# Cykor 2

2020190627 박민용

## Cond 1

1) code

```
#include <stdio.h>
#include <libgen.h>
경로에서 파일명을 추출하는 basename을 위해 사용

#include <limits.h>
시스템 한계값 상수, PATH_MAX를 위해 사용

#include <unistd.h>
POSIX system call libarary, getcwd를 위해 사용

int main() {
    char cwd[PATH_MAX];

    while(1) {
        getcwd(cwd, sizeof(cwd));
        printf("[%s]$ ", basename(cwd));
        fflush(stdout);

        char line[1024];
        if(!fgets(line, sizeof(line), stdin))
            break;
    }

    return 0;
}
```

2) 설명

```
char cwd[PATH_MAX];
cwd array를 PATH_MAX만큼 할당

getcwd(cwd, sizeof(cwd));
cwd에 현재 작업 directory 전체 경로 저장

printf("[%s]$ ", basename(cwd));
```

basename(cwd)로 경로의 마지막 component 추출 후 출력

```
fflush(stdout);
```

buffer에 잠시 존재할 수 있는 출력 데이터 out

```
char line[1024];

if(!fgets(line, sizeof(line), stdin))
    break;
```

line array에 명령 입력받아 저장

## Cond 2

1) 추가 libarary

```
#define _GNU_SOURCE
```

GNU 확장 사용

```
#include <stdlib.h>
```

exit, malloc, free 사용

```
#include <string.h>
```

문자열 처리 사용

```
#include <sys/wait.h>
```

자식 프로세스 종료 대기 사용

```
#define MAX_ARGS 128
```

최대 인자 define

2) code

```
int main() {
    char cwd[PATH_MAX];
    char *line=NULL;
    size_t len=0;

    while(1) {
        getcwd(cwd, sizeof(cwd));
        printf("[%s]$ ", basename(cwd));
        fflush(stdout);

        if(getline(&line, &len, stdin)==-1)
            break;

        if(strspn(line," ₩t₩r₩n")==strlen(line))
            continue;

        char *args[MAX_ARGS];
        int arg_count=0;
        char *tok=strtok(line, " ₩t₩n");
        while(tok&&arg_count<MAX_ARGS-1) {
            args[arg_count++]=tok;
            tok=strtok(NULL, " ₩t₩n");
        }
```

```
            args[arg_count]=NULL;
            if(arg_count==0)
                continue;

            pid_t pid=fork();
            if(pid==0) {
                execvp(args[0], args);
                perror("execvp");
                exit(127);
            }

            else {
                int status;
                waitpid(pid, &status, 0);
            }
        }

        free(line);
        return 0;
}
```

3) 추가 code 설명

```
char *line=NULL;
```

getline이 realloc

```
size_t len=0;
```

line buffer 길이 전달

```
if(strspn(line," ₩t₩r₩n")==strlen(line))
        continue;
```

strspn으로 공백, 탭, 개행 길이를 재서 빈 줄 무시

```
char *args[MAX_ARGS];
int arg_count=0;
```

args array를 MAX_ARGS만큼 할당

```
char *tok=strtok(line, " ₩t₩n");
```

strtok를 이용해 공백, 탭, 개행 문자 기준으로 구별 후 저장

```
while(tok&&arg_count<MAX_ARGS-1) {
```

tok이 NULL이 아니며 인자 수가 MAX_ARGS-1에 도달하지 않았으면 반복

```
args[arg_count++]=tok;
```
현재 토큰을 args에 저장, arg_count 1만큼 증가

```
tok=strtok(NULL, " ₩t₩n");
```
tok에 다음 token 저장

```
args[arg_count]=NULL;
```
execvp()는 인자 배열의 마지막이 NULL로 끝나야 함(C 표준 exec계열 함수 요구사항!!)

```
if(arg_count==0)
    continue;
```
만약 공백만 입력되었다면 다시 반복문으로

```
pid_t pid=fork();
```
부모는 자식 PID, 자식은 0을 return

```
if(pid==0) {
```
pid==0 즉, 자식 process일 때

```
execvp(args[0], args);
```
프로그램 이름으로 execvp

```
perror("execvp");
```
perror까지 도달했다면 실패

```
exit(127);
```
리눅스에서 명령어가 존재하지 않을 경우 실패 코드 127 사용

```
else {
```
부모 proccess일 때

```
int status;
waitpid(pid, &status, 0);
```
자식이 끝날 때까지 block

```
free(line);
```
line buffer 해제

## Cond 3 & 7

1) 추가 libarary

```
#include <errno.h>
```

2) code

```
int builtin(char **args) {
    if(strcmp(args[0], "cd")==0) {
        const char *dest = args[1]?args[1]:getenv("HOME");
        if(chdir(dest)==-1)
            perror("cd");
        return 0;
    }

    if(strcmp(args[0], "pwd")==0) {
        char cwd[PATH_MAX];
        if(getcwd(cwd, sizeof(cwd)))
            puts(cwd);
        else
            perror("pwd");
        return 0;
    }

    if(strcmp(args[0], "exit")==0) {
        exit(0);
    }

    return -1;
}
```

3) 추가 code 설명

```
int builtin(char **args)
```

input 명령어가 cd, pwd, exit인지 확인하고 실행하는 함수

```
if(strcmp(args[0], "cd")==0) {
```

첫 번째 인자가 cd인지 확인

```
const char *dest=args[1]?args[1]:getenv("HOME");
```

cd명령의 인자가 있다면 args[1], destination으로 사용
없다면 HOME으로 이동

```
if(chdir(dest)==-1)
```

현재 directory를 destination으로 바꿈
perror("cd");
실패하면 오류 메시지 출력

if(strcmp(args[0], "pwd")==0) {
첫 번째 인자가 pwd인지 확인

char cwd[PATH_MAX];
cwd array를 PATH_MAX만큼 할당

if(getcwd(cwd, sizeof(cwd)))
    puts(cwd);
성공 시 출력, cwd에 현재 작업 directory 전체 경로 저장

else
    perror("pwd");
실패 시 에러 메시지 출력

if(strcmp(args[0], "exit")==0) {
첫 번째 인자가 exit인지 확인

exit(0);
프로그램 종료

return -1;
cd, pwd, exit에 해당하지 않으면 -1을 return

## Cond 4

1) 추가 define

```
#define MAX_PIPES 32
```

2) code

```
int run_pipeline(char *cmdline) {
    char *seg_save,
        char *segment=strtok_r(cmdline, "|", &seg_save);
    int input_fd=STDIN_FILENO;
    pid_t pids[MAX_PIPES];
    int pid_count=0;

    while(segment) {
        char *next=strtok_r(NULL, "|", &seg_save);
        int fds[2];
        int output_fd=STDOUT_FILENO;

                if(next) {
            if(pipe(fds)==-1) {
                                perror("pipe");
                                    return 1;
                            }

            output_fd=fds[1];
        }

        char *args[MAX_ARGS];
                int arg_count=0;
        char *tok=strtok(segment, " ₩t₩n");

            while(tok && arg_count<MAX_ARGS-1) {
            args[arg_count++]=tok;
            tok=strtok(NULL, " ₩t₩n");
        }

                args[arg_count]=NULL;

                if(arg_count==0) {
            segment=next;
                            continue;
        }
```

```
    if (builtin(args) == 0) {
        if(next || input_fd!=STDIN_FILENO)
            fprintf(stderr, "myshell: built-in commands cannot be piped₩n");
    }

            else {
        pid_t pid=fork();
        if(pid==0) {
            if(input_fd!=STDIN_FILENO)
                dup2(input_fd,  STDIN_FILENO);

            if(output_fd != STDOUT_FILENO)
                                    dup2(output_fd, STDOUT_FILENO);

            if(next)
                                    close(fds[0]);

            execvp(args[0], args);
            perror("execvp");
                            _exit(127);
        }

                    else if(pid>0)
            pids[pid_count++]=pid;

                    else
            perror("fork");
    }

    if(input_fd!=STDIN_FILENO)
        close(input_fd);

            if(output_fd!=STDOUT_FILENO)
                    close(output_fd);

    input_fd=next?fds[0]:STDIN_FILENO;
    segment=next;
}

for(int i=0; i<pid_count; ++i)
```

```
        int status; waitpid(pids[i], &status, 0);

    return 0;
}
```

3) code 설명

```
int run_pipeline(char *cmdline) {
```

cmdline에 들어온 명령줄을 파이프 단위로 잘라 실행하는 함수

```
char *seg_save;
```

내부 상태 저장을 위한 seg_save pointer

```
*segment=strtok_r(cmdline, "|", &seg_save);
```

strok_r을 사용해 cmdline을 첫 번째 | 기준으로 자름

```
int input_fd=STDIN_FILENO;
```

현재 프로세스의 표준입력 대신 스크립터 보관

```
pid_t pids[MAX_PIPES];
```

자식 process pid 저장을 위해 pids array를 MAX_PIPES만큼 할당

```
int pid_count=0;
```

개수 count

```
while(segment) {
```

segment=0이 될 때까지 반복

```
char *next=strtok_r(NULL, "|", &seg_save);
```

다음 segment를 next pointer에 저장

```
int fds[2];
```

새 pipe를 위한 fds[2] 준비 (fds[0]은 읽기용, fds[1]은 쓰기용)

```
int output_fd=STDOUT_FILENO;
```

기본 출력은 표준출력으로 설정

```
if(next) {
```

만약 next가 존재한다면 다시 pipe를 만든다

```
if(pipe(fds)==-1) {
    perror("pipe");
```

```
    return 1;
}
```
pipd(fds)가 실패했다면 오류 메시지, 함수 종료

```
output_fd=fds[1];
```
쓰기용 FD를 현재 출력으로 설정

```
char *args[MAX_ARGS];
int arg_count=0;
```
args array를 MAX_ARGS만큼 할당

```
char *tok=strtok(segment, " ₩t₩n");
```
strtok를 이용해 공백, 탭, 개행 문자 기준으로 구별 후 저장

```
while(tok&&arg_count<MAX_ARGS-1) {
```
tok이 NULL이 아니며 인자 수가 MAX_ARGS-1에 도달하지 않았으면 반복

```
args[arg_count++]=tok;
```
현재 토큰을 args에 저장, arg_count 1만큼 증가

```
tok=strtok(NULL, " ₩t₩n");
```
tok에 다음 token 저장

```
args[arg_count]=NULL;
```
execvp()는 인자 배열의 마지막이 NULL로 끝나야 함(C 표준 exec계열 함수 요구사항!!)

```
if(arg_count==0) {
    segment=next;
    continue;
}
```
만약 공백만 입력되었다면 segment=next, 다시 반복문으로

```
if(builtin(args)==0) {
```
cd, pwd, exit과 같은 내장 명령이면

```
if(next||input_fd!=STDIN_FILENO)
```
pipe 내에서 내장을 쓰는 경우

```
fprintf(stderr, "myshell: built-in commands cannot be piped₩n");
```
경고 출력

```
else {
    pid_t pid=fork();
외부 명령이면 새로운 process를 fork

if(pid==0) {
자식 process면

if(input_fd!=STDIN_FILENO)
    dup2(input_fd,  STDIN_FILENO);
input_fd가 표준 입력이 아니라면 dup2로 읽는 쪽을 표준 입력으로 연결

if(output_fd != STDOUT_FILENO)
    dup2(output_fd, STDOUT_FILENO);
input_fd가 표준 출력이 아니라면 dup2로 읽는 쪽을 표준 출력으로 연결

if(next)
    close(fds[0]);
자식을 읽는 쪽 fds[0]을 닫아야 함

execvp(args[0], args);
명령어 execvp 실행

perror("execvp");
_exit(127);
명령이 없거나 실패, 에러 메시지 출력, exit(127)

else if(pid>0)
    pids[pid_count++]=pid;
부모 process라면 fork한 자식 PID를 배열에 저장

else
    perror("fork");
fork가 실패하면 시스템 에러 출력

if(input_fd!=STDIN_FILENO)
    close(input_fd);
사용 끝난 입력 FD close

if(output_fd!=STDOUT_FILENO)
    close(output_fd);
사용 끝난 출력 FD close
```

```
input_fd=next?fds[0]:STDIN_FILENO;
segment=next;
```
만약 다음 segment가 있다면 fd[0]을 다음 명령어의 stdin으로 연결할 준비

```
for(int i=0; i<pid_count; ++i)
    int status;
    waitpid(pids[i], &status, 0);
```
모든 자식 process wait

## Cond 5

1) code

```
int exec_line(char *line) {
        int last_status=0;
        enum{NONE,AND,OR} prev=NONE;

        char *p=line;

        while(*p){
                while(*p==' '||*p=='₩t')
                        ++p;

                if(!*p||*p=='₩n')
                        break;

                char *cmd_start=p;
                enum{END,SC,SA,SO} sep=END;

                while(*p && *p!='₩n') {
                        if(p[0]==';'){
                                sep=SC;
                                break;
                        }

                        if(p[0]=='&'&&p[1]=='&'){
                                sep=SA;
                                break;
                        }

                        if(p[0]=='|'&&p[1]=='|'){
                                sep=SO;
                                break;
                        }

                        ++p;
                }

                char *cmd_end=p;

                if(*p) {
                        *cmd_end='₩0';
```

```
                        if(sep==SA||sep==SO)
                                p+=2;

                        else
                                p+=1;
                }

                int exec=1;

                if(prev==AND && last_status!=0)
                        exec=0;
                if(prev==OR  && last_status==0)
                        exec=0;

                if(exec)
                        last_status=run_pipeline(cmd_start);

                prev=(sep==SA)?AND:(sep==SO)?OR:NONE;
        }

        return last_status;
}
```

2) code 설명

```
int exec_line(char *line)
```

line을 받아 exec_line 시작

```
int last_status=0;
```

마지막 명령어 종료 상태 성공으로 설정

```
enum{NONE,AND,OR} prev=NONE;
```

prev는 직전 구분자가 무엇이었는지 기억

```
char *p=line;
```

p는 입력받은 문자열을 읽을 pointer

```
while(*p) {
```

문자열 끝까지 반복

```
while(*p==' '||*p=='₩t')
```

```
    ++p;
```
공백이거나 tap이면 한 칸 건너뜀

```
if(!*p||*p=='₩n')
    break;
```
끝이면 break

```
char *cmd_start=p;
```
현재 명령어의 시작 위치 저장

```
enum{END, SC, SA, SO} sep=END;
```
현재 명령어가 끝난 후 어떤 구분자를 만났는지 기억할 변수

```
while(*p && *p!='₩n') {
```
*p가 존재하며 줄바꿈이 아닐 때까지

```
if(p[0]==';' ) {
    sep=SC;
    break;
}
```
;를 만나면 SC

```
if(p[0]=='&'&&p[1]=='&') {
    sep=SA;
    break;
}
```
&&를 만나면 SA

```
if(p[0]=='|'&&p[1]=='|') {
    sep=SO;
    break;
}
```
||을 만나면 SO

```
++p;
```
한 글자씩 p 진행

```
char *cmd_end=p;
```
현재 위치를 끝으로 저장

```
if(*p) {
```

```
    *cmd_end='₩0';
```
지금 위치에 '₩0'을 넣어서 명령어를 문자열 끝으로 만듦

```
if(sep==SA||sep==SO)
    p+=2;
```
&&, ||는 2글자니까 2칸 이동

```
else
    p+=1;
```
;는 1칸 이동

```
int exec=1;
```
기본은 1, 실행하겠다라는 의미

```
if(prev==AND && last_status!=0)
    exec=0;
```
직전 &&였는데 실패면 실행 안함

```
if(prev==OR  && last_status==0)
    exec=0;
```
직전 ||였는데 성공이면 실행 안함

```
if(exec)
    last_status=run_pipeline(cmd_start);
```
만약 실행해야 하면 pipeline 실행

```
prev=(sep==SA)?AND:(sep==SO)?OR:NONE;
```
prev 저장

```
return last_status;
```
마지막 상태 return

## Cond 6

```
int run_pipeline(char *cmdline, int bg) {
run pipeline에 bg 매개변수 추가

if(!bg) {
bg가 background 실행이 아닐 때

for(int i=0; i<pid_count; i++) {
모든 자식 process들을

int st;
waitpid(pids[i], &st, 0);
하나씩 대기

if(i==pid_count-1)
last_status=WIFEXITED(st)?WEXITSTATUS(st):1;
마지막 process의 종료 code를 last_status에 저장

else
background라면

printf("[bg pid %d]₩n", pids[pid_count-1]);
pipeline의 마지막 자식 process의 PID 출력

while(*p) {
    int bg=0;
cmd마다 bg=0으로 초기화

if(exec)
    last_status=run_pipeline(cmd_start, bg);
run_pipeline 호출 시 bg 추가
```

## Code Review

builtin은 cd, pwd, exit와 같은 내장 명령어를 처리하는 함수로
성공하면 0, 외부 명령이면 -1을 반환하도록 하였다.
만약 명령어가 cd라면 args[1]로, 만약 args[1]이 없다면 HOME으로 이동!
chdir(dest)를 통해 현재 process의 작업 directory 변경
만약 명령어가 pwd라면 (현재 directory 출력)
cwd라는 array를 만든 후 getcwd로 directory 경로 저장
만약 명령어가 exit라면
프로그램 자체 종료 exit(0) 호출

**즉, builtin은 case by case로 cd, pwd, exit 내장명령 실행**

run_pipeline은 파이프가 있는 명령어들을 실행하는 함수로
|를 기준으로 첫 번째 segment를 가져온다.
pipe의 정의에 따라 input_fd를 설정하는데, 처음은 표준 입력으로 정의
pids 배열로 자식 process PID, pid_count로 process 수 저장
last_status로 마지막 상태를 저장
while을 통해 segment가 남아 있으면 계속 처리
next를 미리 할당, fds[2] 및 output_fd를 미리 설정해 놓음
다음 명령어가 있으면 pipe를 만든다.
output_fd=fds[1]로 설정, 그러면 next는 fds[0]을 읽을 수 있음
이 부분이 헷갈렸는데, pipe(fds)를 하면 fds[1]에 쓰고 다음 next가 fds[0]으로 읽을 수 있음
이후 args 배열에 인자들 저장, arg_count에 개수 저장
공백, 탭, 줄바꿈 기준으로 인자들 분리 및 저장(마지막에 NULL을 저장해야 함)
빈 명령어면 넘어가고, 내장 명령어면 builtin을 이용해 처리
내장명령인데 pipe를 쓰면 경고(gpt를 통해 인지)
외부 명령어면 따로 실행해야 하므로 fork를 통해 자식 process 만듦
자식 process임을 확인하고
input_fd가 기본 입력 형식이 아니면 표준 입력을 input_fd로
output_fd가 기본 출력 형식이 아니면 표준 출력을 in_oud로
다음 명령어가 있다면 fds[0]은 필요 없으니 close
fds[0]은 다음 process가 사용하기 때문
이후 자식 process 명령어 실행
부모 process임을 확인하고
fork한 자식 PID를 pids 배열에 저장
임시 pipe fd를 썼으면 닫아야 함
다음 명령어가 있다면 next의 입력을 fds[0]으로 연결
background면 기다리지 않고 PID 출력
background아니면 기다림

**즉, run_pipeline은 cmdline을 받아서 segment를 처리하는 함수이며**

**segment가 없을 때까지 pipe를 만들어 연결하는데**
**각 segment를 argument 단위로 쪼개 내장 함수, 외장 함수로 나눠 처리한다고 요약할 수 있다.**

exec_line은 line을 입력받아서
last_status로 종료 상태 저장하는 변수, prev를 통해 이전 명령어 조건자 저장 변수 설정
이후 포인터를 돌면서 명령어 시작 지점 기록 후
;를 만나면 sep=SC 설정, break
&&를 만나면 sep=SA 설정, break
||를 만나면 sep=SO설정, break
이후 그 자리에 ₩0을 넣어 문자열을 끊고
&&, ||은 p+=2, ;는 p+=1을 한다.
명령어 끝에 남아있을 수 있는 공백이나 tap을 제거한다.
명령어 길이를 계산해서 마지막 명령어가 &면 bg=1로 설정한다.
이후 명령어를 실행할 예정으로 초기화 exec=1
이후 조건자 &&와 ||의 조건에 따라 실행할 여부 결정
이후 실행 필요하면 run_pipeline 호출

**즉, exec_line은 line을 입력받아서 명령어를 구분자로 나눈 후 background 실행 여부를 판단해 run_pipeline을 호출**

## 보안 고려 사항 분석

1. 현재 코드는 Background 실행 bg==1인 경우 waitpid를 호출하지 않고 있기 때문에 ppt에서 언급했듯이 zombie process가 발생할 수 있다. 컴퓨터학과의 운영체제 강의를 수강하면서 zombie process와 orphan process의 해결 방안에 대해 접한 적 있는데, 이 경우에서는 Background 실행 후에도 자식 process를 적절히 waitpid하는 방식으로 회수하는 방식으로 해결할 수 있을 것이다.

2. 명령어 buffer overflow 위험

CyKor에서 공부를 하면서 느낀 점이 하나 있다. 일반적인 알고리즘 문제에서는 작동에 크게 문제가 되지 않는다고 넘어갈 수 있던 일들이 보안 부분에서는 문제가 될 수 있다는 것이다. 특히 overflow가 그렇다. CyKor의 선배들과 대화를 하면서 알게 된 점 중 하나였는데 overflow가 나타난다는 것은 보안적으로 큰 취약점이 될 수 있다. args 배열을 선언해 인자들을 각각 나누도록 코드를 짰는데, 명령어 인자가 너무 많을 경우 경계를 초과할 가능성이 있다. 현재는 고정된 크기를 define했지만 이론상으로는 이 크기를 넘어서는 명령어 인자를 입력할 수 있기 때문이다. 이에 arg_count<MAX_ARGS-1과 같은 방법으로 어느 정도의 예방은 하였으나 코드를 완성하고 다시 찬찬히 읽어본 결과 MAX_ARGS 이상 인자가 들어올 경우 나머지 인자는 무시되며 명시적 경고 출력이나 에러 처리 또한 없다는 사실을 알 수 있었다. 따라서 인자 수 초과 시 사용자에게 경고를 출력하거나 애초에 명령어 실행 자체를 거부하는 로직이 필요함을 인지하였다.

3. fork() 호출의 실패

사실 fork 명령어가 어떻게 사용되는지만 알았지 그 자세한 쓰임을 알지는 못했다. 컴퓨터학과의 운영체제 과목에서 자식 process를 만드는 데 쓴다는 사실을 인지하고 있었으나 이론에 너무 치우친 나머지 이런 코드적인 부분에서 깊게 파고들지 않았었다. 그러나 이번 과제를 계기로 찾아본 결과 fork()는 호출 시 실패할 가능성이 항상 존재한다는 사실을 새롭게 알 수 있었다. 현재 코드는 fork()시 perror을 통한 출력만을 하고 이후 로직은 강제로 진행한다. 즉, 예상치 못한 동작을 유발할 수 있다는 것이다. 따라서 fork() 실패 시 명령어 자체를 취소하고 적절히 복구하는 과정이 필요할 것이라고 생각하였다.

4. 악의적 명령어가 있다면?

이 프로그램 코드는 입력된 문자열을 하나하나 잘라 execvp를 호출하는 식으로 작동하고 있다. 그러나 자체적인 명령어 검증이나 필터링을 하지 않으므로 악의적인 명령어 또한 그대로 실행할 가능성이 있다. 디지털포렌식개론 강의를 수강하면서 문제가 될 수 있는 명령어들이 여기저기 숨어 있다는 사실을 깨달았다. 실제로 툴을 잘못 사용하다가 C drive에 있던 모든 파일을 날리던 사례가 있었기 때문이다. 아직 리눅스에 대한 이해가 얕기에 근본적인 해결 방안은 잘 모르겠으나 문제가 되는 명령어들을 몇 개 뽑아 금지시키도록 검증 로직이 필요할 것이라 생각한다.

5. getline의 제한

2에서 언급했듯이 명령어 buffer overflow가 나타날 수 있다. 이와 비슷하게 getline은 자동으로 buffer를 확장시키는 형식으로 입력을 가져온다. 만약 비슷한 식으로 너무 큰 입력이 들어올 경우 시스템 메모리를 소진시킬 가능성이 있기에 애초부터 상한선을 설정하는 것도 좋은 아이디어라고 생각한다.

## 추가 명령어 구현

추가 명령어 구현으로는 기존 shell이 구현하는 명령어 중 일부인 echo, history, help 세 개를 builtin의 내부에서 구현하였다. echo는 문자열을 출력하는 명령어로 -n option을 통해 줄바꿈을 할지 말지를 결정해 주었으며 history는 지금까지 입력한 명령어 기록을 보여주도록 하였다. history를 위해 1024크기의 array 및 추가적인 로직을 일부 구현하였으며 help는 내장 명령어를 모아 보여주는 역할을 담당하도록 하였다.

최종 코드로 reap_zombies()함수를 통해 zombie process를 방지하였고
echo, histoy, help 내장 명령어 구현
그리고 args overflow를 보호하였으며 입력 줄 길이를 제한하였다.
또한 마지막 free(history)를 통해 메모리 누수를 방지하여 최종적으로 수정해 보았다.

+0428
CyKor 교육생 공지에서
명령어에 대해 function pointer를 적극적으로 활용하라는 tip을 주셨다.
이에 builtin의 함수들을 pointer 형태로 구현해서 새롭게 내장 함수를 추가할 경우 보다 쉽게 이용 가능하도록 수정하였다.

최종 code

```
#define _GNU_SOURCE
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <libgen.h>
#include <limits.h>
#include <unistd.h>
#define MAX_ARGS 128
#define MAX_HISTORY 1024
#define MAX_LINE 4096
#define MAX_PIPES 32
char *history[MAX_HISTORY];
int history_count=0;

static void remove_zombies() {
    int status;

    while(waitpid(-1, &status, WNOHANG)>0)
                {}
}

typedef int (*builtin_func)(char **);

typedef struct {
    const char *name;
    builtin_func func;
} bulitincmd;

int cmd_cd(char **argv) {
    const char *dest=argv[1] ? argv[1] : getenv("HOME");

        if(chdir(dest)==-1)
        perror("cd");

    return 0;
}

int cmd_pwd(char **argv) {
```

```
	char cwd[PATH_MAX];

	if(getcwd(cwd, sizeof(cwd)))
		puts(cwd);

	else
		perror("pwd");

	return 0;
}

int cmd_exit(char **argv) {
	exit(0);
}

int cmd_echo(char **argv) {
	int i=1;
	int newline=1;

	if(argv[1]&&strcmp(argv[1], "-n")==0) {
		newline=0;
		i=2;
	}

	for(; argv[i]; i++) {
		printf("%s", argv[i]);

		if(argv[i+1])
			putchar(' ');
	}

	if(newline)
		putchar('₩n');

	return 0;
}

int cmd_history(char **argv) {
	for(int i=0; i<history_count; i++)
		printf("%4d  %s", i+1, history[i]);
```

```
    return 0;
}

int cmd_help(char **argv) {
    puts("Supported built-in commands:");
    puts("cd [dir] Change directory");
    puts("pwd Print working directory");
    puts("exit Exit the shell");
    puts("echo [-n] Print strings");
    puts("history Show command history");
    puts("help Show this help message");

    return 0;
}

bulitincmd builtincmds[]={
    {"cd", cmd_cd},
    {"pwd", cmd_pwd},
    {"exit", cmd_exit},
    {"echo", cmd_echo},
    {"history", cmd_history},
    {"help", cmd_help},
    {NULL, NULL}
};

int builtin(char **argv) {
    if(argv[0]==NULL)
                return -1;

    for(int i=0; builtincmds[i].name!=NULL; i++)
        if(strcmp(argv[0], builtincmds[i].name)==0)
            return builtincmds[i].func(argv);

    return -1;
}

int run_pipeline(char *cmdline, int bg) {
        char *seg_save;
        char *segment=strtok_r(cmdline, "|", &seg_save);
        int input_fd=STDIN_FILENO;
        pid_t pids[MAX_PIPES];
```

```
int pid_count=0;
int last_status=0;

while(segment) {
        char *next=strtok_r(NULL, "|", &seg_save);
        int fds[2];
        int output_fd=STDOUT_FILENO;

        if(next) {
                if(pipe(fds)==-1) {
                        perror("pipe");
                        return 1;
                }

                output_fd=fds[1];
        }

        char *args[MAX_ARGS];
        int arg_count=0;
        int overflow=0;
        char *tok=strtok(segment," ₩t₩n");

        while(tok) {
                if(arg_count>=MAX_ARGS-1) {
                        overflow=1;
                        break;
                }

                args[arg_count++]=tok;
                tok=strtok(NULL, " ₩t₩n");
        }

        if(overflow) {
                fprintf(stderr, "myshell:overflow (max %d)₩n", MAX_ARGS-1);
                return 1;
        }

        args[arg_count]=NULL;

        if(arg_count==0) {
                segment=next;
```

```
            continue;
        }

        if(builtin(args)==0) {
            if(next||input_fd!=STDIN_FILENO)
                fprintf(stderr, "myshell:cannot be piped₩n");

            last_status=0;
        }

        else {
            pid_t pid=fork();

            if(pid==0) {
                if(input_fd!=STDIN_FILENO)
                    dup2(input_fd,STDIN_FILENO);

                if(output_fd!=STDOUT_FILENO)
                    dup2(output_fd,STDOUT_FILENO);

                if(next)
                    close(fds[0]);

                execvp(args[0],args);
                perror("execvp");
                _exit(127);
            }

            else if(pid>0)
                pids[pid_count++]=pid;

            else {
                perror("fork");
                break;
            }
        }

        if(input_fd!=STDIN_FILENO)
            close(input_fd);

        if(output_fd!=STDOUT_FILENO)
```

```
                close(output_fd);

            input_fd=next?fds[0]:STDIN_FILENO;

            segment=next;
        }

        if(!bg) {
        for(int i=0; i<pid_count; i++) {
            int st;
                waitpid(pids[i], &st, 0);

                if(i==pid_count-1)
                    last_status=WIFEXITED(st)?WEXITSTATUS(st):1;
        }
    }

        else
        printf("[bg pid %d]₩n", pids[pid_count-1]);

        return last_status;
}

int exec_line(char *line) {
        int last_status=0;
        enum{NONE,AND,OR} prev=NONE;
        char *p=line;

        while(*p) {
            int bg=0;

            while(*p==' '||*p=='₩t')
                ++p;

            if(!*p||*p=='₩n')
                break;

            char *cmd_start=p;
            enum{END,SC,SA,SO} sep=END;

            while(*p && *p!='₩n') {
```

```
		if(p[0]==';') {
			sep=SC;
			break;
		}

		if(p[0]=='&'&&p[1]=='&') {
			sep=SA;
			break;
		}

		if(p[0]=='|'&&p[1]=='|') {
			sep=SO;
			break;
		}

		++p;
	}

	char *cmd_end=p;

	if(*p) {
		*cmd_end='₩0';

		if(sep==SA||sep==SO)
			p+=2;

		else
			p+=1;
	}

	for(char *q=cmd_end-1; q>=cmd_start && (*q==' '||*q=='₩t'); --q)
		q='₩0';

	size_t len=strlen(cmd_start);

	if(len && cmd_start[len-1]=='&') {
		bg=1;

		cmd_start[len-1]='₩0';

		while(len>1 && (cmd_start[len-2]==' '||cmd_start[len-2]=='₩t'))
```

```
                        cmd_start[--len-1]='₩0';
                }

                int exec=1;

                if(prev==AND && last_status!=0)
                        exec=0;

                if(prev==OR  && last_status==0)
                        exec=0;

                if(exec)
                        last_status=run_pipeline(cmd_start, bg);

                prev=(sep==SA)?AND:(sep==SO)?OR:NONE;
        }

        return last_status;
}

int main() {
        char cwd[PATH_MAX];
        char *line=NULL;
        size_t len=0;

        while(1) {
                remove_zombies();
                if(getcwd(cwd,sizeof(cwd)))
                        printf("[%s]$ ", basename(cwd));
                else
                        perror("getcwd");
                fflush(stdout);

                if(getline(&line, &len, stdin)==-1)
                        break;

                if(strlen(line)>MAX_LINE) {
                        fprintf(stderr, "myshell:MAX_LINE over (max%d)₩n", MAX_LINE);
                        continue;
                }
```

```
            if(strspn(line," ₩t₩r₩n")==strlen(line))
                continue;

        if(line && *line!='₩n')
            if(history_count<MAX_HISTORY)
                history[history_count++]=strdup(line);

            exec_line(line);
        }

        free(line);
        for(int i=0; i<history_count; i++)
            free(history[i]);

        return 0;
}
```