**System Programming Project 1**

담당 교수 : 김영재

이름 : 김승우

학번 : 20181610

1. **개발 목표**

해당 프로젝트에서는 수업시간에 배운 shell을 직접 구현하는 것을 목표로 한다.

Shell에 다양한 기능들이 존재하지만 프로젝트에서 구현할 내용은 다음과 같다.

[Phase 1]

* Fork()로 자식 프로세스를 만들어 stdin으로 입력받은 명령어를 execve로 수행
* Fg로 실행시킨 자식 프로세스가 종료되면 부모 프로세스가 waitpid로 reap

[Phase 2]

* 파이프가 포함된 명령어에 대해 필요한 만큼 자식 프로세스를 recursive 하게 fork
* 자식이 생성된 순서대로 waitpid를 통해 reap 진행
* Fd[2]를 MAXJOBS 만큼 생성하여 이전 자식 프로세스의 output이 현재 fd[WRITE]를 통해 기록되고, 다음 자식 프로세스가 이전 fd[READ]를 통해 input을 받음
* Pipe()와 Dup2()를 이용하여 STDOUT과 STDIN가 pipe로 생성된 fd로 복제되게끔 함

[Phase 3]

* Job Queue를 통해 백그라운드 job과 stopped job을 관리
* Job Queue는 JobQueue라는 구조체의 포인터로 구성하며, JoqQueue 구조체는 Job이라는 구조체의 front, rear 포인터와 job 개수를 담고있음. 즉 링크드 리스트 형태로 이루어짐.
* Job 구조체는 해당 job을 이루는 프로세스들의 pid 배열, pgid, 상태(Running or Suspended), cmd, Job \*next, termios tmodes(터미널 상태)를 포함하고 있음.
* Bg로 명령어가 주어졌을 경우, 부모 프로세스는 명시적으로 wait을 하지 않고 procmask의 block이 해제 이후에 시그널 핸들러가 reap 할 수 있도록 처리.
* Bg job이 생성되는 경우 add\_job을 통해 job을 추가하며, 핸들러에 의해 reap 될 시 delelte\_job을 수행.
* add\_job 에서는 job을 이루는 첫번째 프로세스의 pid가 job의 pgid가 될 수 있도록 setpgid()로 처리
* print\_job을 통해 현재 job들의 정보를 출력할 수 있음 (job 명령어를 위함)
* fg로 실행시킨 job에 대해 ctrl+z 입력이 들어온다면 해당 job을 suspended 상태로 job queue에 추가
* suspended된 job을 fg로 실행시킬 경우 tcsetpgrp(STDIN\_FILENO, job->pgid)을 이용해 해당 job이 터미널 권한을 점유할 수 있도록 처리하고, job의 모든 프로세스에게 SIGCONT를 보내 restart 시킴. 쉘 프로세스는 wait을 통해 job이 끝나길 기다림. 정상 reap되었을 경우 다시 tcsetpgrp을 통해 쉘이 터미널을 점유할 수 있도록 복구시킴.
* Bg job을 fg로 실행시키는 경우도 위와 동일하지만, SIGCONT를 보내도 이미 Running 상태이므로 해당 신호를 무시하게 됨.
* Suspended된 Job을 bg로 실행시키는 경우 해당 job의 프로세스들의 SIGCONT만 보내 restart 시킴.

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

텍스트이(가) 표시된 사진

자동 생성된 설명

Ls, mkdir, rmdir, echo, cat, touch 명령어를 execve로 정상 수행하는 것을 확인할 수 있다. Exit과 cd는 built\_in\_command로 구현했다.

1. Phase 2

텍스트, 스크린샷, 전자기기, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

Ls -al | grep filename 명령어는 파이프가 포함되어 있기 떄문에 파이프 처리를 했다. 특히 grep filename의 경우 “나 ‘가 짝수개로 주어지는 경우에 대해서도 처리를 했다.

1. Phase 3

텍스트이(가) 표시된 사진

자동 생성된 설명

Phase 2에서 수행한 명령어를 bg로 처리했다.

텍스트이(가) 표시된 사진

자동 생성된 설명

sleep 10 을 bg로 실행한 모습이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Bg로 실행시킨 명령어를 stop한 뒤 kill 한 모습이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Fg로 실행시킨 명령어를 stop한 뒤 Kill 한 모습이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Fg로 실행시킨 명령어를 stop한 뒤 bg로 restart 시킨 모습이다.

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

fgets를 통해 stdin으로 입력받은 커맨드를 parseline 함수가 파싱한다. 파싱한 결과는 execve의 파라미터에 맞게 argv에 담긴다. Eval 함수 내부에서 실제 프로세스를 생성하는데, if((pid = fork()) == 0) { … } else { … } 로 자식과 부모를 구분한다. 이 때 자식 프로세스(pid == 0)에서는 ctrl+z, ctrl+c에 반응하기 위해 sigprocmask를 이용해 block된 시그널들을 해제한다.

Execve시에는 file의 full-path를 전달해야한다. 따라서 ls가 아닌 /bin/ls를 전달해야 한다. 또는 어떤 명령어는 /bin/이 아닌 /usr/bin/에 위치한다. 따라서 우선 execve\_status = execve(argv[0], argv, environ);을 수행한 뒤, execve\_status가 0보다 작으면 비정상 수행된 것이므로, /bin/을 입력한 커맨드 앞에 붙여 수행한다. 그럼에도 execve\_status가 0보다 작다면 마지막으로 /usr/bin/을 붙여 수행한다. 그런데도 비정상 수행 되었다면 커맨드를 찾을 수 없다는 메세지를 출력하고 exit(0) 한다.

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

자식 프로세스의 execve 수행 이후 자식이 terminate 되면 부모는 이를 reap 해줘야한다. 이는 waitpid를 이용해 끝이난 자식을 reap하기 위해 부모는 계속해서 suspend 하고 있다. 자식이 정상적으로 종료되어 waitpid를 통해 reap을 했으면 eval 함수를 빠져나와 다시 shell main함수의 초반부로 돌아가 사용자의 입력을 받기를 기다리는 상태가 된다.

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

우선 사용자가 입력한 커맨드에 파이프가 있는지 확인하고 있다면 pipe\_exists flag를 true로 바꾼다. 그리고 파싱을 통해 첫번째 파이프 이전의 명령어에 대해 자식 프로세스를 만들어 execve를 수행한다. 파싱이 끝나면 명령어가 담긴 버퍼는 첫번째 파이프 바로 다음을 가리키고 있다. 이 버퍼를 갖고 eval 함수를 재귀호출한다. 예를 들어 a | b | c 라는 명령어가 들어왔다고 하자. 그럼 파싱을 한 뒤 버퍼는 b | c 를 담고 있으며, a에 대해 fork된 자식이 execve를 수행한다. 그 다음에 재귀 호출을 하여 b | c 를 처리한다. 남아있는 파이프가 없을 때 까지 자식 프로세스를 생성하는 것이다.

자식 pid를 저장하기 위해 pid[MAXJOBS]에 파이프를 저장한다. 첫번째 자식은 pid[count] (count == 0)에 저장한 뒤, 재귀호출시마다 count++를 하여 pid에 순차적으로 자식 pid가 담길 수 있도록 한다.

파이프의 핵심은 이전 프로세스의 output이 다음 프로세스의 input으로 들어가야 한다는 것이다. 이를 위해 pipe()함수로 fd[2] (READ and WRITE)를 생성하고, 이 file descriptor을 dup2로 STDIN과 STDOUT을 통제할 수 있도록 한다. 파이프의 개수만큼 fd[2]가 필요하므로 fd[MAXJOBS][2]로 선언했다. 어떤 이전 프로세스가 자신의 출력을 fd[count][WRITE]에 쓰기 위해 dup2(fd[count][WRITE], STDOUT\_FILENO)를 수행한다. 이를 통해 execve의 수행 결과가 터미널에 출력되는 것이 아닌 fd[count][WRITE]에 write 될 수 있다. 그리고 다음 프로세스는 fd[count][READ]를 통해 이전 프로세스의 출력값을 input으로 읽는다. 이를 위해 dup2(fd[count-1][READ], STDIN\_FILENO)를 수행한다. Count-1 인 이유는 다음 프로세스가 실행되며 count가 증가했기 때문에 이전 프로세스의 count에 해당하는 count-1 에 대해 fd를 참조한다. 수행 결과 이전 프로세스의 fd에 기록된 execve 출력을 stdin으로 받게된다. Read, Write 작업 이후 해당 fd는 close를 해 준다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

앞서 설명한 과정은 파이프의 개수에 상관없이 동일한 논리로 동작한다. 하지만 첫번째 프로세스, 중간 프로세스 그리고 마지막 프로세스는 일부 분기처리를 통해 조금은 다르게 동작하게 처리했다.

마지막 프로세스의 경우 파이프가 존재하지 않으므로 pipe()를 통해 fd를 새로 생성할 필요가 없다. 이는 pipe\_exists flag를 통해 분기처리했다. 또한 첫번째 프로세스의 경우는 자신의 이전 프로세스가 존재하지 않기 때문에 input을 받을 필요가 없다. 따라서 if (count > 0 ) dup2(fd[count-1][READ], STDIN\_FILENO); 를 통해 두번째로 생성된 프로세스 이상일 때부터 input을 받도록 처리했다.

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

Parse\_line 함수를 통해 마지막에 &가 있다면 마지막 argv는 NULL을 가리키도록 한 뒤 True를 한다. 왜냐하면 execve를 위한 argv에 &는 들어가면 안되기 때문이다. 이후 bg = parse\_line()으로 return 값을 받은 bg에 의해 분기처리 된다. Bg의 경우 기존이랑 동일하게 자식을 실행시키되, 부모 프로세스에서는 명시적으로 waitpid를 하지 않는다. 그 이후 add\_job을 통해 해당 프로세스(또는 Job)을 Job Queue에 담는다. Bg 프로세스가 끝나면 sigchld\_handler에 의해 시그널 핸들링이 된다. Waitpid를 통해 종료된 bg child를 reap 하고 delete\_job 함수를 통해 job queue에서 해당 job을 이루는 프로세스가 전부 끝났으면 job을 완전히 삭제한다.

* 1. **개발 방법**

텍스트이(가) 표시된 사진

자동 생성된 설명

Phase 3의 Job을 관리하기 위해 다음과 같은 구조체를 정의했다. Job은 하나 이상의 프로세스들로 이루어져있다. 하나의 Job의 pgid는 job을 이루는 첫번째 프로세스의 pid이다. 이는 추후에 tcsetpgrp를 이용해 bg job또는 suspended job을 fg로 돌리기 위해 터미널의 권한을 해당 Job의 pgid로 위임하는데, 이를 위해 규칙을 정한 것이다. JobQueue는 Job의 front와 rear을 가리키는 Job 포인터, 그리고 개수를 담당하는 count를 갖고있다. 이는 나중에 어떤 job을 검색할 때 front ~ rear의 범위를 지정해서 search할 수 있어 유용할 것이다. 또한 job 안에 정의된 Number는 프로세스의 개수다. 만약 프로세스 3개로 이뤄진 job이 있다 하자. 만약 이 중 하나의 프로세스가 종료되었으면 해당 Job을 아직 삭제하면 안된다. 종료되지 않은 2개의 프로세스가 있기 떄문이다. Job의 number가 1보다 큰지 아닌지를 통해 해당 Job을 완전히 삭제할지 판단하는 기준이 된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

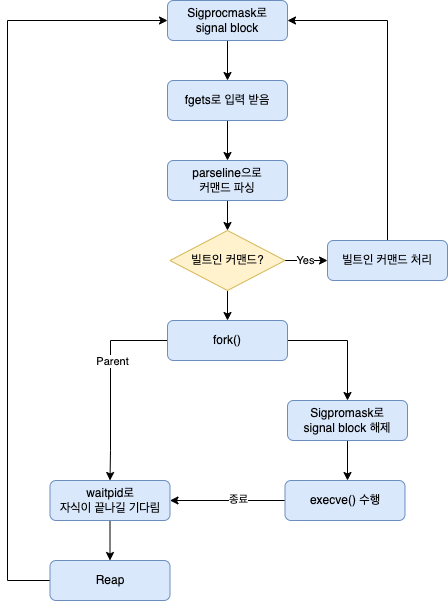
다음은 페이즈 1, 2를 위해 구현한 함수들이다. 이 때 phase1,2,3으로 진행되며 사용자의 입력 명령어에 대해 처리해야 할 문자가 늘어나 parseline 함수를 발전시켰다. 우선 parseline의 기본적인 동작 방법은 ‘ ‘을 기준으로 argv를 구분하여 argv[]에 저장한다. ‘ ‘위치마다 ‘\0’으로 대치하여 해당 문자열의 끝임을 알린다. 그리고는 buf를 ‘ ‘다음으로 옮겨 또다른 단어를 찾도록 한다.

이 때 echo의 경우 “와 ‘를 처리해야한다. 이는 두번째 인자를 처리할 때부터 남은 buf에 “나 ‘가 있는지 확인한다. 있다면, “과 ‘ 이후의 첫번째 ‘ ‘까지 delimiter\_ptr를 이동시킨다. 이후 buf ~ delimiter\_ptr 사이의 모든 “와 ‘를 ‘\0’로 대치한 뒤 또다른 버퍼에 문자열들을 이어붙인다. 이렇게 하면 공백도 포함한 argv를 만들 수 있다.

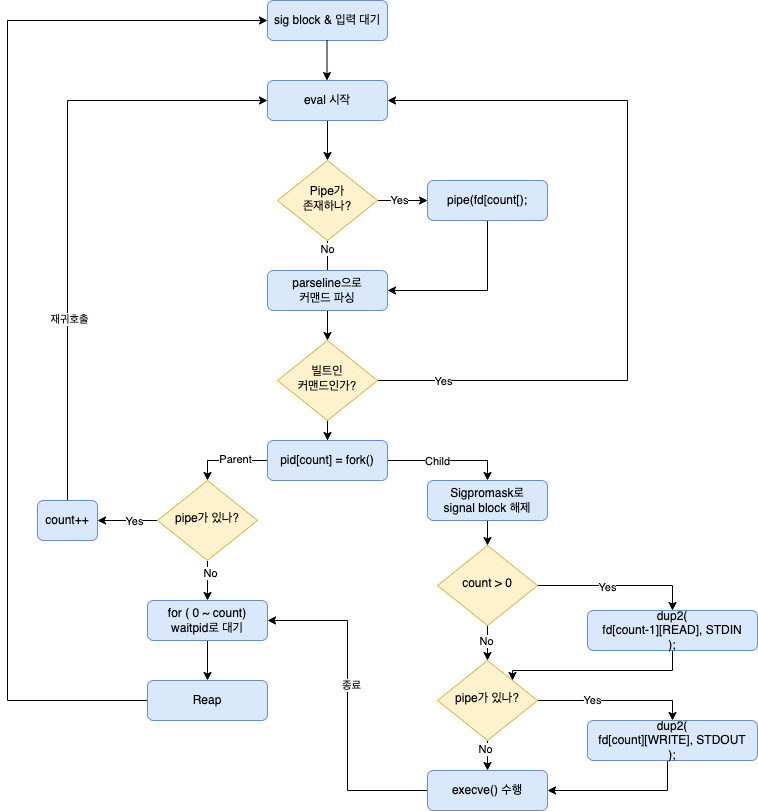
페이즈 2에서는 파이프를 처리한다. 파이프는 ‘ ‘로 대치한 뒤 해당 위치를 pipe\_ptr로 따로 저장한다. 나머지는 기존의 방식과 동일하게 처리하되, buf++를 증가시키며 순회하다 buf > pipe\_ptr이 되면 parse\_line을 종료한다. 또한, “와 ‘의 처리 범위를 설정하기 위해 pipeline이 있다면 최대 pipe\_line까지 “와 ‘를 처리할 수 있도록 제한한다. 그 이유는 ps -al | grep “a” | grep “b” 같은 명령어에 대해 최대 처리 범위를 지정하지 않으면 grep “b”의 마지막 “까지 순회하게 되는 에러가 발생한다.

페이즈 3에서는 &를 처리한다. &가 들어왔다면 arg\_number를 하나 줄이고 마지막을 NULL로 대치한다. 이는 execve에 &가 들어가는 것을 막기 위함이다.

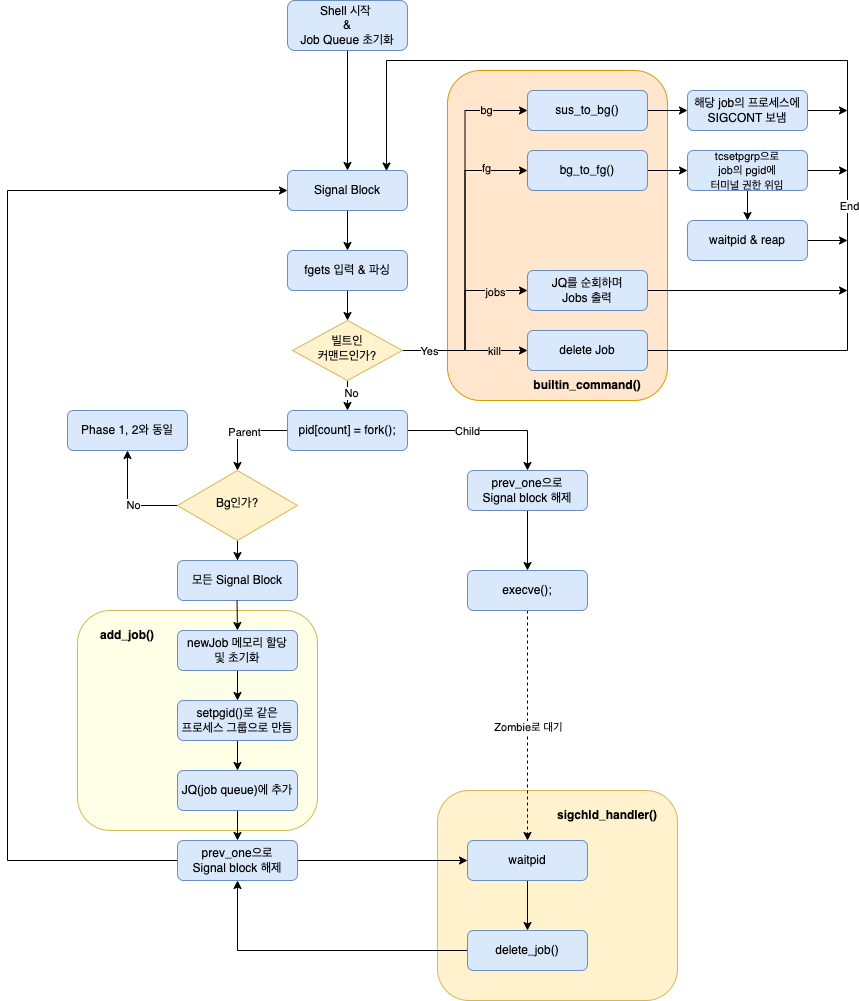
1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****