**System Programming Project 2**

담당 교수 : 박성용

이름 : 이예준

학번 : 20212022

1. **개발 목표**

1. Phase 1

basic internal shell command을 실행할 수 있도록 구현했다.

입력받은 command를 parsing해서 2차원 배열에 저장한 뒤

fork를 통해 child 프로세스에서 명령어를 실행했다. parent 프로세스에서는

child 프로세스가 끝날 때까지 기다린 뒤 reap해준다.

2. Phase 2

Phase1에 추가하여 pipe에 대한 처리도 구현했다.

child 프로세스가 parent 프로세스에 데이터를 보내기 위해 pipe 함수로

파이프를 생성했다. 그 다음 fork를 통해 child 프로세스에서 명령어를 수행한 후

나온 Output을 parent 프로세스로 보내고, pipe 명령어가 존재하면 다시 fork로

child 프로세스에 가서 parent에게 받은 Output을 Input으로 받아 명령을 수행한다.

이 과정을 가장 마지막 명령어까지 수행하고 마지막에 나온 Output을 출력해준다.

3. Phase 3

none

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

과제 자료에 나온 명령어 cd, ls, mkdir, rmdir, touch, cat, echo, exit에 대해

정상적으로 작동하는 것을 확인했다.

1. Phase 2

파이프로 연결된 명령어를 정상적으로 처리하는 것을 확인했다.

예를 들어 cat <filename> | sort -r 을 입력하면 해당 file 내용에 대해

역순으로 정렬한 내용을 출력한다.

1. Phase 3

none

* 1. **개발 내용**
* **아래 항목의 내용만 서술**
* **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명

fork() 함수를 이용해서 child process를 생성한 뒤 execvp()함수로 명령어를

수행하고, parent process는 waitpid() 함수를 통해 child process가 종료될

때까지 기다린 뒤 reap을 해주어 Zombie가 되는 것을 방지한다.

* + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow

child process가 모두 끝나면 parent process에게 SIGCHLD 시그널을 보내게 되는데

child process는 pid 값이 0이고, parent process는 pid 값이 child process의

pid이므로 0 이 아닌 경우에 대해서 wait() 함수를 호출해 signal handling 을

할 수 있다.

* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)

예를 들어 파이프가 1개인 명령어 ls | grep <filename> 이 있다고 한다면

먼저 ls 명령어를 처리하기 위한 child process를 fork()로 생성하고,

parent process는 child process에서 ls에 대한 처리가 끝날 때까지 기다린다.

child process에서는 우선 pipe() 함수로 fds[]에 파이프를 형성하고

close(fds[0])로 Input에 대해 닫고, dup2() 함수로 fds[1]이 Output을

가리키게 한 다음 execvp() 명령어를 수행한다. 따라서 fds[1]에 명령어를

실행한 Output이 들어간다. parent process로 다시 돌아오면 close(fds[1])로

Output에 대해 닫은 다음 fd가 Output을 받도록 한다. 그리고 재귀를 통해

다음 명령어로 이동한다. 다시 똑같이 child process를 생성하고 진행한다.

첫 명령어에서는 fd값이 0이여서 표준입력을 받았지만 이번에는

이전 과정의 Output을 받았기 때문에 그 Output을 이번 과정에서는

dup2()로 Input을 가리키게 한다. 이번에는 마지막 명령어이기 때문에

Output에 대한 fds를 dup2()로 다른 fds를 가리키지 않도록 하여

명령어에 대한 Output이 정상적으로 출력될 수 있도록 한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

위에서 ‘|’를 구분하는 과정에서 pipe의 개수를 구해주고, 그 개수만큼 재귀를

반복했다. 먼저 pipe() 함수를 이용해서 pipe를 생성하고 fork() 함수로

child process를 생성했다. 그 다음 child process에서 dup2()함수로

이전 과정에 대한 Output을 Input으로 사용하고 Output을 다음 과정의

Input으로 보냈다. 만약 처음 과정이라면 dup2()로 Input을 pointing하지 않고,

마지막 과정이라면 dup2()로 Output을 다른 곳으로 Pointing하지 않고

execvp() 함수로 출력한다.

* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명

none

* 1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

소스코드 : 제공된 shellex.c, csapp.c, csapp.h를 사용했다.

**- Phase1**

1. builtin\_command 함수에 명령어 cd, exit 처리과정을 추가

cd에 대한 처리는 chdir함수로 했고, 만약 argv[1]이 없는 경우에는 HOME으로

디렉토리를 이동했다. exit 명령어는 quit 명령어와 똑같이 exit(0)로 shell을 종료했다.

2. user\_command 함수

pid()로 child process를 생성 후 child process에서 execvp()로 명령어에 대한 처리를 했다.

parent process에서는 waitpid()로 child process를 기다린 뒤 reap해주었다.

**- Phase2**

1. eval 함수 수정

먼저 pipeline에 대한 parsing 함수를 추가하여 명령어를 한번 가공했다.

그 다음 만약 파이프 명령어가 존재하면 파이프 명령어를 처리하는 함수를 호출하고,

아니라면 위에서 구현한 user\_command 함수를 호출했다. 마지막으로 pipe 명령어를

저장하기 위해 할당했던 배열을 해제하는 함수를 호출했다.

2. New\_argv 함수

‘|’의 개수를 저장하고, 3차원 배열 pipe\_argv을 동적 할당했다.

pipe 명령어를 기준으로 명령어를 쪼개서 3차원 배열 pipe\_argv에 각각 따로 저장했다.

3. pipe\_command 함수

pipe()와 dup2()로 파이프를 생성해서 입력은 이전 과정의 출력으로, 출력은 다음 과정의

입력으로 연결해주었고, 마지막 명령어는 파이프로 연결하지 않고, execvp()로 정상적으로

출력이 되게 했다.

4. free\_pipe 함수

반복문을 이용해 3차원 배열 pipe\_argv를 할당 해제했다.

**- Phase3**

none

1. **구현 결과**
   1. **Flow Chart**

* **2.B.개발 내용에 대한 Flow Chart를 작성.**
* **(각각의 방법들에서 추가된 내용(fork, pipeline, background)만 특성이 잘 드러나게 그리면 됨.)**

1. **Phase 1 (fork)**

**텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명**

1. **Phase 2 (pipeline)**

**텍스트, 도표, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명**

1. **Phase 3 (background)**