Shell PROJECT_3 보고서

학과 컴퓨터공학과

학번 12181597

이름 김현지

분반 002

제출일 2020.11.29

1. 요구사항 정의

1) chdir()

: cd 명령이 제대로 먹히지 않는 버그를 수정한다(제공된 코드를 실행해 cd 명령어를 입력하면 'main(): no such file or directory'가 출력됨). pwd 명령을 통해 현제 디렉토리를 확인한 뒤 mkdir dirA로 새 디렉토리 dirA를 생성, cd dirA로 디렉토리를 이동하고 다시 pwd 명령으로 현재 디렉토리가 바뀌었음을 확인한다.

2) exit()

: exit()이나 return을 이용해 exit 명령을 구현한다. exit 입력 시 myshell 프로세스가 종료되는 것을 확인한다.

3) Background

: 명령 뒤에 &가 붙으면 Background에서 실행되도록 구현한다. 고아 프로세스의 생성 원리를 이용한다. sleep 10 입력 시(Foreground) 쉘프롬프트가 바로 출력되지 않고 10초 뒤에 출력되는 것과 sleep 10 & 입력 시(Background) 쉘프롬프트가 바로 출력됨을 확인한다. 이후 ps 명령을 통해 좀비 프로세스가 생겼음을 확인하고, 생기는 이유와 다음 테스트 사항에 대해 문제점과 해결 방안에 대하여 고찰한다.

4) SIGCHLD

: 자식 프로세스 wait() 시 프로세스가 온전하게 수행되도록 구현한다. "sleep 50"이 "sleep 20 &"과 "sleep 30 &"에 방해 받지 않고 수행됨과 ps를 통해 좀비 프로세스가 없음을 확인한다.

5) SIGINT, SIGQUIT

: ^C(SIGINT), ^₩(SIGQUIT) 사용 시 쉘프롬프트가 종료되지 않아야 한다. Foreground 프로세스 실행 시에는 제어키(^C, ^₩)을 받았을 때 프로세스가 끝나야 한다. (sleep후 제어키를 입력하면 프로세스 sleep이 종료됨을 확인한다.)

6) Redirection

: Redirection 기능을 구현한다. cat > test.txt, cat < test.txt , cat < test.txt > test1.txt 명령을 통해 redirection이 제대로 동작하는지 확인한다.

7) Pipe

: 파이프 명령 "|"을 통해 별개의 프로세스에서 동작 및 데이터 전달이 제대로 이루어지도록 구현한다. 리다이렉션과 백그라운드 모두 정상적으로 동작하도록 한다.

2. 원인 분석 & 해결 방법 (pseudocode)

1) chdir()

원인 : 제공된 코드에서의 myshell 프로세스는 모든 명령어에 대한 처리를 fork()를 통해 생성한 자식 프로세스에게 맡기고, wait(NULL)로 자식 프로세스의 종료를 기다린다. 그런데 chdir()의 범위는 실행하는 그 프로세스에 한정되기 때문에 명령어를 실행하는 자식 프로세스의 디렉토리만 변경될 뿐 myshell 프로세스의 디렉토리는 변하지 않는다.

해결 방법 : 명령어가 cd일 경우 자식 프로세스를 생성하지 않고 myshell 프로세스에서 처리한다. main함수에서 makelist() 처리를 한 뒤 cmdvector[0]가 "cd"와 일치하면 myshell 프로세스에서 별도의 함수 change_dir()를 호출한다. change_dir 함수에선 cd만 입력되었을 경우 home으로, 디렉토리명이 함께 입력되었을 경우(ex. cd dirA) 해당 디렉토리로 이동하도록 chdir()를 호출한다.

pseudocode :

```
void change_dir(int t){
if t == 1   //cd 만 입력됨
      chdir(home);
else   //디렉토리명이 입력됨
      chdir(cmdvector[1]);
main {
""
int tokens = makelist(cmdline, " ₩t", cmdvector, MAX_CMD_ARG); //명령 라인의 단어 수 저장
if cmdvector[0] == "cd"
      change_dir(tokens);
else
      switch(pid = fork()){
      case 0:
                  execvp(cmdvector[0], cmdvector);
      default:
                  wait(NULL);
      }
}
```

2) exit()

원인 : cd 명령어와 같은 이유로, myshell에서 fork()로 생성한 자식 프로세스가 종료될 뿐 myshell 프로세스는 종료되지 않는다.

해결 방법 : 명령어가 exit일 경우 자식 프로세스를 생성하지 않고 myshell 프로세스에서 종료

명령을 실행한다. main 함수 안에서 if문을 통해 cmdvector[0]가 "exit"과 일치하면 exit(0) 을 실행한다.

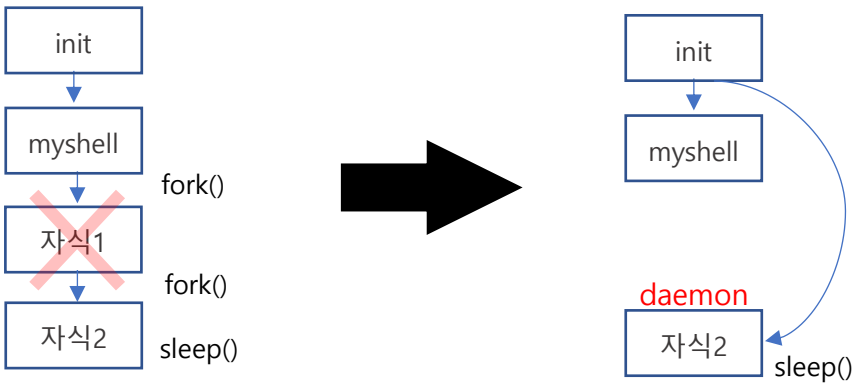
psuedocode :

```
main{
    ""
    if cmdvector[0] == "exit"
            exit(1)
    else
            ""
}
```

3) Background

원인 : sleep 10 & 과 같은 명령에서 "invalid time interval '&'" 메시지가 뜨는 것으로 보아, & 명령에 대해 Background 처리를 지원하지 않고 sleep의 시간 지정을 위한 인자로 받아들였기 때문임을 알 수 있다.

해결 방법 : 명령 문장의 끝에 "&"이 포함되는지 검사하는 라인을 추가한다. 포함될 경우 기존 방식(myshell 프로세스에서 fork후 자식 프로세스에서 실행)과는 다른 방법을 사용한다. 이 때 이용되는 것이 고아 프로세스 생성 원리이다. 고아 프로세스는 자신이 종료되기 전에 부모 프로세스가 먼저 종료된 프로세스로, init(1) 프로세스가 자동으로 해당 프로세스를 '입양'하여 daemon의 동작을 가능하게 한다. 즉, 고아 프로세스는 사용자가 직접적으로 제어하지 않고도 Background에 돌면서 작업하게 되므로 이를 "&" 구현에 이용할 수 있다. -> "&" 명령이 포함될 경우 fork()를 통해 자식1 프로세스를 생성한 뒤 자식1 프로세스에서 fork()를 호출해 자식2 프로세스를 만든다. 그 뒤 자식1 프로세스가 별다른 작업을 하지 않고 종료된다면 자식2 프로세스는 고아 프로세스가 되어 Background에서 sleep과 같은 명령을 수행한다. 그림으로 간략히 나타내면 아래와 같다.



pseudocode :

```
main{
```

```
        ""
        tokens = makelist(cmdline, " ₩t", cmdvector, MAX_CMD_ARG);
        ""
        switch(pid = fork()){
        case 0:   //자식1
                  if cmdvector[tokens -1] == "&"
                            pid2 = fork();
                            if pid2 == 0   //자식2
                                      cmdvector[tokens - 1] = NULL;   // "&" 지움
                                      execvp(cmdvector[0], cmdvector);   // 명령어 실행
                            exit(0);   //자식1 종료
        ""
    }
```

4) SIGCHLD

원인 : 백그라운드 수행 시 좀비 프로세스가 생기는 문제는 SIGCHLD 처리를 통해 해결할 수 있다. 그런데 여러 자식 프로세스를 생성하여 여러 개의 SIGCHLD를 처리하게 될 경우, 자식 프로세스 wait()시 부모 프로세스가 블로킹되어 온전하게 수행되지 못한다.

해결 방법 : SIGCHLD를 처리하는 핸들러(함수)의 waitpid 함수를 통해 좀비 프로세스를 제거한다. 자식 프로세스가 종료된 이유에는 관심이 없고 단순히 좀비 프로세스가 되는 것을 막고자 할 때 while (waitpid(-1, NULL, WNOHANG) > 0); 와 같이 사용하면 된다. (WNOHANG : 기다리는 pid(첫번째 인자가 -1일 경우 임의의 자식 프로세스)가 종료되지 않아 종료 상태를 회수할 수 없을 때 호출자는 차단되지 않고 0을 반환받는다.) 여러 자식 프로세스가 종료될 가능성이 있으므로 while 문으로 처리한다.

pseudocode :

```
main{
 ""
 signal(SIGCHLD, chld_handler);
 ""
}
static void chld_handler (int sig) {
 while (waitpid(-1, NULL, WNOHANG) > 0);
}
```

5) SIGINT, SIGQUIT

원인 : 기본 쉘은 ^C, ^\ 명령을 주어도 종료되지 않지만 myshell은 기본 쉘이 fork를 통해 생성

한 하나의 프로세스이므로 종료(default)된다.

해결 방법 : signal 함수를 통해 SIGINT, SIGTSTP, SITQUIT 명령이 들어올 때 무시(SIG_IGN)한다. myshell에서 sleep 등을 통해 생성한 자식 프로세스는 쉘이 아니므로 위와 같은 명령이 있을 때 종료될 수 있도록 fork()후의 pid가 0일 때는 SIG_DFL 옵션을 준다.

pseudocode :

```
main {
 ""
 signal(SIGINT, SIG_IGN);
 signal(SIGQUIT, SIG_IGN);
 signal(SIGTSTP, SIG_IGN);
 ""
   switch(pid = fork()){
   case 0:   //자식1
     signal(SIGINT, SIG_DFL);
     signal(SIGQUIT, SIG_DFL);
   ""
}
```

6) Redirection

구현 방법 : ">" 명령일 경우 ">" 오른쪽 파일로 출력되도록 dup2함수를 통해 경로를 바꿔준다. "<" 명령일 경우 "<" 오른쪽의 내용을 "<" 왼쪽의 입력 스트림으로 전송하도록 dup2함수를 통해 경로를 바꿔준다.

pseudocode :

```
int redirection(char* cmdvector[], int tokens){
   int i, fd;
   //in
   for( 0<=i<=tokens)
     if( "<" in cmdvector) break;
   if ( i < tokens )
     if( i == tokens - 1) perror("")
   else
```

```
        fd = open(cmdvector[i + 1], O_RDONLY)
        dup2(fd, 0);
        close(fd);
        for ( i<tokens)
           cmdvector[i] = cmdvector[i+2]
        cmdvector[i] = NULL
        tokens = tokens - 2
      // out
       for("")
        if( ">" in cmdvector) break;
       ""
       fd = open(cmdvector[i+1], O_RDWR | O_CREAT, 0644)
       dup2(fd,1)
       close(fd)
       ""
    return tokens
  }
```

7) Pipe

구현 방법 : command line을 "|"를 기준으로 분리해 저장한다. fork 함수를 통해 자식 프로세스들을 생성하고 각각의 입출력 경로를 변경한 뒤 각각 저장된 배열에 해당하는 명령어를 execvp 함수로 실행시킨다. Redirection 명령이 포함되었을 경우 해당 입출력 경로를 변경한다.

pseudocode :

```
  int my_pipe(){
     int fd[2][2]
     char* com1[20]
     char* com2[20]
     char* com3[20]
     for ( 0<=j<tokens)
     if("|' in strcmp) count++
     if(count != 0)
        /* "|"를 기준으로 토큰들을 나눠 com1, com2, com3에 저장 */
```

```
pipe(fd[0]), pipe(fd[1]), pipe(fd[2])
pid=fork()
if (pid = 0)
   pid = fork()
      if(pid>0)       //child1
         redirection(com1, tokens1)
   dup2(fd[0][1], 1)
   ""
   execvp(com1[0], com1)
      else if(pid==0)
   pid=fork()
      if(pid>0)       // child2
         dup2(fd[0][0], 0)
         dup2(fd[1][1], 1)
         ""
         execvp(com2[0], com2)
      else if(pid==0)          // child3
         dup2(fd[1][0], 0)
         redirection(com3, tokens3)
         ""
         execvp(com3[0], com3)
   return 1
else return 0
```

3. 구현 (screenshot)

1) chdir()

```
cmdline[strlen(cmdline) - 1] = '\0';

int tokens = makelist(cmdline, " \t", cmdvector, MAX_CMD_ARG); //tokens: 명령 단어의 수
if (strcmp(cmdvector[0], "cd") == 0) { //cd
    change_dir(tokens);
}
```

위 코드는 main 함수의 while(1)문 안에서 fputs, fgets 함수 실행 이후의 코드이다. makelist를 먼

저 실행한 후 명령어가 "cd"이면(cmdvector[0]가 "cd"이면) change_dir 함수에 명령줄의 단어의 수를 인자로 넘겨 실행한다. "cd"가 아니면 이후의 else문에서 fork()로 자식 프로세스를 생성해 명령을 수행한다.

```
void change_dir(int t) { //change directory
    if (t == 1) //cd만 입력: home으로 이동
        chdir(getenv("HOME"));
    else {
        if (chdir(cmdvector[1]) < 0)
            fputs("no such directory", stdout); //오류가 생겨도 프로그램을 종료하지 않고 메시지 출력만 한다.
    }
}
```

t(tokens)가 1이면 cd만 입력된 경우이므로 home으로 디렉토리를 이동하고, 1이 아닐 경우 (ex. cd dirA일 경우 t = 2) 입력된 디렉토리로 이동한다. 이동에 실패했을 경우 "no such directory" 메시지를 출력한다. 쉘에서 cd 명령어의 디렉토리명을 잘못 입력해도 쉘 자체가 종료되지 않으므로 myshell 프로세스도 종료되지 않도록 perror처리를 하지 않았다.

2) exit()

```
cmdline[strlen(cmdline) - 1] = '\0';

int tokens = makelist(cmdline, " \t", cmdvector, MAX_CMD_ARG); //tokens: 명령 단어의 수
if (strcmp(cmdvector[0], "cd") == 0) { //cd
    change_dir(tokens);
}
else if (strcmp(cmdvector[0], "exit") == 0) { //exit
    exit(0);
}
```

cd 명령과 비슷한 방법으로 cmdvector[0]가 "exit"이면 자식 프로세스를 생성하지 않고 exit(0)을 통해 바로 종료한다.

3) Background

```
else {
    switch (pid = fork()) {
    case 0:
        if (strcmp(cmdvector[tokens - 1], "&") == 0) { //background
            pid_t pid2 = fork();
            if (pid2 == 0) { //자식의 자식
                cmdvector[tokens - 1] = NULL; // "&" 지워줌
                execvp(cmdvector[0], cmdvector);
                fatal("daemon process");
            }
            exit(0);
        }
        else {
            execvp(cmdvector[0], cmdvector);
            fatal("main()");
        }
    case -1:
        fatal("main()");
    default:
        wait(NULL);
    }
}
```

명령줄에 "&"가 포함되는 경우는 cd, exit 명령이 아니므로 else문에 진입하게 된다. 우선 myshell에서의 fork()로 자식1을 생성한 뒤 cmdvector의 마지막 요소가 "&"과 일치할 경우 자식1에서 fork()로 자식2를 생성한다. 이후 자식1이 종료되면 자식2는 고아 프로세스가 되어 Background에서 명령을 수행한다.

4) SIGCHLD

```
int main(int argc, char**argv){
  int i = 0;
  pid_t pid;
  signal(SIGINT, SIG_IGN);
  signal(SIGQUIT, SIG_IGN);
  signal(SIGTSTP, SIG_IGN);
  signal(SIGCHLD, chld_handler);
```

signal 함수를 사용해 SIGCHLD 시그널을 chld_handler 함수로 넘긴다.

```
static void chld_handler(int sig) { //SIGCHLD handler
    while (waitpid(-1, NULL, WNOHANG) > 0);
}
```

while문으로 waitpid 함수를 호출하여 모든 종료된 프로세스를 거둬들인다. WNOHANG 인자를

주어 기다리지 않는다(프로세스가 온전히 수행된다).

5) SIGINT, SIGQUIT

```
int main(int argc, char**argv){
  int i = 0;
  pid_t pid;
  signal(SIGINT, SIG_IGN);
  signal(SIGQUIT, SIG_IGN);
  signal(SIGTSTP, SIG_IGN);
  signal(SIGCHLD, chld_handler);
```

signal 함수를 통해 SIGINT, SIGQUIT, SIGTSTP 시그널을 무시(SIG_IGN)한다.

```
else {
    switch (pid = fork()) {
    case 0:
        signal(SIGINT, SIG_DFL);
        signal(SIGQUIT, SIG_DFL);
        signal(SIGTSTP, SIG_DFL);
        if (strcmp(cmdvector[tokens - 1], "&") == 0) { //background
            pid_t pid2 = fork();
```

자식 프로세스는 SIGINT, SIGQUIT, SIGTSTP 에 대해 종료되거나 중단되도록 signal 함수에 SIG_DFL를 인수로 넘겨준다.

6) Redirection

```c
int redirection(char* cmdvector[], int tokens) {
    int i, fd;
    //in
    for (i = 0; i < tokens ; i++) {
        if (strcmp(cmdvector[i], "<") == 0) //토큰에 "<"가 포함되어 있다면
            break;
    }
    if (i < tokens) { // "<"이 있다면
        if (i == tokens - 1) //입력 오류 ex) " cat <"
            fatal("Redirection_in");
        else {
            if ((fd = open(cmdvector[i + 1], O_RDONLY)) == -1)
                fatal("Redirection_in");
        }
        dup2(fd, 0);
        close(fd);
        for (i; i < tokens; i++) { // "<"와 fd에 해당하는 토큰을 삭제
            cmdvector[i] = cmdvector[i + 2];
        }
        cmdvector[i] = NULL; //마지막 삭제
        tokens = tokens - 2; //전체 토큰 수 변경
    }
```

tokens는 cmdvector의 총 토큰 수이다.

cmdvector에 "<"가 포함되어 있고 입력 오류가 없다면 "<"의 오른쪽 토큰 경로를 읽기 전용으로 open해 fd에 저장한다. 이후 dup2 함수를 통해 표준 입력 경로를 바꿔주고 cmdvector에서 "<"와 fd에 해당하는 토큰을 삭제한다. tokens는 2 감소한다.

```
    //out
    for (i = 0; i < tokens; i++) {
        if (strcmp(cmdvector[i], ">") == 0) //토큰에 ">"가 포함되어 있다면
            break;
    }
    if (i < tokens) { // ">"이 있다면
        if (i == tokens - 1) //입력 오류 ex) "cat >"
            fatal("Redirection_out");
        else {
            if ((fd = open(cmdvector[i + 1], O_RDWR | O_CREAT, 0644)) == -1)
                fatal("Redirection_out");
        }
        dup2(fd, 1);
        close(fd);
        for (i; i < tokens; i++) { // ">"와 fd에 해당하는 토큰을 삭제
            cmdvector[i] = cmdvector[i + 2];
        }
        cmdvector[i] = NULL; //마지막 삭제
        tokens = tokens - 2; //전체 토큰 수 변경
    }
    return tokens;
}
```

">"의 경우도 비슷하다. cmdvector에 ">"가 포함되어 있고 입력 오류가 없다면 ">"의 오른쪽 토큰 경로를 읽기-쓰기로(파일이 없다면 생성) open해 fd에 저장한다. 파일에 대한 권한은 0644로 지정한다. 이후 dup2 함수를 통해 표준 입력 경로를 바꿔주고 cmdvector에서 ">"와 fd에 해당하는 토큰을 삭제한다. tokens는 2 감소한다.

마지막으로 tokens를 리턴한다.

7) Pipe

```
int my_pipe() {
    int fd[2][2];
    int f, j, k;
    int count;
    int tokens1, tokens2, tokens3;
    pid_t pid;

    char* com1[100];
    char* com2[100];
    char* com3[100];

    for (j = 0; j < tokens; j++) {
        if (strcmp(cmdvector[j], "¦") == 0)
            count++;
    }
```

cmdvector에 포함된 "|"의 개수를 센다. count가 0일 경우 아무 처리도 하지 않는다.

```
if (count != 0) {
    j = 0;
    while (1) { // "¦" 앞의 토큰들 저장
        if (strcmp(cmdvector[j], "¦") == 0)
            break;
        com1[j] = cmdvector[j];
        j++;
    }
    com1[j] = NULL;
    tokens1 = j;
```

```
j++;
k = 0;
while (1) { // "¦" 사이의 토큰들 저장
    if (strcmp(cmdvector[j], "¦") == 0)
        break;
    com2[k] = cmdvector[j];
    k++; j++;
}
com2[k] = NULL;
tokens2 = k;
```

```
j++;
k = 0;
while (1) { // "¦" 뒤의 토큰들 저장
    if (cmdvector[j] == NULL)
        break;
    com3[k] = cmdvector[j];
    k++; j++;
}
com3[k] = NULL;
tokens3 = k;
```

cmdvector를 "|"를 기준으로 나누어 com1, com2, com3 배열에 저장한다. tokens1, tokens2, tokens3 변수의 각각의 배열에 포함되는 토큰 수도 함께 저장한다. redirection 함수를 호출할 때 필요하기 때문이다.

```
if (pipe(fd[0]) || pipe(fd[1]) || pipe(fd[2])) {
    fatal("pipe");
}

pid = fork();
if (pid == 0) {
    pid = fork();
    if (pid > 0) { //child1
        redirection(com1, tokens1);
        dup2(fd[0][1], 1); //child1->child2
        close(fd[2][0]); close(fd[2][1]);
        close(fd[0][1]); close(fd[1][0]);
        close(fd[1][1]); close(fd[0][0]);

        execvp(com1[0], com1);
    }
```

pipe함수와 fork함수를 호출한다.

child1 함수의 경우 com1에 속하는 명령을 처리하게 되는데, "<"가 포함될 가능성이 있으므로 redirection 함수를 거쳐 준다. 이후 dup2(fd[0][1], 1)로 표준 출력이 해당 파이프로 가게 한다. 이후 execvp 함수를 호출한다.

```
else if (pid == 0) {
    pid = fork();
    if (pid > 0) { //child2
        dup2(fd[0][0], 0); //child1->child2
        dup2(fd[1][1], 1); //child2->child3
        close(fd[2][0]); close(fd[2][1]);
        close(fd[0][1]); close(fd[1][0]);
        close(fd[1][1]); close(fd[0][0]);
        execvp(com2[0], com2);
    }
```

child2 함수의 경우 com2에 속하는 명령을 처리하게 된다. dup2(fd[0][0], 0)으로 표준 입력을, dup2(fd[1][1], 1)로 표준 출력을 변경한다. 이후 execvp함수를 호출한다.

```
    else if (pid == 0) { //child3
        dup2(fd[1][0], 0); //child2->child3
        redirection(com3, tokens3);
        close(fd[0][0]); close(fd[0][1]);
        close(fd[1][1]); close(fd[2][0]);
        close(fd[1][1]); close(fd[0][0]);

        execvp(com3[0], com3);
    }
```

child3 함수의 경우 com3에 속하는 명령을 처리하게 된다. dup2(fd[1][0], 0)으로 표준 입력을 해당 파이프로 변경하고, ">"가 포함될 가능성이 있으므로 redirection 함수를 거쳐 준다. 이후 execvp 함수를 호출해 명령을 실행하도록 만든다.

```
                else {
                    fatal("pipe");
                }
            }
            else {
                fatal("pipe");
            }
        }
        else if (pid < 0) {
            fatal("pipe");
        }

        return 1;
    }
    else
        return 0;
}
```

에러 처리를 하고 count != 0이었을 경우 1, "|"가 하나도 포함되지 않았을 경우 0을 리턴한다. 이

는 my_pipe()함수에서 redirection과 fork처리를 모두 하기 때문에 main함수에서 동일 함수를 중복 호출하지 않도록 하기 위함이다.

```
case 0:
    signal(SIGINT, SIG_DFL);
    signal(SIGQUIT, SIG_DFL);
    signal(SIGTSTP, SIG_DFL);
    if (my_pipe() == 0) {
        tokens = redirection(cmdvector, tokens);
        if (strcmp(cmdvector[tokens - 1], "&") == 0) { //background
            pid_t pid2 = fork();
            if (pid2 == 0) { //자식의 자식
                cmdvector[tokens - 1] = NULL; // "&" 지워줌
                execvp(cmdvector[0], cmdvector);
                fatal("daemon process");
            }
            exit(0);
```

main함수에서 위와 같이 사용한다.

4. 구현 시 문제점과 해결 방법

1) Background

자식2 프로세스에서 바로 execvp(cmdvector[0], cmdvector);를 실행하려 하니 "invalid time interval '&'" 오류가 떴다. cmdvector의 마지막 요소가 '&'인데 cmdvector를 그대로 실행 인자로 넘겨줬기 때문이다. 바로 윗줄에 cmdvector[tokens - 1] = NULL;를 추가해 "&"를 NULL로 만듦으로써 문제를 해결했다.

2) Segmentation fault

1차 myshell을 구현하면서 가장 어려움을 겪었던 문제이다. 기본 쉘에서는 명령어 입력 없이 엔터를 치면 다음 줄에 $이 나타나 다른 명령을 쓸 수 있는데, myshell에서는 Segmentation fault(이하:Segfault)가 발생해 쉘이 종료되었다. Segfault는 프로그램이 허용되지 않은 메모리 영역에 접근을 시도하거나, 허용되지 않은 방법으로 메모리 영역에 접근을 시도할 경우에 발생하는데, 코드상으로는 문제가 발생한 지점을 알아내기 어려워 디버깅을 해 보았다.

```
hyeonji@hyeonji-VirtualBox:~/Desktop$ gdb -q ./a.out
Reading symbols from ./a.out...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/hyeonji/Desktop/a.out
myshell>

Program received signal SIGSEGV, Segmentation fault.
__strcmp_sse2_unaligned ()
    at ../sysdeps/x86_64/multiarch/strcmp-sse2-unaligned.S:31
31      ../sysdeps/x86_64/multiarch/strcmp-sse2-unaligned.S: No such file or dir
ectory.
```

31번째 줄에서 문제가 생겼다.

```
17 int makelist(char *s, const char *delimiters, char** list, int MAX_LIST){
18   int i = 0;
19   int numtokens = 0;
20   char *snew = NULL;
21
22   if( (s==NULL) || (delimiters==NULL) ) return -1; //s:cmdline
23
24   snew = s + strspn(s, delimiters);      /* Skip delimiters */
25   if( (list[numtokens]=strtok(snew, delimiters)) == NULL )
26     return numtokens;
27
28   numtokens = 1;
29
30   while(1){
31     if( (list[numtokens]=strtok(NULL, delimiters)) == NULL)
32                   break;
33     if(numtokens == (MAX_LIST-1)) return -1;
34     numtokens++;
35   }
36   return numtokens;
```

vim으로 확인해 보니 makelist의 if문이 문제였다. main에서 cmdline[strlen(cmdline) -1] = '₩0'을 통해 개행문자가 ₩0으로 바뀌면서 makelist에서 25번째 줄에 걸려 numtokens로 0을 리턴하게 될 줄 알았는데 그냥 통과해 while까지 진입했다. 조사해 보니 NULL과 같은 것은 0이고, ₩0은 NUL과 일치함을 알 수 있었다. NULL과 널 문자(₩0)는 애초에 달랐던 것이다. 이 문제를 해결하기 위해 많은 시도를 하다가 선택한 가장 간단한 방법은 '₩n'만 입력되었을 경우 아무 코드도 실행하지 않는 것이다. cmdline이 '₩n'과 일치하지 않을 때만 그 후의 작업을 수행할 수 있도록 if문을 만들었다.

```
if (*cmdline != '\n') { //엔터만 입력했을 경우 생기는 segmentation fault 방지
    cmdline[strlen(cmdline) - 1] = '\0';

    int tokens = makelist(cmdline, " \t", cmdvector, MAX_CMD_ARG); //tokens: 명령 단어의 수
    if (strcmp(cmdvector[0], "cd") == 0) { //cd
        change_dir(tokens);
    }
    else if (strcmp(cmdvector[0], "exit") == 0) { //exit
        exit(0);
    }
    else {
```

위 코드로 수정해 실행했더니 엔터만 입력할 때 myshell>이 출력되고 오류가 발생하지 않았다. (if(cmdline != '₩n')이라고 쓸 경우 오류 발생. pointer와 integer 사이의 비교를 수행할 수 없기 때문)

```
hyeonji@hyeonji-VirtualBox:~/project1$ ./a.out
myshell>
myshell>
myshell>
```

3) Redirection 시 Permission denied 문제

```
myshell> cat > t.txt
hhh
myshell> cat < t.txt
Redirection_in: Permission denied
myshell>
```

위와 같이 ">"로 파일에 쓴 후 "<"로 해당 파일을 읽으려 할 때 Permission denied 에러가 출력되었다. ls -l로 해당 파일의 권한을 확인해 보니 '----------'로 설정됨을 알 수 있었다.

```
if ((fd = open(cmdvector[i + 1], O_RDWR | O_CREAT)) == -1)
    fatal("Redirection_out");
```

파일을 open(create)할 때 기존 코드를 아래와 같이 0644 퍼미션을 지정해 읽기 권한을 줌으로써 문제를 해결했다.

```
if ((fd = open(cmdvector[i + 1], O_RDWR | O_CREAT, 0644)) == -1)
    fatal("Redirection_out");
```

```
myshell> cat > test.txt
inha univ.
myshell> cat < test.txt
inha univ.
myshell> cat < test.txt > test1.txt
myshell> cat test1.txt
inha univ.
```

성공적으로 출력되었다.

4) Bad address

```
myshell> cat < ls.txt | grep ^- | wc -l > d.txt
main(): Bad address
```

Redirecton + Pipe 명령 실행 시 위와 같이 Bad address 오류가 떴다. 자료를 찾아 봐도 'EFAULT 14 Bad address' 와 같은 정보 외에는 어떤 원인으로 해당 오류가 났는지 확인할 수 없었다.

```
myshell> cat < ls.txt | grep ^- | wc -l
myshell> 7
```

위와 같은 명령으로 테스트해보니 잘 출력되는데 "> d.txt"를 포함할 경우 오류가 뜨는 것으로 보아 my_pipe()함수의 child3(com3를 실행하는 부분)에서 redirection 함수 또는 execvp 함수를 호출할 때 에러가 난 것으로 추측된다. 그러나 child1 부분과 구현 방식이 크게 다르지 않았기 때문에 특별히 고쳐야 할 부분을 알아내지 못하였다.

5. 테스트 사항에 대한 실행 결과

1) chdir()

```
hyeonji@hyeonji-VirtualBox:~/project1$ ./myshell
myshell>
myshell> pwd
/home/hyeonji/project1
myshell> mkdir dirA
myshell> cd dirA
myshell> pwd
/home/hyeonji/project1/dirA
myshell>
```

: cd 명령을 통해 디렉토리가 정상적으로 바뀌는 것을 확인할 수 있다.

2) exit()

```
myshell> pwd
/home/hyeonji/project1/dirA
myshell> exit
hyeonji@hyeonji-VirtualBox:~/project1$
```

: exit 명령을 통해 myshell 프로세스가 종료되는 것을 확인할 수 있다.

3) Background

```
hyeonji@hyeonji-VirtualBox:~/project1$ ./myshell
myshell> sleep 10
```

```
myshell> sleep 10 &
myshell>
```

(1) sleep 10은 Foreground이므로 쉘프롬프트가 바로 출력되지 않고 10초 후에 출력된다.

sleep 10 &은 Background이므로 쉘프롬프트가 바로 출력되었다.

```
13319 pts/0    00:00:00 myshell
13320 pts/0    00:00:00 ps <defunct>
13321 pts/0    00:00:00 ps
```

(2) mysh> ps -u 1000 &

mysh> ps -u 1000 &

mysh> ps -u 1000 &

mysh> ps -u 1000 에 대한 결과이다.

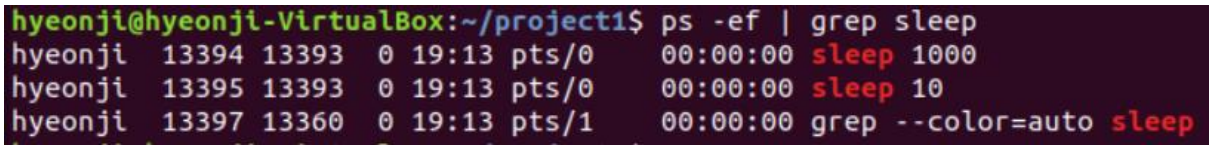
<u>좀비 프로세스가 생기는 이유</u> : 좀비 프로세스는 자식 프로세스가 먼저 종료되었는데 부모 프로세스가 자식 프로세스의 종료 상태를 회수하지 못했을 때 그 자식 프로세스를 말한다. Background에서 실행하기 위해 자식1이 자식2 프로세스를 만든 뒤 종료되므로 자식2는 init 프로세스에 의해 입양되는데, 이때 자식2의 수행이 끝났다고 해서 바로 wait 해주는 게 아니라 주기적으로 wait을 하기 때문에 자식2가 종료되고 회수되기 전까지의 시간에는 좀비 프로세스가 된다.

```
myshell> sleep 10 &
myshell> sleep 20 &
myshell> ps
  PID TTY          TIME CMD
13282 pts/0    00:00:00 bash
13609 pts/0    00:00:00 myshell
13611 pts/0    00:00:00 sleep
13613 pts/0    00:00:00 sleep
13614 pts/0    00:00:00 ps
```

(3) Background를 기다리지 않고 쉘프롬프트가 출력된다.

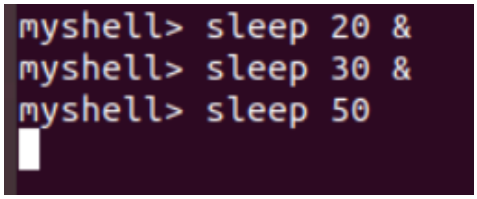
이 테스트의 문제점 : Background 실행을 위해 fork() 함수를 자주 호출하게 되는데, 이때 자식 프로세스는 부모 프로세스와 동일한 데이터 복사본을 갖게 된다. 이는 자식 프로세스가 execvp를 호출할 경우 불필요한 오버헤드가 된다.

해결 방법 : vfork()함수를 사용하여 자식 프로세스가 종료되거나 execvp를 호출하기 전까지 부모 프로세스의 데이터 부분 참조를 통해 부모 프로세스의 주소 공간에서 동작할 수 있도록 한다.

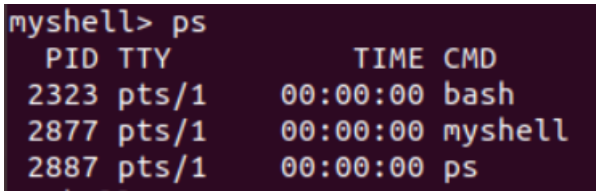


(4) sleep이 잘 실행됨을 알 수 있다.

4) SIGCHLD



(1) sleep 50이 sleep 20 & 과 sleep 30 & 에 방해받지 않고 수행되는 것을 확인



(2) ps를 통해 좀비 프로세스가 없음을 확인

5) SIGINT, SIGQUIT

```
myshell> ^C
myshell> ^C
myshell> ^\
myshell> ^\
myshell>
```

(1) 쉘 프롬프트가 ^C, ^₩에 의해 종료되지 않음을 확인

```
myshell> sleep 30
^Cmyshell>
```

(2) 제어키(^C, ^₩)을 받아 포그라운드 프로세스 sleep이 죽는 것을 확인

6) Redirection

```
myshell> cat > test.txt
inha univ.
myshell> cat < test.txt
inha univ.
myshell> cat < test.txt > test1.txt
myshell> cat test1.txt
inha univ.
```

세 가지 명령에 대해 Redirection이 잘 실행됨을 확인

7) Pipe

```
myshell> ls -l > ls.txt
myshell> cat ls.txt
total 28
-rw-r--r-- 1 hyeonji hyeonji     0 11월 29 01:02 ls.txt
-rwxr-xr-x 1 hyeonji hyeonji 13584 11월 29 00:58 myshell
-rw-rw-rw- 1 hyeonji hyeonji  3718 11월 29 00:58 simple_myshell.c
-rw-r--r-- 1 hyeonji hyeonji    11 11월 29 01:00 test1.txt
-rw-r--r-- 1 hyeonji hyeonji    11 11월 29 00:59 test.txt
```

(1) ls 결과를 ls.txt에 저장해 ls.txt 내용을 확인

```
myshell> cat < ls.txt | grep ^- | wc -l > d.txt
main(): Bad address
```

(2) ls.txt로부터 디렉터리를 필터링하는 명령 - Bad address

```
myshell> cat < ls.txt | grep ^- | wc -l
myshell> 7
```

위와 같이 "<"만 포함된 경우 해당 개수를 출력할 수 있음

```
myshell> ls -l > ls.txt &
myshell> cat ls.txt
total 36
-rw-r--r-- 1 hyeonji hyeonji     2 11월 29 01:14 dir_num.txt
-rw-r--r-- 1 hyeonji hyeonji   310 11월 29 01:11 ls.txt
-rwxr-xr-x 1 hyeonji hyeonji 13584 11월 29 00:58 myshell
-rw-rw-rw- 1 hyeonji hyeonji  3718 11월 29 00:58 simple_myshell.c
-rw-r--r-- 1 hyeonji hyeonji    11 11월 29 01:00 test1.txt
-rw-r--r-- 1 hyeonji hyeonji    11 11월 29 01:05 test.txt
```

(3) ls.txt로의 출력을 백그라운드로 처리