**System Programming Project 2**

담당 교수 : 박성용

이름 : 김명준

학번 : 20201558

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* **(MyShell을 만드는 전체적인 개요에 대해서 작성하면 됨.)**

Unix에서 shell은 컴퓨터와 사용자가 상호작용하도록 하는 인터페이스이다. shell에 사용자가 명령어를 입력하고 실행시킴으로 인하여 파일 및 디렉토리를 관리하고, 기타 프로그램을 실행하거나 시스템에 접근하기도 한다. shell을 통해 사용자는 운영 체제 커널과 상호작용 할 수 있으며, 컴퓨터를 제어하고 작업을 수행한다.

이 프로젝트에서는 Shell을 간소화하여, 유사한 기능을 갖고 있는 MyShell을 구현한다. 최종적인 MyShell이 지원하도록 할 주된 작업은 기본적인 파일 및 폴더에 대한 생성, 수정 및 삭제와 프로그램(명령어)실행, 파이프 명령, 백그라운드실행, 프로세스 관리이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

fork와 excevp함수를 사용하여 기본적인 리눅스 명령을 수행할 수 있다. 또한 추가적으로 ‘&&’를 사용하여 && 이전의 명령이 성공할 때, 이후 명령도 바로 수행하도록 기능을 구현한다.

1. Phase 2

Phase1의 상황에 더하여 더 많은 리눅스 명령을 수행하고, 파이프라인 ‘|’를 활용해 ‘|’ 이전의 명령의 출력을 이후의 명령의 입력으로 사용할 수 있게끔 기능을 구현한다. 예를 들어 ls | grep c 를 사용시, ls의 결과가 터미널에 출력되는 것이 아니라, grep c 의 입력으로 전달되어 ls 의 결과 중 c가 포함된 문자열을 grep c가 출력하게 된다. 파이프가 1개가 아니라 2. 3. … n개 존재하더라도 동일하게 작동될 수 있다.

1. Phase 3

Phase2의 상황에 더하여 jobs, bg, fg, kill과 같은 프로세스의 상태를 관리하는 명령어들을 구현하고, 진행중인 프로세스를 강제 종료하거나, 중지시키는 등 시그널을 사용할 수 있게끔 기능을 구현한다. 정지한 process나, 백그라운드에 진행중인 process에 대하여 위에 언급한 bg, fg, kill을 사용하여 명령을 다시 시행하거나, 강제로 종료키는 있는 기능을 구현한다.

* 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

fork라는 명령어를 사용하면, pid=fork()와 같은 명령어를 사용한 그 상태와 완전히 동일한 상태의 process가 복사되어 실행되게 된다. 이 때, child와 parent를 구분하는 방법은 pid에 0이 저장되어 있으면 child, 그렇지 않다면 parent라고 판단할 수 있다. 때문에 조건문을 사용하여 pid가 0일 때, child process에서 실행하고자 하는 명령을 수행하게 하면 된다. 이러한 방식을 사용하는 방법은, execve등과 같은 방법으로 시스템 명령을 사용할 시 생기는 일 때문이다. 명령어를 불러오게 되면, 시스템에서 해당 명령어의 process를 그대로 복사하여 현재 process (즉 child process)에 복사하여 실행시킨다. 때문에 fork하지 않고 시행 시, 시스템 콜 이후 코드를 진행할 수 없기 때문에 fork를 통해 child process를 생성하고, child process를 시스템 콜로 덮어 쓰며 명령어를 수행하게 된다.

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

Phase 1에서는 Wait 또는 Waitpid를 사용하여 child의 종료를 확인하고, parent process가 reaping 및 이후 작업을 수행하도록 한다.

위에서 설명된 것처럼 pid가 0이 아님을 확인한다면, parent process는 child process가 고아나 좀비가 되는 것을 방지하기 위해 child가 수행되고 있는 동안 더 이상 진행하지 않고, Wait혹은 Waitpid 구문에서 멈춰 child가 종료되기를 대기한다. 만약 child process가 죽으면, wait 함수에서 자신이 생성한 자식의 종료를 감지하고, 해당 process를 reaping해주고, wait함수는 자식 프로세스의 종료 상태를 int로 반환해준다. wait과 waitpid의 차이점은 아무 child를기다리는가, 특정 pid의 child를 기다리는지에 있다. fork시 pid=fork()와 같이 저장한 pid가 그 child의 pid이다.

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

Pipeline은 Standard I/O 대신 파이프 이전 명령의 출력을 인풋으로 받거나, 자신의 output을 파이프 이후 명령에게 input으로 넘겨주는 작업을 한다.

이를 구현하기 위해서는 dup, dup2, pipe 함수를 사용하여 구현했다.

우선 파이프를 구현하기 위해서는 사용자에게서 입력 받은 한 줄의 명령에서 파이프를 찾아내고, 파이프를 기준으로 파싱하는 작업이 먼저 수행되어야 한다. strtok함수를 사용하여 |를 기준으로 파싱하고, 직접 만든 큐 자료구조에 순서대로 저장했으나, 파싱이 주된 기능이 아니기에 자세한 설명은 생략한다.

명령어 한 줄을 모두 분석하여 저장했다면, 파이프를 사이에 둔 명령의 입출력을 제어해야한다. pipe(fd)를사용하여 fd[0]에 파이프 입력, fd[1]에 파이프 출력을 저장한다. 이제 파이프를 사이에 두고, 파이프 이전 명령은 출력을 STDOUT이 아닌, fd[1]으로 출력하게 하고, 파이프 이후 명령은 STDIN대신 fd[0]을 입력으로 받게끔 하여 파이프를 구현할 수 있다. 이 때, 파이프를 매 명령마다 새로 만들어주고 싶다면, 이전 명령에서의 파이프 값을 별도의 변수에 저장해두고 사용해야 한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

명령어 중 파이프의 개수에 따라서 조건을 달리하여 진행했다. 파이프 기준으로 파싱한 문자열이 1개이하라면, 파이프라인이 없는 명령이므로 phase1에서의 명령어 시행과 같이 진행시켰고, 2개 이상이라면, 파이프 기준으로 파싱한 문자열을 큐에 넣어두었다가, 하나씩 빼내며 evalForPipe라는 새로운 함수에 넣어 eval시켰다. 큐의 시작에 있던 명령어는 input을 STDIN으로, 큐의 마지막인 명령어는 output을 STDOUT으로 하게 만들었고, 이외의 input과 output은 모두 파이프 입출력을 사용하여 각 프로세스들의 명령을 서로 주고받게 했다.

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

myshell에 입력된 명령의 마지막에 ‘&’문자가 있을 때, 명령을 백그라운드에서 시행되도록 만들었다. 백그라운드와 포그라운드 실행의 차이점은 백그라운드 실행은 프로세스 실행 중에도 shell이 다른 명령을 받아들일 수 있다는 것에 있다.

때문에 명령어를 파싱하여 만약 마지막에 &가 포함되었다면, 백그라운드 실행을 진행하면 되는데, process가 진행되는 동안, 별도의 wait이나 pause작업 없이, 다시 입력을 받는 단계로 넘어가주면 된다. 이를 위해 eval이나 evalForPipe에서 백그라운드가 아니라면, pause나 wait없이 다시 명령을 받는 단계로 돌아가게끔 조건문을 작성해주었다. 이 때 별도로 대기하지 않았던 child process를 reaping해주는 방법에는 SIGCHLD를 사용하는 것이 있었다. SIGCHLD는 부모가 생성한 자식이 변화할 때 발생하는 시그널으로, 만약 이 시그널이 발생하면 별도의 시그널 핸들러 함수에서 waitpid를 사용한 반복문을 통해 reaping할 자식이 있는지 탐색 후 reaping해주며 고아나 좀비의 생성을 방지할 수 있었다. 또한, Phase 3에서 백그라운드 프로그램이나 중지중인 프로그램이 별도의 CTRL+C, CTRL+Z에 의해 영향받는 상황을 피하고자 모든 자식 프로세스의 생성 과정에서 gpid를 자기 자신의 pid로 바꾸는 작업도 구현한다.

* 1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**
* **Phase 1 : execvp, chdir, &&, 파싱함수, 파싱 큐**

명령어 실행 : fork & execvp

phase 1에서 요구했던 명령어 실행에 대하여는 eval이라는 스켈레톤코드와, fork, evecvp를 사용하여 해결했다. 만약 입력받은 명령어가 builtin functiond 아닐경우, fork를 사용해 child process를 만들고 그 process 안에서 parseline으로 파싱된 argv를 인자로 하여 execvp(argv[0], argv)함수로 명령을 수행한다. execvp는 환경변수 내의 PATH경로 내에서 argv[0]의 명령이 수행 가능한지를 확인한 후 가능하다면 명령을 수행하고 불가하다면, -1을 반환한다. 때문에 execve를 사용한다면 ls가 argv[0]일 때, 앞에 /bin/등을 추가로 문자열에 붙여줘야하는 등의 작업을 execvp로 대채함으로써 해결할 수 있다. 이 execvp는 bin, usr/bin 내의 모든 명령어를 수행할 수 있게 해주기 때문에, phase1에서 요구된 명령이 아닌 모든 bin, usr/bin명령을 수행하게 해준다. 즉 built-in인 cd, phase3의 jobs, fg, bg, kill 제외한 이 프로젝트의 모든 명령어를 execvp로 모두 처리할 수 있었다. execvp는 다른 exec함수들처럼 현재 프로세스를 해당 명령 프로세스로 덮어 씌우기 때문에 fork로 만들어진 child에서 수행하게 한다. fork 이후에 자식 프로세스에서는 execvp를 수행하고, 부모 프로세스에서는 waitpid를 통해 child가 끝날때까지 다른 작업으로 넘어가지 않고, reaping을 위한 대기를 하고 있게 제작했다.

cd 명령어

cd명령어는 위에서 언급한 것처럼 built in함수이기 때문에 직접 c코드안에서 구현하였다. C언어 라이브러리에 있는 chdir함수를 사용해 구현했다. chdir함수는 인자로 받은 char\*에 해당하는 위치로 이동하게끔 도와주는 함수이다. builtin\_command함수안에 strcmp를 사용하여 argv[0]이 “cd”라면 chdir을 통해 디렉토리 이동을 하게 해줬다. 기본적인 이동 외에도 ch ~, ch 를 통해 HOME디렉토리로 이동하는 것을 구현하기 위해 argv[1]을 분석하여 해당 경우에는 chdir(getenv(“HOME”))을 통해 홈 디렉토리로 이동하게끔 하였다. 이를 통해 ch~/폴더와 같은 명령을 통해 홈디렉토리 아래에 있는 폴더로 바로 이동하는 것도 가능하게 구현해줬다.

&&기능 구현

초기 명세서에 존재하던 &&기능도 구현하였다. 이후 구현하지 않아도 되게끔 수정되었지만, Phase2의 |를 처리하는 과정과 유사하다고 느껴 함수로 제작하여 구현해서 굳이 제거하지 않았다.

main함수에서 cmdline에 사용자에게서 입력을 받으면 ParsingUseStr라는 함수를 사용해 큐에 명령을 파싱했다. 이 함수는 int ParsingUseStr(Queue\* Q, char\* command, char\* parsing)의 프로토타입을 갖게끔 제작하였다. 첫번째 인자인 큐에, 두번째 인자인 command를 strtok함수를 사용하여 세번째 인자인 parsing으로 나누어 순서대로 저장해줬다. 반환값은 큐의 사이즈였다.Queue Struct는 char\* Item[], int Front, Rear, size를 갖고 있는 구조체이며, 순환 큐로 구현했다. 추가로 EnQueue, DeQueue, initQueue함수도 구현하였으나 자세한 코드는 생략한다.

위 함수로 파싱하여 큐에 명령들을 &&로 나누어 저장해둔 후, 순차대로 큐에서 명령어를 꺼내며 eval()함수에 넣어 명령을 수행해주었다. 이 과정에서 명령 수행에 실패할 시 &&이후의 명령을 수행하지 않고 끝내는 기존 쉘의 성격을 따라 어떠한 명령이 실패하면 eval함수가 음수를 반환하게 하여 큐의 모든 내용을 강제로 비우고 다음 명령어를 입력받게끔 수정해주었다. 이 작업을 위해 기존 void반환값을 갖던 eval을 int 반환값을 갖게끔 수정해주었다.

* **Phase 2 : pipe ( evalForPipe, evalMakerForPipe )**

phase에서 추가된 명령어는 이미 phase1에서 모두 evecvp를 사용함으로 구현되어있는 상태였기에, 파이프 부분만 추가 구현했다

위에서 사용했던 ParsingUseStr함수를 사용해 메인함수에서 &&파싱된 문자를 eval에 넘기기 전에 한번 더 |를 기준으로 파싱해주었다. 반환값, 즉 |로 파싱한 큐의 크기가 1 이하라면 파이프가 없는 경우이므로 phase1처럼 기존 eval을 사용하였고, 그렇지 않는 경우에는 evalMakerForPipe라는 함수를 제작하여 이 함수에 |를 기준으로 쪼갠 명령어들이 들어있는 큐를 인자로 넘겨주었다.

evalMakerForPipe라는 함수는 파이프가 들어있는 명령어들을 evalForPipe라는함수에게 순차적으로 넘겨주며, 인풋과 아웃풋 제어를 용이하게 하기 위해 제작되었다. int currentOutput이라는 이전 작업의 출력을 뱉어주는 파이프 인풋을 의미하는 변수를 선언했고, 초기값은 STDIN을 의미하는 0으로 설정했다. 이후 순차적으로 evalForPipe(command, i, maxComIdx, &currentOutput)를 사용하여 eval을 수행했다. i는 현재 명령이 큐의 몇번째였는지, maxComIdx는 마지막 명령의 index를 의미한다.

evalForPipe는 기존 eval함수를 약간 변형시킨 파이프용 eval함수이다. 큰 틀에서는 eval과 같지만, fork이후 child process가 수행할 작업에서 시작부분에 pipe(fd)를 통해 파이프를 제작한 후, 만약 현재 명령이 마지막이 아니라면, 파이프로 출력을 바꿔주고, 시작이 아니라면, 인풋을 currentOutput으로부터 받게끔 해주는 작업을 추가했다.(evalMakerForPipe의 초기 currentOutput이 이미 STDIN이라 제거해도 되는 작업이지만 가독성 측면을 위해 추가) 그리고 부모 프로세스의 작업 부분에서는 마지막에 현재 명령이 마지막 명령이 아니라면, \*currentOutput을 현재 pipe의 fd[0]을 갱신시켜주고, fd[1]을 close해주는 작업을 추가하여 다음 함수와의 파이프 연결을 구현했다.

* **phase 3 : job 구조체, signal hander, wait -> sigsuspend 그리고 fg,bg,jobs,kill**

phase 3 구현을 위해 우선 job들을 저장할 수 잇는 jobs 구조체와 jobs들을 저장할 Jobs jobsList[]를 전역변수로 만들었다. jobs구조체에는 해당 작업의 pid, 몇번째 job인지, 해당 명령의 커맨드, 상태를 지니게 만들었다. 이는 각각 pid\_t jobPid, int jobIndex, char comName[], int state로 만들었다. state는 1일 때 foreground, 2는 background, 3은 stop, 0은 없는 job을 의미하게 제작하였다

jobList을 조작하는 함수들로는 아래 함수들을 제작했다.

jobListInit ( jobList를 초기화시키는 함수 )

findLastJobIndx() (마지막 jobList Index를 찾는 함수 )

makeJobElement (명령어, 프로세스의 pid, 상태를 인자로 받아 jobList의 마지막 Index+1번째 element로 추가해주는 함수.)

delJobElent ( pid를 인자로 받아 해당 pid를 갖는 job을 jobList에서 찾아 모든 값을 0으로 초기화 ( 상태도 invalid로 만들어줌 ))

jobsPrint ( jobList들을 마지막 Index까지 순회하며 state가 0이 아니라면 상태를 출력하게끔 만드는 함수. [몇번째 job인지] suspended/running 명령어 순서대로 출력함. )

CallJobbyPid ( pid를 인자로 받고, jobList를 순회하며 해당 pid를 갖는 job element를 Jobs\* 형태로 반환. 없다면 NULL 반환 )

또한 phase 3의 기능들을 구현해주고자 몇가지 signal handler들을 우선 구현했다.

SIGTERM, SIGINT, SIGTSTP, SITCHLD에 대해 구현햇다. 각각 Signal(시그널, 시그널\_handler)로 main 맨 앞부분에 시그널을 캐치하게 만들었고, 각 시그널 핸들러의 작업은 아래와 같다.

우선 SIGTERM은 키보드로 발생하진 않지만, 외부에서 명령이 들어올 수 있기 때문에 발생시 myshell자체가 끝나도록 \_exit(0)을 발생시켰다. \_exit은 exit대신 사용할 수 있는 async-signal-safe 함수이다.

SIGINT는 ctrl + C를 입력 시 발생하는 시그널로 이 시그널이 들어오면 현재 foregroud에 시행중인 명령이 중지될 수 있게끔 jobList에서 foreground인 명령이 있다면 해당 명령에 대해 kill(해당 프로세스 pid, SIGINT)를 수행하고, 해당 jobList요소를 비우도록 제작했다.

SIGTSTP는 현재 foreground에 있는 명령을 ctrl+Z가 입력될 시 중지시키게끔 만들었다. SIGINT와 유사하게 현재 수행중인 job을 찾고, kill(해당 프로세스 pid, SIGSTOP)으로 process를 suspended상태로 만들고, 해당 jobList의 state를 3으로 만들었다.

위의 CTRL+Z, CTRL+C를 통해 sigkill, sigstop을 보내는 과정에서, 동일한 그룹아이디를 갖고 있을 시에, 원치 않는 백그라운드 프로세스나, 중지중인 프로세스에게 신호가 가는 것을 방지하기 위해 모든 프로세스들이 생성될 때, 그룹 pid를 자기 자신의 pid로 바꾸는 작업을 추가로 행해주었다.

SIGCHLD는 만약 자식의 상태가 변한다면 부모에게 전달되는 시그널이다. 이 시그널이 발생한다면 reaping할 수 있는 모든 process를 반복문을 통해 wait해준다.

while((tmppid=Waitpid(-1,&child\_status,WUNTRACED | WCONTINUED | WNOHANG))>0)이와 같은 반복문으로 가능한 모든 자식들을 wait함수로 감지해냈다. 이 과정에서 waitpid에 옵션을 주었는데, WUNTRACED | WCONTINUED | WNOHANG라는 옵션이다. 순서대로 WUNTRACED는 현재 종료된 process가 아니라, suspended된 child에 대한 상태도 보고하게 해주는 옵션으로 이 옵션을 넣어주면 SIGSTOP으로 멈춘 child도 감지할 수 있다. WCONTINUED는 이후에 구현할 SIGCONT로 인해 다시 진행되게 만든 child를 감지하게 해주고, WNOHANG은 wait할 자식이 없을 때, 0을 반환하여 반복문을 탈출하게 해준다. 또한 tmppid가 0이 아니어서 반복문 안에서 내용을 시행할때에는, 이후에 사용할 volpid를 tmppid로 세팅하고, WIF함수들을 사용해 현재 process의 상태에 따라 JobList를 삭제하는 등의 작업을 행한다.

이렇게 시그널 핸들링 세팅을 마친 후에는 eval과 evalForPipe의 부모 process의 wait부분을 수정해주었다. 이를 위해 volatile sig\_atomic\_t volpid을 전역변수로 우선 선언한 후, 우선 child 생성 전에, Sigprocmask를 사용해 SIGCHLD를 block처리한다, 그 후 child를 fork로 생성후, 부모 process에서 volpid를 0으로 초기화 후, 포그라운드 명령에 대해서는 while(!volpid){Sigsuspended(&prev)}를 통해 SIGCHLD를 기다려 자식을 reaping한 후, 자식 리핑으로 인해 sigchld에서 volpid를 0이 아닌 자식의 pid로 바꿔주면, 반복문을 탈출하고 다시 기존의 signalblocking상태로 돌려주며 eval을 마친다. 이러한 방법을 사용한 이유는 이전과 달리 백그라운드 실행이나, suspended등을 구현하기 위해 wait으로는 불가능한 부분들을 보완하고, race가 발생하는 것을 막기 위해 이러한 방법을 사용하였다.

위의 구조체와 시그널 핸들링을 제작한 이유였던 fg, bg, kill, jobs 명령을 builtin으로 제작해주었다.

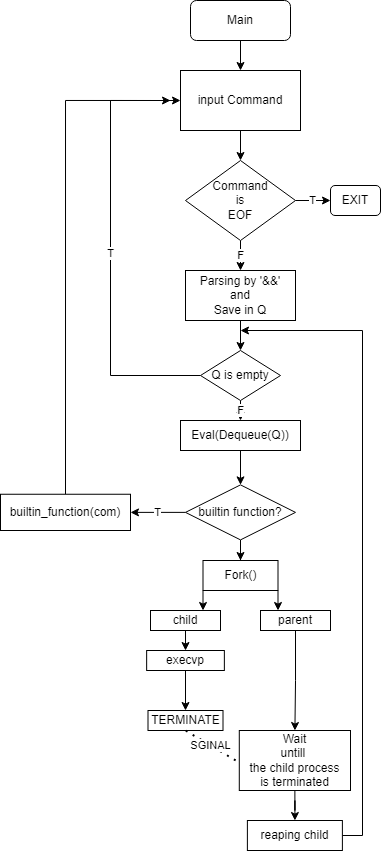
jobs명령에 대해서는 위에 언급한 JobsPrint를 수행하게 하여 제작했고, bg에 대해서는 정지되어있던 명령을 백그라운드 실행으로 돌려주는 작업 수행할 수 있게 하는 것이 목표이기 때문에, 올바른 형태로 명령이 들어온다면, 해당 번호에 맞는 Job을 JobList에서 찾아 Kill(jobList[jobIdx].jobPid, SIGCONT)로 job을 다시 실행상태로 만들고, jobList에서의 state도 background 실행을 의미하는 2로 수정해주었다. fg도 이와 유사한데, 백그라운드 실행중이거나, 중지된 job에 대하여 SIGCONT를 보내고, state를 foreground실행을 의미하는 1로 수정한 뒤, shell과 유사하게 명령어를 다시한번 터미널에 출력해주는 작업을 한다.

kill 명령어는 kill %<joblistidx> 형태로 사용하며 해당 index의 joblist에 있는 process를 sigkill을 통해 종료시키고 해당 joblist element를 delJobElement로 지워준다.

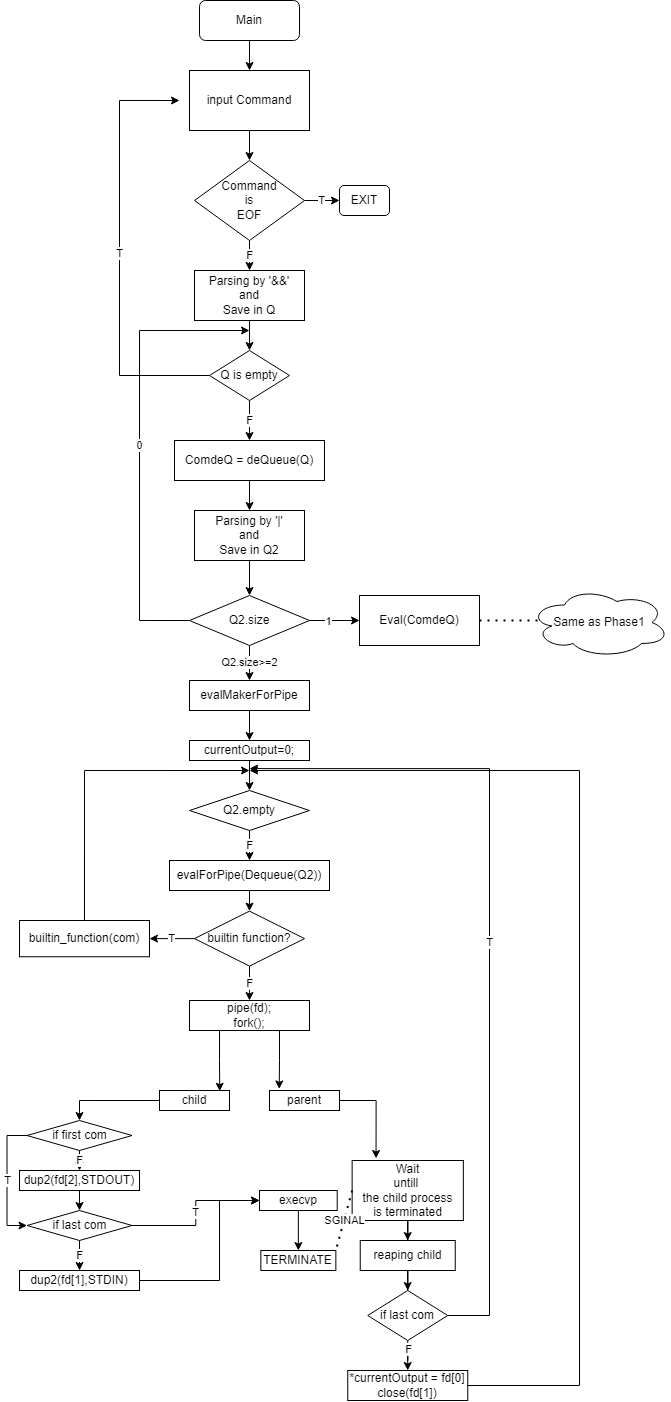
1. **구현 결과**
   1. **Flow Chart**

* **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)**

****

1. **Phase 2 (pipeline)**



1. **Phase 3 (background)**

