< 나만의 Unix Shell 만들기 >

1. 개요

1.1 문제 정의

Unix c 언어를 사용하여, Bourne shell과 같은 형태의 Unix shell을 구현한다. Bourne shell과 같이 커맨드 라인으로 입력을 받아 해당 command를 처리하며, built-in command는 Unix의 command를 그대로 사용한다. 프로그램의 종료는 Ctrl+D 혹은 Exit 명령을 통해 종료하며, Ctrl+C 역시 SIGINT를 일으키며 프로세스의 종료를 수행한다.

프로그램의 실행 경로는 PATH 환경 변수를 참조하며, "&" 심볼을 통해 병렬처리를 구현하여, background process를 생성하도록 한다. 또한, 리다이렉션을 위해, "<", ">" 심볼을 지원하도록 한다. "<"는 stdin의 redirection 기능을 지원하고, ">"는 stdout의 redirection 기능을 지원하도록 한다. "|" 심볼은 pipe 기능을 제공하도록 하며, 앞의 프로세스의 stdout이 뒤의 프로세스의 stdin이 되도록 한다.

1.2 문제 분석

Unix의 일반적인 built-in command를 처리하기 위해 exec 계열의 함수를 사용한다. 문제의 spec에서 PATH 환경 변수를 참조하라 하였으나, exec 계열의 어떠한 함수를 사용하여도 무방하다 하였으므로, function 내부적으로 PATH 환경 변수를 참조하는 function인 execvp와 execlp를 사용한다. 물론, execl이나 execve와 같은 함수를 사용하고, PATH 변수를 별도로 처리하여도 되나 구현의 간결성을 위해 execvp와 execlp 함수를 사용한다.

Ctrl+D의 경우, 별도의 signal을 발생시키지 않고, EOF의 형태로 입력되므로, EOF가 입력되는 경우, Ctrl+D가 typing된 것으로 간주하여, 프로그램을 종료시킨다. exit이 입력되는 경우에도 마찬가지로 프로그램을 종료시킨다. 그러나 Ctrl+C의 경우에는 SIGINT signal을 발생시키며, 이를 위한 handler의 구현이 필요하다.

일반적인 명령의 처리를 위해 fork 함수를 사용하여 해당 command를 수행하는 process를 생성하도록 한다. 이 경우, 부모는 별도의 작업을 수행하지 않고, waitpid를 통해 자식의 종료 코드를 확인하는 과정을 수행하며, 실제의 command는 자식 process에서 수행되도록 한다. "&"를 통한 병렬처리의 경우에는 background process로 해당 process가 동작하도록 해야 하므로, 부모 process에서 자식 process를 제어할 필요가 없다. 따라서 이 경우에는 waitpid를 사용하지 않고, 바로 종료하도록 한다.

Redirection을 위해서는 입력받은 argument가 filename이 되므로, 이를 open 함수를 통해 열고, dup2 함수를 통해 해당 파일을 duplicate하여 사용한다. Pipe를 구현하는 과

정에서도 이와 동일하게 dup2 함수를 사용하며 다만, pipe를 위한 file descriptor는 open으로 열지 않고, pipe 함수를 통해 얻는다.
또한, 입력받은 라인을 token으로 parsing하기 위해, getchar 함수를 사용한다.

1.3 구현 환경

- OS : Linux
- language : C
- editor : VI editor
- compiler : gcc c compiler
- source file : MyShell.c

2. 설계 및 구현

2.1 Framework

전체적인 framework은 smallshell과 교재의 source code를 참조하였다. 기본적인 틀과 함수의 사용을 위해 교재의 소스 코드를 참조하였으며, 여기에 과제의 spec을 추가하는 과정에서 시행착오 끝에 smallshell과 유사한 형태의 코드를 구현할 수 있었고, smallshell의 소스 코드를 참조하여 robustness를 높일 수 있었다. 특히, 각 delimeter를 미리 define해 놓고 switch문을 통해 처리하는 방법과 buffer의 pointer를 global하게 지정하는 방법을 사용해 좀 더 깔끔하게 코드를 구현할 수 있었다.
프로그램의 기본적인 흐름은 다음과 같다. 우선 화면상에 프롬프트를 출력하고, 사용자의 입력을 기다린다. 사용자의 입력이 들어온 경우, getchar 함수를 사용하여, 해당 입력을 buffer에 저장한다. 이때, buffer의 pointer를 두어, 해당 char를 저장할 위치를 기록하고 있도록 한다. 사용자 입력의 하나의 라인이 끝나는 시점은 항상 line feed가 발생하므로, line feed, 즉 '₩n'이 입력되는 시점이 사용자의 입력이 끝나는 시점으로 인식한다. 입력이 진행되는 도중, 사용자가 Ctrl+D를 입력하는 경우, 프로그램으로 EOF가 입력되게 된다. 이 경우, main에 EOF를 리턴하여, 프로그램이 종료하도록 한다. 이외에 사용자가 Ctrl+C를 입력하는 경우, SIGINT interrupt signal이 발생하게 되는데 이를 위해 signal handler를 두어, 해당 signal을 처리하게 한다. 문제의 spec에서 Ctrl+C가 입력되는 경우, 프로그램을 종료하라 하였으므로, 이 경우, exit(0)을 호출하게 한다.
이외의 경우에는 사용자가 모두 정상적으로 입력한 것이라 간주하여, 해당 라인을 수행하는 과정을 진행하도록 한다. 해당 라인을 수행하기 위해서, 우선 몇 개의 symbol을 delimeter로 두어, token으로 parsing하는 작업을 수행한다. 기본 command 이외에 입력될 수 있는 symbol로는 space와 tab이 있을 수 있으며, 이들은 command line을 각 token으로 parsing하는 과정에서 무시한다. 즉, 공백 이외의 기호들의 경우, delimeter로

서의 역할 뿐만 아니라 처리의 지침이 되나, 공백의 경우에는 단순한 delimeter로서의 역할만을 수행하므로 parsing 과정에서는 무시한다. 다른 delimeter symbol들로 '₩n'이 올 수 있는데, 이는 라인의 종료, 즉, End of line을 의미하므로 해당 line의 수행이 완료되었음을 인식하는 용도로 사용한다. '&'는 병렬 작업의 수행을 의미하며, '<', '>'는 stdin, stdout redirection이 수행되어야 함을 의미한다. 마지막으로 '|'의 경우에는 pipe가 수행되어야 함을 의미한다. 따라서, 이들 symbol 이외의 문자들은 command 혹은 argument로 볼 수 있으므로 각각을 하나의 처리 단위로 인식하도록 한다. 즉, 다음과 같은 입력이 들어오는 경우에는 token으로 parsing된 경우 아래와 같은 형태로 저장하도록 한다.

prompt> ls -a & ps > a.txt & who & ls -l | more

ls	-a	ps	a.txt	who	ls	-l	more

위의 경우처럼 공백과 다른 symbol들을 delimeter로 두고, 해당 line을 하나 하나의 token으로 parsing하도록 하는데, 이 경우 해당 token이 공백으로 인해 구분된 것인지 아니면 기능 symbol들로 인해 구분된 것인지 알 수 없으므로, 해당 token이 parsing될 때 마다 delimeter를 리턴하도록 한다. 그리고 delimeter가 공백이었을 경우에는, command 혹은 argument임을 리턴하도록 한다. 따라서 token 처리부에서는 이 리턴값을 통해 어떠한 과정을 수행해야 할 지를 인식하도록 한다.

수행의 단위는 end of line 혹은 &가 된다. 이외의 symbol들의 경우, 해당 작업에서 redirection이 발생하는지의 여부나 pipe가 발생했는지의 여부를 기록하여 처리하기만 하면 되며, 이들 역시 하나의 command 내에 속하게 된다. 따라서, 지금까지의 설계로는 하나의 command에 한번의 redirection만 수행될 수 있다. 일반적으로 command를 입력할 때, 한번의 redirection만을 사용하므로 redirection을 flag로 두어 처리해도 무방하다 생각되어 이러한 방법으로 설계하였고, 또한 redirection이 여러 번 일어나는 경우를 처리하기 위해서는 상당한 cost가 소요되므로 spec을 이렇게 한정하여 설계하였다.

하나의 command의 parsing이 끝나면, 즉, end of line이나 & 기호를 만나면, 해당 token까지의 모든 token을 담은 work buffer를 parameter로 하여, 해당 command를 수행하도록 한다. 이 경우, pipe나 redirection이 모두 포함될 수 있으므로 상당히 복잡한 형태가 될 수도 있다.

command 처리 펑션에서는 우선 입력받은 command가 exit인지를 확인하여, exit이면 프로그램을 종료하도록 하였다. 그리고, 이외의 경우에는 fork 펑션을 통해 자식 process를 생성한 후, 자식 process에서 command가 수행되도록 하였다. Command의 수행은 execvp 펑션을 사용하여 구현하였고, 이 때의 실행파일 이름과 argument로는 입력받은 buffer를 사용하였다. 자식 process는 해당 command를 수행하고 종료하며,

부모 process는 자식 process로부터 signal이 올 때까지 waitpid 함수를 사용하여, wait 한다. 그리고, 자식 process가 종료하면 종료 코드를 리턴한다.
전체적인 수행 과정은 이러한 형태의 루틴이 loop으로 수행되도록 하며, 사용자가 Ctrl+D를 입력하는 경우에 loop이 종료되도록 하였다.

2.2 병렬 Process의 구현

위와 같은 framework에 '&' symbol을 통한 병렬 동작을 추가하였다. 이 경우에 각각의 command가 병렬적으로 동작해야 하므로, 맨 마지막 command를 제외한 나머지 command는 background로 동작하도록 하였다. 그리고 마지막에 오는 command는 foreground로 동작하게 하였다. 따라서, 각각의 process의 수행 순서를 보장할 수 없었는데, 이것은 원래의 background process 역시 마찬가지의 형태로 수행되므로 문제될 것이 없다 하겠다. 병렬 process를 처리하기 위해 기본 framework에서 몇 가지 부분을 수정해야 했다. 원래의 경우, 입력 받은 하나의 라인이 모두 command와 argument라고 생각하여 처리할 수 있었으나, 병렬 process 처리를 위해, 라인을 읽는 도중에 & 기호가 나오는 경우, 그 이전까지 읽은 라인을 하나의 command와 argument 묶음으로 처리해 주고, 다시 남은 입력을 반복하여 같은 방법으로 처리해야 했다.
'&'를 통한 병렬 process를 구현하는데 있어서 우선 해당 token의 마지막에 '&'가 오는 경우는 background로 수행하도록 하였고, 마지막에 '₩n'가 오는 경우에는 foreground로 인식하여 수행하도록 하였다. 이를 위해 입력 받은 buffer를 parsing하는 routine과 해당 command를 처리하는 routine을 수정하였으며, parsing하는 routine에서는 가장 큰 문제가 되었던 점이 exec 계열의 함수를 사용하는 것으로, execvp와 execlp 중에 더 적합한 함수를 찾다가 결국 execvp를 사용하게 되었고, 이 경우 argument가 buffer의 array가 되므로, 입력받은 token을 다시 buffer array에 copy하여 수행 function의 parameter로 넘겨주는 방법을 사용하였다. 이 buffer는 해당 process의 parameter로 넘겨준 후에는 초기화한 다음 다시 마찬가지의 방법으로 command 단위로 끊어서 입력하는 방법을 사용하였다. 그리고, background process로 수행되게 할 것인지 foreground process process로 수행되게 할 것인지를 알리는 flag도 parameter로 넘겨주는 방법을 사용하였다.
background process와 foreground process의 차이점으로 background의 경우, 해당 process에 종속되지 않으며, foreground의 경우 해당 process의 child로 수행되므로, 이를 처리하는데 있어서 background의 경우는 fork() 펑션을 호출하여 자식 process를 생성한 후, wait을 하지 않고 독립적으로 수행되고 종료하게 하였고, foreground process의 경우에는 생성된 자식 process가 종료하는 것을 waitpid 펑션을 통해 기다리게 한 후, 자식이 종료한 다음 부모 process 역시 리턴하도록 하였다.

2.3 Redirection의 구현

redirection의 처리는 기본적인 framework의 흐름을 크게 변경하지 않고 설계가 가능하였다. 즉, 하나의 command에 하나의 redirection만 일어난다고 생각하면, command에 redirection이 포함되어 있는지, 그리고 포함되어 있다면 어떠한 redirection이 포함되어 있는지만 flag로 구분해서 처리해 주는 형태로 가능하였다. 그리고 redirection이 수행되어 file I/O가 발생하는 경우에는 해당 file을 open하고 dup2 function을 통해 duplicate 해주는 처리만 해주면 되었다.

Redirection은 command의 맨 마지막 argument로 오기 때문에 입력받은 argument array에서 맨 마지막의 argument를 file 이름으로 인식하여 해당 file을 open하며, 이때 redirection의 type이 stdin인지 stdout인지를 구분하여, stdin인 경우 O_RDONLY 옵션으로 열어주고, stdout인 경우에는 O_WRONLY 옵션과 더불어 해당 파일이 존재하지 않는 경우에는 해당 파일을 생성해야 하므로 O_CREAT 옵션을 bitwise or로 처리하여 open 했다. 또한 이미 해당 file이 존재하는 경우 해당 file을 사용해야 하는데, 이 경우 flush를 해주고 사용해야 하므로 O_TRUC 옵션도 추가하였다. 해당 file이 성공적으로 open 되어 file descriptor를 리턴하면 리턴받은 file descriptor를 dup2 function으로 duplicate 하여 process 수행의 input 혹은 output이 되도록 하였다. 이외의 process 수행 과정은 기본적인 과정과 동일하다.

2.4 Pipe의 구현

pipe의 구현은 기존 routine을 그대로 사용하며 구현하기 힘든 부분이었다. 왜냐하면 pipe 이외의 경우, end of line 혹은 &로 구분된 한 array가 하나의 처리 단위이며, 이것을 그대로 execvp의 argument로 넣어주면 되었기 때문이며, pipe의 경우에는 복수개의 처리 단위가 입력으로 들어올 수 있었기 때문이다. 즉, &의 경우는 &로 구분된 한 단위가 독립적인 수행의 단위이므로 기본적인 수행 과정과 크게 다른 부분이 없으나, |의 경우에는 |의 좌우의 command가 서로 연관성이 있으므로 독립적으로 처리할 수 없고 하나의 수행 단위로 보아야 했기 때문이다.

따라서, 설계 과정에서 pipe의 처리는 여러 개의 command가 존재하는 array 하나를 입력받고 내부에 존재하는 pipe의 개수 역시 입력받아 처리하는 function을 만들어 처리하였고, 이 function 내부적으로 입력받은 array를 다시 pipe 단위로 구분하는 작업을 한다. 설계 시에 pipe가 복수개 존재하는 경우는 고려하였으나, command의 argument가 일반 text인 경우는 고려하지 않아 grep과 같은 명령의 경우 의도하는 대로 동작하지는 않는다. 즉 여기에서 생각한 수준은 ls -a -s | more 정도의 command 수준이었기 때문에, 하나의 command 단위는 command 이후에 -로 시작하는 옵션이 argument로 추가되는 경우만을 고려하여 parsing하였다. 복수개의 pipe가 존재하는 경우 앞의 command의 결과가 다음 command의 입력이 되고, 다시 다음 command의 결과가 그 다음 command의 입력이 되는 형태이다. 이를 위해, 입력받은 pipe 개수만큼 pipe를 생성하였다. 이후의 과정은 pipe 역시 dup2를 통해 입력과 출력으로 사용할 수 있으므

로 redirection과 유사한 형태로 사용하였다. 다만, pipe에서 입력을 위한 fd와 출력을 위한 fd의 쌍을 제공하므로, 각각의 pipe를 처리할 때, 해당 pipe 이전의 pipe가 있으면 해당 pipe의 입력 fd를 dup2로 duplicate하여 입력으로 사용하고, 해당 pipe의 출력 fd도 dup2로 duplicate하여 출력으로 사용하였다.
즉, 아래와 같은 형태의 코드가 된다.

```
if(해당 iteration의 pipe index != 0)
        dup2(pipes[pipe index - 1][0], 0);
if(해당 iteration의 pipe index < 총 pipe 개수)
        dup2(pipes[pipe index][1], 1);
```

이후의 과정은 여타의 command 처리와 동일하게 execvp function을 사용하여 구현하였다. 이 경우 역시 &를 처리할 때와 마찬가지로 입력받은 command array에서 해당 pipe에 수행되어야 하는 command와 관련 argument의 집합을 별도의 array에 삽입하여, execvp의 argument로 주었다. Pipe의 경우 background로 수행되는 형태가 아니기 때문에, 이 경우에도 wait을 사용하여, child의 종료를 기다리도록 하였다.

3. 함수 설명

과제를 수행하기 위해 구현한 각 함수의 설명은 다음과 같다.

Function Name	signalHandler		
Input param	int sig	Return value	void
SIGINT signal handler 펑션. SIGINT signal이 발생하는 경우 이를 처리하는 handler 펑션으로, SIGINT가 발생했음을 알리는 메시지를 출력하고, shell program을 종료한다.			

Function Name	ScanInputLine		
Input param	void	Return value	int

사용자가 입력한 command line을 읽어 buffer에 저장하는 펑션. getchar() 펑션으로 사용자가 입력한 command line을 읽어 buffer에 append하며, 도중에 Ctrl+C 혹은 Ctrl+D가 입력되는 경우, shell program을 종료한다. 이외의 경우에는 '₩n'가 command line의 단위이므로, '₩n' symbol이 올 때까지 계속해서 buffer에 캐릭터를 저장하는 작업을 loop을 통해 반복한다. Ctrl+D, 즉 EOF가 발생하면 EOF를 리턴하고, 이외의 경우에는 입력받은 캐릭터의 개수를 리턴한다.

Function Name	TokenizeLine		
Input param	char **outputPointer	Return value	int
입력 buffer를 읽어 token의 array로 parsing한다. global로 선언된 입력 buffer를 읽어, 특수 기호를 delimeter로 구분하여 token으로 parsing한다. 여기에서 특수 기호는 '>', '<', '|', '&' 등의 symbol로 구분되어야 할 하나의 단위를 나타낸다. 해당 symbol이 나오는 경우, 해당 delimeter를 만났음을 리턴하고, 이외의 경우에는 수행되어야 할 command로 인식하여, 특수 기호가 나오지 않는 시점까지 계속 읽어 buffer에 저장하고, command임을 리턴한다. 입력받은 outputPointer는 command의 경우 해당 command를 저장할 buffer이며, return value는 어떠한 처리 단위인지를 나타내는 flag이다. 여기에서의 리턴값은 COMMAND, AMPERSAND 등 소스 코드 처음에 선언한 value이다.			

Function Name	ScanCommand		
Input param	char c	Return value	int
입력받은 char가 특수 기호인지를 반환하는 function. Token으로 분리하는 과정에서 사용하며, 특수 기호가 아닌 command가 검출되었을 경우, 해당 command의 모든 char를 읽어 output buffer에 저장하는 경우에 사용한다. 즉, 이 function이 true를 리턴하는 동안 입력 buffer에서 char를 읽어 저장하면 해당 command를 저장할 수 있다. 일반 char인 경우에 1을 리턴하고, 특수 기호인 경우에는 0을 리턴한다.			

Function Name	ProcessLine		
Input param	void	Return value	int

사용자로부터 입력받은 command line을 처리하는 function. 입력받은 line buffer를 각각의 token으로 parsing하여, 처리할 수 있는 단위가 수집되면, 해당 command array를 수행한다. 이 때, 해당 단위가 background인지를 판별하는 flag setting 및, redirection 포함 여부 flag, pipe 포함 여부 flag 등을 setting한 후, 수행 function에 argument로 주어 수행 function을 호출한다. 정상적으로 수행되면 0을 반환하고, 에러가 발생하면 -1을 반환한다.

Function Name	ExecuteCommand		
Input param	char **cline, int isbackground, int redirect, int *cmdcnt, int pipecnt	Return value	int
입력받은 command array를 처리하는 function. Command array와 해당 command array가 background로 동작해야 하는지 flag, redirection이 포함되는지 여부 flag, pipe 포함 여부, 내부의 argument 개수를 입력받아, 해당 setting 값에 따라 command를 수행한다. fork로 자식 process를 생성한 후, 자식 process 내에서 exec 펑션을 호출하여 수행하며, setting 값에 따라 background 처리 및 redirection 처리, pipe 처리 등의 기능도 수행한다. 정상적으로 수행된 경우 0을 리턴하고, 에러가 발생한 경우 -1을 리턴한다.			

Function Name	ExecuteRedirection		
Input param	int redirectionType, char *fileName	Return value	int
Redirection의 수행을 위해 file 처리를 하는 function. 입력받은 redirection 종류에 따라 stdin 혹은 stdout의 redirection을 수행하며, 입력받은 file을 열고, file descriptor를 duplicate하는 작업을 수행한다. 정상적으로 수행되면 0을 리턴하고, 에러가 발생하면 -1을 리턴한다.			

Function Name	ExecutePipes		
Input param	char **cline, int pipecnt	Return value	void
Pipe를 수행하는 function. 기본적인 수행 과정은 ExecuteCommand function과 유사하며, 다만 pipe로 수행되어야 하므로, 입력받은 pipe의 개수만큼 pipe를 생성하고, 각각의 file descriptor를 input과 output으로 처리하는 작업을 수행한다. 그러나, ExecuteCommand와는 달리 입력받은 command array를 다시 각 command 단위로 parsing하는 과정도 수행하며, background의 처리도 하지 않는다.			

Function Name	CloseAllPipes		
Input param	int pipes[][2], int cnt	Return value	void
입력받은 pipe array를 close하는 function. Pipe의 수행이 종료되는 경우 해당 pipe를 close하는 용도로 사용하며, 중간에 남아 있을 지 모르는 pipe 역시 모두 close하는 작업을 수행한다.			

Function Name	main		
Input param	int argc, char *argv[]	Return value	int
main function으로 간단한 shell의 기능을 수행한다. 사용자로부터 command line을 입력받아 해당 line을 처리하는 작업을 수행한다. 기본적인 command의 처리 기능 이외에 병렬 process, redirection, pipe 등의 처리 기능도 수행한다. 정상적으로 종료하는 경우 0을 반환하고, 에러가 발생하는 경우 -1을 반환한다.			

4. 실행 및 결과 분석

대부분의 일반적인 명령에 대해서는 정상적인 결과를 출력하였다. 아래의 결과는 과제 spec의 각각의 단계의 예제를 수행한 결과 화면이다.

Part A

입력 : ls –a [CR] Ctrl+C

Part B

입력 : ls –a & who & ps & ls > a.txt

Part C

입력 : ls –l –s | more

5. 구현의 한계 및 Discussion

* pipe의 처리

&의 경우에는 &가 delimeter가 되어 좌우의 command가 연관성이 없이 독립적으로 수행되지만 |의 경우에는 좌측 command의 output이 우측 command의 input이 되어야 하므로, 전체 line을 parameter로 두고 pipe를 수행하도록 했다. 따라서, pipe가 복수개

올 수도 있으며, 이에 대한 처리도 필요하므로, 각 command 및 argument의 array를 두고, pipe를 수행하는 function에서 이를 다시 parsing하여 해당 process를 수행하도록 했다.
Parsing 과정에서 command와 argument를 구분하는 방법으로 시작 캐릭터가 '-'인 경우는 argument이고 이외의 경우에는 일반 command로 처리하도록 하였는데, 이는 주로 ls를 고려한 방법이었고, grep과 같은 경우 argument가 일반 text가 되므로 grep을 수행하는 경우에는 문제가 발생하였다.
이를 정상적으로 처리하기 위해서는, 설계 단계에서 pipe를 위한 별도의 command 단위 array를 두고, 이를 입력받아 처리하는 형태로 하면 된다. 혹은 pipe의 개수를 세는 대신 각각의 pipe가 수행되어야 하는 index의 array를 기록하여, 이를 토대로 command를 구분하여 처리하면 된다.

* Redirection의 처리

처음 설계 시에 Redirection은 하나의 command array 단위에서 한번만 포함된다는 가정 하에 설계하였으나, 게시판에 보면 하나의 command 단위에서 여러 번 포함되는 경우도 있었다. 처음 설계 시에 고려하지 못한 사항으로, 이 경우 정상적으로 처리되지 않고, 맨 마지막의 redirection type만 처리되며, 처리되지 않은 다른 redirection의 file name은 command의 argument로 입력되게 된다. 이 경우 에러가 발생할 수 있다.
이 경우 역시 pipe의 경우에서와 마찬가지로 각각의 redirection이 발생한 지점의 index를 기록하여 두었다가 execute 시에 각각의 redirection을 따로따로 처리해 주면 된다.

* exit의 처리

exit을 입력하는 경우 대부분 정상적으로 종료가 수행되나, pipe를 수행하는 과정 등에서 에러가 발생하여 입력을 기다리는 경우, exit이 정상적으로 처리되지 않고 무시되는 경우가 발생하였다. 이 경우, carriage return을 여러 번 입력하여 해당 command가 종료되도록 한 후 exit을 type하면 정상적으로 처리된다.

* 기타

전반적으로 unix에 익숙하지 않은 상태에서 각 spec의 기능 구현에 급급하다 보니 예상하지 못한 문제들이 자주 발생하였다. 또한 입력 예제가 있어 입력의 범위를 미리 예상하였다면 좀더 해당 내용을 염두에 두고 설계하여 시행착오를 줄일 수 있었겠으나, 그러지 못해 많은 시행착오를 겪었다. 특히 argument 부분에서 예상하지 못한 경우가 많이 있었고, 위에서 열거한 내용들이 그것이다. 즉, argument는 무조건 -로 시작한다는 전제라던가 redirection은 command에서 한번만 포함된다는 등의 전제 등이다. 이 밖에 more가 오는 경우 만을 생각하여, pipe의 경우에는 argument가 없을 것이라 생각하여 execlp로 구현하였다가, pipe의 경우에도 충분히 많은 argument가 올 수 있다는 생각이

들어 다시 execvp로 변환하는 등 많은 시행착오를 거쳤다. 또한 Buffer 관리에서도 실수가 많아 디버깅에 오랜 시간이 걸렸다. 많은 시행착오를 겪고, 또 다양한 입력을 경험해 본 지금에 와서 다시 프로그램을 구현한다면 앞의 한계가 발생하지 않는 설계를 할 수 있을 것 같고, 좀 더 robust한 shell을 작성할 수 있을 것 같아 아쉬움이 많이 남는다.

6. Source Code

```
/* ============================================
*          나만의 Unix Shell 만들기
============================================ */

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

/* ================
Preprocessor
================ */

#define BUF_MAX      1024                    /* input line의 buffer size */
#define CMD_MAX      128                     /* command의 최대 처리 개수 */

#define COMMAND         1                                   /* 보통 command */
#define EOL                             2                                       /* end of line
*/
#define AMPERSAND           3                                       /* 병렬 처리 */
#define STDOUT_REDIRECTION      4               /* stdin redirection */
#define STDIN_REDIRECTION       5               /* stdout redirection */
#define PIPE                                    6                                       /*
pipe */

/* ================
Global Variables
```

```
================== */

static char inputBuffer[BUF_MAX];                /*input된 argument를 담을 buffer */
static char tokenBuffer[2*BUF_MAX];              /* parsing된 token들을 담을 buffer */
static char *workPointer = inputBuffer, *tokenPointer = tokenBuffer; /* parsing 작업을 위한 pointer
*/

/* parsing과정에서 검출할 symbol 지정 */
static char symbols[ ] = {' ', '₩t', '&', '₩n', '₩0', '|', '>', '<'};



/* ============================
SIGINT signal의 handler
============================ */
void signalHandler(int sig)
{
        printf("Caught SIGINT₩n");
        exit(0);
}

/* ============================
line을 입력받아 buffer에 저장
============================ */
int ScanInputLine()
{
        int c, count;                    /* 입력받은 line을 char 하나씩 읽어 저장 */

        /* buffer pointer 초기화 */
        workPointer = inputBuffer;
        tokenPointer = tokenBuffer;

        printf("> ");                    /* 프롬프트 출력 */
        count = 0;

        while(1) /* loop을 돌며 input을 buffer에 저장*/
        {
```

```
                if(signal(SIGINT, signalHandler) == SIG_ERR)
                {       /* sigint가 발생하는 경우, signal handler에 넘김*/
                        perror("signal error₩n");
                }
                if((c = getchar()) == EOF) {  /* Ctrl+d signal이 오는 경우 종료 */
                        printf("₩n");
                        return EOF;
                }

                if(count <BUF_MAX)          /* buffer 용량을 넘지 않으면 buffer에 삽입 */
                        inputBuffer[count++] = c;

                if(c == '₩n' && count < BUF_MAX)    /* line이 끝나면 입력받은 char 수 리턴*/
                {
                        inputBuffer[count] = '₩0';
                        return count;
                }

                if(c == '₩n')       /* 여기까지 오는 경우 입력된 line이 buffer 크기보다 큼 */
                {       /* command 무시 */
                        count = 0;
                        printf("> ");
                }
        }
}

/* ==================
buffer Tokenizer
================== */
int TokenizeLine(char **outputPointer)
{
        int type;

        /* 현재 token의 pointer를 output의 pointer로 지정 */
        *outputPointer = tokenPointer;
```

```
        /* token에서 공백 제거 */
        while(*workPointer == ' ' || *workPointer == '₩t')
                workPointer++;

        /* 다음 token pointer 지정 */
        *tokenPointer++ = *workPointer;

        /* buffer에 들어있는 token의 종류에 따라 리턴 값 지정 */
        switch(*workPointer++)
        {
        case '>':               /* stdout으로 redirection */
                type = STDOUT_REDIRECTION;
                break;
        case '<':               /* stdin으로 redirection */
                type = STDIN_REDIRECTION;
                break;
        case '&':               /* 병렬 동작 */
                type = AMPERSAND;
                break;
        case '|':               /* pipe */
                type = PIPE;
                break;
        case '₩n':              /* end of line*/
                type = EOL;
                break;
        default:                /* 보통 commands */
                type = COMMAND;
                while(ScanCommand(*workPointer) == 1)       /*      입력      buffer에서      일반
command 분리 */
                        *tokenPointer++ = *workPointer++;
                break;
        }
        /* null terminated */
        *tokenPointer++ = '₩0';

        return type;
```

```
}

/* ==================
buffer를 scan하여  symbol이 아닌 부분만 분리
================== */
int ScanCommand(char c)
{
	char *workpointer;

	for(workpointer = symbols; *workpointer; workpointer++)
	{	/* symbol이면 loop 종료 */
		if(c == *workpointer)
			return 0;
	}

	return 1;
}

/* ==================
Process Line
================== */
int ProcessLine(void)
{
	char *arguments[CMD_MAX + 1];	/* 처리할 token의 pointer */
	int tokenType;		/* 해당 token의 type */
	int commandCnt;		/* 현재 command pointer */
	int isBackground;		/* Background인지 여부 flag */
	int redirectionType;	/* redirection 종류 flag*/
	int pipeCnt;		/* pipe의 개수 */

	/* 초기화 */
	commandCnt = 0;
	pipeCnt = 0;
	redirectionType = 0;

	for(;;)	/* loop을 돌며 token 처리 */
```

```
        {
                /* input buffer를 token으로 잘라 처리 */
                tokenType = TokenizeLine(&arguments[commandCnt]);
                switch( tokenType )
                {
                case COMMAND:   /*일반 command*/
                        {
                                if(commandCnt < BUF_MAX)
                                        commandCnt++;
                        }
                        break;
                case STDIN_REDIRECTION: /* stdin redirection */
                case STDOUT_REDIRECTION:        /* stdout redirection */
                        {
                                /* redirection의 경우 redirection type만 저장 */
                                redirectionType = tokenType;
                        }
                        break;
                case PIPE:      /* pipe */
                        {
                                /* pipe의 count를 증가시키고 command는 제거 */
                                pipeCnt++;
                                arguments[commandCnt] = NULL;
                        }
                        break;
                case EOL:       /* end of line */
                        isBackground = 0;
                        if(commandCnt != 0)
                        {       /* 현재 command는 제거 */
                                arguments[commandCnt] = NULL;
                                ExecuteCommand(arguments, isBackground, redirectionType,
&commandCnt, pipeCnt);
                        }
                        return;
                        break;
                case AMPERSAND:         /* 병렬 처리 */
```

```
                        {
                                isBackground = 1;
                                /* 해당 시점의 command 처리 */
                                if(commandCnt != 0)
                                {       /* 현재 command는 제거 */
                                        arguments[commandCnt] = NULL;
                                        ExecuteCommand(arguments,       isBackground,
redirectionType, &commandCnt, pipeCnt);

                                        /* 수행 후 buffer clear */
                                        for( ;commandCnt>0; --commandCnt)
                                        {
                                                arguments[commandCnt] = NULL;
                                        }
                                }
                        }
                        break;
                }
        }
        return 0;
}

/* ==================
Execute command
================== */
int ExecuteCommand(char **cline, int isbackground, int redirect, int *cmdcnt, int pipecnt)
{
        int pid;
        int status;

        /* exit이 입력된 경우 종료 */
        if(strcmp(*cline, "exit") == 0)
        {
                exit(0);
        }
```

```
        pid = fork();           /* fork로 child process 생성 */

        if(pid == 0)            /* child인 경우 process 처리 */
        {
                if(pipecnt > 0)
                {               /* pipe로 수행되어야 하는 경우 */
                        ExecutePipes(cline, pipecnt);
                }
                else
                { /* pipe가 아닌 경우 */
                        if((redirect == STDIN_REDIRECTION) || (redirect == STDOUT_REDIRECTION) )
                        {               /* redirection이 수행되어야 하면 file redirection 처리 */
                                ExecuteRedirection(redirect, cline[--(*cmdcnt)]);       /* 하나 이전의 command가 file name */
                                cline[*cmdcnt] = NULL;
                        }
                        /* 해당 command execute */
                        {
                                if((execvp(*cline, cline)) < 0)
                           {
                                 perror(*cline);
                                 exit(0);
                           }
                        }

                }

        }

        /* background로 처리되어야 하면 자식 pid 출력 후 종료 */
        if(isbackground == 1)
        {
                printf("[%d] %s₩n", pid, *cline);
                return 0;
        }
```

```
	/* 해당 pid의 자식이 종료하는 경우, status 코드 확인 */
	if(waitpid(pid, &status, 0) == -1)
		return -1;
	else
		return status;
}

/* ==================
redirection 처리
================== */
int ExecuteRedirection(int redirectionType, char *fileName)
{
	int fd;	/* file descriptor */

	if(redirectionType == STDIN_REDIRECTION)
	{	/* stdin redirection인 경우 */
		if((fd = open(fileName, O_RDONLY)) < 0)
		{	/* input 파일 열기 */
			perror("Input file cannot be opened₩n");
			exit(1);
		}
		/* duplicate */
		if((dup2(fd, 0)) < 0)
		{
			perror("Dup cannot be executed₩n");
			exit(1);
		}
	}
	else if(redirectionType == STDOUT_REDIRECTION)
	{	/* stdout redirection인 경우 */
		if((fd = open(fileName, O_WRONLY|O_CREAT|O_TRUNC, 0744)) < 0)
		{	/* output file 생성하기 */
			perror("Output file cannot be created₩n");
			exit(1);
		}
```

```
                /* duplicate */
                if((dup2(fd, 1)) < 0)
                {
                        perror("Dup cannot be executed₩n");
                        exit(1);
                }
        }
        /* file 닫기 */
        close(fd);
}


/* ==================
Execute Pipes
================== */
void ExecutePipes(char **cline, int pipecnt )
{
        int status, i, j;
        int pipes[CMD_MAX][2];     /* pipe의 i/o를 위한 file descriptor */
        int cmdIndex = 0;          /* command의 index */
        char* cmdBuf[CMD_MAX];



        /* 입력받은 pipe 개수만큼 pipe 생성 */
        for(i = 0; i <= pipecnt; i++)
        {
                if((pipe(pipes[i])) < 0)
                {
                        perror("Pipe doesn't exist₩n");
                        exit(1);
                }
        }


        for(i = 0; i <= pipecnt; i++)
        {
                /* command buffer 초기화 */
```

```
for(j=0; j<CMD_MAX; ++j)
{
        cmdBuf[j] = NULL;
}
j = 0;
while(cline[cmdIndex])
{       /* 입력받은 buffer를 다시 command 단위 buffer로 분리 */

        cmdBuf[j] = cline[cmdIndex];
        j++;
        cmdIndex++;
        if(cline[cmdIndex] == NULL)
        {
                cmdBuf[j] = NULL;
                break;
        }
        if(cline[cmdIndex][0] != '-')
        {
                cmdBuf[j] = NULL;
                break;
        }
}

/* 더 이상 남은 command가 없는 경우 종료 */
if(cmdBuf[0] == NULL)
{
        exit(0);
}

if(fork() != 0 ) /* 자식의 경우에만 루틴 수행 */
        continue;

/* input descriptor duplicate */
if(i != 0)
        dup2(pipes[i - 1][0], 0);
```

```
                /* output descriptor duplicate */
                if(i != pipecnt)
                        dup2(pipes[i][1], 1);
                /* pipe file descriptor 해제 */
                CloseAllPipes(pipes, pipecnt);

                /* exit이 입력된 경우 종료 */
                if(strcmp(cmdBuf[0], "exit") == 0)
                {
                        exit(0);
                }

                /* 해당 pipe 처리 */
                /*if(execlp(cline[cmdIndex], cline[cmdIndex], ) == -1)*/
                if((execvp(*cmdBuf, cmdBuf)) < 0)
                {
                        perror(*cmdBuf);
                        exit(1);
                }
                exit(0);

        }
        /* 종료 후 pipe file descriptor 해제 */
        CloseAllPipes(pipes, pipecnt);

        for(i = 0; i <= pipecnt; i++)
                wait(&status);
}

/* ==================
Close Pipes
================== */
void CloseAllPipes(int pipes[][2], int cnt)
{
        int i;
```

```
        /* 입력받은 pipe file descriptor 해제 */
        for(i = 0; i < cnt; i++)
        {
                close(pipes[i][0]);
                close(pipes[i][1]);
        }
}

/* ==================
main function
================== */
int main(int argc, char *argv[])
{
        while(ScanInputLine() != EOF)          /* 라인을 입력 받아 처리 */
                ProcessLine();
        return 0;
}
```