**System Programming Project 1**

담당 교수 : 김영재

이름 : 박성현

학번 : 20181632

1. **개발 목표**

개발 환경 : Intel® Core™ i7-8550U CPU @ 1.80GHz 16.0GB RAM Windows 10

이번 프로젝트에서는 my own shell을 만들어본다. Shell은 Linux에서 명령어와 프로그램을 실행시키기 위해 사용하는 인터페이스이며 여러 프로세스들을 foreground와 background로 실행시킬 수 있다.

Phase 1에서는 기본적인 shell 명령어들을 구현하고 실행시킨다. Parents와 child process를 사용하여 기본적인 단일 명령어를 실행시킨다.

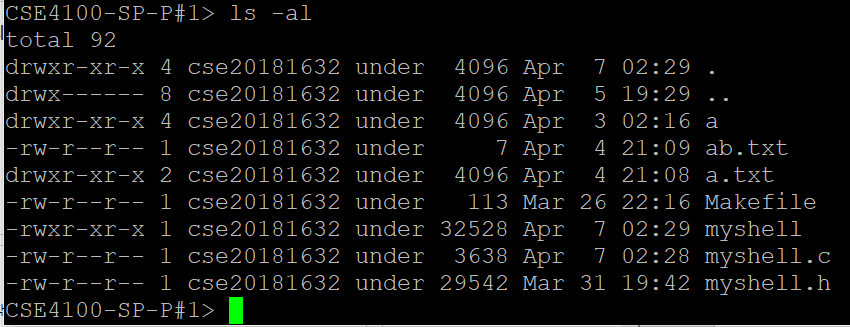
Phase 2에서는 추가적으로 ls | grep “hello”같은 pipe를 이용해 여러 개의 명령어를 한번에 실행시킬 수 있게 구현한다. Phase 1에서와 마찬가지로 parents와 child process를 적절히 사용해 구현할 수 있다..

Phase 3에서는 phase 2에 추가적으로 &기호를 사용하여 process를 background로 실행시키는 기능을 추가로 구현한다. fg%, bg%, kill% , jobs명령어들을 추가로 구현한다. process들의 signal들을 handling하여 phase 3을 완성한다.

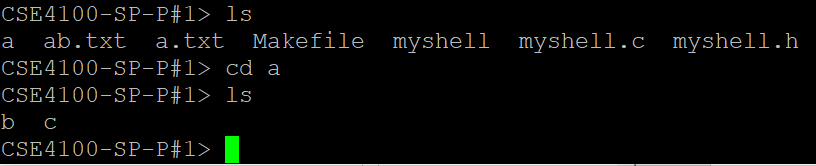
**개발 범위 및 내용**

* 1. **개발 범위**
* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. **Phase 1**

Phase 1을 수행했을 때, CSE4100-SP-P#1>라는 Prompt와 함께 프로그램이 시작된다. Ls -al과 cd, 그리고 exit의 구동결과를 예제로 첨부한다.

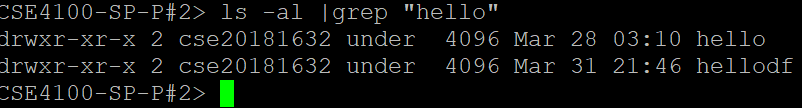
위 사진과 같이 기본적인 명령어를 실행하면 결과가 출력되고 다시 prompt가 출력된다.

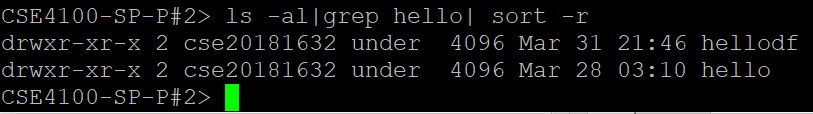
cd 명령어는 직접 구현했다.ls 명령어로 현재 directory의 파일들을 출력 후, a라는 directory로 cd를 이용해 이동한다. a라는 directory안의 또다른 directory가 출력된 것을 볼 수 있다.

C:\Users\User\AppData\Local\Packages\Microsoft.Office.Desktop_8wekyb3d8bbwe\AC\INetCache\Content.Word\p1-3.png

마지막으로 exit command 입력 시, 프로그램이 종료된 것을 볼 수 있다.

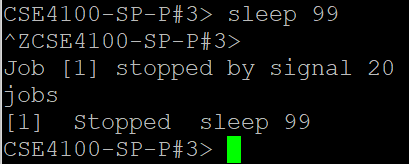
1. **Phase 2**

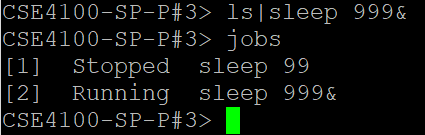
Phase 2 에서는 ls -al |grep hello 같이 pipe로 구성된 명령어들을 실행시킬 수 있게 한다. 예시로 ls -al |grep “hello”와 ls -al | grep hello | sort -r을 첨부한다.

Ls -al로 나온 결과를 grep “hello”의 input으로 넣어 hello가 들어간 파일 정보를 모두 출력했다. Ls -al의 결과는 stdout으로 출력되지 않고 pipe를 통해 grep의 input으로 사용되었다. “문자가 나타나도 “ “안의 문자열만 사용된 것을 볼 수 있다.

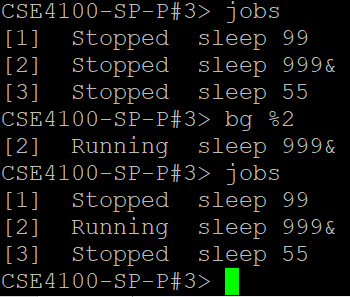
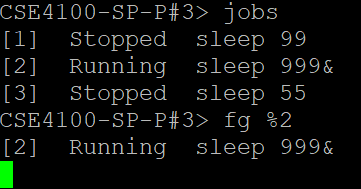
위 사진에서는 두개의 pipe를 사용했다. Ls -al의 결과를 grep hello의 input으로, grep hello는 ls -al의 output을 사용하여 작동하고 output을 또다시 sort -r에 전달한다. Sort -r은 grep hello의 output을 사용하여 결과를 출력한다. Sort -r은 거꾸로 정렬하라는 명령어이기에 이 전 예시의 출력결과와 반대인 것을 볼 수 있다.

1. **Phase 3**

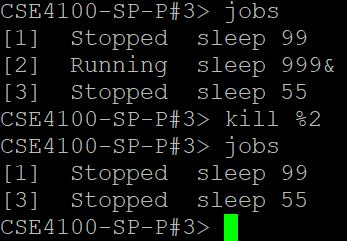
Phase 3에서는 &기호로 background에서 명령을 작동시키고 fg %로 지정된 job을 foreground에서 running 하게, bg %로 지정된 job을 background에서 running하게 만들어준다. 또한 crtl+z가 키보드 입력으로 주어졌을땐 현재 foreground에서 작동중인 job이 중지되며 crtl+c가 주어지면 현재 작동중인 foreground의 job이 terminate가 된다. Kill % 로 지정된 job을 강제 종료시킬 수 있다. Jobs를 입력하면 현재 background에서 멈추거나 작동중인 job들을 출력한다.

Crtl+z의 예시이다. foreground에서 작동중인 sleep 99를 crtl+z로 중지시키면 signal 20, 즉 SIGSTP로 인해 작업이 중지된다. 여기서 jobs 명령어로 job을 띄워보면 1번 job이 stopped된걸 볼 수 있다.

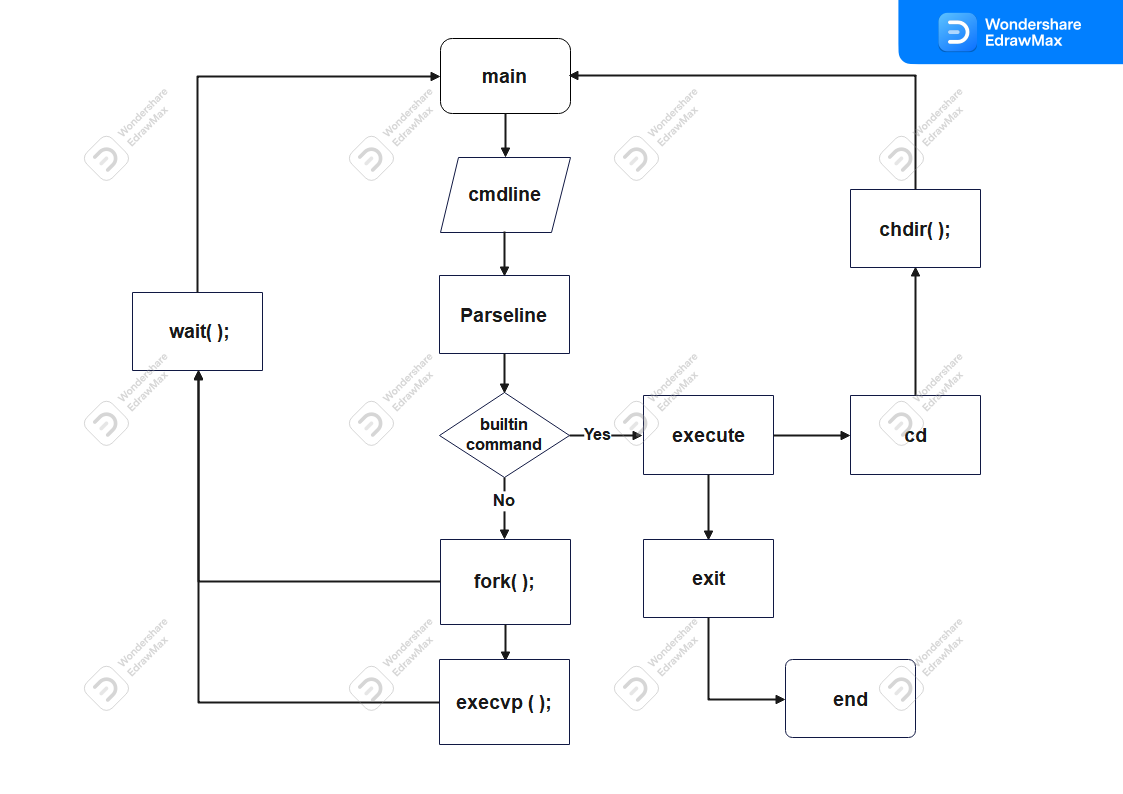
두번째는 background로 돌리는 예시이다. Pipe로 sleep 999를 background로 실행시켰다. 실행시켜도 program이 sleep 999를 기다리지않고 바로 prompt를 출력한다. Jobs를 입력해서 background에서 작동중인 job을 출력한 결과, 전 단계에서 stop 시킨 sleep 99가 1번, 그리고 현재 작동중인 sleep 999&가 background 에서 running 중인 것을 볼 수 있다.

다음은 fg%의 예시이다. Sleep 55라는 정지된 작업을 하나 더 추가시켰다. 그후 background에서 작동중인 2번 job을 fg로 foreground로 가져와서 작동시켰다. 그 결과 sleep 999가 종료되기 직전까지 프로그램이 waiting 하고 있는 것을 볼 수 있다. Fg는 running중인 background job뿐만 아니라 stopped된 background joba에 대해서도 작동한다.

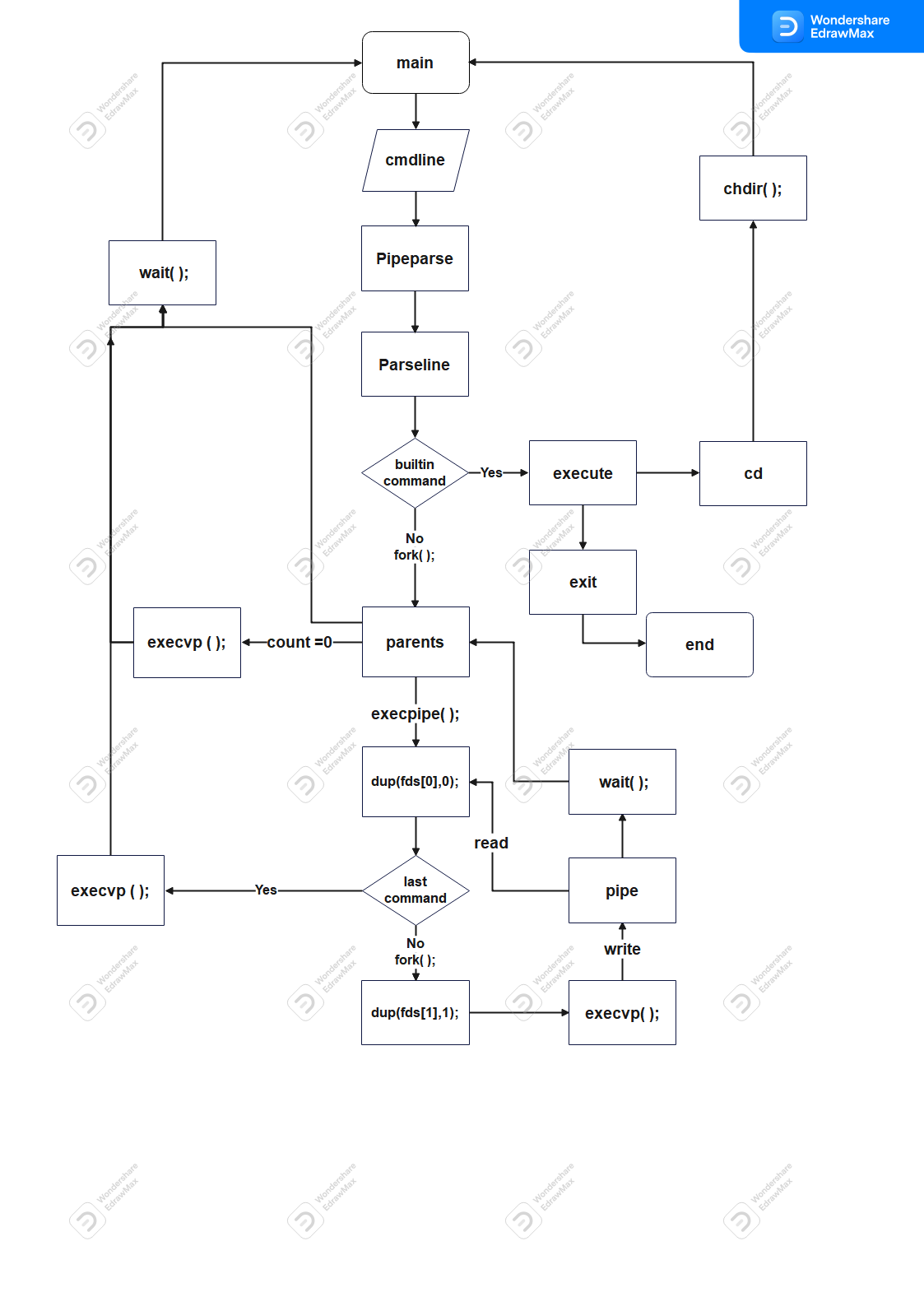
다음 예시는 bg이다. 이전에 정지된 2번 작업을 bg %2로 background에서 작동시켰다. 그 결과 background에서 running이 된다. background에서 작동중이므로 foreground에서는 다음 job을 받을 준비를 한다. Jobs로 띄워본 결과 2번은 background에서 running중이고 프로그램은 prompt를 띄워 새로운 job을 받을 준비를 한다.

마지막 kill예시이다. 이전의 예시에서 bg %2로 작동시킨 2번 job을 kill하였다. kill한 후, jobs를 통해 background의 작업들을 띄운 결과, 2번 job이 kill되서 사라진 것을 볼 수 있다.

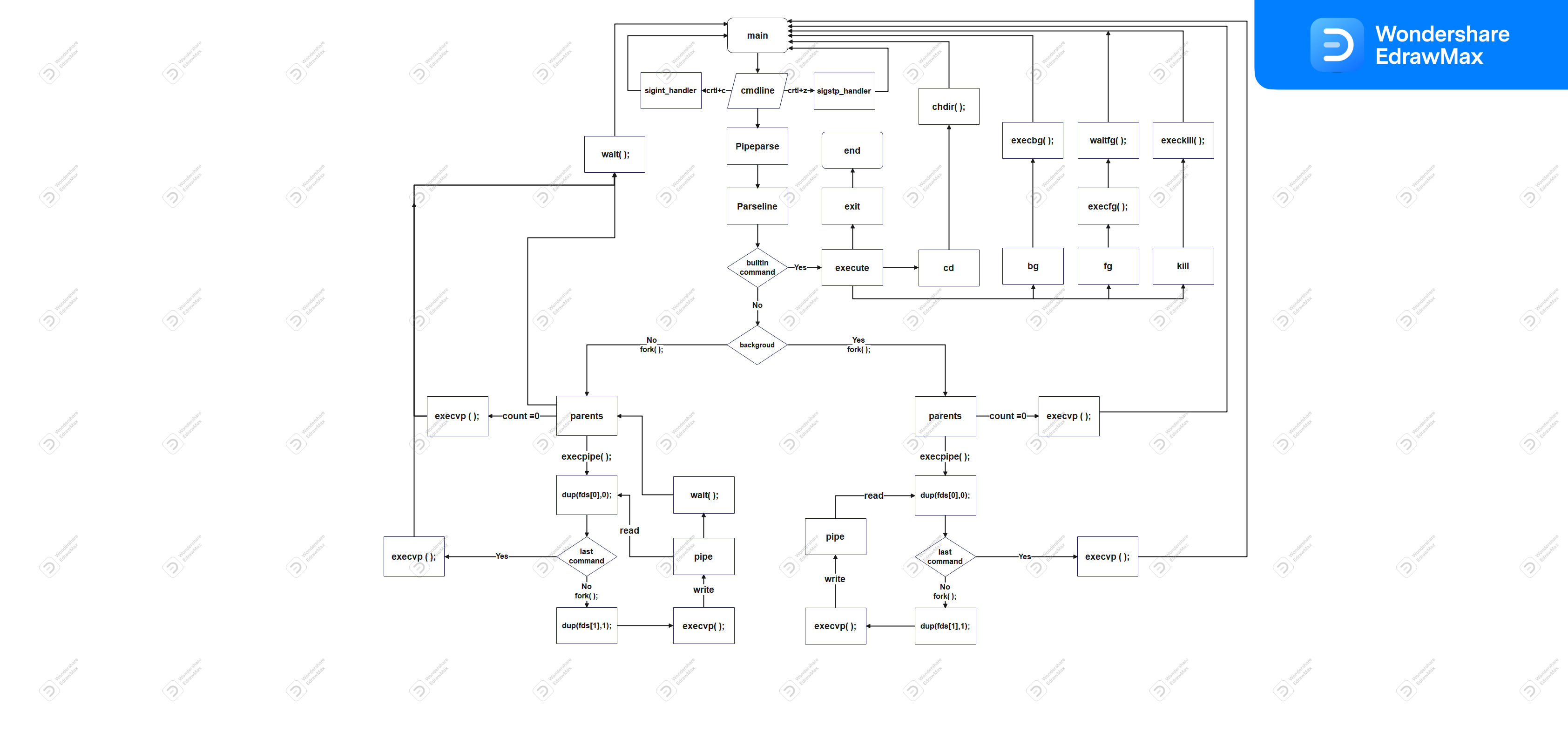
* 1. **개발 내용**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명
    - cmd로 받은 명령어를 공백으로 parsing한 값들은 argv배열에 들어있다. Argv의 첫번째 값, 즉 실행할 명령어가 cd, exit 등의 builtin command가 아닐 경우 하나의 child를 fork한다. Fork가 된 child에서 execvp를 이용해 argv의 명령어를 실행한다.
  + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow
    - child가 종료되면 parent에서 SIGCHLD signal을 보낸다. parents에서는 child가 종료될 때까지 waitpid함수로 기다리고 reaping해준다.
* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)
    - 새로운 pipeparse함수를 생성했다. pipeparse함수는 parseline함수와 동일하지만 parseline함수는 공백으로 나누는 반면 pipeparse함수는 | 기호를 기준으로 나눠준다.
    - eval함수에서 pipeparse를 실행한 후, count값으로 |의 개수가 저장된다.첫번째 명령어를 parseline후, bulitin command가 아닐 경우 child를 생성한다. Count가 0일경우 바로 실행한다. 아닐 경우 child에서 execpipe라는 함수를 count만큼 for문으로 실행시킨다. Execpipe는 인자로 pipeparse된 첫번째 문자열과 두번째 문자열, 그리고 for문의 i값이 넘어간다. execpipe에서 넘어온 첫번째 문자열을 공백으로 parseline해준다. 그후, 하나의 child를 더 만들어 pipe의 쓰기에 stdout을 할당하고 pipe의 reading은 닫아준다. 그후 명령어를 실행한다. 이로써 넘어온 첫번째 명령어의 결과 값은 pipe에 저장되어있다. parents에서는 reading을 해야하기에 pipe의 읽기에 stdin을 할당했다. Pipe의 쓰기는 사용하지 않으므로 닫아주었다. 만약 i의 값이 count-1,즉 두번째 명령어가 마지막 명령어일경우, 두번째 명령어를 parseline해주고 실행해준다. 이로써 parents는 pipe에 있던 값을 가지고 명령어를 실행후 stdout으로 출력할 수 있다. Parents는 wait함수로 child가 종료되길 기다리고 grandparents는 parents가 종료되길 기다린다.
  + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명
    - for문을 돌면서 execpipe를 실행시킨다. 넘어가는 명령어는 항상 두개이고 두번째 명령어가 마지막 명령어일 경우 실행을 parents에서 실행시키고 출력한다. 그렇지 않을 경우, 이 parent에서 fork된 child에서 실행시켜 값을 pipe에 넣어준다. 이때, parents는 pipe에서 값을 받아 그대로 child에게 다시 넘겨준다.
* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명
    - &입력을 받을 경우 parseline에서 return값, 즉 bg의 값이 1이된다. bg값이 1이면 child를 fork하기전에 SIGCHILD를 block시킨다. 이는 insert job을 하는 도중 child가 끝날 경우 job을 넣는 도중에 job을 지우는 경우가 발생할 수 있으므로 insert job이 끝나고 나서 unblock해준다. Background job이 아닐 경우 setpgid로 설정된 child의 값, 즉 같은 process group의 child가 종료될 때까지 waitpid함수로 기다린다. Background job의 경우 wait하지 않음으로써 다시 main의 while loop로 돌아가 다음 명령어를 기다린다.
    - Sigchld\_handler 함수에서 SIGCHLD를 받았을 때 경우에 따라 다른 action을 취해준다. 만약 signal에 의해 중지되었을 경우 waitpid의 option인 WUNTRACED 로 정지된 signal을 파악하여 WSTOPSIG로 그 signal의 번호를 출력한다. Background job이고 이 job이 정상적으로 종료되었을 경우 job이 종료되었다는 문구를 출력한다. Foreground의 경우 종료되었다는 것을 시각적으로 알 수 있기에 출력하지 않는다.
  1. **개발 방법**
* **Phase 1**
  + 예제코드 shellex.c에 추가 작성하였다. 첫번째로 buildin\_command 함수에 cd를 추가했다. Cd가 첫번째 argv의 값일 경우 chdir을 이용해 입력 받은 directory로 이동하게 구현했다. 아울러 exit command로 프로그램을 종료 시킨다.
  + parseline에서 “ 와 ‘를 handling해주었다. “ 나 ’ 가 있을 경우, 제거시킨 후 그 이후에 같은 “ 혹은 ‘ 가 나타나기 전의 모든 문자열을 argv에 저장했다. 이 작업을 수행하면 argv의 맨 앞에 있는 ” 혹은 ‘ 은 사라지지 않기 때문에 del\_colon이라는 함수로 지워주었다. 또한, cd가 command로 들어왔을 때, /로 directory들이 구분될 수 있기에 cd가 입력 값이라면 다음 문자열들을 /로 parsing해주는 부분도 추가했다. 기본 linux shell에서 cd a/ b를 입력하면 a로만 이동한다. 뒤에 공백이 존재하면 무시하기에 myshell 에서도 /이후에 공백이 있으면 뒤의 directory는 무시하게끔 구현했다.
  + del\_colon이라는 함수를 추가했다. 이는 가장 앞의 “ 혹은 ‘ 를 지워주는 함수로 문자열을 한 칸씩 당겨 가장 앞의 값을 지우고 가장 뒤에는 NULL 값을 넣어 중복된 문자를 지워주었다.
* **Phase 2**
  + Phase 1의 코드에 추가로 작성하였다. Pipeparse의 함수로 문자열을 먼저 | 로 나누었다. 이 문자열을 다시 parseline으로 나누어 실행 가능한 명령어로 만든다. pipeparse함수는 parseline과 동일하지만 공백으로 parsing하는대신 |로 parsing해준다. 또한 |의 개수를 count에 저장한다.
  + Eval 함수에서 count가 0일 경우, 즉 pipe가 존재하지 않는 경우엔 fork 후 바로 명령을 실행한다. 그렇지 않을 경우 pipe의 개수만큼 for문을 돌려 execpipe함수를 실행한다.
  + execpipe함수는 두개의 명령어와 for문의 i값을 인자로 받는다. 이 함수에서 공백을 기준으로 parsing해준 후, 다시한번 child를 생성한다. child에서는 값을 출력하는 대신 pipe에다가 써야 하므로 dup(fds[1],1)로 stdout을 pipe의 쓰기로 할당했다. Pipe의 읽기 기능은 쓰지 않으므로 닫았다. 이후 함수를 실행한다. 여기서 parent는 값을 pipe에서 읽어와야 하므로 dup(fds[0],0)으로 stdin을 pipe의 읽기로 할당하고 pipe의 쓰기는 닫았다. 이로써 child에서 실행된 값은 pipe에 써지고 parent에서는 값을 읽는다. 그리고 다시 실행되는 eval의 다음 for문에서 다시한번 생성된 child는 이 값을 읽고 다시 쓴다. 만약 i의 값이 count -1, 즉 두번째 명령어가 마지막 명령어일 경우 화면에 출력해야 한다. Parent는 stdout을 따로 pipe에 할당하지 않았으므로 i의 값이 count-1일 경우 두번째 명령어를 parent에서 실행시킨다. For문이 돌때마다 wait함수로 child를 reaping해준다.
* **Phase 3**
  + Phase 2의 코드에 추가 작성하였다.
  + Job을 저장할 구조체 배열을 생성했다. 구조체에는 pid, id, state, fb, cmd가 들어있다. Pid는 job의 pid, id는 job에 할당된 id, state는 running인지 stopped인지 구별하는 값이다. Fb는 0이면 foreground, 1이면 background임을 나타낸다. Cmd는 job의 명령어이다.
  + getjbid -> 이 함수는 id를 인자로 받아 일치하는 id를 가진 job구조체를 반환 시킨다.
  + Getjbpid -> 이 함수는 pid를 인자로 받아 일치하는 pid를 가진 job 구조체를 반환 시킨다.
  + Getidpid -> 이 함수는 pid를 인자로 받아 일치하는 pid를 가진 job의 id를 반환 시킨다.
  + Getfgpid -> 이 함수는 현재 foreground로 돌고있는 job의 pid를 반환 시킨다.
  + Waitfg -> 이 함수는 현재 running중인 foreground job이 종료될 때까지 기다린다.
  + Initjobtable ->이 함수는 job table에 초기값을 넣어준다.
  + Insertjb -> 이 함수는 job들을 table에 넣어준다. Background일 경우 id값이 증가하면서 들어가지만 foreground일 경우 id의 값은 항상 0이다.
  + Deljob -> 이 함수는 pid를 인자로 받아 그 job의 값들을 초기화 시킨다. Fb의 0과 1값은 사용되고 있기에 2로 초기화 시킨다.
  + Printjob -> 이 함수는 job table에 있는 background job을 모두 출력한다.
  + Prsinglejob -> 이 함수는 pid를 인자로 받아 그 pid를 가지고 있는 job만 출력한다.
  + Execfg -> fg %명령어를 실행한다. Background에서 멈추거나 작동중인 job을 foreground에서 실행되도록 바꾸는 명령어이다. Format에 맞지 않는 입력이 들어왔을 경우, message를 출력한다. 입력 받은 id를 getjbid로 job을 찾는다. 없으면 에러를 출력한다. Return된 값이 있을 경우, job을 foreground, running으로 바꾸고 prsinglejob으로 job을 출력한다. SIGCONT signal을 보내 job을 실행시킨다. Foreground로 실행되는 job이기 때문에 waitfg함수로 이 job이 끝날때까지 기다린다.
  + Execbg -> bg %명령어를 실행한다. background에서 멈춰 있는 job을 background에서 실행되도록 하는 명령어이다. Execfg와 id로 job을 찾는 부분까지는 동일하다. 만약 job을 찾았을 경우, background job으로 다시한번 설정해주고, running상태로 바꾼다. prsinglejob으로 job을 출력 후 SIGCONT signal 을 보내 재 가동시킨다. background이기에 기다릴 필요가 없으므로 waitfg 함수는 호출하지 않는다.
  + Execkill ->execbg와 execfg와 같다. 하지만 Job의 속성을 바꾸지 않고 SIGKILL signal만 보내 강제 종료시킨다.
  + Sigstp\_handler -> ctrl+z 키보드 입력을 받았을 때 실행된다. Getfgpid로 실행되고 있는 foreground job의 pid를 가져와 그 pid를 가진 job을 stop, background, 그리고 id를 설정해준다. Table에 없는 job은 새로운 id를 설정해주고, 이미 존재한다면 id는 그대로 둔다. SIGSTP signal을 job에 보내고 return한다. 실행중인 foreground job이 없다면 아무 작업도 실행하지 않고 return한다.
  + Sigchld\_handler -> SIGCHLD signal을 받았을 때 실행된다. Waitpid에 WNOHANG과 WUNTRACED 옵션을 주어 정지된 child로부터 어떤 signal에 의해 정지되었는지 알 수 있다. Cur\_job에 signal을 보낸 job을 할당한다. 만약 child가 signal에 의해 stopped 되었을 경우, 문구를 출력하고 그 cur\_job의 state를 stop으로 바꾼다. 만약 정상적으로 종료되었을 경우, 그 job을 deljob을 이용해 초기화 시킨다. 여기서 종료된 job이 background job일 경우, 시각적으로 알기가 어렵기 때문에 종료되었다는 문구를 띄운다. Foreground job이면 문구를 출력하지 않는다. kill이나 crtl+c 같은 signal에 의해 종료되었을 경우 그 job을 deljob을 사용해 초기화시킨다.
  + Sigint\_handler -> crtl +c를 받았을 때 실행된다. Getfgpid로 실행중인 foreground job의 pid를 받아 SIGINT signal을 보낸다. 실행중인 foreground job이 없을 경우, 그냥 return 한다.
  + main함수에 SIGCHLD, SIGINT, SIGSTP signal handler을 install하고 initjobtable로 job table을 초기화시켰다.
  + Eval 함수에선 bg 값을 parseline으로부터 받는다. 또한 명령어에 less가 있을 경우, sigttou\_hanlde값을 1로 설정한다. Less가 실행되면 signal 22, SIGTTOU signal에 의해 process의 작동이 중지된다. Background job일 경우, SIGCHLD를 blocking해준다. 또한 less가 없거나 background job일 때, setpgid로 프로세스 child의 pid값을 group process의 값으로 설정한다. Parents 에서는 입력 받은 명령어들을 job table에 넣어준다. insert하는 도중 SIGCHLD를 받을 경우 job table을 동시에 지우고 쓰는 경우가 발생하기에 insert를 완료하고 SIGCHLD를 unblock한다.
  + Parseline의 반환 형을 int로 설정했다. parseline에서 마지막에 &문자가 있을 경우(background job), bg를 1로 설정한다. 이 bg의 값을 return해 eval에서 명령어가 background인지 foreground인지 판별할 수 있게 하였다.
  + Building\_command 함수에 여러가지 if 문을 추가했다. Cmd가 jobs일 경우 printjob, kill 일 경우 execkill, fg일 경우 execfg, 그리고 bg일 경우 execbg를 실행시킨다.

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

Builtin\_command가 아닐 경우 fork 후 명령어를 실행한다. Parent는 child의 종료를 기다린다.

1. **Phase 2 (pipeline)**

Pipe가 존재하는 경우 fork를 두 번 실행한다. 처음 생성된 자식을 parent, 처음 생성된 자식으로부터 생성된 자식을 child라고 할 때, parent에서는 pipe로부터 data를 읽고, child는 pipe로 data를 쓴다. 마지막 command일 경우 명령어를 parents에서 실행하고 grandparents로 돌아간다.

1. **Phase 3 (background)**

Phase 3의 경우 builtin\_command check후 background이면 wait없이 child들을 실행한다. 실행 종료를 기다리지 않고 바로 main으로 돌아간다. Foreground인 경우 wait함수를 사용하여 child의 종료를 기다린다. Crtl+z와 crtl+c가 입력되었을 때, 각각 user-define signal handling 함수가 실행된다.