**Multicore Programming Project 1**

담당 교수 : 박성용

이름 : 장준영

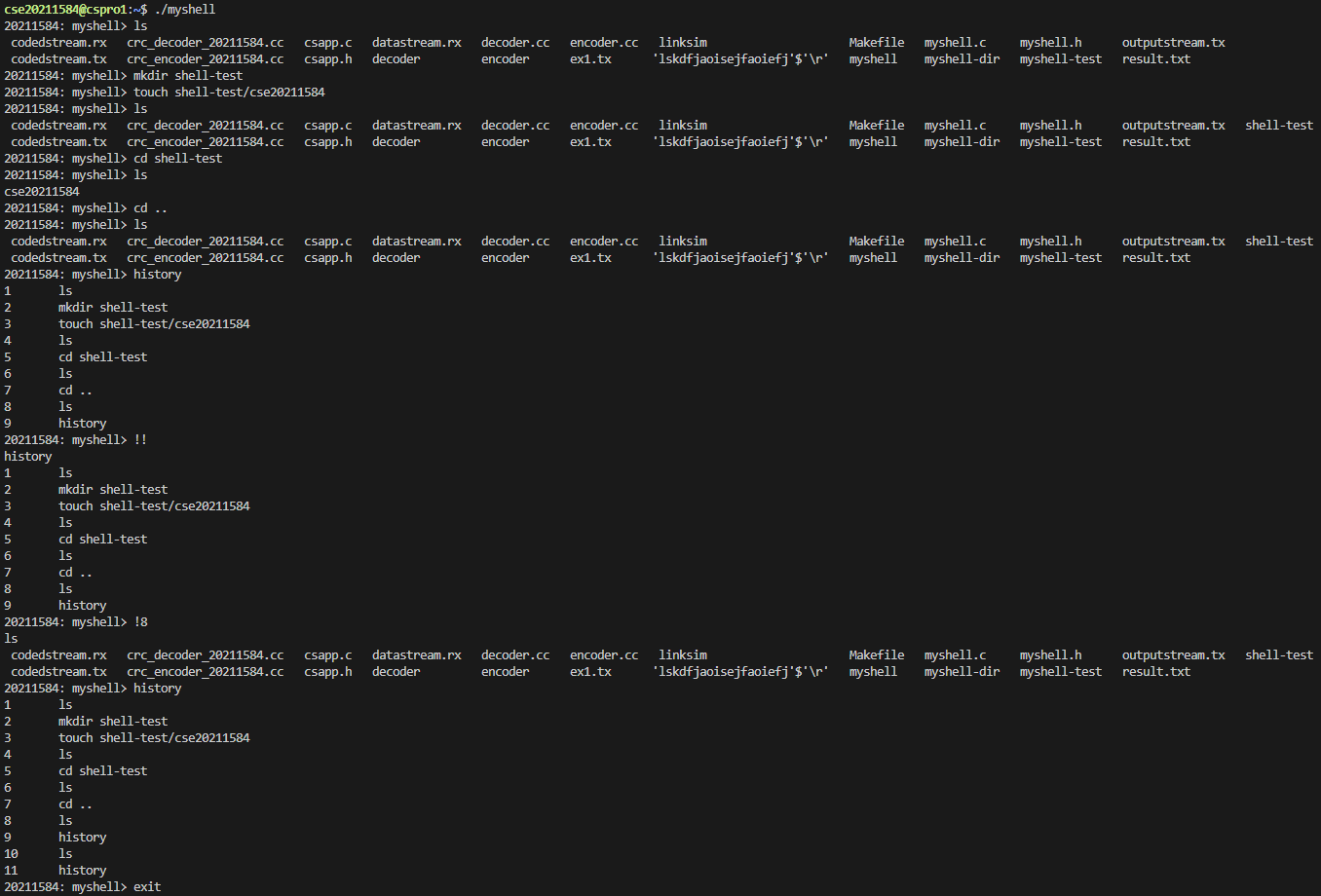
학번 : 20211584

1. **개발 목표**

해당 프로젝트에서는 Linux Shell을 C 코드로 직접 구현한다. Shell에선 유저가 입력한 command line을 읽어 해석하고 프로그램을 실행시킨다. Shell 자체의 process는 계속해서 작동하면서, 유저가 입력하는 명령에 대한 process를 다중적으로 진행해야 하므로 시스템 레벨에서의 프로그래밍이 필요하다. 이를 위해 1)사용자가 입력한 command line을 해석하기 좋게 만드는 parsing, 2)멀티코어프로그래밍 수업시간에 배운 fork, exec 등의 시스템 레벨 함수, 3)여러 process가 충돌하지 않고 잘 돌아갈 수 있도록 해주는 signal 설정 등이 필요하다. 추가적으로 pipe 동작을 가능하게 하는 dup2 함수나, background에서 실행되는 process를 효율적으로 관리하기 위한 구조체 리스트 등을 통해 Linux Shell과 거의 유사하게 작동하는 myshell을 구현해볼 것이다.

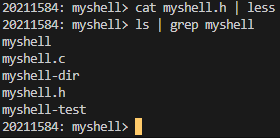
1. **개발 범위 및 내용**
   1. **개발 범위**
2. Phase 1

cd, ls, mkdir, rmdir, touch, cat, echo, exit같은 Linux Shell의 기본적인 명령어를 fork와 exec 등의 함수를 통해 작동 가능하게 만들었다. 또, history, !!, !# 명령어도 작동이 가능하다.



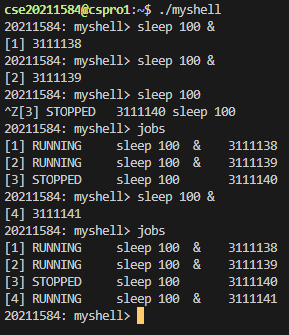
1. Phase 2

pipe 명령을 통한 redirection을 dup2 함수로 구현하였다.



1. Phase 3

Job 구조체를 통해 background process를 관리하도록 구현하였다.



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

유저가 입력한 cmdline을 parsing하여 실행해야 하는 command를 인식하고, builtin command인 경우 동작을 실행하고 프로세스를 중단한다. builtin command가 아닌 경우 fork를 통해 명령을 실행할 child process를 만들고 parent process가 reaping해주어야 한다. if((pid=Fork())==0) 조건문을 통해, Fork의 return 값이 0인 child process를 생성하여 명령어를 Exec 해준다.

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

위와 같은 방법의 else 문에선 parent process가 해야 할 동작이 설계되어있다. phase1에서는 fg process에 대한 reaping만 진행하므로, global variable인 volatile sig\_atomic\_t fg\_flag와 SIGCHLDhandler를 통해 구현하였다. child process가 어떠한 이유에서든 끝나서 SIGCHLD 시그널이 발생하면 fg\_flag가 1이 되고, while 문과 Sigsuspend를 통해 fg\_flag가 1이 될 때까지 기다리단 parent process가 child process를 reaping한다.

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

pipeline을 구현하기 위해선 parsing을 통해 pipe로 나뉜 cmdline을 자료구조에 각각 저장하고, 그 개수만큼 반복하여 input과 output을 redirection 해주어야 한다. eval과 parseline을 진행하면서 pipe가 있다면 cmdline을 pipe command array에 각각 저장하고, Exec 대신 ExecPipe 함수를 만들어 사용한다. ExecPipe 함수에선 pipe command array의 명령에 대해 반복을 진행하면서 dup2 함수와 fdpipe(file pointer를 저장하는 배열)로 각각을 redirection 해준다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

위의 설명과 동일하다. eval에서 pipe command의 경우 pipe command의 개수, pipe command array, pipe command의 시작점을 각각 저장한다. 이후 이 정보를 ExecPipe에 함께 전송하여 명령을 실행한다. redirection에 이용할 파일 포인터들도 fdpipe라는 배열에 각각 저장한다.

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

Background process를 위해선 생성된 process를 모두 관리할 수 있는 새로운 자료구조가 필요하다. pid, state(DONE, TERMINATED, STOPPED, BG RUNNING, FG RUNNING, …), cmdline 등을 저장하는 Job structure를 새로 선언하여 Job linked list로 process를 관리하였다. 이에 따라 linked list에 process 정보를 삽입하기 위한 함수들도 구현이 필요하다.

* 1. **개발 방법**

- 기본 함수(eval, parseline, builtin\_command, cmd-, cmdnormalize …)

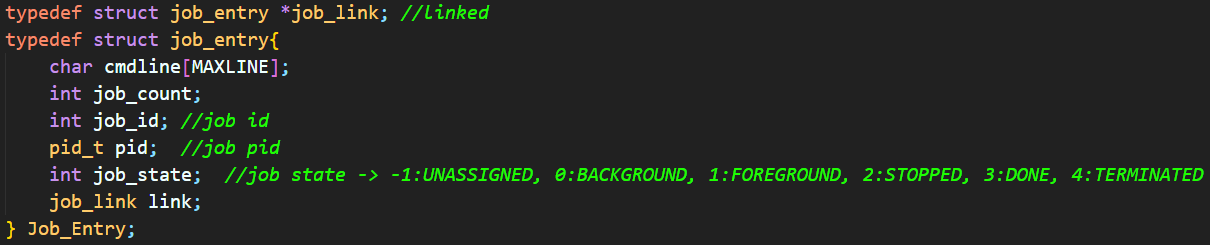
eval 함수는 기본 형식을 따르지만, history, pipe, background process 등의 구간을 확실히 나누기 위해 cmdnormalize 함수와 분기를 추가하였다. cmdnormalize는 주어진 명령어를 argv마다 스페이스로 확실히 구분할 수 있도록 cmdline을 최적화해주는 함수이다. 먼저, history를 builtin\_command에 추가하였고, cmdline의 0번 index가 ‘!’인 경우 !! 혹은 !# 명령어로 판단하여 history에서 알맞은 명령을 cmdline으로 설정하도록 하였다. ‘|’를 찾아 pipe command를 위한 정보를 저장하는 과정이 추가되었다. 또, background process를 고려해야 하므로 fork 이후 모든 process를 Job linked list에 삽입하여 일괄적으로 처리하였다. parseline에서는 마지막의 &을 찾아 bg인지 아닌지를 return하고, history 기능을 위해 전역 변수 char\*\* history를 선언해 지금까지 입력된 모든 명령을 저장하도록 하였다.

- 시그널 관련 함수(siginit, SIGINThandler, SIGSTPhandler, SIGCHLDhandler, SIGCHLDpipehandler, …)

process를 Job list에서 일괄적으로 처리하기 때문에, foreground에서 작동하는 SIGINT, SIGTSTP도 Job list를 참조하여 foreground process를 참조하도록 변경해야 한다. 따라서 handler 함수를 새로 작성하였다. SIGCHLD 또한 Job list를 참조해야 하는데, child process가 SIGCHLD를 보내는 경우는 foreground process와 background process 모두 해당하므로 조건 분기를 통해 처리해준다. pipe에선 ExecPipe 함수를 따로 사용하므로 parent process가 reaping할 수 있도록 pipe\_flag와 SIGCHLDpipehandler를 구현하였다. 추가적으로 함수 내 모든 시그널 과정에서 시그널이 필요할 때 열리고 닫혀야 하므로, sigset\_t로 선언된 네 개의 비트 벡터와 Sig- 함수로 조절해야 한다.

- Background process 관련 함수(Job 구조체 list, Job 관련 함수, printreaped, cmdprintjobs…)

앞서 이야기한 것처럼 Job 구조체는 다음과 같이 이루어져야 한다.

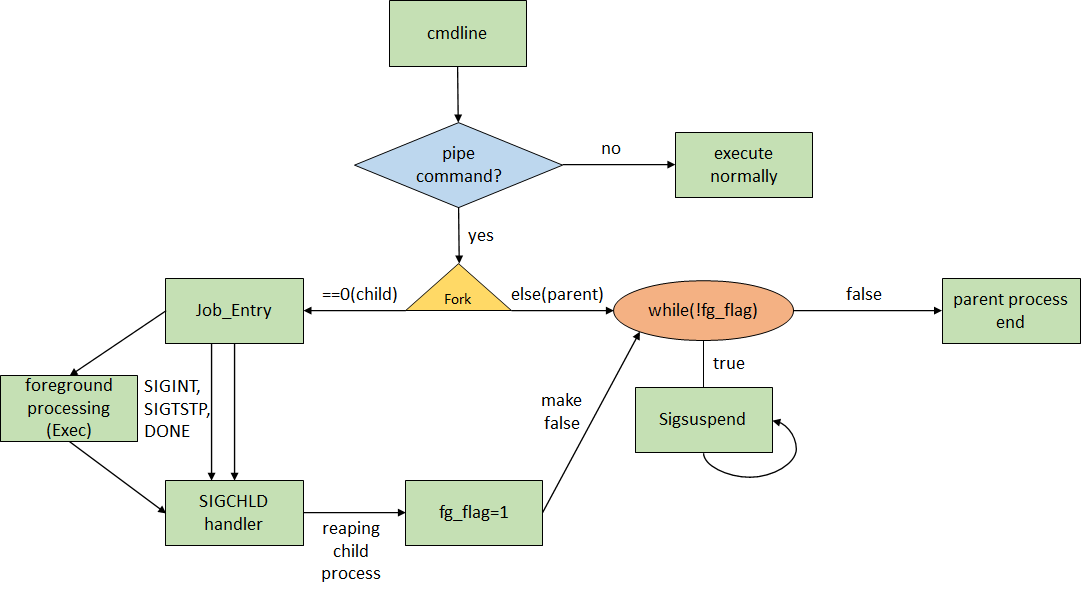


process가 생성되어서 명령을 수행하고 마지막에 수확될 때까지 필요한 모든 정보를 입력한다. jobadd는 새로운 process를 Job 구조체에 넣어 return한다. jobdelete는 job\_id를 받아 해당하는 job을 리스트에서 삭제한다. jobgetid는 pid에 맞는 job\_id를 return한다. jobgetnextid는 jobadd나 SIGCHLDhandler에서 사용할 수 있도록 비어있는 다음 id를 return한다. jobget은 pid에 해당하는 Job을 return한다. printreaped는 job list에서 state가 terminated이거나 done인 process를 reaping하고 터미널에 print하는 함수이다. 프롬프트가 입력될 때마다 출력되도록 main 함수의 while에서 호출된다. 명령어로 jobs를 입력하면 현재 실행중인 process의 상태, pid, 실행 순서를 print해준다. builtin\_command에 jobs를 넣어 실행되게 하였다.

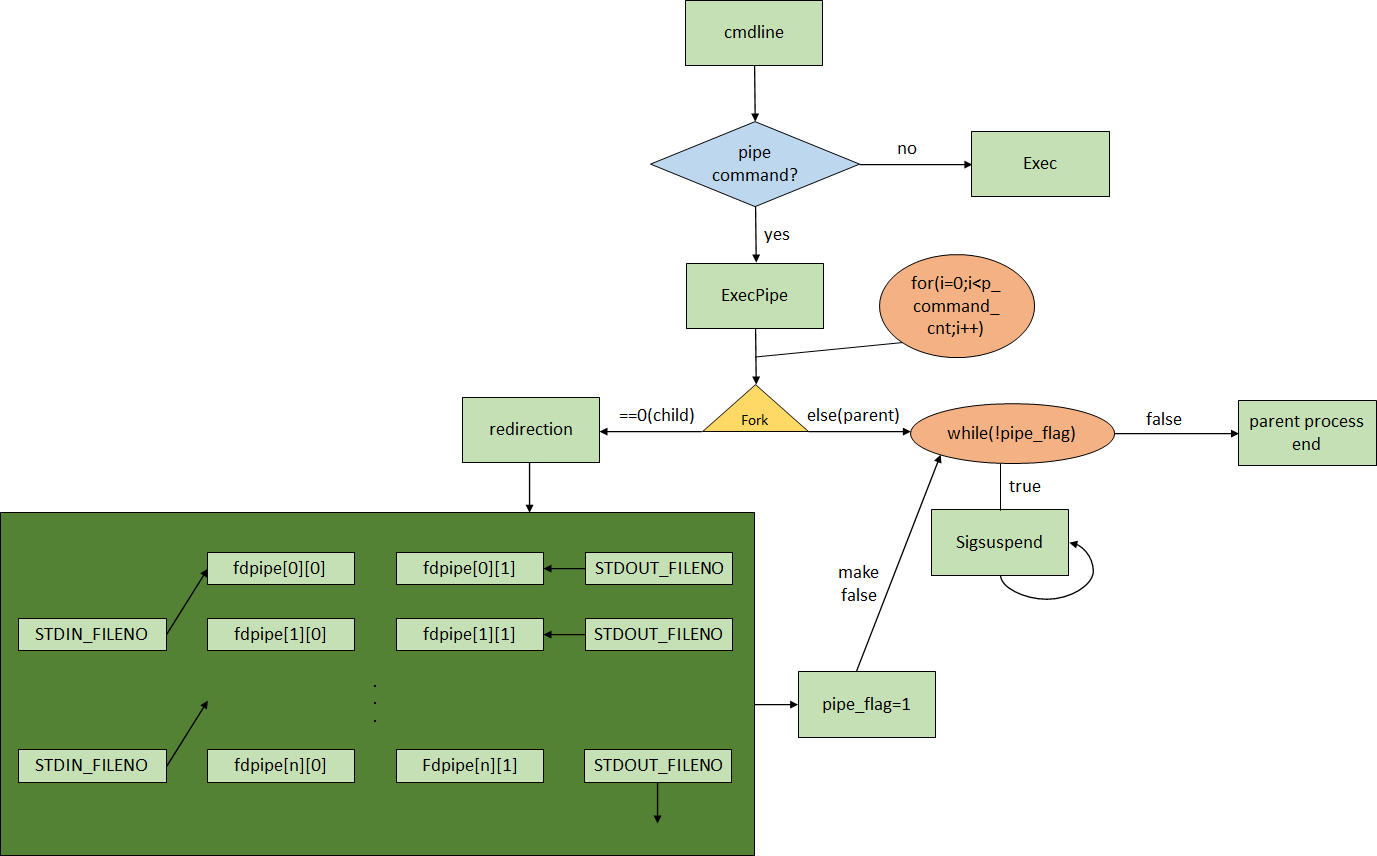
- ExecPipe

pipe 명령어를 수행하는 독립적인 함수인 ExecPipe에서는 pipe로 나뉜 각각의 명령어마다 존재하는 file descriptor를 저장하는 int fdpipe[MAXARGS][2]를 사용한다. 반복문을 통해 pipe 명령어의 개수만큼 반복하여 redirection 하는 과정을 거친다. 그리고 그렇게 생성된 각각의 명령어를 Fork하고 Exec한다.

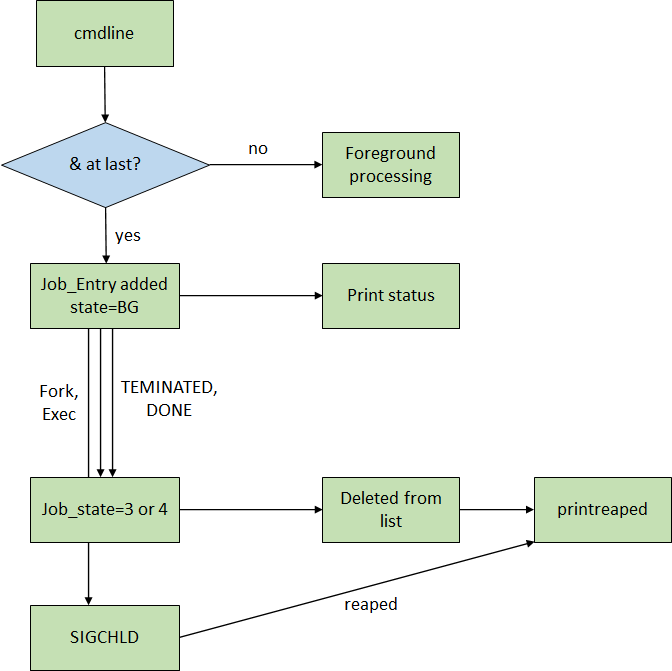
1. **구현 결과**
2. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****