System Programming Project 2

담당 교수 : 이영민

이름 : 방지혁

학번 : 20190808

1. 개발 목표

- 본 프로젝트의 목표는 리눅스 환경에서 동작하는 기존의 bash shell을 모방한 간단한 사용자 정의 shell을 만드는 것이다. 해당 프로젝트를 통해 학기 전반부에 배운 포그라운드 프로세스 생성 및 제어, 시그널, 파이프 처리, 백그라운드 프로세스 생성 및 제어 등의 개념을 학습할 수 있다.
- 사용자 정의 shell인 Myshell은 사용자의 입력을 parsing하여 점차적으로 자식 프로세스 fork, 백그라운드 작업 관리, job 관리, signal handler, 파이프를 통한 IPC 같은 기능을 추가함으로써 실제 bash shell과 가깝게 구현해 나가고자 한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Phase 1

   Phase 1에서는 fork() system call을 이용해서 자식 프로세스를 생성하여 execvp() 함수를 호출한다. 사용자가 입력한 명령어를 참고하여 /bin 경로에 있는 실행파일로 덮어 씌워 명령을 수행할 수 있다. 해당 단계에서는 foreground 자식 프로세스를 생성하여 ls, mkdir, rmdir, touch, cat, echo, exit 등을 수행할 수 있다.

   예시 input)

mkdir testdir ls	Makefile README.txt myshell myshell.c myshell.h myshell.o testdir
touch testdir/cse4100 cd testdir ls	cse4100
echo "Hello World"	Hello World

2. Phase 2

   기존의 Phase1에서 나아가 프로세스 간의 입력과 출력을 연결하는 pipe 기능을 추가한다. pipe(), dup2() 그리고 fork() 함수를 적절히 호출하여 명령어

간의 파이프를 구현한다.

예시 input)

ls -al | grep ".c"	-rw-r--r-- 1 bbangjee bbangjee 21526 Apr 18 01:39 myshell.c
ps aux | grep "bash" | wc -l	3

3. Phase 3

명령어 뒤에 &가 붙은 경우, 해당 명령을 background process로 실행할 수 있도록 한다. 이때 fork된 자식 프로세스 이전의 phase와 다르게 독립적으로 실행되며 child handler를 통해 관리된다. 실행 또는 중지된 back ground process들과 중지된 fore ground process들은 jobs 목록에 추가될 수 있도록 한다. Jobs 명령어를 통해 현재 jobs 목록을 조회하고, fg 명령어를 통해 실행 및 중지된 bg 프로세스를 fg로 돌리고, bg 명령어를 통해 중지된 fg 프로세스를 bg로 돌리고, kill 명령어를 통해 jobs에 있는 fg 및 bg 프로세스를 죽일 수 있다.

예시 input)

sleep 30 &	[1] 56347
kill %1	[1] Terminated sleep 30 &
sleep 60 &	[1] 56430
bg %1	bg: job 1 already in background

B. 개발 내용

- **Phase1 (fork & signal)**

✓ fork를 통해서 child process를 생성하는 부분에 대해서 설명

fork() 시스템 콜을 통해 child process를 생성한다. 이 fork 함수의 경우 두 개의 서로 다른 값을 return 하는데, child process에서는 0을 return 하고, parent process 에서는 child process의 PID를 return 한다. 이러한 차이를 이용해 흐름을 분기할 수 있다. 자식 프로세스에서는 앞서 서술했듯 execvp 함

수를 통해 명령어를 실행하고, 부모 프로세스에서는 waitpid()를 통해 child process의 종료를 기다린다.

✓ connection을 종료할 때 parent process에게 signal을 보내는 signal handling 하는 방법 & flow

parent process인 shell은 waitpid 함수를 통해 child process가 종료되거나 상태가 변경될 때까지 기다린다. 이 과정에서 shell은 블로킹된 상태이다. waitpid함수는 SIGCHILD와 같은 signal을 받으면 return한다. 이러한 과정을 거친 후, 다시 다음 명령어를 받는 state로 넘어간다.

- **Phase2 (pipelining)**

✓ Pipeline('|')을 구현한 부분에 대해서 간략히 설명 (design & implementation)

Parseline 함수 내부에서 명령어 라인을 strtok 함수를 사용하여 파이프 기호를 기준으로 분리한다. 각 명령어를 인자 단위로 분리하여 commands[i][j]에 저장한다. 이후 eval 함수로 넘어가면 pipe 개수가 1개 이상일 시 각 프로세스의 ID를 저장할 pid_list와 파이프 파일 디스크립터인 fd를 위해 동적으로 메모리를 할당한다. Fork 전에 pipe를 생성하고 Dup2 함수를 호출하여 표준 입력을 이전 명령의 출력으로, 표준 출력을 다음 명령의 입력으로 설정한다.

✓ Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

Parseline 함수에서 파이프 개수를 확인한다. 파이프가 없다면 단일 명령어로 프로세스를 execvp로 실행한다. Phase2에서는 포그라운드까지만 구현했기에 부모가 하나의 프로세스를 waitpid로 대기한다. 파이프가 있다면 파이프를 기준으로 분리된 프로세스를 별도로 실행한다.

Setpgid(0, 0) 함수를 통해 첫 번째 프로세스가 그룹 리더로 설정하고, 이후 프로세스는 setpgid(0, pid_list[0]) 함수를 통해 같은 그룹으로 들어가도록 한다.

- **Phase3 (background process)**

✓ Background ('&') process를 구현한 부분에 대해서 간략히 설명

우선 parseline함수에서 입력한 명령어 끝에 &가 있다면 bg를 1로 설정한다. eval 함수에서 bg가 1이면 add_job을 호출해 jobs 배열에 bg job을 추가하고, 부모는 waitpid로 대기하지 않고 즉시 프롬프트로 돌아간다. 파이프가 있는

경우에도 동일하게 bg job으로 처리하며, 모든 자식 프로세스를 같은 프로세스 그룹으로 묶어 관리한다.

sigchld_handler

SIGCHLD 시그널을 받아 자식 프로세스의 종료를 처리한다. waitpid(-1, &status, WNOHANG)을 사용해 비동기적으로 종료된 자식 프로세스를 감지한다. WIFEXITED 또는 WIFSIGNALED 케이스에서 delete_job(pid)을 호출해 jobs 배열에서 해당 job을 제거한다.

sigint_handler

SIGINT(Ctrl+C) 신호를 처리한다. 우선 전역 변수 fg_pgid를 확인한 후 포그라운드 프로세스가 존재한다면 kill(-fg_pgid, SIGINT)로 해당 프로세스 그룹에 시그널을 전달해 종료한다.

sigtstp_handler

SIGINT(Ctrl+C) 신호를 처리한다. 전역 변수 fg_pgid를 확인해 포그라운드 프로세스가 존재하면 kill(-fg_pgid, SIGINT)로 해당 프로세스 그룹에 시그널을 전달해 종료시킨다.

C. 개발 방법

- **Phase1 (fork & signal)**

eval 함수: fork()와 execvp()를 사용해 자식 프로세스를 생성하고 명령어를 실행하도록 구현했다.. 자식 프로세스에서는 execvp(commands[0][0], commands[0])를 호출해 입력된 명령어를 실행한다. fg 프로세스만 구현했기에 부모 프로세스에서는 waitpid(pid, &status, WUNTRACED)를 호출해 자식 프로세스가 종료되거나 중지될 때까지 대기하도록 한다.

main 함수: fgets(cmdline, MAXLINE, stdin)를 사용해 사용자로부터 명령어를 입력받고, 입력받은 명령어를 eval(cmdline)으로 전달한다. 또한, char cmdline[MAXLINE] 배열을 선언해 최대 MAXLINE 길이의 명령어를 저장한다.

builtin_command 함수: exit이 입력되었을 경우 strcmp(argv[0], "exit")로 확인 후 exit(0) 호출해 shell을 종료한다. cd가 입력되었을 경우 strcmp(argv[0], "cd")로 확

인. argv[1]이 없거나 "~"면 getenv("HOME")으로 홈 디렉토리를 가져와 chdir(dir)로 디렉토리 변경한다. 그 외의 경우는 일반적인 경우로 chdir(argv[1])로 지정된 디렉토리로 변경한다. 내장 명령어가 아닌 경우 0을 return하여 외부 명령어 처리로 넘긴다.

- **Phase2 (pipelining)**

  전역 변수: pipe_cnt, command_cnt, commands[10][MAXARGS], pgid 추가

  main 함수 추가 내용: clear_commands()를 호출해 명령어 배열(commands)과 카운터(pipe_cnt, command_cnt)를 초기화.

  eval 함수 추가 내용: 수정된 parseline(buf)를 통해 명령어를 파이프 단위로 분리하고, 백그라운드 여부(bg)를 확인한다.

  만약 단일 명령어(pipe_cnt == 0)라면 기존처럼 처리한다. 내장 명령어가 아니면 Fork()로 자식 프로세스 생성하는데 자식의 경우 setpgid(0, 0)으로 본인을 포그라운드 프로세스 그룹 리더로 설정 후 execvp(commands[0][0], commands[0]) 실행한다. 부모의 경우 pgid = pid로 프로세스 그룹 설정 후 Waitpid(pid, &status, WUNTRACED)로 대기한다.

  만약 단일 명령어가 아닌 경우 (파이프 명령어(pipe_cnt > 0)), command_cnt = pipe_cnt + 1로 명령어 개수를 계산한다. pid_list와 fd 배열을 동적 할당하여 프로세스 ID와 fd 초기 설정을 한다. pipe(fd[i])로 각 파이프를 생성하는데, 파싱된 각 명령어에 대해 Fork()로 자식을 생성한다. 첫 번째 프로세스의 경우 setpgid(0, 0)으로 그룹 리더 설정하지만 이후 프로세스는 setpgid(0, pid_list[0])으로 동일 그룹에 포함되도록 한다. 또한, 파이프 관련해서는 dup2(fd[i-1][0], STDIN_FILENO)으로 이전 파이프의 읽기 끝을 표준 입력에 연결하고, dup2(fd[i][1], STDOUT_FILENO)으로 다음 파이프의 쓰기 끝을 표준 출력에 연결한다. 이후, 모든 파이프 파일 디스크립터 닫고 execvp를 실행한다. 부모는 모든 파이프 파일 디스크립터를 닫고, fg일 경우 pgid = pid_list[0] 설정 후 각 프로세스에 대해 Waitpid(pid_list[i], &status, WUNTRACED)로 대기한다.

  parseline 함수 추가 내용: strtok(buf, "|")로 파이프를 기준으로 명령어 분리하고, pipe_cnt 계산한다. 또한, 각 명령어에 대해: 공백 및 따옴표(", ')로 인자를 분리하는데, 따옴표 내 인자는 공백을 유지하며 저장될 수 있도록 한다. 일반 인자는 공백으로 분리해 commands[i][argc++]에 저장한다. 또한, 마지막 명령어에서 &가

단독이거나 마지막 인자의 끝이 &면 bg = 1 설정 후 제거한다.

- **Phase3 (background process)**

typedef struct job {
  int jid; /* job id [1] [2] 이런거*/
  pid_t pid; /* process ID */
  pid_t pgid; /* process group ID */
  char cmdline[MAXLINE];
  int state;
  int is_bg;
} job;
job_t jobs[MAXJOBS];
	- pid, pgid: 프로세스 ID와 그룹 ID 저장.
	- jid: Job ID
	- state:
	JOB_STATE_RUNNING,
	JOB_STATE_STOPPED,
	JOB_STATE_UNDEF로 상태 관리.
	- cmdline: 명령어 문자열 저장.
	is_bg:
	JOB_BG_BG,
	JOB_BG_FG,
	JOB_BG_UNDEF로 fg/bg 구분.

main 함수 추가 내용: signal(SIGCHLD, sigchld_handler), signal(SIGINT, sigint_handler), signal(SIGTSTP, sigtstp_handler) 시그널 핸들러를 등록했다. 또한, init_jobs() 호출로 jobs 배열을 초기화했다.

eval 함수 추가 내용: bg 처리에 관하여 단일 명령어일 때 부모는 add_job(pid, pid, JOB_STATE_RUNNING, cmdline, JOB_BG_BG)로 jobs 배열에 추가한다. Pipe를 사용해서 단일 명령어가 아닌 경우, add_job(pid_list[0], pid_list[0], JOB_STATE_RUNNING, cmdline, JOB_BG_BG)로 job 추가한다.

builtin_command 함수 추가 내용:

jobs가 입력되었을 경우 print_jobs() 호출로 jobs 배열을 출력하고, fg의 경우 to_fg(argv) 호출로 지정된 job을 포그라운드로 전환하고, bg의 경우 to_bg(argv) 호출로 중지된 job을 백그라운드로 재개한다. kill의 경우 do_kill(argv) 호출로 지정된 job을 종료한다. 마지막으로 exit의 경우 SIGCHLD 차단 후 jobs 배열을 순회하며 kill(-jobs[i].pid, SIGKILL)로 bg job을 종료하고 delete_job(jobs[i].pid)으로 정리 후 끝낸다.

sigchld_handler:

waitpid(-1, &status, WNOHANG)으로 종료된 자식 프로세스를 감지한다. WIFEXITED 또는 WIFSIGNALED의 경우 delete_job(pid)으로 jobs 배열에서 제거한

다.

sigint_handler: fg_pgid != 0이면 즉. Foreground process가 있을 경우 kill(-fg_pgid, SIGINT)로 fg 프로세스 그룹 종료한다.

sigtstp_handler: fg_pgid != 0이면 즉. Foreground process가 있을 경우 kill(-fg_pgid, SIGTSTP)로 fg 프로세스 그룹 중지한다. 또한, add_job(fg_pgid, fg_pgid, JOB_STATE_STOPPED, cmdline, JOB_BG_FG)으로 jobs 배열에 추가하고, fg_pgid는 0으로 설정한다.

job 관련 함수

init_jobs: jobs 배열을 JOB_STATE_UNDEF로 초기화한다.

add_job(pid, pgid, state, cmdline, is_bg): jobs 배열에 새 job을 추가한다.

delete_job(pid): pid에 해당하는 job 제거, 상태를 JOB_STATE_UNDEF로 설정한다.

print_jobs: jobs 배열의 실행 또는 중지된 job들을 출력한다.

to_fg(argv): job을 포그라운드로 전환, kill(-jobs[i].pid, SIGCONT)로 재개 후 Waitpid로 부모는 기다린다.

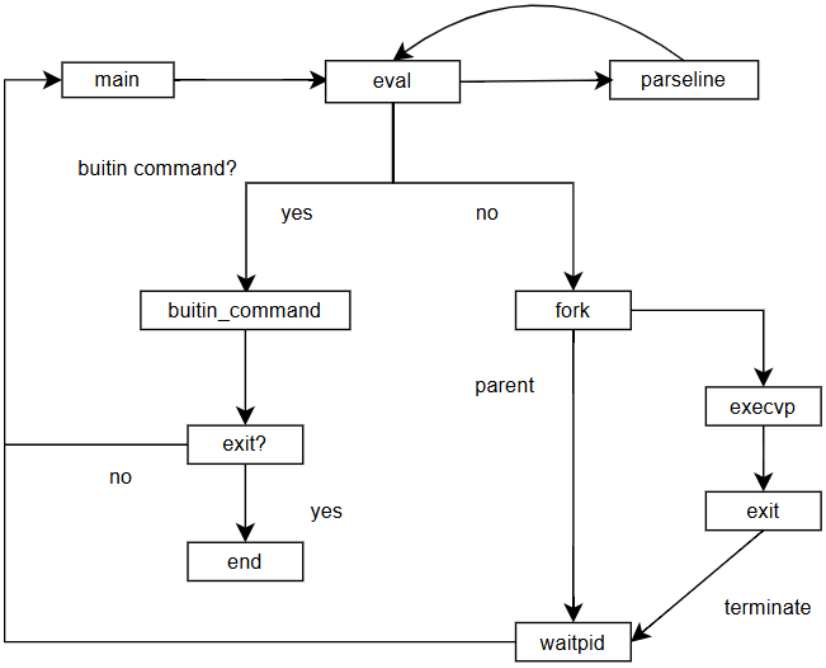
to_bg(argv): job을 백그라운드로 재개한다. kill(-jobs[i].pid, SIGCONT)로 신호를 준다.

do_kill(argv): %jid로 지정된 job 종료한다. kill(-jobs[i].pid, SIGTERM)로 신호를 준다.

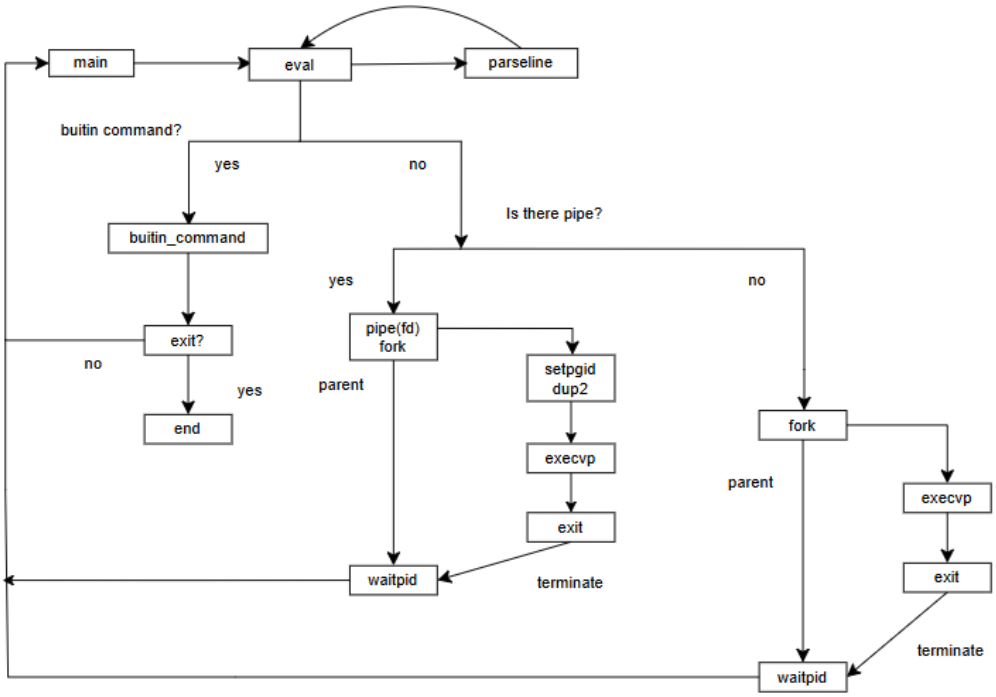
3. 구현 결과

A. Flow Chart

1. Phase 1 (fork)



2. Phase 2 (pipeline)



3. Phase 3 (background)