시스템 프로그래밍
smsh project

소프트웨어학부 2017012251 윤영훈

1. 소스코드 설명

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <fcntl.h>

#define MAX_LINE 128 // the maximum length command
#define READ_END 0
#define WRITE_END 1

// to delete element in string
void delete_element(char *string, char element);

int main(void)
{
    char *args[MAX_LINE/2 + 1];         // command line arguments.
    char *args_pipe[MAX_LINE/2 + 1];    // arguments for pipe.
    int alive = 1;                      // flag to determine when to exit program.
    char buffer[MAX_LINE];              // sub string to make args.
    char cwd[MAX_LINE];                     // to save current working directory.
    int background = 0, status;         // background : to check background. status : to check error when do execvp.
    int fd;                             // to save command to history file.
    pid_t c_pid, pid;                   // to fork.
    char *semi[2];                      // to separate buffer by semi_colon (;) and save.

    // open history_file to save history.
    if ((fd = open("history_file.txt", O_CREAT | O_APPEND | O_RDWR, 0666)) == -1){
        perror("OPEN ERROR");
        exit(1);
    }
```

소스코드를 위한 header와 define, delete_element를 미리 선언해두고 main 함수 진입. 다양한 변수들을 초반에 미리 선언해둠.
사용한 command들을 저장해두기 위해 history_file.txt를 O_CREAT | O_APPEND | O_RDWR flag로, permission은 0666으로 설정하고 open.

```c
    while (alive) {
        int num = 0, pipe_index = 0;
        int num_semi=0;

        getcwd(cwd, MAX_LINE-1);
        printf("\n%s $", cwd);

        // clear all char variable and STDIN, STDOUT.
        memset(buffer, '\0', MAX_LINE);
        memset(args, '\0', MAX_LINE);
        memset(semi, '\0', MAX_LINE);
        fflush(stdin);
        fflush(stdout);

        // get command.
        fgets(buffer, sizeof(buffer), stdin);
        printf("\n");

        // save command to history_file.
        if (write(fd, buffer, strlen(buffer)) != strlen(buffer)){
            perror("WRITE HISTORY_FILE ERROR");
            exit(1);
        }
```

quit를 입력하거나 오류가 발생하지 않는 이상 계속해서 프로그램을 실행. (alive == 1)
‘현재 경로명$ (command)’ 와 같이 최대한 리눅스 shell 환경과 비슷하게 설정하기위해 getcwd() 로 현재 경로명을 받음.
이후 fgets() 로 command를 입력받아 buffer에 저장하고 입력된 command를 history_file에 write로 써서 저장한다.

```c
// check how many semi_colon (;) is used.
for (int i=0; i<strlen(buffer); i++){
    if (buffer[i] == ';')
        num_semi++;
}

// separate buffer by semi_colon (;).
// to execute each buffer (separated by semi_colon) independently, use fork.
for (int i=0; i<num_semi; i++){
    switch(c_pid = fork()){
        case -1:
            perror("SEMI_COLON FORK ERROR");
            break;
        case 0:
            semi[0] = strtok(buffer, ";");
            strcpy(buffer, semi[0]);
            break;
        default:
            waitpid(c_pid, NULL, 0);
            strtok(buffer, ";");
            semi[0] = strtok(NULL, "\n");
            strcpy(buffer, semi[0]);
            break;
    }
}

// check background
if (buffer[strlen(buffer)-2] == '&'){ // If there are background sign &,
    background = 1;                   // Delete & sign and background = 1
    buffer[strlen(buffer)-2] = '\n';
    buffer[strlen(buffer)-1] = '\0';
}

// if command is non_semi_colon, add '\n' for executing well at below code.
if (num_semi != 0)
    strcat(buffer, "\n");
```

command의 semi colon 사용 여부를 판단하기 위해 총 몇 번의 semi colon이 사용되었는지 파악하고, 해당 횟수만큼 fork를 통해 각각의 명령어에 대해 독립적으로 프로세스가 시행될 수 있도록 환경 구축. 이때 strtok을 통해 semi colon 좌우 명령어를 구분하여 buffer에 입력함으로 이후 소스코드에서 정상적으로 독립된 명령어가 인식될 수 있도록 함.

이후 background 기호 '&'가 입력되었는지 확인한 뒤 입력되었으면 background 변수를 1로 설정하고 buffer에서 '&'를 삭제.

```c
// if command is nothing, continue.
if (strcmp(buffer, "cd \n") == 0 || strcmp(buffer, "cd\n") == 0 || buffer[0] == '\n')
    continue;
else
    buffer[strlen(buffer) - 1] = '\0';

// parse buffer to args.
args[0] = strtok(buffer, " ");
if (strcmp(args[0], "quit") == 0)
    break;

while (args[num] != NULL){
    args[++num] = strtok(NULL, " ");
}

// cd command (internal)
if (strcmp(args[0], "cd") == 0){
    if (strncmp(args[1], "/", 1) == 0){
        strcpy(cwd, args[1]);
        if (chdir(cwd) == -1){
            perror("CHDIR ERROR 1");
            exit(1);
        }
    } else{
        strcat(cwd, "/");
        strcat(cwd, args[1]);
        if (chdir(cwd) == -1){
            perror("CHDIR ERROR 2");
            exit(1);
        }
    }

    continue;
}

// history command (internal)
if (strcmp(args[0], "history") == 0){
    lseek(fd, 0, SEEK_SET);
    while(read(fd, buffer, MAX_LINE-2) != 0){
        printf("%s", buffer);
        memset(buffer, 0, MAX_LINE);
    }

    continue;
}
```

command에서 'cd 공백(\n)', '공백(\n)'과 같은 의미없는 값이 입력되면 continue를 통해 다시 입력받도록 함.
이후 execvp를 실행하기 위해 buffer의 명령어를 공백을 기준으로 나눠 args에 저장. (cd src -> cd / src)
이때 command에 quit가 입력되었다면 프로그램을 종료.

이후 cd command에 대해 상대경로, 절대경로 두 가지 경우에 대해 구현함.
history command의 경우 그동안 저장된 모든 command, 즉 history_file 전체 내용을 불러오도록 설정.

```
140         // for other external command
141         for (int i=0; i < num; i++){
142 
143             // redirection command
144             if (strcmp(args[i], ">") == 0 | strcmp(args[i], "<") == 0 | strcmp(args[i], ">>") == 0){
145                 int fd;
146                 mode_t mode = S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IROTH;
147                 if (strcmp(args[i], ">") == 0){                                          // Case 1 : >
148                     if ((fd = open(args[i+1], O_CREAT | O_WRONLY | O_TRUNC, mode)) == -1){
149                         perror("OPEN ERROR AT >");
150                         break;
151                     }
152                     dup2(fd, STDOUT_FILENO);
153                     close(fd);
154                 }
155                 else if (strcmp(args[i], "<") == 0){                                     // Case 2 : <
156                     if ((fd = open(args[i+1], O_RDONLY, mode)) == -1){
157                         perror("OPEN ERROR AT <");
158                         break;
159                     }
160                     dup2(fd, STDIN_FILENO);
161                     close(fd);
162                 }
163                 else{                                                                    // Case 3 : >>
164                     if ((fd = open(args[i+1], O_CREAT | O_WRONLY | O_APPEND, mode)) == -1){
165                         perror("OPEN ERROR AT >>");
166                         break;
167                     }
168                     dup2(fd, STDOUT_FILENO);
169                     close(fd);
170                 }
171 
172                 args[i] = NULL;
173                 num -= 2;
174                 alive = 0;                  // To check the change, stop program.
175             }
```

cd, history를 제외한 external command를 처리하는 영역. 그 중 redirection command를 처리하는 영역.
‘>’, ‘<’, ‘>>’ 각각의 명령어를 구분하고, dup2를 통해 STDIN 또는 STDOUT이 각각 화면 또는 file로 향하도록 설정.
이때 ‘>>’는 O_APPEND를 통해 기존 파일의 뒷부분에 내용을 추가함.

```
177             // pipe command.
178             else if (strcmp(args[i], "|") == 0){
179                 pipe_index = i;
180                 args[i] = NULL;
181                 int k=0;
182 
183                 // If command is A | B, Make A -> args / B -> args_pipe.
184                 while (k < num - i - 1){
185                     args_pipe[k] = args[i + k + 1];
186                     if (args_pipe[k][0] == '"')
187                         delete_element(args_pipe[k], '"');
188                     else if (args_pipe[k][0] == '\'')
189                         delete_element(args_pipe[k], '\'');
190                     args[i + k + 1] = NULL;
191                     k++;
192                 }
193 
194                 if (pipe_index != i)
195                     num++;
196                 num = num - k - 1;
197                 args_pipe[k] = NULL;
198             }
199         }
```

pipe command를 처리하는 영역.
pipe 기호 ‘|’가 위치하는 pipe_index를 설정하고 해당 index를 기점으로 좌우 명령어를 구분하여 다음과 같이 저장한다.

ls -la	more
args	**args_pipe**
ls -la	more

```
201         // fork to execute command.
202         pid = fork();
203 
204         if (pid < 0){                           // fork error
205             perror("FORK FAILED");
206             return -1;
207         }
208         else if (pid == 0){                 /// child
209             // if command is pipe.
210             if (pipe_index != 0){
211                 int pipe_fd[2];
212                 pid_t pid_pipe;
213 
214                 if (pipe(pipe_fd) == -1){
215                     perror("PIPE FAILED");
216                     return -1;
217                 }
218                 pid_pipe = fork();
219 
220                 if (pid_pipe < 0){
221                     perror("PIPE FORK FAILED");
222                     return -1;
223                 }
224                 else if (pid_pipe == 0){        // pipe_child
225                     close(pipe_fd[WRITE_END]);
226                     dup2(pipe_fd[READ_END], STDIN_FILENO);
227                     close(pipe_fd[READ_END]);
228                     status = execvp(args_pipe[0], args_pipe);
229                     if (status == -1){
230                         perror("FAIL TO EXECUTE THE PIPE COMMAND");
231                         exit(1);
232                     }
233                 }
234                 else {                          // pipe_parent
235                     close(pipe_fd[READ_END]);
236                     dup2(pipe_fd[WRITE_END], STDOUT_FILENO);
237                     close(pipe_fd[WRITE_END]);
238                 }
239             }
240 
241             status = execvp(args[0], args);
242             if (status == -1){
243                 perror("FAIL TO EXECUTE THE COMMAND");
244                 exit(1);
245             }
246 
247         }
```

fork()를 통해 child process를 생성한다.

이후 child process는 command가 pipe라면 if문에 진입하여 다시 fork()하여 pipe_child를 생성한다.

pipe_child는 사용하지 않는 WRITE_END를 닫고 dup2를 통해 표준입력으로 pipe_fd[READ_END]를 복사하고 args_pipe를 execvp를 통해 실행한다.

pipe_parent는 READ_END를 닫고 dup2를 통해 표준출력으로 pipe_fd[WRITE_END]를 복사한다.

이때 pipe를 통해 args의 execvp output이 args_pipe의 input으로 입력된다.

command가 pipe가 아니라면 if문에 진입하지 않고 line 241부터 바로 args가 execvp를 통해 실행된다.

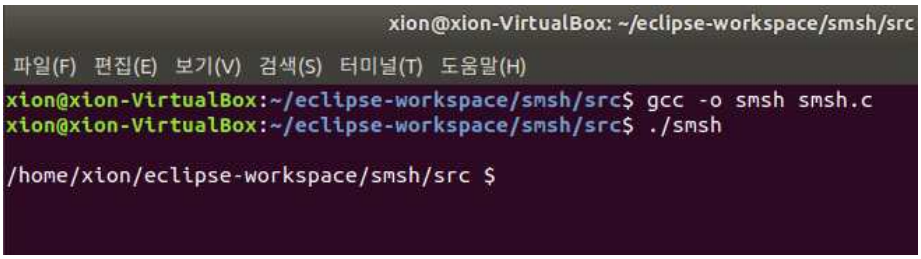
```
250             else{                                   // parent
251                 if (background){                    // background, no wait.
252                     printf("PID #%d IS WORKING IN BACKGROUND : %s\n", pid, buffer);
253                 } else{                         // foreground, wait child.
254                     waitpid(pid, NULL, 0);
255                 }
256             }
257 
258             // clear variable for next loop.
259             num = 0;
260             background = 0;
261 
262             // if command was semi_colon, finish process.
263             if (num_semi > 0){
264                 if (c_pid == 0)
265                     exit(1);
266                 else
267                     kill(c_pid, SIGKILL);
268             }
269         }
270 
271         return 0;
272     }
273 
274     void delete_element(char *string, char element){
275         for (; *string != '\0'; string++){
276             if (*string == element){
277                 strcpy(string, string+1);
278                 string--;
279             }
280         }
281     }
```

parent process는 앞서 설정했던 background 값을 확인하여 background가 설정되어있으면 wait하지 않고 background에서 실행되는 process의 관련 정보를 출력해준다.

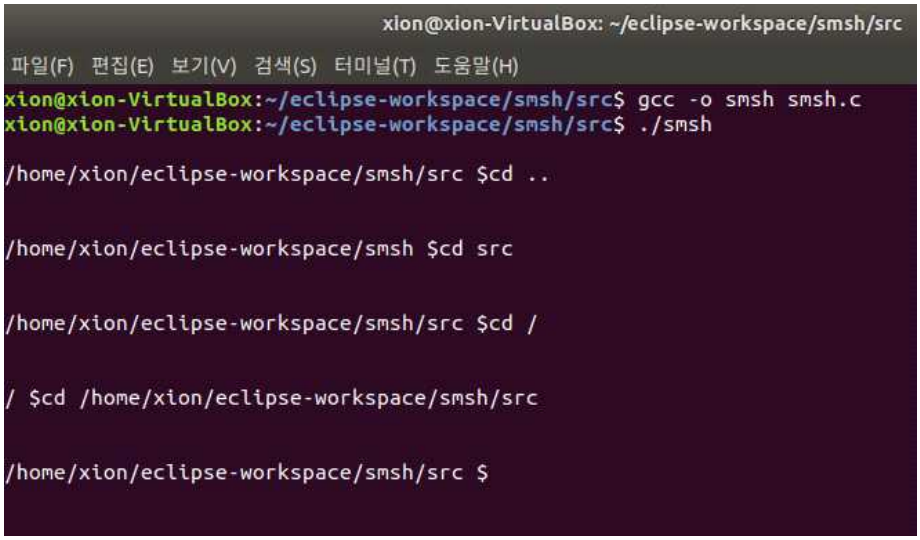
만약 background가 설정되어있지 않다면 waitpid()을 통해 child가 끝날 때 까지 대기한다.

이후 다음 loop를 위해 num과 background를 0으로 초기화하고, semi colon을 사용하여 fork를 했다면, 여분의 process들을 종료시켜준다.

2. 결과물

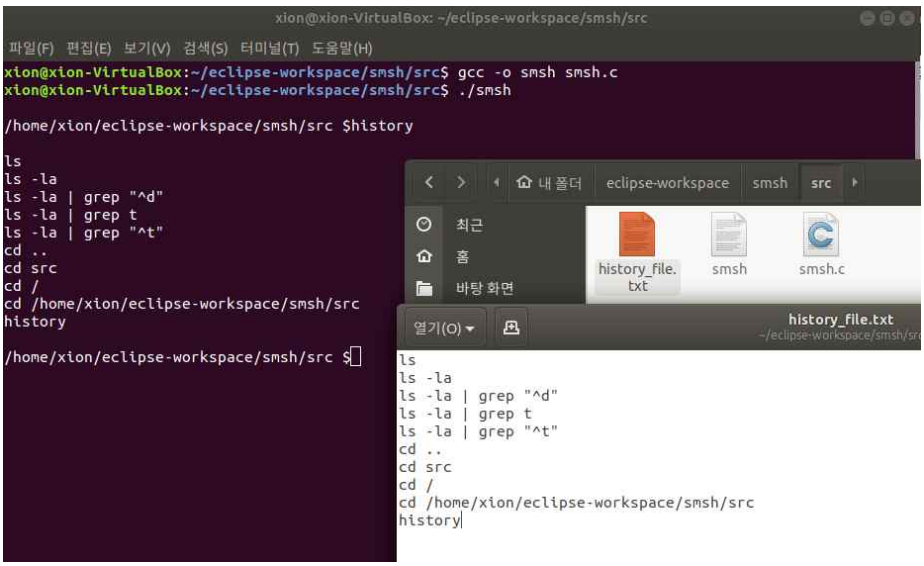


compile



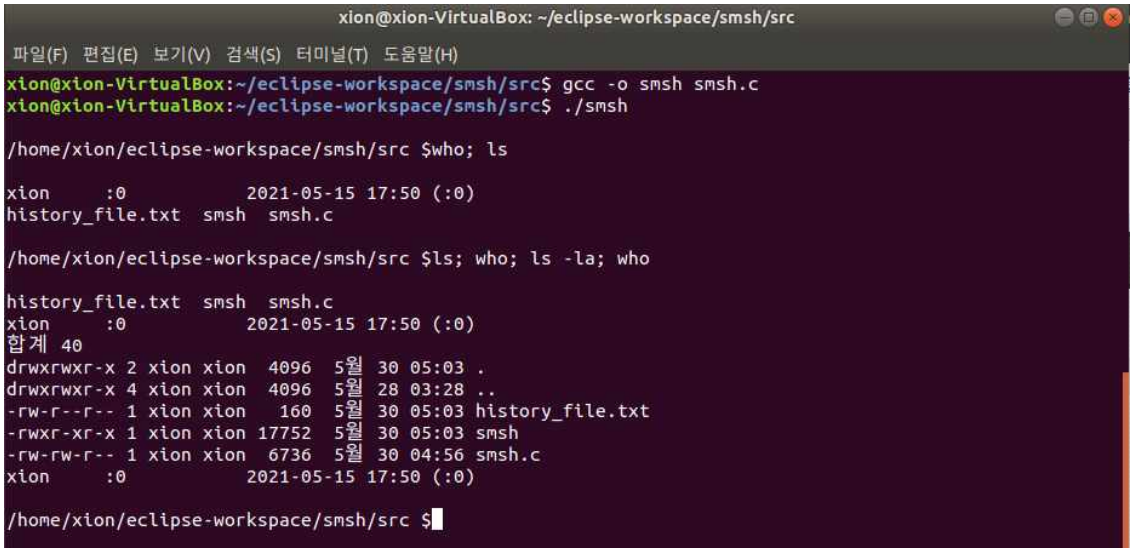
cd command

상대경로, 절대경로 모두 적용됨을 확인



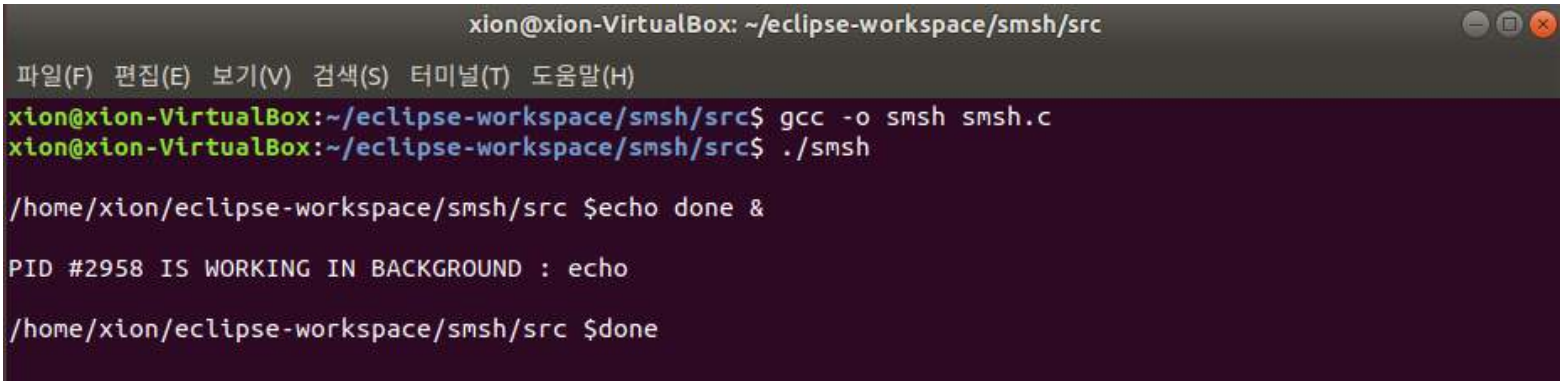
history command

history_file이 생성되었고, 그동안 입력했던 command들이 저장된 것과 history command 입력 시 저장된 내용과 같은 명령어들이 출력되는 것을 확인



multiple command separated by semi colons

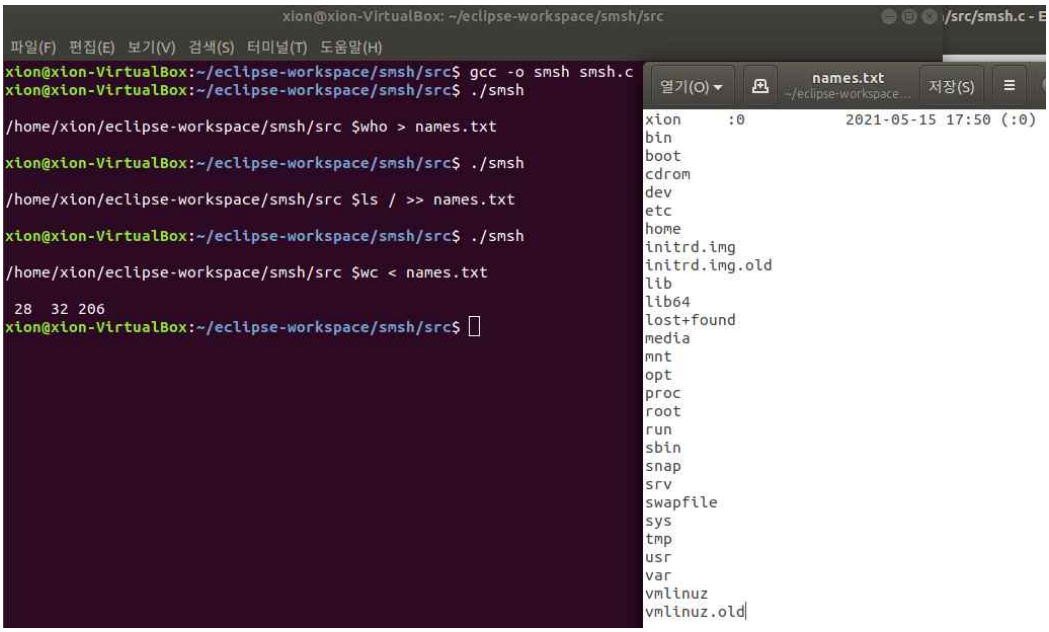
2개 이상의 semi colon 또한 처리되는 것을 확인



background command

미리 printf()로 지정해둔 background 출력메세지가 출력되는 것을 확인

background로 처리되었으므로 process가 먼저 끝나고 command 입력을 기다리는 와중에 done이 출력되는 것을 확인



redirection command

who > names.txt를 통해 names.txt에 who의 결과가 첫 번째 줄에 저장되었고

ls / >> names.txt를 통해 who의 결과인 첫 번째 줄 이후 ls / 의 내용이 append 되었음

이후 wc < names.txt를 통해 names.txt의 관련 정보를 shell에 출력함을 확인

```
xion@xion-VirtualBox: ~/eclipse-workspace/smsh/src
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
xion@xion-VirtualBox:~/eclipse-workspace/smsh/src$ gcc -o smsh smsh.c
xion@xion-VirtualBox:~/eclipse-workspace/smsh/src$ ./smsh

/home/xion/eclipse-workspace/smsh/src $ls -la | more

합계 44
drwxrwxr-x 2 xion xion  4096  5월 30 05:09 .
drwxrwxr-x 4 xion xion  4096  5월 28 03:28 ..
-rw-r--r-- 1 xion xion   300  5월 30 05:09 history_file.txt
-rwxr--r-- 1 xion xion   206  5월 30 05:08 names.txt
-rwxr-xr-x 1 xion xion 17752  5월 30 05:09 smsh
-rw-rw-r-- 1 xion xion  6736  5월 30 04:56 smsh.c

/home/xion/eclipse-workspace/smsh/src $ls -la | grep "^d"

drwxrwxr-x 2 xion xion  4096  5월 30 05:09 .
drwxrwxr-x 4 xion xion  4096  5월 28 03:28 ..

/home/xion/eclipse-workspace/smsh/src $
```

pipe command

3. 잘 구현된 부분

```
57          // check how many semi_colon (;) is used.
58          for (int i=0; i<strlen(buffer); i++){
59              if (buffer[i] == ';')
60                  num_semi++;
61          }
62
63          // separate buffer by semi_colon (;).
64          // to execute each buffer (separated by semi_colon) independently, use fork.
65          for (int i=0; i<num_semi; i++){
66              switch(c_pid = fork()){
67                  case -1:
68                      perror("SEMI_COLON FORK ERROR");
69                      break;
70                  case 0:
71                      semi[0] = strtok(buffer, ";");
72                      strcpy(buffer, semi[0]);
73                      break;
74                  default:
75                      waitpid(c_pid, NULL, 0);
76                      strtok(buffer, ";");
77                      semi[0] = strtok(NULL, "\n");
78                      strcpy(buffer, semi[0]);
79                      break;
80              }
81          }
82
83          // check background
84          if (buffer[strlen(buffer)-2] == '&'){ // If there are background sign &,
85              background = 1;                   // Delete & sign and background = 1
86              buffer[strlen(buffer)-2] = '\n';
87              buffer[strlen(buffer)-1] = '\0';
88          }
89
90          // if command is non_semi_colon, add '\n' for executing well at below code.
91          if (num_semi != 0)
92              strcat(buffer, "\n");
```

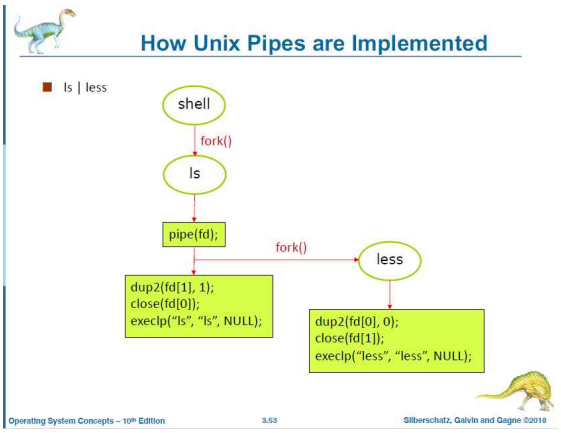
semi colon을 처리하는 코드이다. 그동안 child-parent process, fork() 등과 관련된 내용은 이론적으로는 알고 있었지만, 예제를 풀어보거나 스켈레톤이 주어진 실습에서만 사용해봤었다. 그래서 단 한번도 능동적으로 '이 부분은 fork와 child-parent를 사용해서 코드를 구현해봐야지'라는 생각을 떠올리면서 fork()를 사용해본 적이 없었다. 하지만 semi colon을 처리하기위해 고민하면서 '각 명령어가 독립적으로 수행되려면 어떻게 해야할까?'를 생각하다가 불현듯 fork()와 child-parent process가 떠올랐고, 'semi colon으로 나눠진 각 명령어를 각각 다른 process에서 처리하면 독립적으로 수행될 것이다'라는 결론을 도출했다. 도출된 결론을 통해 코드를 구축하고, 실제로 동작하는 것을 봤을 땐 큰 희열을 느꼈으며 막막하여 포기를 고민하던 이번 project를 포기하지 않도록 마음을 다잡아준 계기가 되었다. 이 코드를 구현하면서 process, fork()에 대해 부족했던 이해를 완성할 수 있었고, 이후 external command와 pipe에서 아주 복잡한 process 관계를 구축하는데에도 많은 도움이 되었다.

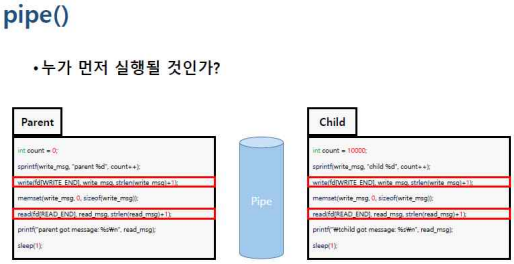
추가적으로 history의 경우 'command를 무한대로 저장할 수 있어야 한다. 배열을 사용할 시 0점이다.'라는 교수님의 어드바이스를 참고하여 배열을 제외한 다른 방법을 찾아보다가, 실제 linux나 window shell에서 history가 어떤 식으로 구현되었나를 찾아보았다. linux는 '~/.bash_history' file에 command를 저장한다. 이를 따라하여 open과 read, write를 통해 text file을 생성하여 command를 저장하는 방법을 생각했다. text file에 저장하면 system의 용량적 한계가 아닌 이상 command를 무한히 write 할 수 있으며 read 또한 while문을 통해 전체 command를 모두 불러올 수 있으며, 배열이 필요없다는 점에서 좋은 구현이라고 생각한다.

4. 어려웠던 점

```
201         // fork to execute command.
202         pid = fork();
203 
204         if (pid < 0){                           // fork error
205             perror("FORK FAILED");
206             return -1;
207         }
208         else if (pid == 0){                 /// child
209             // if command is pipe.
210             if (pipe_index != 0){
211                 int pipe_fd[2];
212                 pid_t pid_pipe;
213 
214                 if (pipe(pipe_fd) == -1){
215                     perror("PIPE FAILED");
216                     return -1;
217                 }
218                 pid_pipe = fork();
219 
220                 if (pid_pipe < 0){
221                     perror("PIPE FORK FAILED");
222                     return -1;
223                 }
224                 else if (pid_pipe == 0){        // pipe_child
225                     close(pipe_fd[WRITE_END]);
226                     dup2(pipe_fd[READ_END], STDIN_FILENO);
227                     close(pipe_fd[READ_END]);
228                     status = execvp(args_pipe[0], args_pipe);
229                     if (status == -1){
230                         perror("FAIL TO EXECUTE THE PIPE COMMAND");
231                         exit(1);
232                     }
233                 }
234                 else {                          // pipe_parent
235                     close(pipe_fd[READ_END]);
236                     dup2(pipe_fd[WRITE_END], STDOUT_FILENO);
237                     close(pipe_fd[WRITE_END]);
238                 }
239             }
240 
241             status = execvp(args[0], args);
242             if (status == -1){
243                 perror("FAIL TO EXECUTE THE COMMAND");
244                 exit(1);
245             }
246 
247         }
```

구현에 있어서 어려웠던 부분은 왼쪽 코드에 해당하는 부분이다. 해당 코드에 대한 부분은 아직도 완전히 이해하지 못한채 그저 이 값, 저 값 건드려보며 억지로, 운좋게 끼워맞춰서 결과를 도출해냈다. 특히 process, execvp, dup2 파트가 가장 어려웠는데, 시스템프로그래밍 강의 및 실습 예제만으로는 이해에 어려움을 겪는 부분이 있어서 운영체제론 강의 및 실습 예제를 참고했다.





구현에 있어서 아쉬웠던 부분은 괄호 처리 ( ) 및 multiple pipe를 구현하지 못한 부분이 있다.

괄호 처리에 대한 부분은, 코드 구축 순서에 문제가 있었다고 생각한다. 이미 상당 부분의 명령어들을 구현해둔 상태였고, 그 명령어 구현을 buffer parsing에 의존하였기에 이미 parsing 된 buffer를 괄호 처리하여 묶는 것에도 문제가 생겼고, 먼저 묶어버리더라도 이후 parsing하는 것에 문제가 생기고, 연쇄적인 문제로 이어져서 결국 괄호 처리 구현을 포기하게 되었다.

multiple pipe 역시 이미 single pipe에 대해서 process 관계 및 dup2, execvp 등 복잡하고 어려운 구현을 겨우 끝낸 상태였고, 여기서 multiple pipe를 구현하기 위해 아주 작은 변경만 적용해도 연쇄적인 문제와 오류로 이어져서 결국 구현을 포기하게 되었다.

추가적으로 아쉬웠던 점은 모든 기능을 main 함수 내부에 구현한 점이다. internal command 및 execvp 초기 코드를 구현해보고 코드가 간단할 것이라고 생각하여 main 함수 내부에서만 구현을 진행했지만 pipe, redirection 등 점점 복잡하고 코드의 양도 늘어나면서 이를 후회했다. 관련 기능들을 main 외부의 함수로 구현하여 호출하는 식으로 진행했다면 코드의 가독성, 접근성 및 기능 자체도 효율적으로 구현할 수 있었을 것이다.