[운영체제론] 프로젝트1 : 미니 쉘 만들기

> 2017012642 강동진

- 알고리즘

1. fgets를 통해 명령어를 입력받는다.
2. 입력받은 명령어를 strtok를 통해 공백 단위로 split하여 캐릭터 포인터 배열에 삽입한다.
3. fgets로 받은 문자열은 개행 문자를 포함하고 있으므로 개행 문자를 제거해준다.
4. 공백단위로 쪼개져 있는 명령어, 옵션, |, <, > & 를 파악하기 위해 각각의 flag를 만들고, 반복문을 통해 각각의 위 치를 파악한다.
5. 만약 파이프가 없는 경우, 한가지 명령어만 처리한다.
    1. 자식 프로세스를 하나 생성한다.

> 자식 프로세스 :
>
> 1. '>' 플래그가 켜져있는 경우,
>   표준 출력을 주어진 파일로 바꾸어 준다. 주어진 명령어를 실행한다.
> 2. '<' 플래그가 켜져있는 경우,
>   표준 입력을 주어진 파일로 바꾸어 준다. 주어진 명령어를 실행한다.
> 3. 둘다 아닌 경우,
>   주어진 명령어를 실행한다.

> 부모 프로세스 :
>
> 1. '&' 플래그가 켜져있는 경우,
>   백그라운드 실행을 위해 waitpid 에 WNOHANG 옵션을 준다.
> 2. '&' 플래그가 꺼져있는 경우,
>   waitpid의 옵션에 0을 주어 wait과 같은 기능을 하도록 한다.

6. 파이프가 있는 경우, 2개의 명령을 처리한다.
    1. '|' 이후의 문자열을 2번째 명령으로 간주하고 새로운 캐릭터 포인터 배열로 만든다.
    2. 자식 프로세스를 생성한다.

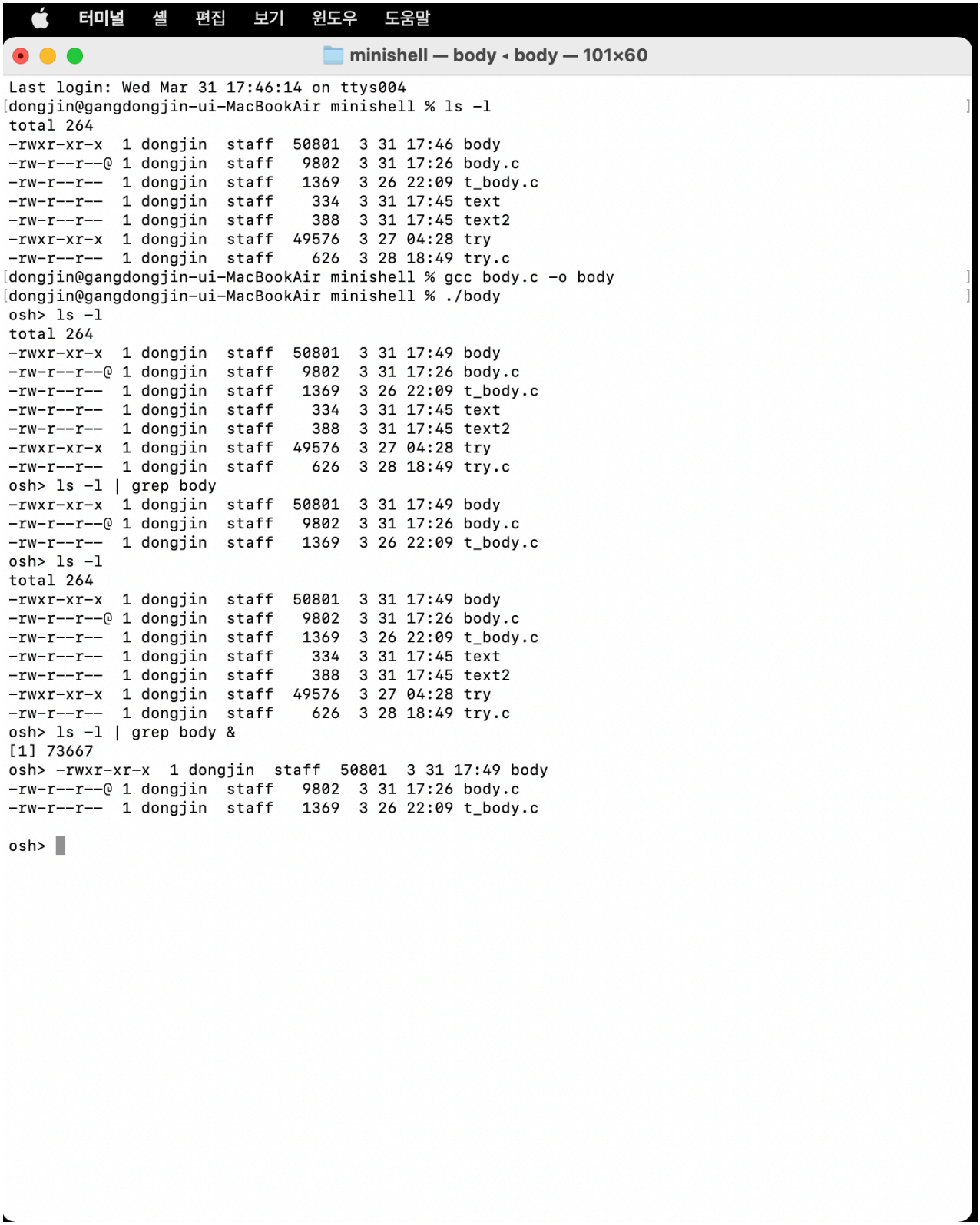
> 자식 프로세스:
>
> 1. 자손 프로세스를 생성한다.
>
>> 자손 프로세스 :
>>
>> 1. 파이프를 만든다.
>> 2. 표준 출력을 파이프로 지정한다.
>> 3. 처음 캐릭터 포인터 배열에 대해 "파이프가 없는 경우" 를 실행한다.

> 2. 표준 입력을 파이프로 지정한다.
> 3. 자손 프로세스를 기다린다.
> 4. 새로만든 캐릭터 포인터 배열에 대해 “파이프가 없는 경우”를 실행한다.
>   (단, 자손과 자식 프로세스는 순차적으로 일어나야 하기 때문에, wait을 사용해 준다.)

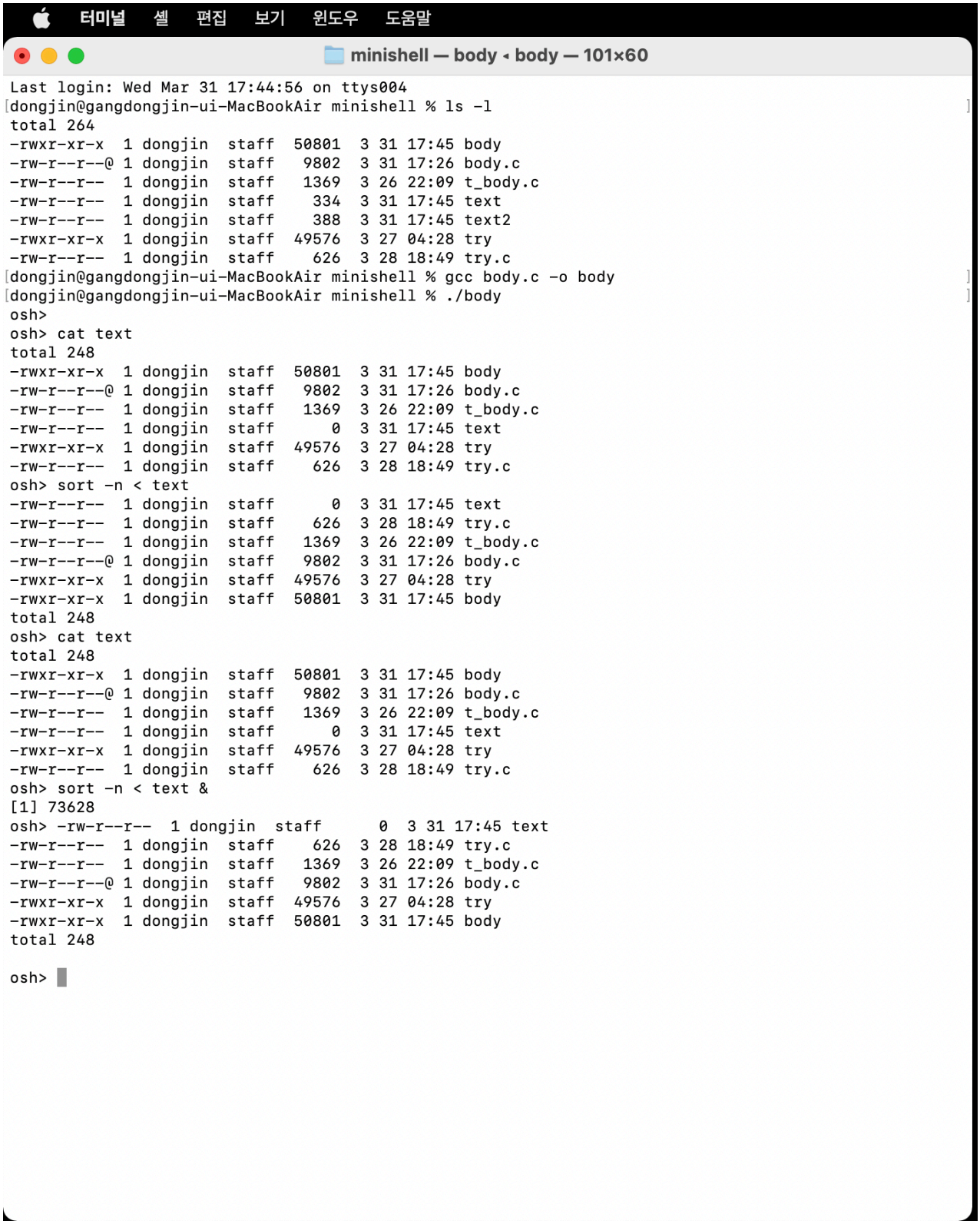
> 부모 프로세스 :
>
> 1. ‘&’ 플래그가 켜져있는 경우, 백그라운드 실행을 위해 waitpid 에 WNOHANG 옵션을 준다.
> 2. ‘&’ 플래그가 꺼져있는 경우, waitpid의 옵션에 0을 주어 wait과 같은 기능을 하도록 한다.

- 컴파일 과정

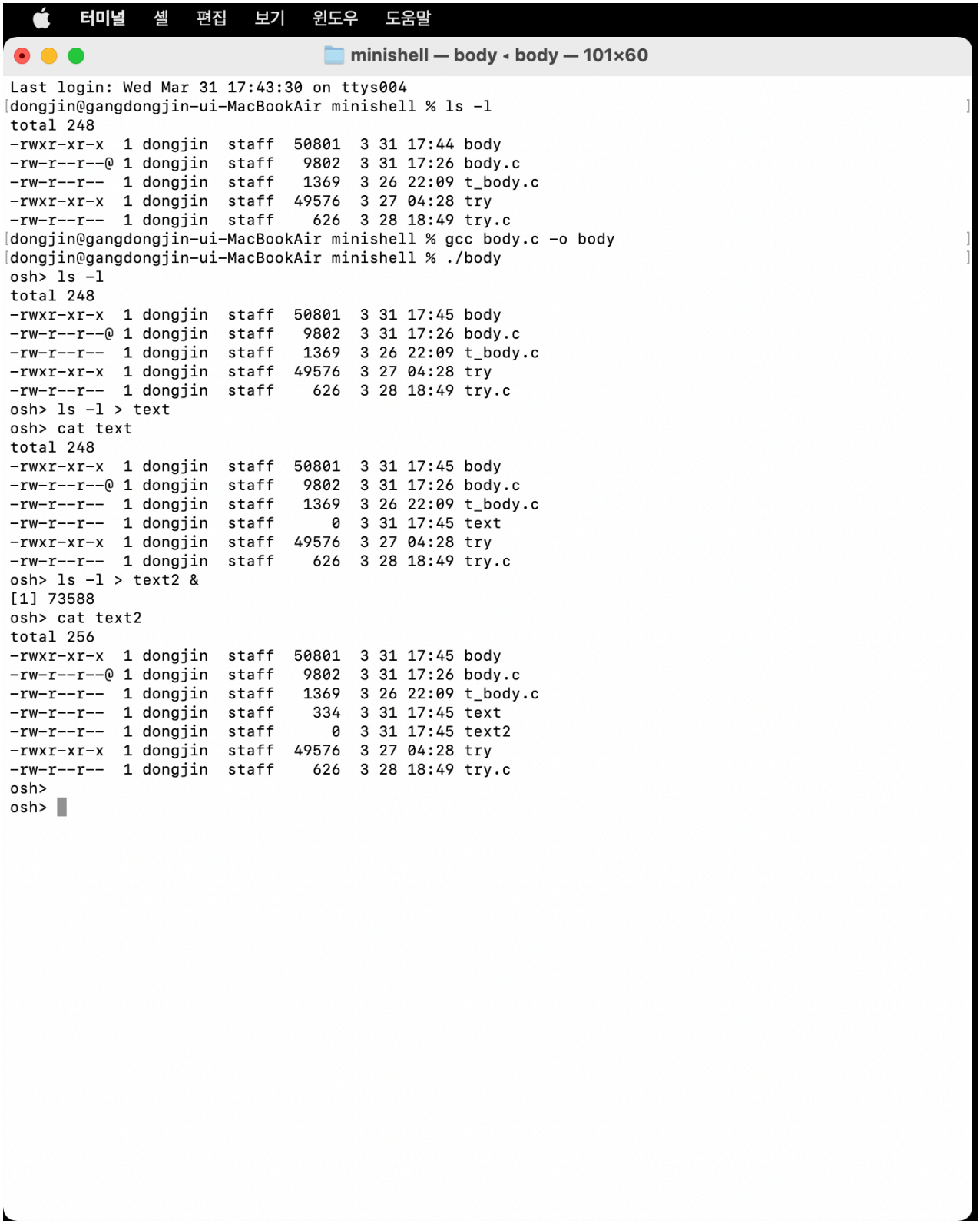
1. `ls -l` 명령어와 `ls -l &` 명령어를 수행하였습니다.
    1. `ls -l &` 명령어의 경우, 백그라운드로 실행되어 "osh>" 이후에 결과물이 출력되었습니다.



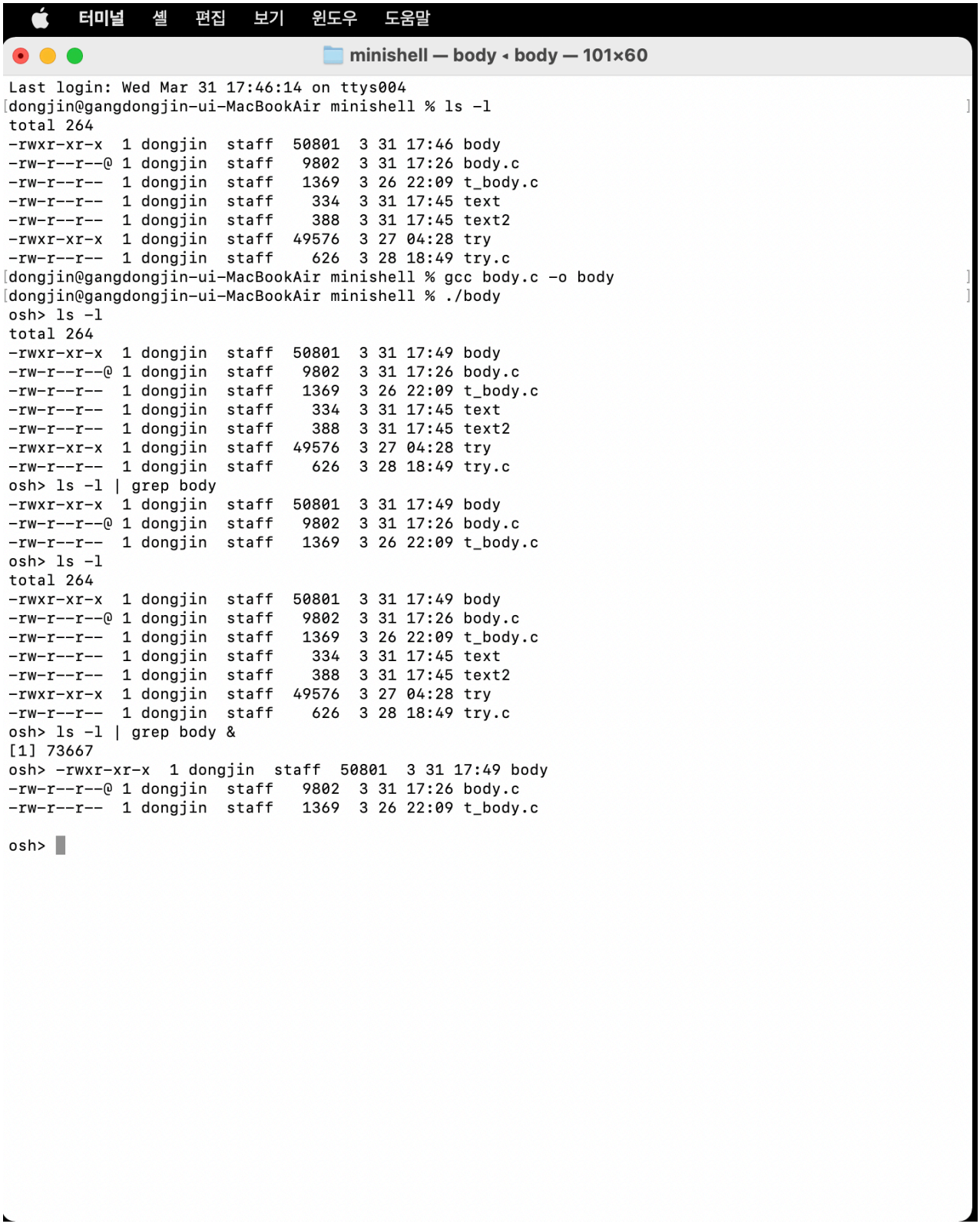
2. `sort -n < text` 명령과 `sort -n < text &` 명령을 수행하였습니다.
    1. 처음 `cat text` 를 통해 본 text파일은 정렬이 되어있지 않다.
    2. `sort -n < text` 명령을 수행 후 결과물을 보면, 문자열이 아닌 숫자로 정렬된 결과를 볼 수 있고, 이는 옵션이 성공적으로 수행되었다는 것을 알 수 있다.
    3. `sort -n < text &` 명령의 경우 백그라운드로 실행되어 "osh>" 가 먼저 출력된 후, 결과물이 출력되는 것을 볼 수 있다.



3. `ls -l > text` 명령 과 `ls -l > text2 &` 명령을 수행하였습니다.
    1. 이전에 수행한 `ls -l` 명령을 보면 text파일이 없다.
    2. `ls -l > text` 명령을 수행 후, `cat text` 명령을 통해 text의 내용을 파악해 보면, text파일이 추가된 결과가 들어있는 것을 볼 수 있다.
    3. `ls -l > text2 &` 명령의 경우, 이전의 명령과 차이점을 주기 위해 text2 파일을 대상으로 하였고, 성공적으로 수행되었다.
    4. `ls -l > text2 &` 명령의 경우, 백그라운드로 실행되기 때문에 "osh>" 가 출력된 후, 결과물이 출력되었다.



4. `ls -l | grep body` 명령과 `ls -l | grep body &` 명령을 수행하였다.
    1. `ls -l` 한 결과 와 `ls -l | grep body` 한 결과를 비교해 보면, `ls -l | grep body` 가 잘 수행된 것을 볼 수 있다.
    2. `ls -l | grep body &` 를 수행한 결과 백그라운드로 실행되기 때문에, "osh>" 가 먼저 출력되고, 그 이후에 결과물이 출력된 것을 볼 수 있다.



- 소스 코드

```
/**
 * @file body.c
 * @author 2017012642 강동진
 * @brief
 *  [운영체제론] 1차 프로젝트 과제
 *  미니 쉘 구현하기.
```

```
 *
 *  -- 기능 --
 *  명령어 or 명령어 &
 *  명령어 > 파일 or 명령어 > 파일 &
 *  명령어 < 파일 or 명령어 < 파일 &
 *  명령어1 | 명령어2 or 명령어1 | 명령어2 &
 *  (단 명령어는 옵션을 포함함.)
 *
 * @version 0.1
 * @date 2021-03-28
 *
 * @copyright Copyright (c) 2021
 *
 */

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

#define MAX_LINE 80 /* The maximum length command */

// "파이프가 없는 경우" 명령어를 실행하는 함수.
int exeInstruction(int RD_Lflag, int RD_Rflag, int BG_flag, char *argp[]);

int main(void)
{
    char *args[MAX_LINE / 2 + 1]; /* command line arguments */
    int should_run = 1;           /* flag to determine when to exit program */

    while (should_run)
    {
        printf("osh> ");
        fflush(stdout);

        // fgets을 통해 표준 입력으로 명령을 받는다.
        char input[MAX_LINE];
        if (fgets(input, MAX_LINE, stdin) == NULL)
        {
            perror("fgets error");
        }

        // strtok를 통해 공백 단위로 명령을 split하여 chat* args[] 에 저장한다.
        char *strtok_p = strtok(input, " ");
        int args_length = 0;
        while (strtok_p != NULL)
```

```
        {
            args[args_length] = strtok_p;
            strtok_p = strtok(NULL, " ");
            args_length++;
        }

        // execvp 의 args의 마지막 인자로 NULL이 필요하기 때문에 NULL문자를 삽입해 준다.
        args[args_length] = NULL;

        // 입력에 포함되어있는 개행문자를 제거한다.
        int leng = strlen(args[args_length - 1]);
        *(args[args_length - 1] + (leng - 1)) = '\0';

        // 명령어 탐색
        int pipe_flag = 0; // args에서 '|'의 위치
        int BG_flag = 0;   // args에서 '&'의 위치
        int RD_Lflag = 0;  // args에서 '>'의 위치
        int RD_Rflag = 0;  // args에서 '<'의 위치
        for (int i = 0; i < args_length; i++)
        {
            if (*args[i] == '|')
            {
                pipe_flag = i;
            }

            if (*args[i] == '&')
            {
                BG_flag = i;
            }

            if (*args[i] == '>' && RD_Rflag == 0)
            {
                RD_Rflag = i;
            }

            if (*args[i] == '<' && RD_Lflag == 0)
            {
                RD_Lflag = i;
            }
        }

        /* 파이프가 있는 경우, 명령어2를 저장할 배열을 만든다. */
        char *args2[MAX_LINE / 2 + 1];

        // 마찬가지로 플래그들을 생성해준다.
        int RD_Lflag2 = 0;
        int RD_Rflag2 = 0;
        int BG_flag2 = 0;
```

```
        // pipe가 있는 경우, 명령어2 부분을 args2[]에 저장한다.
        if (pipe_flag)
        {
            // args의 '|' 문자를 NULL로 바꾸어 준다.
            args[pipe_flag] = NULL;

            // args의 '|' 이후의 내용을 args2로 복사한다.
            for (int i = 1; i < (MAX_LINE / 2 + 1); i++)
            {
                if (args[pipe_flag + i] == NULL)
                {
                    args2[i - 1] = NULL;
                    break;
                }
                else
                {
                    args2[i - 1] = args[pipe_flag + i];
                }
            }

            // 명령어2 에서 사용될 플래그들의 위치를 저장한다.
            for (int i = 0; args2[i] != NULL; i++)
            {

                if (*args2[i] == '>')
                {
                    RD_Rflag2 = i;
                }

                if (*args2[i] == '<')
                {
                    RD_Lflag2 = i;
                }

                if (*args2[i] == '&')
                {
                    BG_flag2 = i;
                }
            }
        }

        // & 제거
        if (BG_flag)
        {
            args[BG_flag] = NULL;
            args2[BG_flag2] = NULL;
        }

        // exit 검사
```

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

        // ------------------------------------------------------------------------
------------
        // pipe(|) 없을 때 동작

        if (pipe_flag == 0)
        {
            exeInstruction(RD_Lflag, RD_Rflag, BG_flag, args);
        }

        // ------------------------------------------------------------------------
------------
        // pipe(|) 있을 때 동작

        else
        {
            // 자식 프로세스 생성
            pid_t child_pid = fork();

            if (child_pid < 0)
            {
                perror("pipe fork error");
            }

            // 자식 프로세스
            else if (child_pid == 0)
            {
                // 파이프 생성
                int pipe_fd[2];

                if (pipe(pipe_fd) == -1)
                {
                    perror("pipe error");
                }

                // 자손 프로세스를 만든다.
                pid_t child2_pid = fork();
                if (child2_pid < 0)
                {
                    perror("pipe fork error");
                }

                // 자손 프로세스
                else if (child2_pid == 0)
```

```
            {
                // 사용하지 않는 파이프를 닫아준다.
                close(pipe_fd[0]);
                // 파이프를 표준 출력으로 바꾸어 준다.
                dup2(pipe_fd[1], 1);
                // args에 대해 "파이프가 없는 경우" 함수를 실행한다.
                exeInstruction(RD_Lflag, RD_Rflag, 0, args);
                return 0;
            }

            // 자식 프로세스
            else
            {
                // 사용하지 않는 파이프를 닫아준다.
                close(pipe_fd[1]);

                // 파이프를 표준 입력으로 바꾸어준다.
                dup2(pipe_fd[0], 0);

                // args2에 대해 "파이프가 없는 경우" 함수를 실행한다.
                exeInstruction(RD_Lflag2, RD_Rflag2, 0, args2);
                return 0;
            }

            return 0;
        }

        // 부모 프로세스
        else
        {
            // & 를 사용한 경우
            if (BG_flag)
            {
                // not wait
                int status;
                int process_cnt = 1;
                int retval = waitpid(-1, &status, WNOHANG);
                if (retval == 0)
                {
                    // 백그라운드로 실행되었음을 알려주기 위한 출력
                    printf("[%d] %d\n", process_cnt, child_pid);
                }
            }
            // & 를 사용하지 않는 경우
            else
            {
                // wait
                int status;
                int retval = waitpid(-1, &status, 0);
```

```
                }
            }
        }

        // 반복
    }
    return 0;
}

//
================================================================================
=

int exeInstruction(int RD_Lflag, int RD_Rflag, int BG_flag, char *argp[])
{
    // 자식 프로세스를 생성
    pid_t child_pid = fork();

    if (child_pid < 0)
    {
        perror("child_pid error");
        return -1;
    }

    // 자식 프로세스
    else if (child_pid == 0)
    {
        // '<' 가 사용된 경우
        if (RD_Lflag)
        {
            int fd;
            // 파일이 없는 경우 에러
            fd = open(argp[RD_Lflag + 1], O_RDONLY | O_NONBLOCK, 0600);
            if (fd < 0)
            {
                perror("no file");
            }
            else
            {
                // 표준 입력을 파일로 바꿔준다.
                dup2(fd, 0);

                // execvp가 처리할 수 있게 '<'를 null 로 바꿔준다.
                argp[RD_Lflag] = NULL;

                // 명령어 실행
                execvp(argp[0], argp);
                perror("execvp Error Occured");
            }
```

```
            close(fd);
            return 0;
        }
        // '>' 를 사용한 경우.
        else if (RD_Rflag)
        {
            int fd;
            // 파일이 없는 경우 생성
            fd = open(argp[RD_Rflag + 1], O_CREAT | O_WRONLY | O_TRUNC, 0600);

            // 표준 출력을 파일로 바꾸어 준다.
            dup2(fd, 1);

            // execvp가 처리할 수 있게 '>'를 null 로 바꿔준다.
            argp[RD_Rflag] = NULL;

            // 명령어 실행
            execvp(argp[0], argp);
            perror("execvp Error Occured");
            close(fd);
            return 0;
        }
        else
        {
            // 명령어만 실행하는 경우
            execvp(argp[0], argp);
            return 0;
        }
    }

    // 부모 프로세스
    else
    {
        // '&' 를 사용한 경우, WNOHANG 을 사용하여 background process 구현
        if (BG_flag)
        {
            // not wait
            int status;
            int process_cnt = 1;
            int retval = waitpid(child_pid, &status, WNOHANG);
            if (retval == 0)
            {
                printf("[%d] %d\n", process_cnt, child_pid);
            }
        }
        // '&' 를 사용하지 않은 경우
        else
        {
            // wait
```

```
            int status;
            int retval = waitpid(child_pid, &status, 0);
        }
    }
    return 0;
}
```