System Programming Project 4

담당 교수 : 김영재

이름 : 조보현

학번 : 20160641

1. 개발 목표

이번 프로젝트에서는 쉘의 구현을 목표로 한다. 기본적인 명령어의 실행과 더불어 pipeline을 이용한 명령어 실행, background 명령어 실행을 할 수 있다. signal handler를 통한 signal 처리 또한 익힐 수 있다.

2. 개발 범위 및 내용

A. 개발 범위

1. Phase 1

   - simple shell 구현. 간단한 명령어들의 수행이 가능하다. 명령어의 수행은 built-in command를 제외하고는 fork() 명령을 통해 자식 프로세스에서 수행된다.

2. Phase 2

   - pipeline을 이용한 piping 기능을 수행할 수 있다. 어떠한 명령어의 output이 다음 명령의 input이 될 수 있다.

3. Phase 3

   - background 명령을 수행 할 수 있다. background 명령을 진행하면서 foreground 명령도 진행 할 수 있다. signal 처리를 적절히 하여 shell 에 오류가 발생하지 않도록 한다.

B. 개발 내용

- **Phase1 (fork & signal)**

  ✓ 우선 built-in command(cd, exit 등) 을 제외하고는 모두 fork()를 통해 child process 를 생성한다. pid = Fork() 명령을 통해 pid를 받아 0 일때는(자식 프로세스) 사용자로부터 받은 명령을 수행하고, pid != 0 일때는(부모 프로세스) 자식프로세스가 종료되기를 기다린다.

  ✓ fork()를 통해 생성된 자식 프로세스가 사용자가 입력한 명령어를 실행하는 동안 부모 프로세스는 waitpid()를 통해 자식프로세스가 종료되기를 기다린다. 이 때 waitpid(pid, &status, 0) 을 통해 특정 pid를 가진 프로세스(자식프로세

스)가 종료될 때까지 기다리게 된다. 이후 자식 프로세스가 종료되면 SIGCHLD signal을 부모에게 전달하게 되고 종료된다.

- **Phase2 (pipelining)**

  ✓ pipeline 은 pipe() 함수와 fork()를 통해 구현했다. 우선 fork()를 통해 자식 프로세스를 2개 생성하여 첫번째 명령어와 두번째 명령어를 순서대로 수행한다. 여기서 pipe() 함수를 통해 file descriptor를 pipe처럼 이용하게 된다. 이때 file descriptor를 통해 첫번째로 실행되는 명령어는 input file descriptor(fd[0])를 닫고, output file descriptor(fd[1]) 를 열어 수행된 결과를 보낸다. 이후 다음으로 수행되는 명령어는 input file descriptor(fd[0])를 열고 이전 명령어로부터 받은 output을 input으로 사용하게 된다. 부모 프로세스는 해당 자식 프로세스들이 종료될 때까지 기다린다.

  ✓ pipeline이 하나 일 때는 위와 같은 과정을 거치고, 2개일때는 pipe()함수를 두 번 호출하고, 자식 프로세스는 fork()를 세 번 호출하여 3개를 만든다. 또한 file descriptor 를 2개 만들어 첫번째 명령어-두번째 명령어의 pipe, 두번째 명령어-세번째 명령어의 pipe 로 이용하게 된다. 여기서 한계점은 pipe를 최대 2개까지밖에 이용할 수 없다는 점이다.

- **Phase3 (background process)**

  ✓ Background ('&') process를 구현한 부분에 대해서 간략히 설명. 우선 main 함수에서 SIGCHLD, SIGINT, SIGTSTP 신호를 받을 경우 구현한 handler가 handle 하게 설정한다. 사용자의 입력에 & 가 마지막에 올 경우 background로 명령을 수행하는 것으로 간주하고 자식 프로세스가 명령을 수행하고 종료되지 않았더라도 부모 프로세스는 기다리지 않고 다음 명령을 받게 된다. 이는 기존 waitpid() 과정에서 option을 waitpid(pid, &status, WNOHANG)으로 설정하여 구현하였다. 또한 foreground 작업을 수행중 ctrl+z 를 입력해 SIGTSTP 이 발생한다면 해당 작업을 Stopped 상태로 바꾼다. 이후 bg 명령을 통해 다시 명령을 재개할 수 있다.

C. 개발 방법

- **Phase 1**

  void eval(char* cmdline)

  사용자의 명령을 input으로 받아 수행한다. parseline()함수를 통해 명령어를 적절하게 parsing 하여 eval()함수에서 수행한다. 기존 eval() 함수에서 fork()함수를 이용해 자식 프로세스인 경우에 명령을 수행하도록 수정한다. built-in 명령어로 exit 과 cd 를 구현한다. exit 의 경우 쉘을 종료하도록 하고 cd 의 경우 chdir() 를 이용해 현재 디렉토리를 변경하게 된다.

- **Phase 2 (pipeline)**

  char* pipe_arg1[MAXARGS] : pipeline의 첫번째 명령어들을 저장

  char* pipe_arg2[MAXARGS] : pipeline의 두번째 명령어들을 저장

  char* pipe_arg2[MAXARGS] : pipeline의 세번째 명령어들을 저장

  int num, bp1, bp2 : pipeline을 parsing 할 때 사용되는 변수

  int mode : mode == 1 – 파이프라인 없이 수행되는 모드, mode == 2 – 파이프라인 모드

  int pipe_cnt : pipe('|') 의 개수를 저장하는 변수

  void parseline(char* buf, char argv, int* argc)**

  기존 parseline() 함수에서 '|' 를 처리해주는 구문을 추가한다. 만약 '|' 가 한 개라면 pipe_cnt를 1로, 두 개라면 2로 설정한다. '|' 를 기준으로 명령어를 parsing 하여 적절한 pipe_arg 배열에 넣어준다.

  void eval(char* cmdline)

  기존 eval() 함수에서 파이프라인 모드를 처리해주는 구문을 추가한다. mode 변수를 통해 normal모드와 pipeline 모드를 구별해 수행한다. 만약 pipeline 모드일 경우에는 do_pipe() 함수를 통해 pipeline 모드로 명령을 수행한다.

  void do_pipe(char* arg1[], char* arg2[], char* arg3[], int bg, char* cmd)

  파이프 모드는 우선 fork()를 통해 자식 프로세스를 2개 생성하여 첫번째 명령어와 두번째 명령어를 순서대로 수행한다. 여기서 pipe() 함수를 통해 file descriptor

를 pipe처럼 이용하게 된다. 이 때 file descriptor를 통해 첫번째로 실행되는 명령어는 input file descriptor(fd[0])를 닫고, output file descriptor(fd[1]) 를 열어 수행된 결과를 보낸다. 이후 다음으로 수행되는 명령어는 input file descriptor(fd[0])를 열고 이전 명령어로부터 받은 output을 input으로 사용하게 된다. 부모 프로세스는 해당 자식 프로세스들이 종료될 때까지 기다린다. pipeline이 하나 일 때는 위와 같은 과정을 거치고, 2개일때는 pipe()함수를 두 번 호출하고, 자식 프로세스는 fork()를 세 번 호출하여 3개를 만든다. 또한 file descriptor 를 2개 만들어 첫번째 명령어-두번째 명령어의 pipe, 두번째 명령어-세번째 명령어의 pipe 로 이용하게 된다.

- **Phase 3 (background)**

```
typedef struct _Node{
    int job_id;
    pid_t pid;
    char status[20];
    int state;
    char cmd[100];
    struct _Node* next;
    struct _Node* prev;
}Job;
```

구조체 Job : 현재 수행되고 있는 명령어들을 저장하기 위한 구조체. job_id, pid, status(running, stopped, terminated, done), state(1-foreground, 2-background), cmd(명령어)를 멤버로 갖는다.

void eval(char* cmdline)

built-in 명령어로 jobs, kill, bg를 수행한다. jobs는 현재 background 에서 수행되고 있는 명령어들을 출력한다. 이는 print_jobs() 함수를 이용한다. kill 명령은 해당 job_id를 가지고 있는 작업을 terminated 시킨다. 이는 kill()함수를 통해 SIGINT signal을 발생시켜 구현한다. bg 명령은 해당 job_id를 가지고 있는 중단된 background 작업을 재개하게 한다. 이는 kill()함수를 통해 SIGCONT signal을 발생시켜 구현한다. 여기서 여러가지 handler를 통해 signal을 관리하는데 자세한 사항은 아래에서 서술한다.

void sigchild_handler(int sig)

SIGCHLD signal 을 받을 경우 수행되는 handler. waitpid() 함수를 통해 자식 프로세스의 pid를 구해 수신한 signal 에 맞게 작업을 수행한다. WIFSIGNALED(status)를 통해 자식 프로세스가 시그널에 의해 종료되었을 경우 해당 job의 상태를 terminated로 바꿔주고, WIFSTOPPED(status)을 통해 자식프로세스가 시그널에 의해 중단되었을 경우 해당 job의 상태를 stopped로 바꿔준다. 혹은 정상적으로 종료된 경우에는 해당 job의 상태를 done으로 바꿔준다.

void sigint_handler(int sig)

ctrl+c 입력을 수신할 경우 해당 프로세스를 종료한다. main의 경우에는 해당 시그널을 무시하고 줄바꿈만 한다.

void sigtstp_handler(int sig)

ctrl+z 입력을 수신할 경우 해당 프로세스를 중단한다. 해당 pid 를 getpid_usingstate()함수를 통해 구해 kill() 함수를 통해 SIGTSTP 시그널을 보낸다.

void change_job_state(pid_t pid, int stat)

파라미터로 받은 pid를 통해 joblist 에서 해당 pid를 가진 job의 상태를 stat 에 맞게 바꿔준다. stat 이 1인 경우 해당 job의 status를 "Done"으로 바꿔준다. 2인 경우 "Terminated", 3인 경우 "Stopped", 4인 경우 "Running" 으로 바꿔준다.

void addjob(pid_t pid, char* cmd, int state)

새로운 job을 joblist에 추가해준다. 기본값을 설정하고 사용자로부터 받은 입력들을 이용해 새로운 newNode를 만들어 기존 joblist에 이어붙이는 식으로 구현한다.

void print_done_list()

joblist의 job들 중 "Done", "Terminated", "Stopped" 상태인 job들을 출력한다.

void print_jobs()

joblist의 모든 job들을 화면에 출력한다.

void deletejob()

joblist의 job들 중 "Done", "Terminated", "Stopped" 상태인 job들을 제거한다.

int getjid(pid_t pid)

pid를 통해 해당 job의 job_id를 반환한다.

pid_t getpid_usingjid(int jid)

job_id를 통해 해당 job의 pid를 반환한다.

pid_t getpid_usingstate()

foreground job들중 가장 최근의 job의 pid를 반환한다.

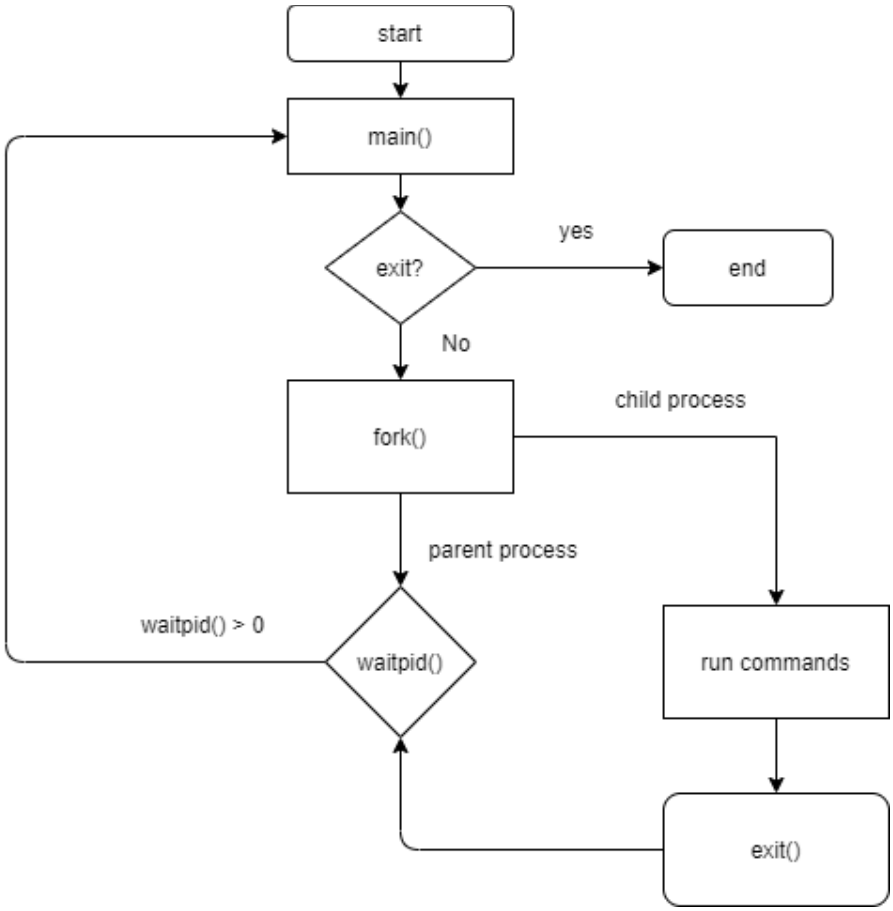
char* getcmd (pid_t pid)

pid를 통해 해당 job의 cmd를 반환한다.

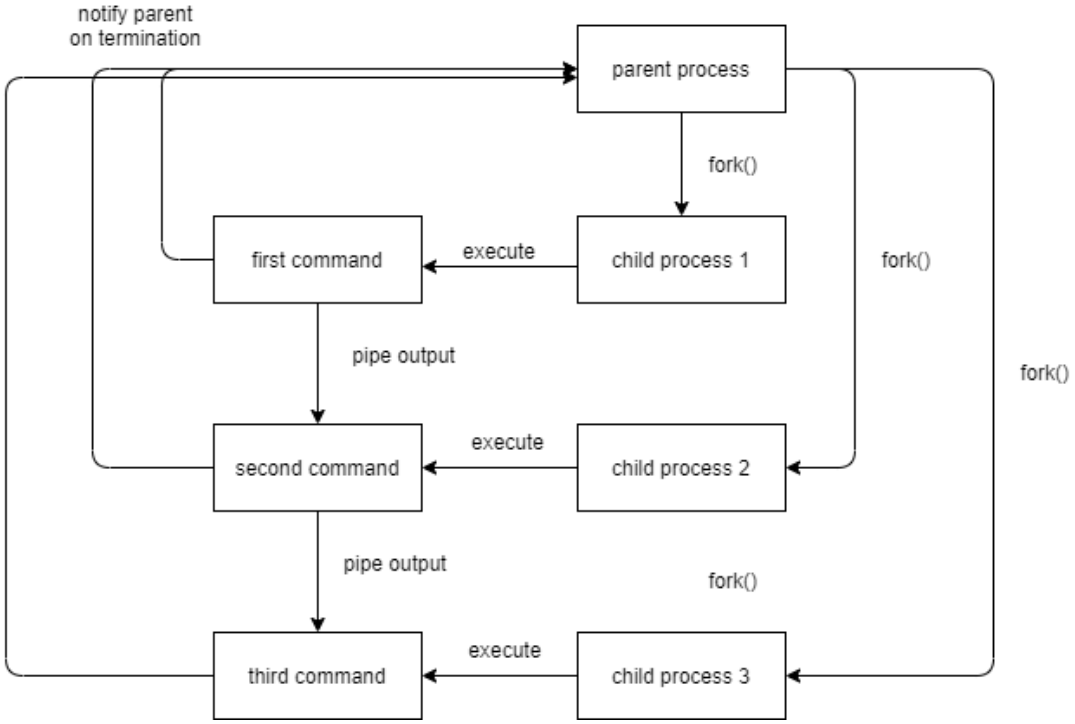
3. 구현 결과

A. Flow Chart

1. Phase 1 (fork)



2. Phase 2 (pipeline)



3. Phase 3 (background)