**System Programming Project 1**

담당 교수 : 김영재 교수님

이름 : 안도현

학번 : 20181650

1. **개발 목표**

* **이 프로젝트에서는 CSPRO 환경에서 기존에 사용하던 쉘의 기능을 일부 도입한 MyShell을 직접 구현하여 돌려봄으로써 SIGNAL 및 각종 SYSTEM CALL 등을 숙달한다. 주요 기능에는 exec로 프로그램 실행하기, pipe 구현, 프로세스의 백그라운드 실행이 있다.**
* **이 프로젝트는 총 3단계로 나뉘어져 있다. 페이즈 1에서는 단순히 커맨드 라인으로부터 입력받은 것을 처리해 실행하는 것이므로 fork()를 사용해 자식 프로세스와 부모 프로세스 간 적절한 상호작용을 통해 구현한다. 페이즈 2에서는 pipe를 구현하는 것인데, pipe()와 dup2()를 통해 파일 디스크립터를 적절히 분배해 각 프로세스 간 IPC를 구현하는 것이 핵심이었다. 페이즈 3에서는 커맨드 라인 끝에 &로 입력받은 것을 백그라운드로 실행하여 포어그라운드에서는 다시 커맨드를 받거나 다른 프로세스를 진행하는 식으로 구현해야 했는데, 결국 fork로 자식 프로세스를 적절한 위치에 사용하면서 구현할 수 있고, 추가적으로 fg,bg,kill,jobs는 builtin command로 추가하면 될 것이다.**

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

myshell을 실행한 뒤 builtin command로 작성된 cd를 포함해 ls,pwd,cat,touch 등 /bin, /usr/bin 디렉토리 내에 존재하는 커맨드들을 실행할 수 있게 한다. 한 명령어를 처리한 뒤 다른 커맨드 라인을 다시금 입력받을 수 있게 된다.

1. Phase 2

페이즈 1의 기능에 더하여, pipe 기호(|)를 각 명령어 간에 삽입하여 이전 명령어의 출력이 다음 명령어의 입력으로서 주어지도록 한다. 예를 들어 현재 작업 디렉토리의 파일 및 디렉토리 정보를 숨김없이 디테일하게 출력하는 ls -al에 |를 붙인 뒤 어떤 문자열이 포함된 라인을 찾는 grep myshell을 쓰고 엔터를 치면(ls -al | grep myshell), 현 디렉토리의 파일들 중 myshell이라는 문자열이 정보에 포함된 것들에 대한 라인들이 출력되게 된다.

1. Phase 3

페이즈 1,2의 기능에 더하여, 커맨드 라인에 &를 붙임으로서 프로세스를 백그라운드로 시작할 수 있도록 한다. ctrl+c를 누르면 포어그라운드 프로세스가 종료되고 ctrl+z를 누르면 포어그라운드 프로세스가 정지된 백그라운드 프로세스가 되도록 하는 기능도 구현한다. 빌트인 커맨드로서 fg는 백그라운드 프로세스를 포어그라운드로 전환하며, bg는 정지된 프로세스를 백그라운드에서 진행시키며, jobs는 모든 백그라운드 프로세스를 인덱스와 함께 출력해주며 kill은 프로세스를 종료시킨다.

* 1. **개발 내용**
* **Phase1 (fork & signal)**
  + eval 함수 내에서 파싱된 라인을 바탕으로 빌트인 명령어인지를 검사 후 if문 내부로 진입한다. (pid=fork())==0일 경우 진행하는 구문은 자식 프로세스가 따라가는 코드이므로 이 부분에 execve를 사용해 명령어를 처리할 수 있도록 한다. execve는 정상적으로 실행될 경우 리턴되지 않고 프로세스를 종료하므로 이후 코드는 부모 프로세스가 진행하는 부분이다.
  + 그래서 자식 프로세스가 execve와 함께 종료되면 부모 프로세스는 SIGCHLD signal을 받으며 wait()또는 waitpid()를 사용하여 정상적으로 자식 프로세스를 reap하여 좀비 프로세스가 되는 것을 미리 막을 수 있다. wait()를 통과하면 다시 메인 함수로 리턴되어 새로운 커맨드 라인을 받는다.
* **Phase2 (pipelining)**
  + 커맨드 라인을 입력받은 뒤 빌트인 커맨드가 아니고 빈 라인이 아니라는 것이 확인된 이후에, |를 기준으로 명령어를 comlist에 분리해 저장한다. 페이즈 1에서는 eval함수 내에서 fork를 사용했던 반면 페이즈 2에서는 exec\_pipe함수를 별도로 만들어 이 함수 내에서 fork를 수행한다. 크기가 2\*sizeof(int)인 int 배열을 pipe()함수의 인자로 주고 실행시키면 부모 프로세스와 자식 프로세스 간 통신에 사용되는 파일 디스크립터가 할당된다. 그리고 fork를 호출하는데, 기본적으로 여기서 부모 프로세스는 파일 디스크립터를 적절히 관리하고 자식 프로세스를 wait하며, 최종적으로 모든 파일 디스크립터를 닫는 역할까지 수행한다. 자식 프로세스에서는 dup2를 통해 표준 입력으로 받을 것을 파일 디스크립터로 바꾸거나 표준 출력으로 결과를 내보내는 execve의 출력을 다른 파일 디스크립터로 지정하는 과정 이후 적절히 파일 디스크립터를 닫고 프로그램을 실행한다.
  + pipe 사용은 다중 연결이 가능하도록 설계되었기 때문에 여기서 이 exec\_pipe함수를 부모 프로세스가 재귀적으로 호출한다. 자식 프로세스가 적절한 파일 디스크립터에 내용을 출력하고 종료하면 부모는 그 시그널을 받아 자식 프로세스를 정상적으로 종료시킨 뒤, 이 파일 디스크립터를 재귀함수 호출의 인자로 넘겨 다음 명령어가 이전 명령어로부터 적절한 내용을 전달받을 수 있도록 설계했다.
* **Phase3 (background process)**
  + 커맨드 라인으로부터 &를 인식해 bgflag를 set하고 백그라운드 프로세스로 실행한다. 이 경우 기존에 돌아가던 부모 프로세스에서 fork()를 띄워 자식 프로세스에서 exec\_pipe함수를 호출하고, 부모 프로세스는 백그라운드 프로세스들의 목록을 저장할 적절한 자료구조를 생성해 그것을 관리한 뒤 다시 커맨드 라인을 입력받는다. 자식 프로세스는 내부에서 또다시 fork()를 통해 새로운 프로세스를 만들어 그 자식 프로세스에서 execve를 수행해 프로그램을 실행하고 자신은 waitpid를 사용해 타겟 명령어가 종료될 때까지 대기하는 역할을 한다.
  + SIGINT와 SIGTSTP에 대한 handler를 추가로 정의해 현재 포어그라운드 프로세스를 종료시키거나 중단시키는 역할을 하도록 한다.
  + built-in command로서 fg,bg,jobs,kill이 존재한다. kill은 타겟 프로세스가 정지된 상태면 다시 진행시킨 뒤 SIGKILL 시그널로 종료시킨다. jobs는 부모 프로세스에서 지속적으로 관리한 백그라운드 프로세스를 저장한 것들을 모두 출력하도록 한다. fg와 bg는 타겟 프로세스에 SIGCONT를 날려 다시 진행시키게 하는데, fg에서는 추가적으로 wait\_proc을 호출해 해당 프로세스가 다시금 중단되거나 끝나기를 기다리도록 함으로써 포어그라운드에서 해당 프로세스가 진행되는 것으로 만든다.
  1. **개발 방법**

모든 기능이 전부 구현된 페이즈 3의 코드를 기준으로 설명하면, 이 프로젝트는 단순한 기능을 작은 함수로 분리해놓은 부분과 중요한 기능을 하는 함수 몇 가지로 이루어져 있고, 이 코드의 핵심적인 변수 및 함수의 구조를 코드의 위에서부터 살펴보겠다.

우선 jobs 명령어를 사용할 때 백그라운드 프로세스를 표시해야 하므로 이 프로세스를 관리할 타입으로서 jobtype 구조체를 정의한다. 구조체가 num,pid,state,command string을 저장하는데, 여기서 num은 job list의 인덱스이자 프로세스의 생존 여부를 나타내는 변수이다(0일 경우 종료된 프로세스). jobs array로 프로세스를 저장하고, 같은 인덱스의 c\_procs는 백그라운드로 시작한 프로세스의 종료를 기다리고 있는 waiting process의 pid를 저장한다. cur\_pid는 현재 포어그라운드에서 진행중인 프로세스의 pid로서 signal handler가 호출되었을때 사용되는 전역변수이다.

메인 함수에서는 기본적으로 ctrl+z ctrl+c를 blocking한다. 그리고 jobs array를 초기화한 뒤 무한루프인 while을 통해 표준 입력으로부터 커맨드 라인을 입력받는다. 입력받은 라인은 eval로 전달된다.

sigint\_handler와 sigtstp\_handler는 ctrl+c, ctrl+z가 입력되었을 때 cur\_pid가 가리키는 프로세스에 각각 SIGKILL과 SIGSTOP을 보낸다. sigint쪽은 추가적으로 해당 job의 num을 0으로 바꾸어 해당 프로세스의 종료를 관리하고, sigstop쪽은 같이 멈춰있던 waiting process또한 같이 진행해주는 방식이다.

eval 함수에서는 우선 각 waiting process를 WNOHANG 옵션과 함께 wait한다. 그렇게 함으로써 백그라운드로 돌아가던 프로세스들이 종료되었다면 그것을 적절히 reap해주게 된다. 이후 커맨드 라인에 &가 포함되어 있다면 bgflag를 set하고, '|'를 기준으로 커맨드를 분리해주는 devideCommand함수를 호출한 뒤 분리된 커맨드를 파싱한 뒤, 그것이 빈 라인이거나 빌트인 명령어인지를 검사한다. 빌트인 명령어라면 적절한 수행 뒤 메인으로 리턴되고, 아닌 경우에는 exec\_pipe 함수를 호출한다. 문제는 bgflag가 set 되어 있는 경우인데, 이 때는 프로세스를 백그라운드로 처리하도록 해야 하므로 fork()를 한번 call한다. 부모 프로세스라면 jobs에 백그라운드로 돌아갈 자식 프로세스의 정보를 저장하고 다시 메인으로 돌아가 입력을 받는다.

exec\_pipe는 실제로 '|' 기호가 포함되어 있는지와는 관계없이 명령어를 처리할 때 무조건 호출되는 함수이다. 명령어는 fork()가 호출된 뒤 자식 프로세스에서 execve를 쓰게 되는 별도의 함수인 exec\_com를 호출해 돌아가며, 부모 프로세스는 적절한 파일 디스크립터 관리와 waiting 및 재귀 호출을 진행한다. 크기가 2인 int 배열을 선언한 뒤 pipe()를 호출해 부모 프로세스와 자식 프로세스 간의 통신을 가능하게 하며, 파이프 기호가 여러개일 때 함수 호출의 인자로 fd를 넘겨 이전 자식 프로세스에서 쓴 내용을 다음 자식 프로세스가 받을 수 있도록 했다. 각 명령어는 표준 입력으로부터 받아 표준 출력으로 내보내는 경우가 대부분이기 때문에 dup2를 호출해 표준 입출력과 각 파일 디스크립터를 적절히 mapping해주는 작업도 필요했다.

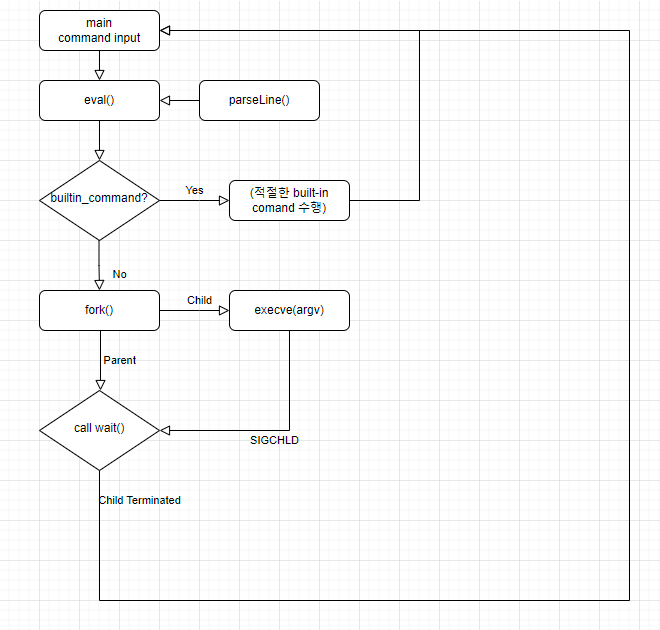
wait\_proc은 백그라운드 프로세스 관리의 핵심이다. 우선 blocking 해두었던 SIGINT와 SIGTSTP를 위에서 정의한 각 핸들러로 연결해준다. 그리고 인자로 받은 pid를 전역변수인 cur\_pid로 할당하고 처음 백그라운드로 시작했는지 또는 포어그라운드에서 중단된 지를 구분하는 flag에 따라 waiting process의 종료 또는 실행 프로세스 자체의 종료를 wait한다. 물론 SIGTSTP로 중단될 수 있기 때문에 WUNTRACED 옵션을 인자로 전달한다. 중단되어 wait를 탈출하면 jobs에서 그 상태를 적절히 바꿔준다. 포어그라운드에서 실행되다 처음으로 중단된 프로세스의 경우 jobs에 그 프로세스가 등록되어 있지 않으므로 새로 추가해준다.

마지막으로 builtin\_command 함수는 각 빌트인 함수를 처리한다. exit는 부모 프로세스의 종료, 즉 쉘의 종료를 진행한다. jobs는 jobs array에 저장된 jobs들을 for loop로 훑으며 출력하는데, num이 0인 경우 출력하지 않는다. kill은 해당 프로세스를 num으로 탐색한 뒤 0이 아니라면 SIGKILL을 전달한다. fg와 bg는 각 프로세스에 SIGCONT를 전달하고 fg의 경우 추가로 wait하여 포어그라운드 상에서 진행될 수 있도록 했다.

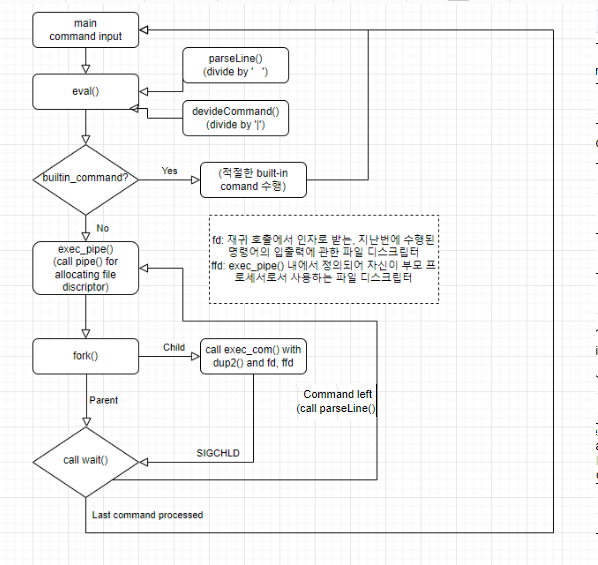
1. **구현 결과**

**Flow Chart**

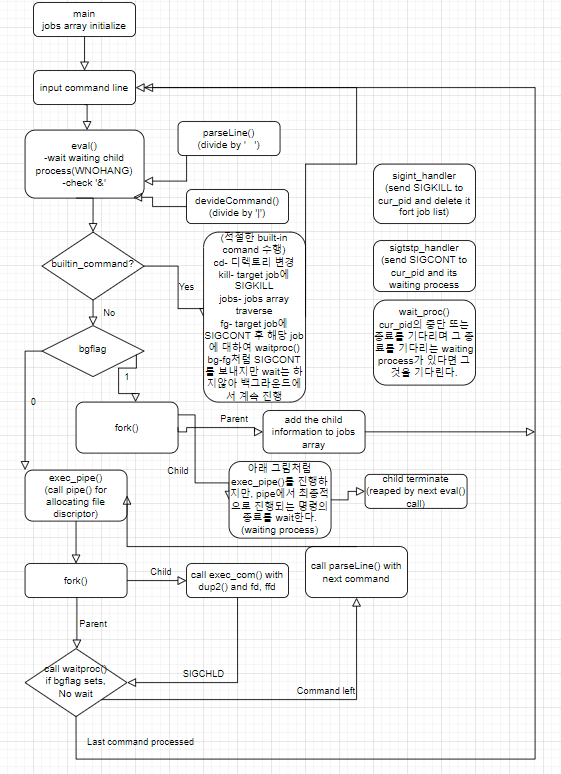
1. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****