Operating Systems Project - 1

소프트웨어학부 2017012279 윤찬웅

설계한 Simple Shell 알고리즘과 소스파일

헤더파일 및 프로토타입

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <ctype.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <malloc.h>

#define MAX_LINE 80 /* The maximum length command */
#define READ_END  0 /* define stdin */
#define WRITE_END 1 /* define stdout */

void redirect(char *str,char **args,int redirect_pos);
void ex_pipe(char **args_pipe);
```

메인함수 - 1

```
int main(void){
  char *line = (char*)malloc(sizeof(char) * MAX_LINE); /* first input command line(include spaces) */
  int should_run = 1; /* flag to determine when to exit program */
  pid_t pid; /* process id to distinguish parent and child */
  int pipe_pos = 0; /* flag to store index of "|" in args array. default is 0(not founded) */

  while(should_run){
    printf("osh>");
    fflush(stdout); /* flush last stdout */

    fgets(line,MAX_LINE,stdin); /* read line from stdin and store in line */
    line[strlen(line) - 1] = '\0'; /* remove "\n" in last element of array */

    int i = 0;
    int params_num = 1;
    while(line[i] != '\0'){ /* find out the number of parameters by counting spaces */
      if(line[i] == ' '){
        params_num++;
      }
      i++;
    }

    /* make dynamic array to store command line arguments separated by space */
    char **args = (char **)malloc(sizeof(char *) * params_num);
    /*make dynamic array to store another command when using pipe */
    char **args_pipe = (char **)malloc(sizeof(char *) * params_num);

    i = 0;
    args[i] = strtok(line," ");
    while(args[i] != NULL){ /* insert parameters in array */
      i++;
      args[i] = strtok(NULL," "); /*arguments seperated by space */
    }
```

메인함수-1에서는 command 전체를 받아올 line(char *)과 process id(pid_t)를 저장할 pid를 선언하고 필요한 flag들을 정의한다. 그 뒤 while문을 통해 shell을 실행시킨다. shell이 실행되고 있는 동안 fgets를 통해 사용자가 입력한 command 전체를 받은 다음 이것을 space 단위로 잘라 동적배열로 선언한 args 배열에 집어넣는다.

메인함수 - 2

```
i = 0;
char *redirect_file;

while(args[i] != NULL){
  if (!strcmp(args[i],">") | !strcmp(args[i],"<")){ /* when args include redirect operands */
    redirect_file = args[i+1];
    redirect(redirect_file,args,i); /* redirect */

    if(!strcmp(args[params_num -1],"&")){ /* after redirection, modify command line */
      args[params_num -1] = NULL;
      args[i] = "&";
    }
    else {
      args[i] = NULL;
    }
    args[i+1] = NULL;
    params_num = params_num - 2;
  }
  else if(!strcmp(args[i],"|")){ /* when args include pipe operands */
    pipe_pos = i;
    int t = 0;
    args[i] = NULL;
    while(t < params_num - i - 1){ /* args_pipe store pipe arguments */
      if(!strcmp(args[i + t + 1],"&")){
        args[i] = "&";
        args_pipe[t] = NULL;
        pipe_pos++;
      }
      else {
        args_pipe[t] = args[i + t + 1];
        args[i + t + 1] = NULL;
      }
      t++;
    }
    if (pipe_pos != i){
      params_num++;
    }
    params_num = params_num - t - 1;

  }
  i++;
}
```

메인함수 - 2은 redirection과 pipe 작업을 위해 args 검사하고 수정한다. while문을 돌며 args 배열요소들을 하나씩 검사하는데

첫번째 케이스로 ">" 과 "<" 을 찾으면 redirect할 파일을 저장하고 redirect()가 호출된다. 그 뒤 args 를 수정하여 execvp가 제대로 작동할수 있게 한다.

두번째 케이스로 "|" 를 찾으면 args배열을 "|" 기준으로 나누어 왼쪽부분은 args 에 저장되고 오른쪽 부분은 또 다른 동적배열인 args_pipe 에 저장이 된다. 또한 이번 커맨드에 pipe operand가 있는지 없는지 확인할수 있는 pipe_pos라는 플래그를 정의해준다.

메인함수 - 3

```
pid = fork();

if (pid < 0){ /* error occurred */
  fprintf(stderr, "Fork Failed");
  return -1;
}
else if (pid == 0){ /*child process */
  if(!strcmp(args[params_num -1],"&")){ /* if child process include "&", remove it */
    args[params_num -1] = NULL;
  }
  if (pipe_pos != 0){
    ex_pipe(args_pipe); /* pipe */
  }
  //sleep(5); /* give sleep command to check background operation */
  execvp(args[0],args); /* operate command */

}
else { /* parent process */
```

```
      if(!strcmp(args[params_num -1],"&")){ /* Background,the parent does not wait child process */
        waitpid(pid,NULL,WNOHANG);
      }
      else { /* Foreground,the parent waits until child processes are completed */
        waitpid(pid,NULL,0);
      }
    }
    should_run = 0; /* after parent and child process, change flag */
  }
return 0;
}
```

메인함수 - 3 에서 부모 프로세스는 사용자에게서 받은 입력을 실행시키기 위해서 자식 프로세스를 생성한다. (pid = fork();) 부모 프로세스는 pid 가 양수인 값을 가지는데 반해 자식 프로세스는 pid가 0 인것을 이용하여 if-else문을 통해 부모 프로세스와 자식 프로세스의 작업을 분리한다.

자식 프로세스에서는 execvp()명령어를 직접 실행한다. 그 전에 background operand를 없애고 위에서 설정한 pipe_pos 를 통해 ex_pipe()함수가 호출이 될지 안될지를 결정한다.

부모 프로세스에서는 자식 프로세스의 백그라운드 실행여부를 args 를 검사하여 확인한다. 만약 백그라운드로 실행이 되어야 한다면 waitpid()에 WNOHANG 옵션을 주어 부모는 자식 프로세스를 기다리지 않게 된다. 백그라운드 실행이 아니라면 waitpid()에 NULL 옵션을 주어 자식 프로세스가 완료될때까지 부모가 기다리게 한다.

sleep()함수는 자식 프로세스가 execvp()를 수행하기 전에 몇초동안 대기하게 만든다. 만약 백그라운드로 이것을 수행한다면 자식 프로세스가 대기중일때도 부모 프로세스는 명령어를 받을 수 있는 상태가 된다. 이것을 통해 background 옵션이 잘 동작 되는 것을 확인할수 있다.

redirect함수

```
void redirect(char *str,char **args,int redirect_pos){
  int fd; /* file descriptor number to redirect */
  mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IROTH; /* file permission when opening file */

  if(!strcmp(args[redirect_pos],">")){ /* case 1 : redirect ">" */
    if ((fd = open(str,O_CREAT | O_WRONLY | O_TRUNC,mode)) == -1){ /* open file to write stdout */
      fprintf(stderr, "File Can Not Open");
    }
    dup2(fd, STDOUT_FILENO); /* redirect stdout to fd */
    close(fd);
  }
  else { /* case 2 : redirect "<" */
    if ((fd = open(str,O_RDONLY,mode)) == -1){ /* open existing file to give stdin*/
      fprintf(stderr, "File Not Found");
    }
    dup2(fd,STDIN_FILENO); /* redirect stdin to fd */
    close(fd);
  }
}
```

redirect() 함수는 args 배열요소에 ">" 나 "<" 가 있으면 호출되는 함수이다. 인자로 redirect 가 이뤄질 파일(char *str), args 배열(char **args), redirect operand 가 저장되 있는 인덱스(redirect_pos) 를 받는다. if -else문으로 케이스 두가지를 나눈다.

첫번째 케이스는 standard output을 redirect 할 경우이다. redirect가 이뤄질 파일을 open( )함수로 열고 반환되는 file descriptor number를 fd 에 저장한다. 그 후 dup2()함수를 이용해 standard output을 fd로 redirect 한다.(file descriptor number 를 복사)

두번째 케이스는 standard input을 redirect 한다. dup2를 이용해 standard input을 fd 로 redirect한다.

ex_pipe 함수(pipe를 이용한 프로세스간 통신)

```
void ex_pipe(char **args_pipe){
  /* if args include pipe operand("|"),
   * process creates another child process and communicate using pipe */
  int fd[2];
  pid_t pid_pipe;

  if (pipe(fd) == -1) { /* create pipe */
```

```
    fprintf(stderr, "Pipe failed");
  }

  pid_pipe = fork(); /* create another child process */

  if (pid_pipe < 0){ /* error occurred */
    fprintf(stderr, "Pipe Fork Failed");
  }
  else if (pid_pipe == 0){ /*pipe child process */
    close(fd[WRITE_END]); /* close unused write pipe */
    dup2(fd[READ_END],STDIN_FILENO); /* redirect stdin */
    close(fd[READ_END]); /* after duplicating file descriptor,close read pipe */
    execvp(args_pipe[0],args_pipe);
  }
  else { /* pipe parent process */
    close(fd[READ_END]); /* close unused read pipe */
    dup2(fd[WRITE_END], STDOUT_FILENO); /* redirect stdout */
    close(fd[WRITE_END]); /* after duplicating file descriptor,close write pipe */
  }

}
```

ex_pipe함수는 args에 "|" 가 있으면 호출된다. 또 다른 자식 프로세스를 생성하고 프로세스간에 pipe 통신이 이뤄지기 때문에 pid_pipe(pid_t)와 fd (int *)를 선언한다.

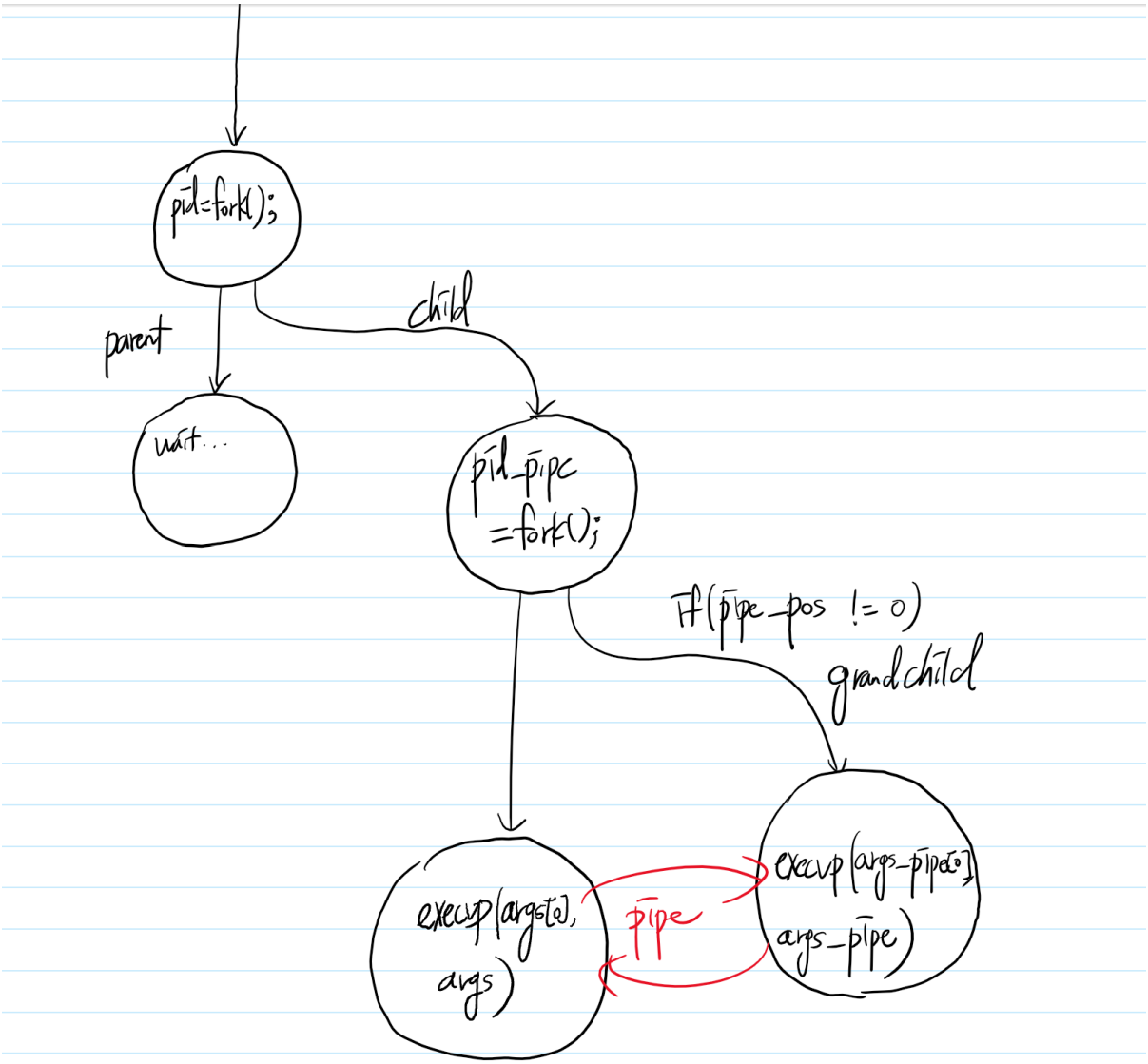
pipe(fd)로 pipe를 생성하고 pid_pipe의 값을 통해 자식 프로세스와 부모 프로세스를 구분한다.

부모 프로세스에서는 write pipe(fd[WRITE_END])만 이용 하기 때문에 read pipe(fd[READ_END])를 닫아준다. 그 후 dup2를 통해 standard output을 write pipe로 redirect 해주면 원래의 부모 execvp()함수에서 나온 stdout이 write pipe로 보내진다.

자식 프로세스에서는 반대로 read pipe만 이용 하기 때문에 write pipe를 닫아준다. 부모 프로세스에서 stdout을 redirection 했기 때문에 자식 프로세스에서 read pipe로 stdout을 받아올수 있게 된다.

이것을 다시 dup2를 통해 stdin 으로 redirection을 해주면 execvp(args_pipe[0],args_pipe)의 input으로 execvp(args[0],args) output이 들어가게 된다.

그림으로 표현해보면



컴파일 과정과 명령어

컴파일 및 커맨드 실행 - 1

gcc 명령어를 통해 main.c 파일을 os_project1로 컴파일한 뒤 실행시켜 각 커맨드를 입력해보았다.

```
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ gcc -o os_project1 main.c
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ ./os_project1
osh>ls -la
total 40
drwxrwxr-x 2 qic qic  4096  3월 28 18:06 .
drwxrwxr-x 5 qic qic  4096  3월 28 17:50 ..
-rw-rw-r-- 1 qic qic  5019  3월 28 17:54 main.c
-rwxrwxr-x 1 qic qic 17552  3월 28 18:06 os_project1
-rw-rw-r-- 1 qic qic    10  3월 27 22:36 test.txt
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ ./os_project1
osh>ls -la &
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ total 40
drwxrwxr-x 2 qic qic  4096  3월 28 18:06 .
drwxrwxr-x 5 qic qic  4096  3월 28 17:50 ..
-rw-rw-r-- 1 qic qic  5019  3월 28 17:54 main.c
-rwxrwxr-x 1 qic qic 17552  3월 28 18:06 os_project1
-rw-rw-r-- 1 qic qic    10  3월 27 22:36 test.txt

qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ ./os_project1
osh>ls -la > out.txt
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ cat out.txt
total 40
drwxrwxr-x 2 qic qic  4096  3월 28 18:07 .
drwxrwxr-x 5 qic qic  4096  3월 28 17:50 ..
-rw-rw-r-- 1 qic qic  5019  3월 28 17:54 main.c
-rwxrwxr-x 1 qic qic 17552  3월 28 18:06 os_project1
-rwxr--r-- 1 qic qic     0  3월 28 18:07 out.txt
-rw-rw-r-- 1 qic qic    10  3월 27 22:36 test.txt
```

ls -la , ls -la & , ls -la > out.txt 명령어를 실행시켰다.

ls -la > out.txt 에서 stdout 이 out.txt를 연 file descriptor로 redirection 되었기 때문에 out.txt 에 결과가 있는것을 확인할수 있음.

커맨드 실행 - 2

```
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ ./os_project1
osh>ls -la > out        ^C
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ ./os_project1
osh>ls -la > out.txt &
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ ./os_project1
osh>sort -n < test.txt
35
194
522
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ cat test.txt
194
35
522qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ ./os_project1
```

```
qic@qic-VirtualBox:~$ cd eclipse-workspace/OS_project1/src/
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ ls
main.c  os_project1  out.txt  test.txt
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ ./os_project1
osh>sort -n < test.txt &
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ 35
194
522
```

ls -la > out.txt & , sort -n < test.txt , sort -n < test.txt & 를 실행시켰다.

test.txt는 194,35,522가 개행되있는 텍스트파일이다. 이것을 input으로 받아 sort -n 옵션을 주어 실행하면 오름차순으로 정렬된것을 확인할수 있다.

커맨드 실행 -3

```
522qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ ./os_project1
osh>ls -al | grep -i src &
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ ./os_project1
osh>ls -la | grep -i main &
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ -rw-rw-r-- 1 qic qic  5019  3월 28 17:54 main.c

qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ ./os_project1
osh>ls -la | grep -i main
qic@qic-VirtualBox:~/eclipse-workspace/OS_project1/src$ -rw-rw-r-- 1 qic qic  5019  3월 28 17:54 main.c
```

ls -al | grep -i src & , ls -la | grep -i main & , ls -la | grep -i main 을 실행시켰다.

ls -al | grep -i src & 에서는 ls -al 의 output이 grep -i src 의 input으로 들어갔다. 현재 파일 디렉토리에는 src 파일이 없으므로 결과가 출력되지 않는 것을 확인 할 수 있다.

ls -la | grep -i main & 과 ls -la | grep -i main 에서는 ls -al 의 output이 grep -i main 의 input으로 들어갔다. 현재 디렉토리에 main.c파일이 있기 때문에 결과가 출력되는 것을 확인 할 수 있다.