- Outline how programs are launched and arguments are passed

이번 과제로 주어진 프로그램은 main함수에서 시작하여, main함수 내의 while문을 통해 prompt를 화면에 출력하고, 수행하고자 하는 명령을 입력받아 해당기능을 실행하는 것을 반복하다가 exit가 명령어로 들어왔을 때 종료하는 프로그램이다. 이때 기능을 수행하는 함수는 크게 main() -> append_history() -> __process_command() -> run_command()의 방향으로 실행되는 것을 알 수 있다. 입력받는 명령어는 크게 두 가지로 나눌 수 있는데, 기능을 직접 구현하는 내장함수와 exec()를 통해 기능을 수행하는 외장함수이다.

1. Execute external commands

fork()를 통해 새로운 process를 생성하여 command로 입력받은 명령어를 execvp()함수를 통해 수행하도록 구현한다. 이때, 부모노드는 새로 생성된 process(child)가 기능을 수행하고 종료할 때까지 기다린다.

2. change working directory

command에 cd 명령어가 들어왔을 경우에는 "home" directory로 이동하는 경우, 그 외의 directory로 이동하는 경우로 나누어 실행한다. command가 "cd" 또는 "cd ~" 일 경우에는 home directory로 이동하고, 그 외에는 cd 뒤에 붙은 경로로 directory를 변경해주도록 구현하였다.

3. keep the command history

- command에 history 명령어가 들어왔을 경우에는 연결리스트로 연결된 command들을 순회하며 모두 출력한다. (밑에 자세한 설명 있음)
- "! (num)" 명령어가 입력되었을 경우에는 연결리스트로 연결된 리스트들을 순회하며 num과 같은 번호를 가진 command들을 찾아 __process_command()함수를 통해 해당 command명령을 재실행시킨다.

4. connect two processes with a pipe

| (파이프) 전과 후의 command를 두 개의 token들로 나누어 저장하고, 첫 번째 token의 출력결과가 두 번째 token의 입력이 되어 실행되도록 구현하였다. (밑에 자세한 설명 있음)

5. argument passing방법

위의 네 가지 기능은 run_command에서 실행되며 모두 입력으로 받은 command를 단어 단위로 parsing한 char *tokens[] 형태의 argument를 받아 수행한다. 파이프를 제외한 모든 내장함수에 대한 명령어는 tokens의 0번 index에 저장되어있

으므로, pipe만 따로 확인해줄 수 있다면, 나머지 기능은 tokens[0]과 비교하여 찾아 실행할 수 있다.
run_command()외에 구현한 함수들의 argument passing을 간단하게 설명하겠다.

- void append_history(char *command)

main함수에서 입력받은 char* 형의 command를 연결리스트에 순차적으로 연결시킨다. append된 history는 미리 선언된 연결리스트에 저장하기 때문에 따로 반환값은 없다.

- void dump_history()

argument값이나 return값이 따로 없이, 연결리스트를 참조해 순회한다.

- struct entry *find_history(char* num)

"! (num)" 형태의 command를 입력받았을 때, num 부분을 argument로 받는다. num은 char*형이므로, 함수 내에서 문자열을 정수로 변환하여 수행하여 해당 entry를 찾아 반환한다.

- int is_pipe(char *tokens[])

run_command()에서 argument로 받은 tokens를 그대로 argument로 받아 tokens안에 파이프가 있는지 확인하고 있으면 파이프가 있는 index 번호를, 없으면 0을 반환한다.

- How the command history is maintained and replayed later

main함수에서 command를 입력받을 때마다 append_history()를 실행하여 command를 연결리스트에 순차적으로 연결한다. 이때 command들을 연결리스트에 연결하기 위하여 command들을 각각의 리스트헤드와, string, command의 순번(int)을 가진 struct entry 구조체로 abstract해 command에 대한 entry를 정의하였다. 따라서 append_history()에서 struct entry*형 변수를 생성해 command와 순번을 지정해주고, 리스트헤드를 연결리스트에 add해서 각 command에 대한 entry들을 연결리스트에 순차적으로 연결하였다. (pa0의 push_stack()과 동일)

이후 history명령어가 실행되면, 현재까지 생성된 list들을 순회하며 list를 포함한 entry들의 command를 출력하도록 dump_history()함수를 구현하여 사용하였다. dump_history()에서는 list_for_each_entry_reverse()함수를 통해 연결리스트에 접근하며, 각 리스트를 포함하는 entry의 주소를 찾아내, 해당 entry가 참조하는 command와 순번을 찾아 출력하도록 하였다. 또한 "! (num)"명령어를 사용하기 위

해 find_history()함수를 정의하였는데, find_history()는 dump_history()와 같은방법으로 연결리스트를 순회하며, num과 같은 순번을 가진 entry를 찾아 반환하는 함수이다. (+ argument로 받는 num은 char* 형이므로, atoi함수를 사용해 정수로 변환하여 순번을 비교하였다.) 따라서 find_history()로 찾은 entry의 command를 __process_command()의 argument로 주어 해당 명령어를 다시 실행할 수 있도록 하였다.

- Your STRATEGY to implement the pipe

먼저 command로 들어온 명령이 pipe인지를 확인하고, pipe가 맞다면 | 이전의 command와 | 이후의 command를 두 개의 token으로 나누어 저장한다. 이를 각각 tokens1, tokens2라고 한다면, tokens1의 결과(출력)를 tokens2의 입력으로 넣어준다. 이를 위해서 file descriptor인 fd를 정의하여 fd[0]를 read, fd[1]은 write으로 구성된 pipe를 만들었다. 이후 두 개의 process를 만들어 tokens1과 tokens2를 각각 실행하도록 하였다. tokens1(파이프 이전의 command)의 결과는 pipe의 write부분에 들어가도록 하였고, tokens2(파이프 이후의 command)의 입력이 pipe의 read에서 읽을 수 있도록 구현하였다. 또한 각각의 process를 실행시키며 각각 불필요한 부분, 즉 tokens1에서는 pipe의 read부분, tokens2에서는 pipe의 write부분을 사용하지 않으므로, close를 통해 닫아주었다. 이후 parent process에서는 pipe전체를 close해주고, wait을 통해 child process들이 종료하는 것을 기다릴 수 있도록 구현하였다.

- AND lessons learned

먼저 이론수업에서 들었던 process 작동법을 이번과제에서 명확하게 이해할 수 있었다. 특히 이론수업에서 parent process를 fork()를 통해 복사하여 child process를 만드는 것까지는 쉽게 이해했지만, 왜 새로운 process를 생성하는지, 새로운 process를 생성하면 어떻게 따로 동작하게 할 수 있는지 등이 명확하지 않았었는데, 직접 fork(), exec(), wait()을 통해 process를 생성하고, 프로그램을 구현해보니 process작동법과 효율성을 받아들일 수 있게 되었다.

또한 pipe의 이론수업에서 pipe가 어떻게 작동하는지, 어떻게 사용하는지에 대해서 배웠지만, 어떻게 pipe를 구현하고, pipe를 통해 두 token의 입력과 출력을 통제하는지 이해하기가 어렵고 매우 추상적이었는데, pipe를 직접 구현해봄으로 인해서 pipe를 read와 write로 나누어 tokens1과 실행결과를 tokens2의 입력으로 넘기는 방법을 완벽하게 터득하고 pipe의 구조를 완벽하게 이해할 수 있었다.

마지막으로 당연하지만, 프로그래밍 문법이나 동적할당의 개념에 대해서 더욱 꼼꼼

하고 명확하게 이해할 수 있었다. 기존에는 문법을 많이 틀려서 한 번에 돌아가는 코드를 작성하기 어려웠고, segment오류가 어디서 발생하는지, 어떻게 해야 해결할 수 있는지 단번에 알아내기 어려웠는데, 이제는 무엇을 조심해야하고, 어떻게 해결해야하는지에 대해서 많이 능숙해졌고, 특히 동적할당을 통해 메모리를 할당해주고 사용하는 것을 자유롭게 사용할 수 있게 되어, 코드를 보다 효율적으로 구현할 수 있게 되었다.