Programming Assignment 3

2017313264 홍진기

코드 전체적으로 입력값을 평가하기 위해 pa3 과제 설명에 안내된 순서를 따른다.

1) 명령어의 개수를 파악 (pipeline 등으로 명령어들이 연결된 경우)
(각 명령에 대해)
2) Input/output redirection이 사용되었는지 파악
3) 명령어 종류를 분석하여,
A. 직접 구현한 경우, (필요하다면) 자식 프로세스를 생성하여 해당 루틴을 호출
B. 구현하지 않은 경우, 자식 프로세스를 생성하여 해당 프로그램을 로드하여 실행
4) 자식 프로세스를 생성했을 경우, 종료될 때까지 기다림

1. swsh.c의 함수 개관

```
27  /* function prototypes */
28  void eval(char *cmdline);
29  void parseline(char *buf, char **argv);
30  int builtin_command(char **argv);
31  /* my function prototypes */
32  int redirect_in(char ** myArgv);     /* myArgv should be ended by NULL */
33  int redirect_out(char ** myArgv);
34  void loop_pipe(char ** cmd[], int command);
35  void handler();
36
37  int pgid;
```

eval, parseline, buildtin_command 함수는 기존에 있던 것을 코드에 맞게 수정했다.
redirect_in, redirect_out, loop_pipe, handler 함수는 필요에 의해 새로 만든 함수이다.
각각의 기능과 역할은 다음과 같다.

1) eval() : 명령어 문자열을 입력받아 명령어 각각에 맞는 동작을 수행한다.
2) parseline() : 명령어 문자열을 받아 공백문자를 기준으로 분리해서 배열에 저장한다.
3) builtin_command() : 주어진 명령어가 type3, 4인지 확인하고, 만약 그렇다면 적절한 동작을 수행하고 1을 리턴, type3, 4가 아니라면 0을 리턴한다.
4) redirect_in() : '<' 문자가 명령어에 섞여 있는지 확인하고 만약 그렇다면 입력을 redirect 해준다.
5) redirect_out() : '>'나 '>>'가 명령어에 섞여 있는지 확인하고 만약 그렇다면 출력을 redirect 해준다.
6) loop_pipe() : 파싱된 명령어 배열을 받아 규칙에 맞게 fork하고 pipe해서 명령어들을 실행합니다.
7) handler() : SIGINT, SIGTSTOP 시그널을 핸들링하는 함수이다.

2. eval()

코드의 주 흐름이 되는 함수이므로 eval 함수를 먼저 설명하겠다.

```
63  void eval(char *cmdline)
64  {
65      char *argv[MAXARGS]; /* Argument list execve() */
66      char buf[MAXLINE];   /* Holds modified command line */
67      int inredir, outredir, command; /* [inredir : 0-no 1-exist], [outredir : 0-
68
69      strcpy(buf, cmdline);
70      parseline(buf, argv);
71      if (argv[0] == NULL)
72      return;   /* Ignore empty lines */
73
74      /* Check redirection and pipeline */
75      inredir = outredir = 0; command = 1;
76      for (int i = 0; argv[i] != NULL; i++){
77          if (!strcmp(argv[i], "<")) inredir = 1;
78          if (!strcmp(argv[i], ">") || !strcmp(argv[i], ">>")) outredir = 1;
79          if (!strcmp(argv[i], "|")) command++;
80      }
81      int tempfd_in, tempfd_out;
82      if (inredir) tempfd_in = redirect_in(argv); /* Redirection */
83      if (outredir) tempfd_out = redirect_out(argv);
```

65-67 : argv는 파싱된 단어들을 저장하는 배열, buf는 입력된 명령어를 저장할 배열, inredir, outredir, command는 각각 redirection 과 pipeline을 해야하는지 판단하기 위해 선언한 플래그 변수이며, 각각 다음과 같은 값을 가질 수 있다.

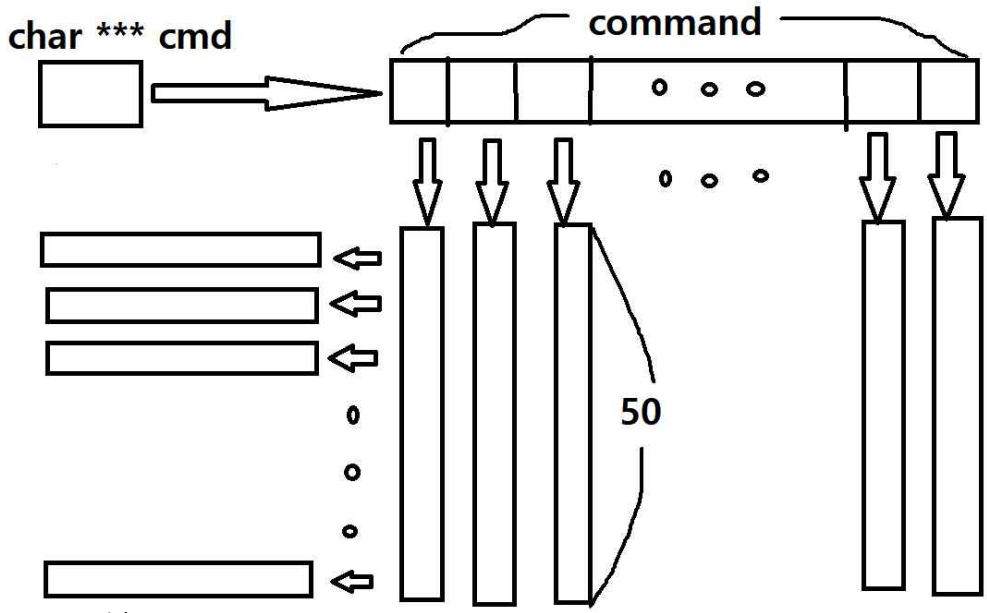
- inredir : 0-in_redirection이 없음, 1-있음
- outredir : 0-out_redirection이 없음, 1-있음
- command : 파이프라인으로 연결된 명령의 개수

69-72 : 규칙에 따라 명령어를 파싱한 후 argv에 저장한다. 자세한 규칙은 밑에서 parseline() 함수를 설명할 때 같이 설명하겠지만, 기본적으로 띄어쓰기를 기준으로 파싱하며 따옴표를 만나면 그 따옴표 안에 있는 문자열은 띄어쓰기 관계없이 그대로 저장한다.
만약 명령어가 없다면 eval함수를 끝낸다.

75-80 : 명령어들을 검사해서 '>' '>>' '<'를 찾아서 inredir, outredir, command를 규칙에 맞게 저장한다.
81-83 : 만약 redirection이 되었다면 fd값을 임시로 저장해 놓는다. 나중에 dup2로 stdin, stdout 값을 원상 복귀하기 위해서이다.

```
85      /* Parse commands for pipeline */
86      char ***cmd = malloc(sizeof(char**)*(command+1));
87      int ptr, pos = 0;
88      cmd[command] = NULL;
89      for (int i = 0; i < command; i++){
90          ptr = 0;
91          cmd[i] = malloc(sizeof(char*)*50);
92          while(argv[pos] != NULL && strcmp(argv[pos], "|") != 0){
93              cmd[i][ptr] = malloc(sizeof(char)*strlen(argv[pos]+3));
94              if (argv[pos][0] == '\'' || argv[pos][0] == '\"') {
95                  argv[pos][strlen(argv[pos])-1] = '\0';
96                  argv[pos]++;
97              }
98              strcpy(cmd[i][ptr++], argv[pos++]);
99          }
100         cmd[i][ptr] = NULL;
101         pos++;
102     }
103
104     /* Execute Command! */
105     if (!builtin_command(argv)) {
106         loop_pipe(cmd, command);
107     }
```

86-89 : 아래와 같은 구조를 갖게 할 cmd 포인터 변수를 선언했다.



89-102 : '|'를 기준으로 파싱된 명령어 문자열들을 저장한다. 이 때 배열의 마지막에 NULL을 저장해 배열의 끝을 판단할 수 있도록 한다.

105-107 : built_command() 함수를 통해 주어진 명령어가 type3, 4인지 판단하고 만약 그렇다면 적절한 동작을 수행하고, 만약 그렇지 않다면 0을 리턴해 loop_pipe() 함수를 실행하도록 한다. loop_pipe()가 실행되면, 위와 같이 저장되어 있는 명령어들을 파이프라인을 고려하며 수행한다.

```
105     /* Backup Redirection */
106     if (inredir){
107         dup2(tempfd_in, STDIN_FILENO);
108         close(tempfd_in);
109     }
110     if (outredir){
111         dup2(tempfd_out, STDOUT_FILENO);
112         close(tempfd_out);
113     }
114
115     /* free (char ***)cmd mallocs */
116     for (int i = 0; i < command; i++){
117         for (int j = 0; cmd[i][j] != NULL; j++){
118             free(cmd[i][j]);
119         }
120         free(cmd[i]);
121     }
```

106-113 : inredir, outredir 변수를 확인해 만약 앞에서 redirection이 이루어졌다면 dup2를 통해 stdin, stdout을 디폴트 값으로 되돌려준다.

116-121 : 위에서 메모리를 할당한 cmd의 메모리들을 해제해줍니다.

2. parseline()

parseline 함수는 명령어 배열을 받아 규칙에 따라 단어로 분리한 후 argv 배열에 저장합니다. 따옴표를 구별하기 위해 기존의 parseline 함수를 수정했습니다.

```
402  void parseline(char *buf, char **argv)
403  {
404      int argc;            /* Number of args */
405
406      buf[strlen(buf)-1] = ' ';  /* Replace trailing '\n' with space */
407      while (*buf && (*buf == ' ')) /* Ignore leading spaces */
408      buf++;
409
410      /* Build the argv list */
411      argc = 0;
412      while(1){
413          if (*buf == '\'' || *buf == '\"'){
414              argv[argc++] = ++buf;
415              while (*buf && *buf != '\'' && *buf != '\"')
416                  buf++;
417              if (!*buf) break;
418              *(buf++) = '\0';
419          }
420          else if (*buf == ' '){
421              buf++;
422          }
423          else if (*buf == '\0'){
424              break;
425          }
426          else {
427              argv[argc++] = buf;
428              while(*buf && *buf != ' ')
429                  buf++;
430              *(buf++) = '\0';
431          }
432      }
433      argv[argc] = NULL;
434
435      if (argc == 0)  /* Ignore blank line */
436      return;
437  }
```

406-408 : 앞의 띄어쓰기를 무시합니다.
412 : 문자열이 끝날 때까지 문자열을 한 글자씩 검사하고 평가합니다.
413-419 : 만약 따옴표를 발견한다면 따옴표 안의 문자열을 그대로 argv에 저장합니다.
420-422 : 만약 띄어쓰기를 만나면 무시합니다.
423-425 : 만약 널 문자를 만나면 문자열의 끝이므로 검사를 끝냅니다.
426-431 : 만약 따옴표가 없는 단순 문자열이라면 띄어쓰기를 기준으로 argv에 저장합니다.
433 : argv에 저장된 문자열의 마지막이 어디인지를 판단하기 위해 배열의 마지막에 NULL을 저장합니다.

3. redirect_in()

redirect_in 함수는 명령어에 '<'이 있는지 검사해서 만약 있다면 stdin의 fd를 '<' 뒤의 파일로 덮어쓰게 하는 함수입니다.

```
189  int redirect_in(char ** myArgv) {
190      int i;
191      int fd;
192      int tempfd = dup(STDIN_FILENO);
193
194      /* search forward for < */
195      for (i = 0; myArgv[i] != NULL; i++) {
196          if (!strcmp(myArgv[i], "<")) {
197              break;
198          }
199      }
200
201      if (myArgv[i]) {      /* found "<" in vector. */
202
203      if (!myArgv[i+1]) { /* File for redirection has not been provided*/
204          return -1;
205      }
206      else {            /* Open file. */
207          if ((fd =open(myArgv[i+1], O_RDONLY)) == -1) {
208              perror(myArgv[i+1]);
209              return -1;
210          }
211      }
212
213      dup2(fd,STDIN_FILENO);      /* Redirect to use it for input. */
214      close(fd);
215
216      for (i = i + 2; myArgv[i] != NULL; i++) { /* close up the vector */
217          myArgv[i-2] = myArgv[i];
218      }
219      myArgv[i-2] = NULL;      /* NULL pointer at end */
220    }
221    return tempfd;
222  }
```

192, 220 : stdin_fd의 디폴트 값을 저장했다가 리턴합니다. 이는 나중에 dup2로 바뀌어 버린 stdin_fd의 값을 복원하기 위해 필요합니다.
195-119 : "<"이 있는지 검사하고 만약 있다면 그 index를 저장합니다.
201 : myArgv[i]가 NULL이 아니라면, 즉 "<"를 찾았다면 아래의 작업을 수행한다.
203-205 : 만약 "<" 뒤에 아무 파일명도 없다면 함수를 종료한다.
206-211 : 만약 파일이 제대로 있다면 그 파일을 open합니다.
213-214 : dup2함수를 이용해 stdin_fileno가 파일을 가리키는 fd가 되게 하고, 기존의 fd는 종료합니다.
216-220 : 이후 명령어를 쉽게 검사하기 위해 명령어 배열에 있는 "<", "$filename"을 그 뒤의 문자열들이 덮도록 문자열을 왼쪽으로 두 칸씩 옮깁니다.

4. redirect_out()

redirect_out 함수는 위의 redirect_in 함수와 매우 비슷하게 동작하고 코드 또한 거의 비슷합니다. 따라서 차이점만을 짧게 짚고 넘어가겠습니다.

```
224 ▼ int redirect_out(char ** myArgv) {
225       int i, fd;
226       int flag; /* distinguish between '>' and '>>' */
227       int tempfd = dup(STDOUT_FILENO);
228
229     /* search forward for > or >> */
230 ▼     for (i = 0; myArgv[i] != NULL; i++) {
231 ▼         if (!strcmp(myArgv[i], ">")) {
232               flag = 0;
233               break;
234           }
235 ▼         else if (!strcmp(myArgv[i], ">>")) {
236               flag = 1;
237               break;
238           }
239       }
240
241 ▼     if (myArgv[i]) {      /* found ">" or ">>" in vector. */
242
243           if (!myArgv[i+1]) { /* File for redirection has not been provided*/
244               return -1;
245           }
246 ▼         else {            /* Open file. */
247 ▼             if (flag == 0){
248 ▼                 if ((fd =open(myArgv[i+1], O_RDWR | O_TRUNC | O_CREAT, 0644)) == -1) {
249                       perror(myArgv[i+1]);
250                       return -1;
251                   }
252               }
253 ▼             else if (flag == 1){
254 ▼                 if ((fd =open(myArgv[i+1], O_RDWR | O_APPEND | O_CREAT, 0644)) == -1) {
255                       perror(myArgv[i+1]);
256                       return -1;
257                   }
258               }
259           }
```

226 : flag라는 변수를 선언했습니다. redirection이 “>”에 의해서인지 “>>”에 의해서인지 구분하기 위해서입니다.
230-239 : 만약 “>”가 있다면 flag에 0을 “>>”가 있다면 flag에 1을 저장합니다.
246-258 : 만약 “>”이 있다면 O_TRUNC 옵션을 주어 파일의 내용을 모두 지우고 새로이 작성합니다. 만약 “>>”이 있다면 O_APPEND 옵션을 주어 파일의 기존 내용에 이어서 작성할 수 있도록 합니다.

```
261           dup2(fd,STDOUT_FILENO);      /* Redirect to use it for input. */
262           close(fd);
```

dup2를 STDOUT_FILENO에 해준다는 것을 제외하고는 거의 모든 코드나 동작이 redirect_in함수와 같습니다.

5. builtin_command()

builtin_command 함수는 type3, 4의 명령어들을 처리하기 위한 함수입니다. 프로세스를 따로 만들지 않고, 쉘 프로세스 내부적으로 명령들을 처리합니다.

```
127  /* If first arg is a builtin command, run it and return true */
128  int builtin_command(char **argv)
129  {
130      if (!strcmp(argv[0], "quit")) /* quit command */
131          exit(0);
132      if (!strcmp(argv[0], "&"))    /* Ignore singleton & */
133          return 1;
134
135      if (!strcmp(argv[0], "pwd")){
136          char buff[1024];
137          char *trash = getcwd(buff, 1024);
138          int trash2 = write(1, buff, strlen(buff));
139          trash2 = write(1, &"\n", 1);
140          trash++; trash2++;
141          return 1;
142      }
143      if (!strcmp(argv[0], "exit")){
144          fprintf(stderr, "exit\n");
145          if (argv[1] != NULL)
146              exit(atoi(argv[1]));
147          exit(0);
148      }
```

130-131 : “quit" 명령어를 받으면 쉘을 종료합니다.
132-134 : “&” 명령어를 받으면 무시합니다.(스켈레톤 코드에 존재했던 코드인데, 굳이 없애지 않았습니다.)
135-141 : "pwd" 명령을 받으면 getcwd()함수로 현재의 주소를 받아 출력합니다. 1을 리턴합니다. trash 변수는 warning을 없애기 위해 사용했습니다.
143-148 : "exit" 명령을 받으면 exit()함수로 쉘을 종료합니다. 기본적으로 0을 반환하지만, 만약 exit 뒤에 숫자가 있다면 프로그램의 반환값으로 그 숫자를 반환합니다.

```
149      if (!strcmp(argv[0], "rm")){
150          int ret;
151          for (int i = 1; argv[i] != NULL; i++){
152              if ((ret = unlink(argv[i])) < 0){
153                  perror("rm");
154              }
155          }
156          return 1;
157      }
158      if (!strcmp(argv[0], "cd")){
159          if (argv[2] != NULL){
160              int trash = write(2, &"cd: too many arguments\n", 23);
161              trash++;
162              return 1;
163          }
164          int ret = chdir(argv[1]);
165          if (ret < 0){
166              perror("cd");
167          }
168          return 1;
169      }
```

```
170     if (!strcmp(argv[0], "mv")){
171         if (argv[3] != NULL){
172             int trash = write(2, &"cd: too many arguments\n", 23);
173             trash++;
174             return 1;
175         }
176         int ret = rename(argv[1], argv[2]);
177         if (ret < 0){
178             perror("mv");
179         }
180         return 1;
181     }
182     return 0;                       /* Not a builtin command */
183 }
```

149-156 : “rm"을 받으면 unlink 함수를 이용해 그 뒤에 이어지는 파일들을 모두 제거합니다. 1을 리턴합니다.
158-169 : “cd"를 받으면 chdir을 이용해 디렉토리를 이동하고 1을 리턴합니다. 만약 디렉토리명이 한 가지가 아니라면 too many argument를 출력하고 1을 리턴합니다.
170-181 : “mv"을 받으면 rename함수를 이용해 파일의 이름을 바꿉니다. 파일명의 개수가 2개보다 많다면 too many arguments를 출력하고 1을 리턴합니다.

그리고 built에 쓰인 함수들에는 모두 제대로 작동되지 않는 경우 perror함수를 통해 적절한 에러를 출력하도록 했습니다. 설명서의 rm, cd, mv에 대한 에러 출력 조건에 거의 부합하는 것을 확인했습니다.

6. loop_pipe()

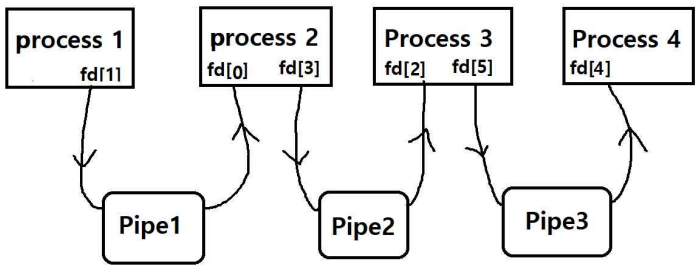
loop_pipe 함수는 명령어들을 파이프로 연결해주고, 자식 프로세스를 만들어 명령어를 실행하는 함수입니다. 필요한 명령어만큼 fork()를 한 후, 자식 프로세스를 exec를 통해 요구된 프로세스로 바꿉니다.

```
273     if (strcmp((*cmd)[0], "man") || command != 1){  //Case of not one man command
274         int status;
275         pid_t pid;
276         int *fds = malloc(sizeof(int)*2*(command-1));
277         char *cmd_type1[6] = {"ls", "man", "grep", "sort", "awk", "bc"};
278         char *cmd_type2[4] = {"head", "tail", "cat", "cp"};
279         char *cmd_type2_2[4] = {"./head", "./tail", "./cat", "./cp"};
280 
281         char tmp[200];
282         for (int j = 0; j < 4; j++){
283             if (!strcmp((*cmd)[0], cmd_type2[j])) {
284                     sprintf(tmp, "./%s", (*cmd)[0]);
285                     strcpy((*cmd)[0], tmp);
286                     break;
287                     }
288                 }
289 
290         for (int i = 0; i < command-1; i++){
291             if (pipe(fds + i*2) < 0) {
292                 perror("pipe");
293                 exit(1);
294             }
295         }
```

273 : man 명령어의 경우 위의 방법으로 실행이 되지 않았습니다. man의 결과는 제대로 나오는 것 같아 파이프라인으로 출력의 방향을 바꿀 경우 정상적으로 동작하지만, man 명령어 하나만 들어온 경우나 파이프라인 마지막 명령어가 man인 경우 man의 설명 창이 뜨지 않는 상황이 발생했습니다. 그래서 man이 들어오는 경우만 별도로 코드를 짰습니다. 일단 man 명령어가 없는 경우를 설명합니다.
276 : fds는 pipe 함수로 연결한 fd들을 저장하는 배열입니다.
277-279 : 명령어를 검사해 type1, 2에 포함되어 있지 않은 명령어는 실행시키지 않기 위해 type1, 2 명령어들을 저장한 배열을 만들었습니다.
281-288 : type2에 해당하는 명령어들, 즉 head, tail, cat, cp 앞에 ./를 붙여 제가 만든 프로그램을 실행시키도록 합니다.
290-295 : pipe함수를 통해 fds를 파이프로 연결합니다. 명령어의 개수-1개 만큼의 파이프가 필요합니다.

```
296         int j = 0;
297         pgid = 0; // Global Variable
298         while (*cmd != NULL) {
299
300             if (*(cmd + 1) == NULL && !strcmp((*cmd)[0], "man"))
301                 goto man;
302
303             pid = fork();
304             if (pgid == 0) pgid = pid;
305             setpgid(pid, pgid);
306
307             if (pid == 0) {
308                 if (*(cmd+1) != NULL){  /* if not last command */
309                     if (dup2(fds[j+1], 1) < 0){
310                         perror("dup2");
311                         exit(1);
312                     }
313                 }
314                 if (j != 0){    /*if not first command && j!=2*command */
315                     if (dup2(fds[j-2], 0) < 0){
316                         perror("dup2");
317                         exit(1);
318                     }
319                 }
320
321                 for (int i = 0; i < 2*(command-1); i++)
322                     close(fds[i]);
```

297 : 전역변수 pgid는 파이프라인을 수행할 때 첫 번째 생긴 자식 프로세스의 pid를 저장한 후, 그 다음부터 생기는 자식 프로세스의 group id를 모두 pgid로 설정합니다.
298 : cmd가 NULL이 될 때까지, 즉 파이프라인으로 연결된 명령어의 개수만큼 fork를 반복합니다.
300-301 : 우선 파이프라인의 마지막 명령어가 man일 경우 goto를 써 man 전용 코드로 보냈습니다.
303-305 : fork를 해 자식 프로세스를 만듭니다. 그리고 pgid를 설정해줍니다. 이는 첫 번째 자식은 자신의 pid를 group id로 하고 그 다음 자식들은 첫 번째 자식의 group id를 따른다는 조건을 위해서입니다.
307 : 자식 프로세스의 경우입니다.
308-313 : 파이프라인의 마지막 명령어가 아니면 stdout_fileno를 fds[홀수]로 덮어씌웁니다.
314-318 : 파이프라인의 첫 번째 명령어가 아니면 stdout_fileno를 fds[짝수]로 덮어씌웁니다. 이 두 작업을 통해 프로세스간의 파이프를 연결시킵니다.
그림으로 표현하면 아래와 같습니다.



321-322 : stdin, stdout을 이미 파이프로 연결시켜 주었으니 fds들은 닫습니다.

```
324                 /* Check if input command is in cmd_types list */
325                 int is_in_type = 0;
326                 for (int i = 0; i < 6; i++)
327                     if (!strcmp(cmd_type1[i], (*cmd)[0]))
328                         is_in_type = 1;
329                 for (int i = 0; i < 4; i++)
330                     if (!strcmp(cmd_type2_2[i], (*cmd)[0]))
331                         is_in_type = 1;
332                 if ((*cmd)[0][0] == '.' && (*cmd)[0][1] =='/')
333                     is_in_type = 1;
334 
335                 if (is_in_type == 0 || execvp((*cmd)[0], *cmd) < 0){
336                     char tmp[200];
337                     sprintf(tmp, "%s: Command not found.\n", (*cmd)[0]);
338                     int trash = write(2, tmp, strlen(tmp));
339                     trash++;
340                     exit(1);
341                 }
342             }
343             cmd++;
344             j += 2;
345         }
```

325-331 : 실행하려는 명령어가 type1, 2에 속해있는지 검사합니다. "./"이 붙은 명령어도 type1에서 파생 가능하므로, 이 또한 검사합니다. 만약 속해있다면 is_in_type 에 1을 저장합니다.

335-342 : 만약 명령어가 type에 속하지 않는다면 command not found를 출력합니다. 속한다면 execvp로 명령어를 실행시킵니다.

343 : cmd++로 '|'로 구분된 다음 명령어 덩어리로 넘어갑니다.

344 : pipe를 위해 j를 2씩 증가시킵니다.

```
347         /*Parent closes the pipes and wait for children*/
348         for(int i = 0; i < 2 *(command-1); i++){
349             close(fds[i]);
350         }
351 
352         for(int i = 0; i < command; i++)
353             waitpid(-1, &status, WUNTRACED);
354 
355         pgid = 0;
356         free(fds);
357     }
```

348-350 : 열려있는 파이프들을 모두 닫습니다.

352-353 : 명령어의 수만큼 실행된 프로세스들이 끝나는 걸 기다렸다가 시체를 회수합니다.

355 : 다음 명령어를 위해 pgid를 다시 0으로 만듭니다.

356 : fds에 할당된 메모리를 해제합니다.

```
358     else {  //Case of one man command
359         man:;
360         int   p[2];
361         pid_t pid;
362         if (pipe(p) < 0){
363             perror("pipe");
364             exit(1);
365         }
366         if ((pid = fork()) == -1) {
367             exit(EXIT_FAILURE);
368         }
369         else if (pid == 0) {
370             if (*(cmd + 1) != NULL) dup2(p[1], 1);
371              close(p[0]);
372              if (execvp((*cmd)[0], *cmd) < 0) {
373                  fprintf(stderr, "%s: Command not found.\n", (*cmd)[0]);
374                  exit(0);
375              }
376          }
377         /* Parent waits for foreground job to terminate */
378         else{
379              int status;
380              waitpid(pid, &status, 0);
381              close(p[1]);
382         }
383         pgid = 0;
384     }
385 }
```

이 코드의 경우 man 명령어를 위해 만들어졌습니다. 위의 코드에서는 man 명령어가 제대로 실행되지 않습니다.

358-359 : man 명령어만 들어왔거나 파이프라인의 맨 마지막이 man 명령어인 경우 이 코드를 실행합니다.

362-365 : p배열을 통해 파이프를 연결해줍니다.

366-368 : fork를 통해 자식 프로세스를 만듭니다.

369-376 : 출력을 파이프에 연결시킨 후 exec 함수로 man을 실행시킵니다.

378-382 : 부모 프로세스는 자식 프로세스를 기다렸다가 시체를 회수합니다.

383 : 다음 명령어를 위해 pgid를 0으로 만듭니다.

7. 시그널 처리

```
39  int main()
40  {
41      char cmdline[MAXLINE]; /* Command line */
42      char* ret;
43
44      signal(SIGINT, handler);
45      signal(SIGTSTP, handler);
46
47      while (1) {
48      /* Read */
49      printf("swsh> ");
50
51      ret = fgets(cmdline, MAXLINE, stdin);
52      if (feof(stdin) || ret == NULL)
53          exit(0);
54
55      /* Evaluate */
56      eval(cmdline);
57      }
58  }
```

main 함수에서 signal 함수를 통해 SIGINT와 SIGTSTP 시그널을 handler로 처리하도록 했다.

```
387  void handler()
388  {
389      int trash;
390      if (pgid != 0) {
391          kill(-1*pgid, SIGKILL);
392          trash = write(1, &"\n", 1);
393      }
394      else trash = write(1, &"\nswsh> ", 7);
395      trash++;
396  }
```

핸들러 함수는 다음과 같은데, 만약 pgid가 0이 아니면, 즉 명령어가 실행이 되고있는 상태라면 pgid에 속한 그룹에게 SIGKILL을 날려 강제로 종료시킵니다. write 함수들은 시그널을 보내면 쉘에 남아있는 ^C, ^Z를 보기 좋게 하기 위해 사용되었습니다.

8. type2 명령어

type2 명령어는 각각 c파일을 폴더 안에 만든 후 Makefile을 다음과 같이 수정해 실행파일이 바로 만들어질 수 있게 하였습니다.

```
CC              = gcc
CFLAGS  = -g -O -Wall
TARGET = swsh cp cat head tail

$(TARGET): swsh.c cp.c cat.c head.c tail.c
        $(CC) $(CFLAGS) -o swsh swsh.c
        $(CC) $(CFLAGS) -o cp cp.c
        $(CC) $(CFLAGS) -o cat cat.c
        $(CC) $(CFLAGS) -o head head.c
        $(CC) $(CFLAGS) -o tail tail.c

clean:
        rm -f $(TARGET)
```

make하면 다음과 같이 됩니다.

```
imnotubuntu@DESKTOP-2LFAT71:~/desktop/pa3_r$ make
gcc -g -O -Wall -o swsh swsh.c
gcc -g -O -Wall -o cp cp.c
gcc -g -O -Wall -o cat cat.c
gcc -g -O -Wall -o head head.c
gcc -g -O -Wall -o tail tail.c
```

이제 cp.c, cat.c, head.c, tail.c의 코드와 동작을 리뷰하겠습니다.

1) cp.c

```c
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]){
    int src = open(argv[1], O_RDONLY);
    if (src < 0){
        perror("cp");
        exit(1);
    }
    int dest = open(argv[2], O_RDWR | O_CREAT, 0644);
    if (dest < 0){
        perror("cp");
        exit(1);
    }

    int nbread;
    char buff[1000];
    while((nbread = read(src, buff, 1000)) > 0){
        int trash = write(dest, buff, nbread);
        trash++;
    }
    close(src);
    close(dest);
    return 0;
}
```

7-16 : 파일을 2개 받아 open합니다. src는 원본 파일, dest는 새로 만들 파일입니다. 만약 에러가 나면 perror 함수를 실행시키고 프로세스를 종료합니다.
18-23 : src의 내용을 read로 읽어와 dest에 write로 그대로 써줍니다.
24-26 : src, dest를 닫아주고 함수를 끝냅니다.

2) cat.c

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

int main(int argc, char *argv[]){
    for (int i = 1; i < argc; i++){
        int fd = open(argv[i], O_RDONLY);
        if (fd < 0){
            perror("cat");
            exit(1);
        }
        char buf[2];
        while(read(fd, buf, 1) > 0){
            int trash = write(1, buf, 1);
            trash++;
        }
        close(fd);
    }
    return 0;
}
```

10 : 여러 개의 파일이 들어올 경우 그만큼 반복한다.
11-15 : 파일을 열고 만약 오류가 난다면 perror를 통해 에러를 출력한다.
16-20 : 파일의 모든 내용을 stdout으로 내보낸다.
21 : 파일을 닫는다.

3) head.c

```
 9 int main(int argc, char *argv[]){
10     int line;
11     int trash;
12     if (strcmp(argv[1], "-n")){
13         for (int i = 1; i < argc; i++){
14             line = 1;
15             int fd = open(argv[i], O_RDONLY);
16             if (fd < 0){
17                 perror("head");
18                 exit(1);
19             }
20             if (argc > 2) {
21                 char tmp[100];
22                 sprintf(tmp, "==> %s <==\n", argv[i]);\
23                 if (i > 1) trash = write(1, &"\n", 1);
24                 trash = write(1, tmp, strlen(tmp));
25             }
26             char buf[2];
27             while(read(fd, buf, 1) > 0 && line <= 10){
28                 if (buf[0] == '\n') line++;
29                 trash = write(1, buf, 1);
30             }
31             close(fd);
32         }
33     }
```

12 : option으로 -n이 들어오냐에 따라 경우를 나눈다. 첫 번째는 들어오지 않은 경우, 두 번째는 들어온 경우이다.
13 : 여러 개의 파일을 입력받을 경우 그만큼 아래 작업을 반복한다.
15-19 : 파일을 열고 만약 실패하면 perror로 에러를 출력한다.
20-25 : 만약 파일이 하나보다 많다면, 즉 여러개의 파일을 읽어야 하는 경우 ==>[filename]<==를 띄울수 있도록 한다.(bash 쉘을 동작을 참고함)
26-30 : \n을 10번 만날 때까지, 즉 10줄이 될 때까지 파일을 읽어 stdout으로 출력한다.
31 : 파일을 닫는다.

```
34     else{
35         for (int i = 3; i < argc; i++){
36             line = 1;
37             int fd = open(argv[i], O_RDONLY);
38             if (fd < 0){
39                 perror("head");
40                 exit(1);
41             }
42             if (argc > 4) {
43                 char tmp[100];
44                 sprintf(tmp, "==> %s <==\n", argv[i]);
45                 if (i > 3) trash = write(1, &"\n", 1);
46                 trash = write(1, tmp, strlen(tmp));
47             }
48             char buf[2];
49             while(read(fd, buf, 1) > 0 && line <= atoi(argv[2])){
50                 if (buf[0] == '\n') line++;
51                 trash = write(1, buf, 1);
52             }
53             close(fd);
54         }
55     }
56     trash++;
57     return 0;
58 }
```

34 : -n 옵션이 들어온 경우이다.
35 : index 1, 2에는 각각 -n과 숫자가 저장되있을 것이므로, index 3부터 파일로 보고 읽어온다. 그리고 파일의 개수만큼 for문을 반복한다.
37-41 : 파일 하나를 open합니다. 실패할 경우 perror로 에러를 출력합니다.
42-47 : 만약 파일이 하나보다 많다면 ==>[filename]<== 식의 표지를 파일마다 출력해줍니다. 이는 bash쉘의 동작을 참고했습니다.
48-52 : argv[1]에 -n이 있다면 argv[2]에 숫자가 들어있을 것이라고 생각할 수 있고 따라서 줄 수가 argv[2]의 숫자만큼 될 때까지 파일을 읽어서 출력해줍니다.
53 : 파일을 닫습니다.

4) tail.c

```
 9  int main(int argc, char *argv[]){
10      int line;
11      int trash;
12      if (strcmp(argv[1], "-n")){
13          for (int i = 1; i < argc; i++){
14              line = 1;
15              if (argc > 2) {
16                  char tmp[100];
17                  sprintf(tmp, "==> %s <==\n", argv[i]);
18                  if (i > 1) trash = write(1, &"\n", 1);
19                  trash = write(1, tmp, strlen(tmp));
20              }
21              int fd_front = open(argv[i], O_RDONLY);
22              int fd_rear = open(argv[i], O_RDONLY);
23              char buf[2];
24              while(read(fd_front, buf, 1) > 0){
25                  if (buf[0] == '\n') {
26                      line++;
27                      if (line > 10+1) {
28                          while(read(fd_rear, buf, 1)){
29                              if (buf[0] == '\n'){
30                                  break;
31                              }
32                          }
33                      }
34                  }
35              }
36              while(read(fd_rear, buf, 1) > 0){
37                  trash = write(1, buf, 1);
38              }
39              close(fd_front);
40              close(fd_rear);
41          }
42      }
```

12 : 우선 -n 옵션이 들어오지 않은 경우를 생각한다.
13 : 파일의 개수만큼 동작을 반복한다.
15-20 : bash 쉘에서 tail의 동작을 참고해 여러개의 파일이 들어올 경우 파일 각각에 대해 ==>[filename]<==을 출력시켜줍니다.
21-22 : 두 개의 file descripter를 만듭니다. 하나는 파일의 끝을 찾는 역할, 또 하나는 일정 거리(디폴트로 10줄)를 두고 앞선 파일 디스크립터를 쫓습니다.
23-35 : fd_front 파일 디스크립터가 파일을 한 줄씩 읽다가, 10줄을 읽었을 때부터 fd_rear가 fd_front와 같이 한 줄씩 읽으며 일정한 줄간격을 유지합니다. 그리고 fd_front가 파일의

끝에 도달했을 때, fd_rear는 파일의 끝을 기준으로 10번째 줄에 와있을 것입니다.
36-38 : 그리고 fd_rear로 파일을 읽어 파일의 끝에서부터 10번 째 줄을 출력합니다.
39-40 : 파일 디스크립터들을 닫습니다.

```
43      else{
44          for (int i = 3; i < argc; i++){
45              line = 1;
46              if (argc > 4) {
47                  char tmp[100];
48                  sprintf(tmp, "==> %s <==\n", argv[i]);
49                  if (i > 3) trash = write(1, &"\n", 1);
50                  trash = write(1, tmp, strlen(tmp));
51              }
52              int fd_front = open(argv[i], O_RDONLY);
53              int fd_rear = open(argv[i], O_RDONLY);
54              char buf[2];
55              while(read(fd_front, buf, 1) > 0){
56                  if (buf[0] == '\n') {
57                      line++;
58                      if (line > atoi(argv[2])+1) {
59                          while(read(fd_rear, buf, 1)){
60                              if (buf[0] == '\n'){
61                                  break;
62                              }
63                          }
64                      }
65                  }
66              }
67              while(read(fd_rear, buf, 1) > 0){
68                  trash = write(1, buf, 1);
69              }
70              close(fd_front);
71              close(fd_rear);
72          }
73      }
74      trash++;
75      return 0;
76  }
```

43 : -n [NUM] 옵션이 들어간 경우 아래 동작을 실행합니다. 이 경우 -n 옵션이 없을 때와 거의 비슷하게 동작합니다.
44-51 : 파일이 여러개일 경우 ==>[filename]<==을 파일 각각에 대해 출력해줍니다.
52-53 : fd_front, fd_rear 파일 디스크립터를 2개 만들어 하나는 끝을 찾도록, 하나는 끝에서 일정 줄 위를 찾도록 합니다.
54-66 : fd_front가 파일의 끝을 찾도록 내려가도록 하고, fd_rear는 fd_front와 일정 줄 간격을 유지하도록 합니다. 만약 fd_front가 파일의 끝에 도달한다면 fd_rear는 파일의 끝을 기준으로 일정 줄 간격 위에 있게 될 것입니다.
67-69 : 그러면 fd_rear의 위치에서부터 파일의 내용을 출력합니다.
70-71 : 파일 디스크립터들을 닫습니다.