System Programming Project 2

담당 교수 : 박성용 교수님

이름 : 최호진

학번 : 20201654

1. 개발 목표

프로젝트의 목표는 직접 리눅스에서 사용하는 쉘에서 사용하는 기본적인 명령어와 동작들을 사용할 수 있는 myshell을 구현하는 것이다. 구현하고자 하는 동작 및 명령어는 아래와 같다.

- 명령어 파싱: 사용자가 입력한 명령어를 띄어쓰기 단위, 또는 파이프('|') 단위로 파싱하여 적절한 명령어를 실행한다.
- 기본적인 쉘 명령어 (ls, echo, mkdir, rmdir, cat, touch 등)을 exec 함수와 fork()를 이용해 작동하도록 구현한다.
- 쉘에 내장된 명령어(cd, jobs, kill, bg, fqg, exit) 등을 사용자가 직접 구현한다.
- 파이프 단위로 연속적으로 입력된 명령어를 파싱하고, pipe(), dup()을 이용해 파이프 작업을 구현한다.
- 쉘에서 프로세스를 백그라운드로 실행할 때 입력하는 '&'를 동일하게 구현하고, 시그널 핸들러를 직접 구현하여 백그라운드에서 작동하는 프로세스를 처리할 수 있도록 한다.
- 이외에 쉘에서 프로세스 실행 시 ctrl+c, ctrl+z와 같은 입력을 처리하는 것과 동일하게 작동할 수 있는 시그널 핸들러를 직접 구현한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Phase 1

   Phase 1에서는 기본적인 쉘 프롬프트를, 쉘에서 사용하는 기본적인 명령어에 작동할 수 있도록 fork()와 구현하고exec(본인의 코드에서는 execvp 사용) 함수를 적절히 사용하여 구현하였다.

1 – 1) Shell Prompt

사용자가 exit을 입력하기 전까지 계속해서 입력을 받도록 해야 한다. 따라서, 무한 루프(do{ … }while(1))를 입력으로 "exit"이 입력되기 전까지 계속해서 입력을 받을 수 있도록 하였다.

1 – 2) 내부 명령어(“exit”, “cd”) 구현

exit: exit이 입력되면 프로그램이 exit(0)으로 종료되도록 하였다.

cd: cd의 경우 chdir() 함수를 이용해 현재 디렉토리의 위치를 옮겨주는 방식으로 구현하였다.

다양한 옵션이 있는데, 먼저 cd와 함께 '~'이 다음 인자로 들어오게 되면 ~은 home 디렉토리를 나타내므로, ~을 getenv(HOME)를 이용해 환경 변수 HOME에 저장된 값으로 변경해주었다. 또한, '-'가 인자로 주어졌을 땐 이전의 위치로 돌아가야 하므로, 프롬프트에서 입력 받을 때마다 현재 directory의 위치를 저장해두고, 다음 입력이 'cd –' 라면 저장해둔 위치로 돌아가는 방식으로 구현하였다. 이외의 입력에 대해서는 현재 위치에서 이동할 수 있는 경로라면 정상적으로 이동하고, 절대경로가 입력으로 주어졌을 때도 정상적으로 이동할 수 있도록 하였다. 이외의 입력에 대해서는 에러 메세지와 함께 위치를 변경하지 않도록 하였다.

1 – 3) /bin/* 명령어 (ls, echo, touch, mkdir, rmdir 등)

쉘의 /bin 디렉토리 내의 명령어를 myshell에서 이용하기 위해서 exec 계열의 함수를 이용하였다. 본인의 프로그램에서는 execvp 함수를 이용했는데, 명령어의 경로를 입력해주지 않아도 환경변수 PATH의 경로에 있는 명령어들에 대해 작동하기 때문에 이 함수를 사용했다. 한편, exec 함수를 이용하면 현재 프로세스의 메모리가 덤프 되어 함수 종료 시 프로그램이 종료되기 때문에, 이는 우리가 구현하고자 하는 쉘과 달라진다. 따라서 fork()함수를 이용해 자식 프로세스를 생성하고, 자식 프로세스에서 exec 함수가 실행되도록 구현하였다. 정상적인 입력에 대해서는 명령어가 작동하지만 이외의 입력은 에러 메세지를 출력하도록 하였다.

1 - 4) 문자열 파싱

기본적인 문자열 파싱은 뼈대 코드에 구현되어 있지만, echo와 함께 따움표가 입력으로 주어졌을 때엔 따움표를 제거해줄 필요가 있다. 이를 뼈대 코드의 parseline 함수에 추가하여, echo “hello”와 같이 입력이 주어졌을 때 hello만 출력될 수 있도록 구현하였다.

2. Phase 2

Phase 2에서는 파이프 라인이 포함된 명령어를 처리할 수 있어야 한다. 이를 위해서 Phase 1에서 파이프를 파싱하는 부분을 추가하고, 여러 명령어가 입력되었을 때 앞의 명령어의 결과가 뒷 명령어의 입력으로 사용될 수 있도록 구현해주었다.

2 - 1) 문자열 파싱 – 파이프 추가

Phase 1의 파싱에 추가해 파이프를 파싱 해주어야 한다. 또한, 파이프가 띄어쓰기 없이 입력되더라도 이를 구분할 수 있어야 하기 때문에 parse_with_pipe 함수를 추가해 이들을 파싱 할 수 있는 함수를 구현하였다. 또한, 이때 입력된 파이프의 개수를 같이 구한다.

2 - 2) 파이프 라인 처리

2 -1)에서 파싱한 인자들을 이용해 파이프 라인으로 입력된 명령어에 대해 정상적으로 작동할 수 있도록 구현한다.

이때, pipe()와 dup2() 함수를 이용해 해당 작업을 수행한다. 앞 명령어의 출력 결과를 뒷 명령어에서 사용할 수 있도록 구현한다.

3. Phase 3

Phase 3에서는 백그라운드에서 프로세스가 작동할 수 있도록 구현하고, 이 백그라운드의 프로세스를 다루는 여러 명령어(jobs, fg, bg, kill)들을 구현한다. 또한 이를 위해서 시그널 핸들러 코드를 직접 작성하여 사용한다.

먼저, 프롬프트에서 프로세스를 실행할 때 & 입력이 마지막에 같이 주어지면, 이를 적절하게 파싱하여 백그라운드 프로세스로 인식하고, 백그라운드에서 실행될 수 있도록 구현한다.

그 다음, 사용자가 실행한 프로세스들을 백그라운드에서 포어그라운드로, 그 반대로의 전환이 원활하게 이루어지도록 구현하였고, 백그라운드에서 종료된 프로세스를 reaping할 수 있도록 구현하였다.

또한, 포어그라운드의 작업에 대해 ctrl+c, ctrl+z와 같은 입력이 주어졌을 때 적절한 시그널 핸들러를 구현하여 쉘과 유사하게 작동할 수 있도록 하였다.

B. 개발 내용

- **Phase1 (fork & signal)**

1) fork()를 사용한 자식 프로세스 생성 및 exec 함수 실행

execvp() 함수를 사용해 환경변수 PATH의 경로에 있는 명령어를 실행할 수 있도록 구현했다. 이때, 현재 프로세스에서 실행하게 되면, execvp 함수 실행 시 현재 프로세스의 메모리가 덤프되고, 실행 후 프로세스가 종료되기 때문에 현재 프로세스가 아닌 fork() 함수를 이용해 자식 프로세스를 생성하고, 자식 프로세스에서 execvp 함수가 실행될 수 있도록 구현하였다. 이렇게 구현함으로써, execvp가 성공적으로 실행되어 올바른 작업을 한 뒤에 자식 프로세스는 종료되더라도, 부모 프로세스는 남아있기 때문에, 계속해서 쉘 프롬프트에서 입력을 받을 수 있다.

2) Waitpid() 함수를 이용해 자식 프로세스 reaping

자식 프로세스와 부모 프로세스 중 어떤 프로세스가 먼저 종료될 지 모르기 때문에 자식 프로세스가 좀비 프로세스, 또는 고아(orphan) 프로세스가 될 수 있다. 이를 방지하기 위해 Waitpid()함수를 이용해 자식 프로세스가 종료될 때까지 부모 프로세스가 대기하고, 자식 프로세스가 종료되었을 때(SIGCHLD 시그널이 발생했을 때) 적절하게 reaping할 수 있도록 구현하였다.

이를 간단하게 설명하면 다음과 같은 순서의 Flow를 가진다.

[1] 쉘 프롬프트에서 입력을 받는다.

[2] 입력된 명령어가 내장 명령어인지 builtin command 함수에서 확인한다.

[3] 내장 명령어가 아니라면 fork() 함수를 이용해 자식 프로세스를 생성하고, 자식 프로세스에서 execvp 함수를 호출한다.

[4] execvp 함수가 정상적으로 실행되어 프로세스가 종료되거나, 또는 정상적으로 실행되지 않았을 땐 에러 메세지를 출력하고, exit 함수로 프로세스를 종료한다.

[5] 자식 프로세스가 종료될 때까지(SIGCHLD 시그널이 발생할 때까지) 부모 프로세스는 Waitpid() 함수로 대기하고 있는다. 자식 프로세스가 종료되면, 부모 프로세스에서 자식을 reaping해준다.
[6] 다음 명령어를 입력 받을 준비를 한다.

- **Phase2 (pipelining)**

  1) 파이프 파싱

  파이프 라인으로 연결된 명령어를 처리하기 위해서, 먼저, 입력받은 명령어를 파싱할 때 파이프의 개수를 정확하게 파악하고, 파이프 단위로 명령어를 나눠서 실행할 수 있도록 해야 한다. 따라서 파이프의 개수를 먼저 센다. 다음으로, 파이프가 명령어와 띄어쓰기 되어있을 지 없을 지 모르기 때문에 먼저, 입력된 명령어에서 파이프('|')가 있을 때마다 '|'를 ' | '로 바꾸어 준다. 이렇게 하면, 띄어쓰기가 없는 파이프들에 대해서도 모두 띄어쓰기를 하여 parse파싱할 때 띄어쓰기 단위로 명령어를 구분해줄 수 있게 된다. 또한, 띄어쓰기 단위로 파싱 한 이후에는 한 명령어 단위로 구분해주어야 한다. 이 또한 새 함수를 작성해 구현하였다.

  2) 파이프의 개수에 따라 명령 구분

  파이프가 하나도 주어지지 않았을 때엔 phase1에서 구현한 부분을 동일하게 사용하여 명령어를 실행할 수 있도록 하였다.

  파이프가 입력되었을 땐 파일 디스크립터(file descriptor)와 pipe()함수를 이용해 파이프를 생성한다. 이때 파일 디스크립터는 파이프의 개수만큼 만들어준다. 각 파일 디스크립터는 2개의 원소를 지닌 배열로 index 0은 읽기, index 1은 출력을 나타낸다.

  명령어를 차례대로 실행하는데, 이때 명령어의 개수는 파이프의 개수+1 개이다. 먼저 fork하여 자식 프로세스를 만들고, dup2() 함수를 이용해 표준 입력 및 출력(STDIN, STDOUT)을 파이프의 파일 디스크립터로 리다이렉션한다.

  한편, 부모 프로세스에서는 phase1과 마찬가지로 waitpid() 함수를 사용해 자식 프로세스가 종료될 때까지 대기한다.

- **Phase3 (background process)**

  1) & 파싱

  Phase2에서 이용한 방법과 마찬가지로 띄어쓰기가 없이 입력된 &에 대해 파싱을 하기 위해 파싱 전 '&'를 모두 " & "로 바꿔준다.

  2) 백그라운드 프로세스 처리

  먼저, 입력된 명령어가 내장 명령어(builtin_command)인지 확인한다. 내장 명령어

라면 자식 프로세스를 생성한 뒤 따로 작업을 수행하지 않고, 바로 exit한다.

이외의 명령에 대해서는 백그라운드 프로세스는 포어그라운드처럼 프로세스가 종료될 때까지 wait하면 안되기 때문에, 프로세스와 관련된 정보(child pid, cmdline, state, jobid)를 저장해둘 구조체를 선언하고, 백그라운드 프로세스가 종료될 때 이를 reaping해주기 위해 SIGCHLD handler를 직접 작성하여 적절하게 reaping한다.

3) 백그라운드 프로세스 관련(jobs, fg, bg, kill) 명령어

백그라운드와 관련된 앞서 선언한 구조체를 이용하여 각 명령어에 대한 적절한 명령을 실행한다. Jobs는 구조체에 저장된 백그라운드 프로세스들의 jobid와 state, command를 출력한다. Fg 와 bg 명령어에 대해서는 입력된 jobid로 백그라운드 프로세스를 찾고, 해당 프로세스의 state를 적절하게 바꿔준다. 이때 SIGCONT 시그널을 프로세스에 보내어 state를 변경한다.

C. 개발 방법

Phase1 (fork & signal)

1) fork()를 사용한 자식 프로세스 생성 및 exec 함수 실행

```
455     if (!builtin_command(argv)) {
456         Sigprocmask(SIG_BLOCK, &mask_chld, &prev_one);
457         if ((pid = Fork()) == 0) {
458             Sigprocmask(SIG_SETMASK, &prev_one, NULL);
459             if (execvp(argv[0], argv) < 0) {
460                 char* str = "Command not found.\n";
461                 printf("%s: %s", argv[0], str);
462                 exit(1);
463             }
464         }
465
```

위에서 작성한 코드는 eval 함수에서 호출되어 실행된다. 입력받은 명령어가 builtin_command가 아닐 때 (cd, exit이 아닐 때) Fork() 함수를 이용해 자식 프로세스를 생성한다. (Fork()는 fork()의 wrapper 함수이다.) Fork()의 return이 0일 때, 즉 자식 프로세스일 때엔 execvp 함수를 이용해 입력받은 명령어를 수행한다. 이때, 명령어가 정상적으로 실행되면 정상적으로 종료되지만, 해당 명령어를 execvp 함수에서 실행할 수 없다면 에러 메세지와 함께 프로세스를 종료한다. 이때 exit(1)로 종료함으로써, 정상적으로 종료되지 않았다는 것을 알린다.

2) Waitpid() 함수를 이용해 자식 프로세스 reaping

```c
if (!bg) {
    bg_job[++bg_cnt].pid = pid;
    insert_job(pid, 2, cmdline);

    Waitpid(pid, &status, WUNTRACED);
    for (int i = 0; i < MAXARGS; i++) {
        if (bg_job[i].pid == pid) {
            bg_job[i].state = -1;
            bg_job[i].pid = 0;
            memset(bg_job[i].cmd, 0, MAXLINE);
            break;
        }
    }

}
```

Phase 1은 foreground 명령에 대해서만 구현하므로 해당 부분의 코드 중 Waitpid() 함수만 이용한다. 자식프로세스가 종료되기 전까지 Waitpid()함수로 기다리고, 자식 프로세스가 종료되면 부모 프로세스는 자식 프로세스를 reaping 해준다.

Phase2 (pipelining)

1) 파이프 파싱

B에서 설명한 부분을 구현하기 위해 세가지 함수를 구현했다. 먼저, count_pipe 함수에서 입력된 명령어에 있는 파이프의 개수를 센다.

```c
400  int count_pip(char* buf) {
401      int count = 0;
402      for (int i = 0; buf[i] != '\0'; i++)
403          if (buf[i] == '|') count++;
404
405      return count;
406  }
```

다음으로, replace_bar_ with_space_bar_space() 함수에서는 함수의 이름에서 알 수 있듯이 '|;를 ' | '로 바꿔주는 작업을 하였다. 이렇게 함으로써 띄어쓰기로 구분되어있지 않은 파이프에 대해서도 파싱할 때 명령어를 구분할 수 있게 된다. (아래 코드에는 phase3를 위해 '&'를 구분하는 코드가 함께 삽입되어있다.)

```c
char* replace_ba_and_with_space_bar_and_space(char* buf) {
    char* ret = (char*)malloc(MAXLINE * 3);
    int i = 0;
    int j = 0;
    while (buf[i]) {
        if (buf[i] != '|' && buf[i] != '&') ret[j++] = buf[i++];
        else if (buf[i] == '|') {
            strcpy(&ret[j], " | ");
            j += 3;
            i++;
        }
        else if (buf[i] == '&') {
            strcpy(&ret[j], " & ");
            j += 3;
            i++;
        }
    }
    ret[j] = '\0';
    return ret;
}
```

그 다음 뼈대코드에 있는 parseline() 함수로 파싱을 해주었다. 한편, 띄어쓰기 단위로 구분한 이후에 파이프 단위로 명령어를 잘라 저장해주었다. 이렇게 함으로써 명령어를 파이프 단위로 실행하도록 했다. parse_with_pipe() 함수를 작성하여 구현하였다.

```c
void parse_with_pipe(char** argv, char*** newargv) {

    int i = 0;
    int j = 0;
    int k = 0;
    newargv[0] = (char**)malloc(sizeof(char*) * MAXARGS);
    while (argv[k] != NULL) {
        // printf("argv[%d](%d): %s\n",k,strlen(argv[k]),argv[k]);
        if (!strcmp(argv[k], "|")) {
            if (i == 0) {
                printf("bash: syntax error near unexpected token '|'\n");
                exit(0);
            }
            newargv[j][i] = NULL;
            j++;
            newargv[j] = (char**)malloc(sizeof(char*) * MAXARGS);
            i = 0;
        }
        else {
            newargv[j][i] = (char*)malloc(strlen(argv[k]) + 1);
            if (!strcmp(argv[k], "&") && i == 0) {
                if (i == 0) {
                    printf("bash: syntax error near unexpected token '|'\n");
                    exit(0);
                }
            }
            if ((argv[k][0] == '"' && argv[k][strlen(argv[k]) - 1] == '"') || argv[k][0] == '\'' && argv[k][strlen(argv[k]) - 1] == '\'') {
                argv[k][strlen(argv[k]) - 1] = '\0';
                strcpy(newargv[j][i], &argv[k][1]);
            }
            else strcpy(newargv[j][i], argv[k]);
            i++;
        }
        k++;

    }
    argv[j][i] = NULL;
}
```

해당 함수에서는 띄어쓰기로 구분된 명령어(argv)를 2차원의 문자열 배열(newargv)에 문자열을 파이프가 나올 때마다 row를 바꿔주어(j++) 하나의 row에 한 명령어 세트가 저장되도록 하였다.

2) 파이프의 개수에 따라 명령 구분

파이프가 하나도 주어지지 않았을 때엔 (count_pipe에서 센 pipe의 개수가 0개일 때엔 no_pipe_command 함수에서 phase1과 동일하게 명령어를 처리해주었다. (Phase1 코드 참조)

```
496  void pipe_comman(pid_t pid, int status, int bg, char* cmdline, char*** argv, int pipe_count) {
497      int** fd = (int**)malloc(sizeof(int*) * pipe_count);
498      for (int i = 0; i < pipe_count; i++) fd[i] = (int*)malloc(2 * sizeof(int));
499      for (int i = 0; i < pipe_count; i++) pipe(fd[i]);
500      for (int i = 0; i < pipe_count + 1; i++) gethome(argv[i]);
501
502      sigset_t mask_chld, mask_all, prev_one;
503      Sigfillset(&mask_all);
504      Sigemptyset(&mask_chld);
505      Sigaddset(&mask_chld, SIGCHLD);
506
507      for (int i = 0; i <= pipe_count; i++) {
508          if ((pid = Fork()) == 0) {
509              if (i > 0) {
510                  Dup2(fd[i - 1][0], STDIN_FILENO);
511                  close(fd[i - 1][0]);
512                  close(fd[i - 1][1]);
513              }
514              if (i < pipe_count) {
515                  Dup2(fd[i][1], STDOUT_FILENO);
516                  close(fd[i][0]);
517                  close(fd[i][1]);
518              }
519              if (!builtin_command(argv[i])) {
520                  if (execvp(argv[i][0], argv[i]) < 0) {
521                      printf("%s: Command not found. \n", argv[i][0]);
522                      exit(0);
523                  }
524              }
525              exit(0);
526          }
527          if (i > 0) close(fd[i - 1][0]);
528          if (i < pipe_count) close(fd[i][1]);
529
530          // pid = Waitpid(pid, &status, WUNTRACED);
531          if (!bg) {
532              bg_job[++bg_cnt].pid = pid;
```

한편, 파이프가 입력되었을 땐 pipe_command 함수에서 명령어를 처리해주었다. 파일 디스크립터(fd)와 pipe()함수를 이용해 파이프를 생성한다. 이때 파일 디스크립터는 파이프의 개수만큼 만들어주었다.(2차원 배열을 이용해 구현) 각 파일 디스크립터는 2개의 원소를 지닌 배열로 index 0은 읽기, index 1은 출력을 나타낸다.

반복문을 이용해 명령어의 개수만큼 자식 프로세스를 생성하는데, 이때 명령어의 개수는 파이프의 개수 +1 개가 된다. 자식 프로세스에서는 B에서 설명한 것 처럼, Dup2()(dup2의 wrapper) 함수를 이용해 표준 입력 및 출력(STDIN, STDOUT)을 파이프의 파일 디스크립터로 리다이렉션한다. 이렇게 하여 명령어를 순차적으로 실행한다.

부모 프로세스에서는 phase1에서 pipe가 없는 명령어와 마찬가지로 waitpid() 함수를 사용해 자식 프로세스가 종료될 때까지 대기한다.

```
542         if (!bg) {
543 
544             Waitpid(pid, &status, WUNTRACED);
545             if(!status){
546                 bg_job[++bg_cnt].pid = pid;
547                 insert_job(pid, 2, cmdline);
548                 for (int i = 0; i < MAXARGS; i++) {
549                     if (bg_job[i].pid == pid) {
550                         if(bg_job[i].state != 0)
551                         {
552                             bg_job[i].state = -1;
553                             bg_job[i].pid = 0;
554                         }
555                         //memset(bg_job[i].cmd, 0, MAXLINE);
556                         break;
557                     }
558                 }
559             }
560         }
```

마지막으로 모든 file descriptor를 닫아주면서 끝낸다.

```
559     for (int i = 0; i < pipe_count; i++) {
560         close(fd[i][0]);
561         close(fd[i][1]);
562     }
```

Phase3 (background process)

1) '&' 파싱

Phase2에서 파이프를 처리한 함수(replace_bar_with_space_bar_space)에 추가적으로 '&'에 대해서도 동일하게 처리하도록 추가한다.

```
425 char* replace_bar_and_with_space_bar_and_space(char* buf) {
426     char* ret = (char*)malloc(MAXLINE * 3);
427     int i = 0;
428     int j = 0;
429     while (buf[i]) {
430         if (buf[i] != '|' && buf[i] != '&' && buf[i] != '"' && buf[i] != '\'') ret[j++] = buf[i++];
431         else if (buf[i] == '|') {
432             strcpy(&ret[j], " | ");
433             j += 3;
434             i++;
435         }
436         else if (buf[i] == '&') {
437             strcpy(&ret[j], " & ");
438             j += 3;
439             i++;
440         }
441         else if (buf[i] == '"' || buf[i] == '\''){
442             strcpy(&ret[j], " ");
443             j++;
444             i++;
445         }
446     }
447     ret[j] = '\0';
448     return ret;
449 }
```

그 다음으로 똑같이 앞선 Phase들과 동일하게 파싱 해주고, 이때 뼈대 코드를 보면 parseline 함수에서 백그라운드 프로세스인지(&가 입력되었는지) 반환하는 것을 볼 수 있다.

2) 백그라운드 프로세스 처리

먼저 자식 프로세스를 관리할 구조체를 선언하고 구조체 배열을 생성한다.

구조체에는 pid, jobid, state, 입력된 command 문자열이 저장된다.

```
28  volatile sig_atomic_t ctrl_c = -1;
29  typedef struct jobs {
30      pid_t pid;
31      int jobid;
32      int state;
33      char cmd[MAXLINE];
34  } Jobs;
```

백그라운드로 명령을 실행할 때엔 해당 구조체에 알맞은 정보를 저장한다. State는 stopped = 0, running = 1, foreground = 2, terminated = -1에 각각 대응했다.

백그라운드 프로세스에 대해서는 파이프 라인 명령과, 파이프가 아닌 명령 2가지 case로 나누어 처리했다.

파이프가 없는 명령에 대해선 먼저, 내장 명령어인지 확인한다. 한편, 기존의 builtin_command 함수에는 argv만 인자로 전달했지만, 백그라운드 명령인지 확인해야 하므로 int bg를 인자로 추가하였다. 내장 명령어라면 Fork()후 아무 명령을 실행하지 않고 바로 exit()하고 "Done"을 출력한다.

(builtin_command에 해당 코드 추가)

```
if(bg){
    if(!strcmp(argv[0],"cd") || !strcmp(argv[0],"&") || !strcmp(argv[0],"jobs") || !strcmp(argv[0], "bg") || !strcmp(argv[0], "fg") || !strcmp(argv[0], "kill") || !strcmp(argv[0], "exit")){
        pid_t pid;
        int status;
        if((pid = Fork()) == 0){
            exit(0);
        }
        Waitpid(pid,&status,WUNTRACED);
        printf("[%d] Done          %s\n",bg_cnt + 1, argv[0]);
        return 1;
    }
    return 0;
}
```

내장 명령어가 아니라면 jobid와 해당 프로세스의 pid를 출력하고 insert_job 함수를 이용해서 구조체에 해당 프로세스의 정보를 저장한다. (no_pipe_command 함수에 코드 추가)

```
else {
    printf("[%d] %d\n", ++bg_cnt, pid);
    insert_job(pid, 1, cmdline);
}
```

```
626 void insert_job(pid_t pid, int state, char* cmd) {
627     int k;
628     char* temp = strtok(cmd, "&");
629     strcpy(cmd,temp);
630     if(!strchr(cmd, '\n')) strcat(cmd, "\n");
631 
632     if (pid > 0) {
633 
634         memset(bg_job[(++ps_cnt)].cmd, 0, sizeof(bg_job[ps_cnt].cmd));
635         bg_job[ps_cnt].pid = pid;
636         bg_job[ps_cnt].state = state;
637 
638         strcpy(bg_job[ps_cnt].cmd, cmd);
639 
640         if (state != 2 && state != -1) {
641             bg_job[ps_cnt].jobid = bg_cnt;
642             bg_cnt++;
643         }
644         else bg_job[ps_cnt].jobid = ps_cnt;
645 
646         return;
647     }
648     return;
649 }
```

파이프가 있는 명령에 대해서는 명령어의 Sequence 중 마지막 명령에 대해서만 프로세스 pid를 출력하고, 구조체에 저장한다.(pipe_command 함수에 추가) 따라서, 반복문을 통해 Fork()를 할 때 마지막으로 Fork를 할 때(i=pipe_count일 때)에만 해당 pid를 출력하고 insert_job 함수를 이용해 구조체에 저장한다.

```
581             if (i == pipe_count) {
582                 printf("[%d] %d\n", ++bg_cnt, pid);
583                 insert_job(pid, 1, cmdline);
584             }
```

그리고, 백그라운드 프로세스가 종료되면 SIGCHLD 시그널을 부모 프로세스에 보내는데, 이를 처리하기 위해 sigchld_handler를 구현했다.

```
680  void sigchld_handler(int sig) {
681      if(ctrl_c == 1){
682          ctrl_c = -1;
683          return;
684      }
685      sigset_t mask, prev;
686      int t_errno = errno;
687
688      Sigfillset(&mask);
689
690      pid_t pid;
691      int status;
692
693      while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
694          Sigprocmask(SIG_BLOCK, &mask, &prev);
695          if (!status) {
696
697              for (int i = 0; i < MAXARGS; i++) {
698                  if (pid == bg_job[i].pid) {
699                          printf("\n[%d] Done            %s", bg_job[i].jobid, bg_job[i].cmd);
700                          bg_job[i].state = -1;
701                          bg_job[i].pid = 0;
702                          memset(bg_job[i].cmd, 0, MAXLINE);
703                          break;
704                  }
705              }
706
707          }
708          else {
709              for (int i = 0; i < MAXARGS; i++) {
710                  if (pid == bg_job[i].pid && bg_job[i].state == 2) {
711                      printf("\n[%d] Exit          %s",bg_job[i].jobid, bg_job[i].cmd);
712                      bg_job[i].state = -1;
713                      bg_job[i].pid = 0;
714                      memset(bg_job[i].cmd, 0, MAXLINE);
715                      break;
716                  }
717              }
718          }
719          Sigprocmask(SIG_BLOCK, &prev, NULL);
720      }
721      int k = 0;
722      for (int i = 0; i < MAXARGS; i++){
723          if (bg_job[i].state == 2) bg_job[i].state = -1;
724          if (bg_job[i].state == 0 || bg_job[i].state == 1) k = 1;
725      }
726      if (!k) bg_cnt = 1;
727      errno = t_errno;
728  }
```

3) 백그라운드 프로세스 관련 내장 명령어(builtin_command에 추가)

- jobs

```
168      if (!strcmp(argv[0], "jobs")){
169          for (int i = 0; i <= MAXARGS; i++){
170              if(bg_job[i].pid > 0)
171                  switch (bg_job[i].state)
172                  {
173                  case 0: //stopped
174                      printf("[%d] Suspended         %s", bg_job[i].jobid, bg_job[i].cmd);
175                      break;
176                  case 1: //running
177                      printf("[%d] Running         %s", bg_job[i].jobid, bg_job[i].cmd);
178                      break;
179                  default:
180                      break;
181                  }
182          }
183          Sigprocmask(SIG_SETMASK, &prev, NULL);
184          return 1;
185      }
```

구조체의 배열에서 state가 0(Stopped)과 1(Running)인 명령어를 jobid 순서대로 출력한다.

- fg

입력된 jobid를 가진 백그라운드 프로세스를 포어그라운드로 전환한다.

Index의 범위가 잘못된 경우엔 에러 메세지를 출력한다. (아래 코드에는 fg 관련 코드 중 가장 핵심적인 부분만 캡처해 첨부하였다.)

```
217         if (index >= 0) {
218             int status;
219             pid_t pid;
220             switch (bg_job[index].state)
221             {
222             case 1: case 0:
223                 printf("%s\n", bg_job[index].cmd);
224 
225                 bg_job[index].state = 2;
226                 Kill(bg_job[index].pid, SIGCONT);
227 
228                 Waitpid(pid, &status, WUNTRACED);
229 
230                 if(!status){
231                     for (int i = 0; i < MAXARGS; i++) {
232                         if (bg_job[i].pid == pid) {
233                             if(bg_job[i].state != 0) bg_job[i].state = -1;
234                             bg_job[i].pid = 0;
235                             break;
236                         }
237                     }
238                 }
239 
240                 break;
241 
242             default:
243                 printf("bash: fg: %%%d: no such job\n", atoi(argv[1] + 1));
244                 break;
245             }
246 
247         }
248         else{
249             printf("bash: bg: no such job\n", index);
```

-bg

입력된 job id를 가진 백그라운드 프로세스를 Stopped state에서 Running state로 전환한다. fg와 마찬가지로 입력된 index가 잘못된 경우 에러 메세지를 출력한다.

(아래 코드에는 bg 관련 코드 중 가장 핵심적인 부분만 캡쳐해 첨부하였다.)

```
217         if (index >= 0) {
218             int status;
219             pid_t pid;
220             switch (bg_job[index].state)
221             {
222             case 1: case 0:
223                 printf("%s\n", bg_job[index].cmd);
224 
225                 bg_job[index].state = 2;
226                 Kill(bg_job[index].pid, SIGCONT);
227 
228                 Waitpid(pid, &status, WUNTRACED);
229 
230                 if(!status){
231                     for (int i = 0; i < MAXARGS; i++) {
232                         if (bg_job[i].pid == pid) {
233                             if(bg_job[i].state != 0) bg_job[i].state = -1;
234                             bg_job[i].pid = 0;
235                             break;
236                         }
237                     }
238                 }
239 
240                 break;
241 
242             default:
243                 printf("bash: fg: %%%d: no such job\n", atoi(argv[1] + 1));
244                 break;
245             }
246 
247         }
248         else{
249             printf("bash: bg: no such job\n", index);
```

-kill

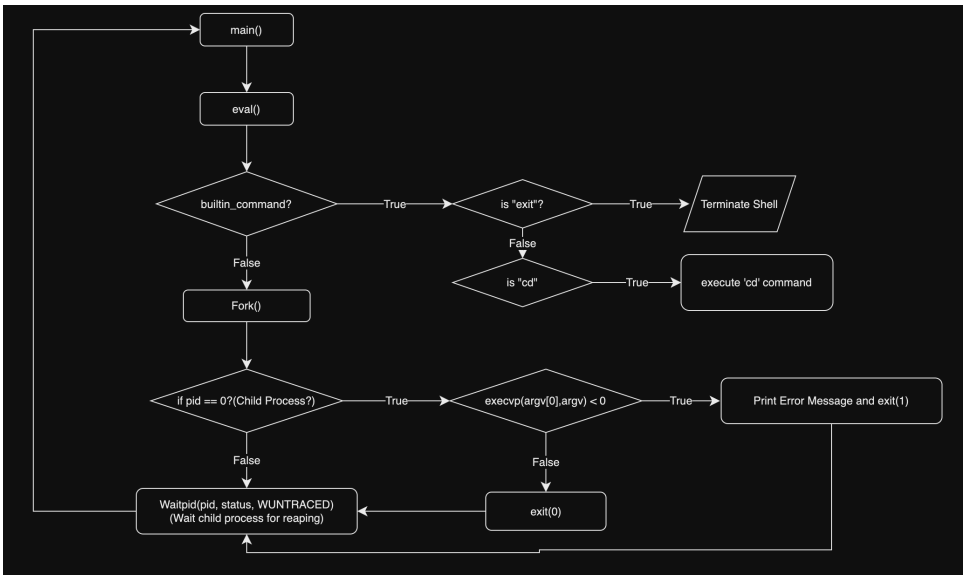
입력된 job id를 가진 백그라운드 프로세스에 SIGKILL 시그널을 보내 종료한다.

```
if (index >= 0) {
    switch (bg_job[index].state)
    {
    case -1:
        printf("bash: kill: %%%d: no such job\n", atoi(argv[1] + 1));
        break;
    default:
        Kill(bg_job[index].pid, SIGKILL);
        bg_job[index].state = -1;
        bg_job[index].pid = 0;
        printf("[%d] Terminated          %s \n", bg_job[index].jobid, bg_job[index].cmd);
        break;
    }
}
else {
    printf("bash: kill: %%%d: no such job\n", atoi(argv[1] + 1));
}
Sigprocmask(SIG_SETMASK, &prev, NULL);
return 1;
```

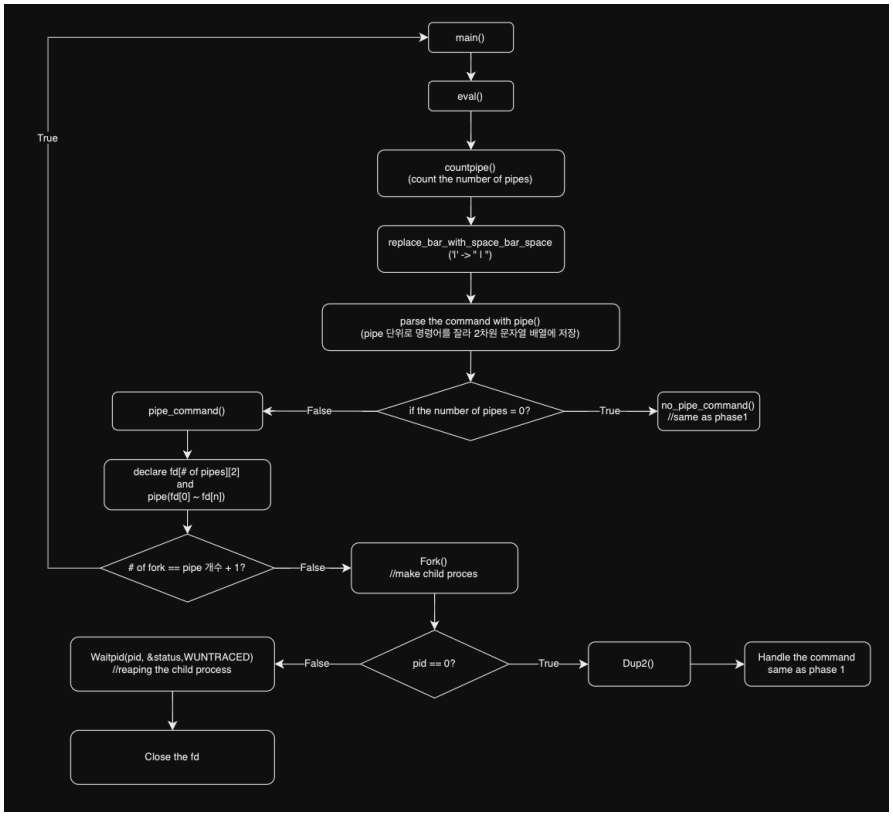
3. 구현 결과

A. Flow Chart

1. Phase 1 (fork)



2. Phase 2 (pipeline)



3. Phase 3 (background)