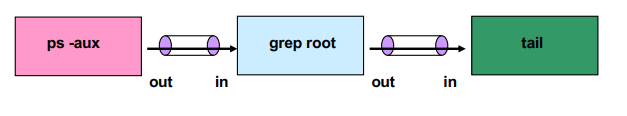
**System Programming Project 4**

담당 교수 : 김영재 교수님

이름 : 조이준

학번 : 20161647

1. **개발 목표**

* **Fork()를 이용한 나만의 쉘을 구현하는 것이 이번 프로젝트의 최종 목표이다.**
* **첫 단계로는 fork()를 이용하야 명령어의 실행을 자식 프로세스에서 진행하도록 쉘을 구성하는 것이다.**
* **두 번째 단계는 ‘|’로 연결된 명령어를 inter-process communication을 통해 구현한다. 구조는 다음과 같다.**
* ****
* **마지막 단계는 프로세스의 백그라운드 실행과 builtin-command인 jobs, kill, fg, bg를 구현하는 것이다.**

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Phase 1

기본 쉘 명령어로

cd, ls, mkdir, rmdir, touch, echo, cat, exit를 입력 받고 결과를 출력한다.

1. Phase 2

기본 쉘 명령어를 pipe로 구성한 명령어를 입력 받고 결과를 출력한다.

즉, inter-connection process를 구현한다.

1. Phase 3

&를 붙인 명령어를 입력 받아 background에서 실행한다.

Builtin-command로 추가한 jobs는 job 배열에 저장된 모든 작업을 출력한다. kill명령어는 SIGKILL signal을 전송한다.

fg명령어는 background에 위치한 작업을 foreground로 옮긴다.

bg명령어는 foreground에 위치한 작업을 background로 옮긴다.

* 1. **개발 내용**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명

Fork() 함수를 통해 child process를 생성하는데 child process는 0값을 반환하기 때문에 해당 반환 값으로 부모 프로세스와 자식 프로세스를 구분한다.

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

자식 프로세스가 종료하면서 발생하는 SIGCHLD 시그널을 handling하는 sigchild\_handler 함수를 만들었고 Signal 함수를 통해 SIGCHLD가 들어오면 handler 함수를 통해 처리되도록 했다. 부모 프로세스는 Signal handler를 통해 reap한 자식 프로세스의 process id가 volatile 변수를 통해 들어올 때까지 Sigsuspend 함수를 통해 기다리고 자식 프로세스가 reap되면 다음 명령을 받게 넘어간다.

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

일단 입력 받은 명령어에 ‘|’가 존재하는지 검사하여 존재하면 ‘|’를 기준으로 명령어를 나누었다. ‘|’ 기준으로 앞에 있던 하나의 명령어를 먼저 fork()를 통해 실행하고 뒤의 명령어는 다시 ‘|’ 검사를 통해 추가적인 ‘|’가 존재하는 지 검사하여 동일하게 실행한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

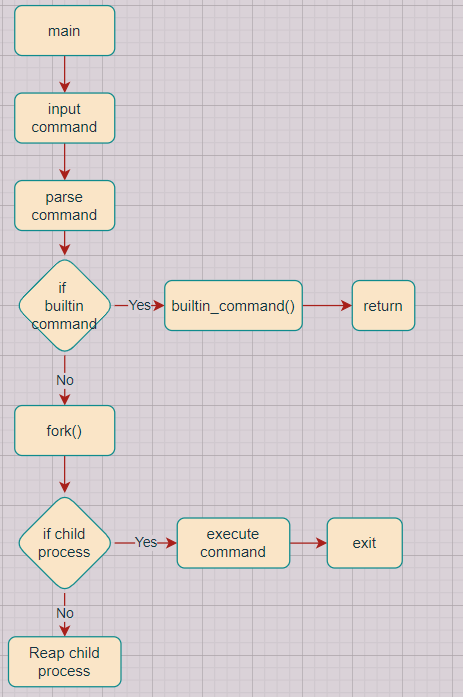
Pipe 개수에 따라 명령어 처리를 다르게 하지 않았고 pipelined 명령어를 실행하는 함수를 재귀 함수를 통해 실행했다. 매번 함수의 끝에서 생성했던 자식 프로세스를 reap 해주도록 했다.

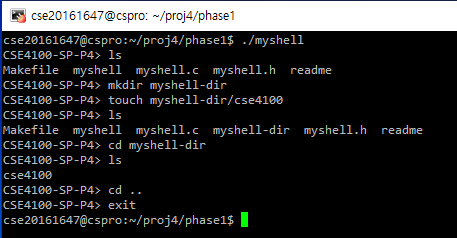
* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명

입력 받은 명령어의 맨 뒤에 ‘&’ 문자가 존재하면 background flag를 설정해주고 &문자는 명령어 문자열에서 지워주고 background 인 경우에는 부모에서 바론 return하여 reap되지 않고 기다리고 foreground 명령이 들어오면 기다리던 자식 프로세스까지 모두 reap하도록 구현했다.

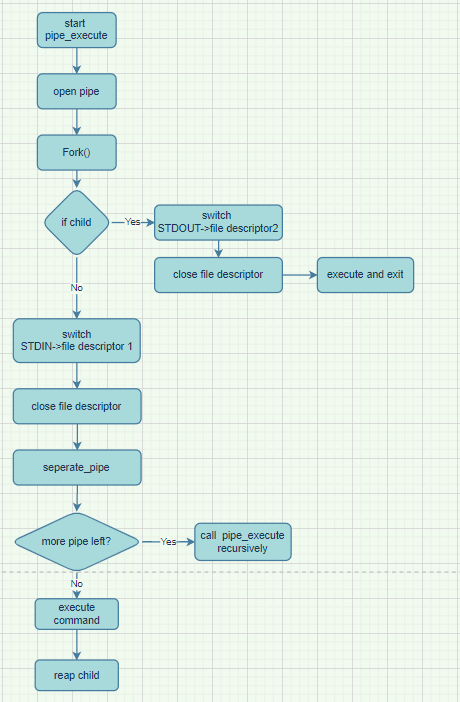
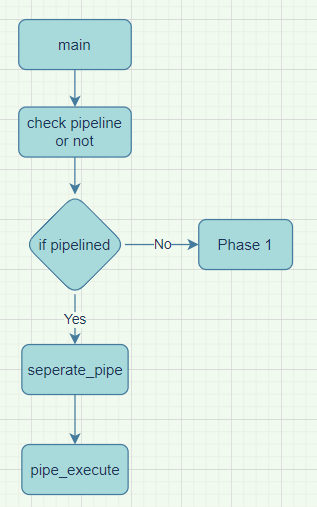
* 1. **개발 방법**
* **프로젝트에 공개된 csapp.c csapp.h에서 필요하다고 느끼는 함수만 헤더 파일에 추가했다.**
* **Phase1의 경우 주어진 shellex.c 소스코드에서 명령어의 실행을 fork()를 통해 하도록 수정했다. Execve 함수는 Execvp 함수로 수정하여 내부적으로 execvp를 호출하여 명령어를 수행하게 수정했다. Signal handling을 원하는 방식으로 하기 위해서 SIGCHLD, SIGINT 시그널에 대한 handler 함수를 구현했다.**
* **Phase2에서는 Phase1의 코드를 기반으로 작성했다. Pipelined command를 처리하기 위해 separate\_pipe, pipe\_execute 함수를 추가했고 myshell\_execute 함수에도 pipe인지 아닌지 검사하는 내용을 추가했다. Separate\_pipe는 ‘|’가 존재할 때 명령어를 파이프를 기준으로 나누는 함수이고, pipe\_execute 함수는 pipelined command를 pipe를 열어 Dup2 함수를 통해 자식의 표준 출력을 pipe로 부모의 표준 입력을 pipe로 바꾸어 inter-process commnunication을 구현했다.**
* **Phase3에서는 기존에 구현했던 myshell\_execute 함수를 수정하여 background processing과 job에 관련된 built-in 명령어들을 구현했다.**
* **부모 프로세스는 fork 전까지 sigchld, sigtstp를 block하게 했고 자식 process는 sigchld만 block 하게 했다. 이후에 실행한 명령어를 job 배열에 추가하고 foreground이면 signal unblock을 하여 reap을 하고 background이면 return을 하여 reap하지 않도록 했다. 처음에 sigsuspend를 이용했는데 sigsuspend를 이용했을 때 계속 하나 이상의 명령어 reap을 놓쳐서 wait\_fg\_Reap이라는 foreground job을 reap하는 함수를 작성했다.**
* **자식이 실행되는 과정에 sleep(1)을 추가했는데 ctrl-z 키보드 입력을 조금 더 직관적으로 받고 실행하기 위해 추가했고 부모 프로세스에서도 어느 정도 실행 속도를 맞춰주려고 sleep(1)을 추가했다.**
* **Jobs, kill, fg, bg는 built-in command로 작성했다. Jobs를 입력하면 현재 job 배열에 등록되어 있는 모든 job을 출력한다.**
* **kill을 했을 때 signal이 block되어 해당 프로세스가 reap되지 않아서 (fork이전에 signal을 block하여 sigchld가 원하지 않는 타이밍에 오지 않게 했다.) kill 명령어 수행 시에 9 값을 return하게 하여서 9가 반환되는 순간에 시그널을 unblock하여 sigchld를 기다리고 따라서 kill을 하자마자 해당 프로세스가 reap되게 작성했다.**
* **Kill, fg, bg 명령어는 job id, process id 둘 다 인자로 받을 수 있게 구분했다. Kill 명령어는 입력되면 해당 프로세스에 9번 signal(SIGKILL)을 보내주었다. Fg 명령어는 입력된 프로세스(또는 job)의 상태를 FG로 바꾸어주고 sigcont 신호를 보내주어 다시 실행시킨다. Bg 명령어는 입력된 프로세스(또는 job)의 상태를 BG로 바꾸어주고 sigcont 신호를 보내주어 background에서 실행시킨다.**

1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

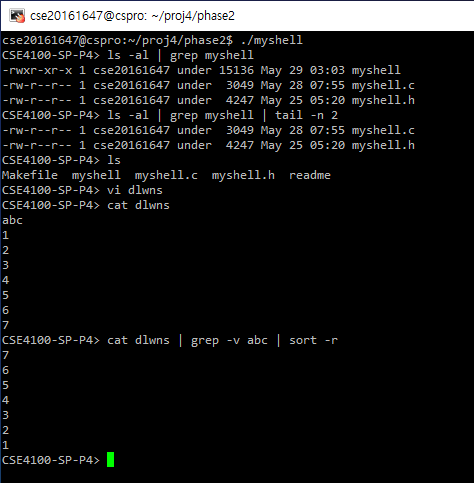
****



1. **Phase 2 (pipeline)**

****

**// pipe\_execute**



1. **Phase 3 (background)**

