해서 기존에 잘 되던 case까지 돌아가지 않을 때가 가장 심적으로 지쳤다. 어느 정도 Slide나 Spec에 힌트로 주어진 것들이 있었지만, 내용이 개인적으로 길다고 느껴졌을뿐더러, 사실 코드 작성을 시작하기 전에 어떻게 작성해야 할지 와닿는 것이 적어, 추후 오랜 시간 고민하다가 다시 pdf를 꺼내보았을 때 발견하는 경우가 많아서, "내가 왜 이걸 몰랐지"하면서 허탈해지는 것이 많았다.

그래도 trace 파일을 통해 1번부터 16번까지 해당 case가 잘 작동하도록 하나씩 작성할 수 있게 해주는 milestone같은 역할을 해주어, 비록 많은 고뇌와 피로가 있었지만 그래도 마무리를 할 수 있었던 것 같다.

새롭게 배운 점

여러 프로세스가 어떻게 foreground, background로 나뉘어 동작하며, 그 사이에 SIGINT, SIGTSTP가 어떤 프로세스를 멈추는지 알 수 있었다. 그를 통해 발생하는 SIGCHLD의 존재, 그리고 이 SIGCHLD가 SIGTSTP 등에 의해 작동이 멈출 때에도 발생한다는 것을 깨달았으며, 이는 waitpid에서 WUNTRACED를 통해 받을 수 있다는 것을 알 수 있었다. 그리고, signal 공부를 하면서 까먹었던 내용들(ex: SIGINT, SIGTSTP같은 signal이 parent에 전달되었을 때, 해당 프로세서의 pgid와 같은 모든 child에도 동일한 signal이 전달되는 것)을 remind할 수 있었으며, 이에 대해 setpgid로 fore/back에 따라 분리하여 signal control을 할 수 있음을 깨달았다.