**System Programming Project 4**

담당 교수 : 김영재 교수님

이름 : 김민선

학번 : 20181603

1. **개발 목표**

해당 프로젝트의 목표는 총 3개의 phase를 수행하여 My shell을 완성하는 것이다.

Phase1에서는 fork를 통해 사용자가 입력한 command를 child process에서 수행할 수 있도록 한다.

Phase2에서는 phase1에 pipe기능을 추가한다. ‘|’를 기준으로 command를 자르고, 한 파이프의 output이 다음 pipe의 input이 되도록 하고, 마지막 파이프의 output을 화면에 출력할 수 있도록 한다. 이 때 pipe기능은 child process에서 recursive하게 이루어지도록 구현한다.

Phase3에서는 background processing 기능이 추가된다. command에 ‘&’가 포함될 경우, 해당 process는 background job이다. background job은 parent process에서 child process를 기다려주지 않으며, ctrl\_c와 ctrl\_z와 같은 signal을 무시한 채 실행될 수 있도록 구현한다.

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

사용자 입력 command에 대한 결과가 화면에 출력된다. 만약 built-in command일 경우 command가 곧바로 실행되어 그 결과를 화면 상에 출력한다. built-in command가 아닌 경우, command는 child process에서 실행되며, parent process는 child process가 끝날 때까지 기다린다.

1. Phase 2

사용자 입력 command에 pipe가 포함된 경우, 각 pipe의 output은 다음으로 이어지는 pipe의 input으로 들어가며, 가장 마지막 pipe의 output이 화면에 출력된다. 이 작업은 child process에서 각 pipe에 대한 fork를 통해 recursive하게 이루어진다. parent process에서는 각 pipe의 command가 끝날 때까지 기다린다. 만약 command에 pipe가 포함되지 않은 경우에는 phase1에서와 동일하게 처리된다.

1. Phase 3

사용자 입력 command에 ‘&’가 포함된 경우, 이 command는 background에서 수행된다. background job은 다음과 같이 처리된다. child process에서는 background job에 대한 ctrl\_z, ctrl\_c signal이 blocked된다. parent process에서는 background job이 JOB배열에 추가되며, child process가 끝날 때까지 기다리지 않고 다음 작업을 수행한다. foreground에 대한 작업은 phase2에서 구현한 것과 동일하다.

* 1. **개발 내용**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명
  + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow

사용자가 command를 입력한 후 eval()함수가 호출된다. eval()에서 command는 ‘ ‘를 기준으로 parse된다. 이 때 command가 built-in command가 아니라면, pid = fork(); 를 통해 child process를 생성한다.

만약 pid<0이라면 fork 오류이다. 이 경우 오류메시지를 출력한 후 prompt를 출력하여 다음 사용자 input을 기다린다.

만약 pid == 0이라면 child process이다. 이 경우 execvp함수를 통해 command에 대한 실질적인 처리를 수행한다. 종료 후에는 parent process에게 종료했다는 signal을 보낸다.

만약 pid > 0이라면 parent process이다. 이 경우 waitpid함수를 통해 child process가 command를 처리하고 종료 signal을 보낼 때까지 기다린다. child process가 종료 signal을 보내면 parent process는 waitpid 이후의 작업을 처리할 수 있다.

* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)
  + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

eval()함수에서 parent process와 fork error에 대한 처리는 phase1과 동일하다. child process의 경우, 만약 commands에 ‘|’가 포함되어있다면, 맨 앞에 있는 ‘|’를 기준으로 commands를 둘로 parse한다. 그 결과로 첫 번째 command와 그 이후의 commands를 얻을 수 있다. 이후 make\_pipe(현재 처리할 command, 나머지 commands)라는 함수를 호출하여 앞서 얻은 첫 번째 command와 그 이후의 commands를 parameter로 넘긴다. make\_pipe에서는 pipe()를 통해 파이프를 생성하고, 또 다시 pid = fork(); 를 수행한다.

pid < 0인 경우 fork error이므로 오류메시지를 출력한 후 종료한다.

pid == 0인 경우 child process에 대한 작업을 수행한다. child process에서는 현재 command를 처리하여 결과를 output으로 저장한다.

pid > 0인 경우 parent process에 대한 작업을 수행한다. parent process에서는 child process의 작업이 끝날 때까지 기다렸다가 child process의 output을 input으로 저장한다. 이후 나머지 commands를 ‘|’기준으로 또 다시 맨 앞의 command와 나머지 commands로 나눈 뒤 make\_pipe를 호출하여 재귀적으로 처리한다. 만약 마지막 command일 경우, make\_pipe를 호출하지 않고 바로 command를 처리한 후 함수를 종료한다.

사용자 입력으로 a | b | c가 들어왔을 경우를 예로 들어 설명하면 다음과 같다. 첫째로, eval()의 child process에서는 make\_pipe(a, b | c)가 수행된다. 이후 make\_pipe의 parent process에서는 make\_pipe(b, c)가 수행된다. 이 때 a가 수행된 뒤 그에 대한 output이 b에 대한 input으로 저장된다. 이후 b에 대한 output이 c에 대한 input으로 저장된다. c가 수행된 뒤에는 다시 eval()로 돌아간다.

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

background process에 대해서는 다음과 같이 처리한다.

1) prompt에서 &를 포함한 command가 입력된 경우, 해당 process의 pid를 job배열에 추가한다. 이 process의 상태는 background running이다.

2) foreground에서 running하던 command에 ctrl\_z signal을 가한 경우, 해당 process의 pid를 job배열에 추가한다. 이 process의 상태는 background stopped이다.

3) bg <job> command를 통해 background stopped process를 깨우는 경우, 해당 process의 상태는 background running이다.

background process의 경우, child process에서는 ctrl\_c나 ctrl\_z에 대한 signal을 막아둔다. parent process에서는 addJob()함수를 통해 background process를 job 배열에 추가하고, child process가 끝날 때까지 기다리지 않는다.

* 1. **개발 방법**

**[Structure]**

typedef struct \_jobStruct{

int status; //job의 현재 상태 [0: stopped] [1: sleeping] [2: running] [3: zombie]

int bg; //job이 background process이면 1로 set

int pid; //job의 process id

char cmd[100]; //job의 command

} jobStruct;

**[Global variables]**

(1) jobStruct bg\_job[MAXJOBS] : background jobs가 저장될 배열이다.

(2) static int bg\_jobNum : bg\_job에 들어있는 background jobs의 개수이다.

(3) int fg\_pid : 현재 돌고 있는 foreground job의 pid이다.

(4) char fg\_cmd[100] : 현재 돌고 있는 foreground job의 command이다.

**[Functions]**

(1) void eval(char \*cmdline)

이 함수는 fork를 통해 child process에서 command를 처리하는 함수이다.

cmdline은 사용자가 입력한 command이다. parseline함수를 호출하여 command를 ‘ ‘를 기준으로 parse한 뒤 잘려진 각 단어들을 argv에 저장한다. 이후 builtin\_command함수를 호출하여 argv[0]가 built-in command인지 검사한다. built-in command라면 각 built-in command에 해당하는 작업을 수행한 후 eval함수를 종료한다.

built-in command가 아니라면, fork를 통해 생성된 child process에서 argv에 대한 처리를 수행한다. child process에서 고려해야 할 사항은 두 가지이다. 첫째로, command가 background인지 아닌지 구분한다. command가 background job이라면 ctrl\_z, ctrl\_c에 대한 signal을 막아두고, foreground job이라면 열어둔다. 둘째로, command가 pipe인지 아닌지 구분한다. pipe라면 makepipe라는 재귀함수 호출을 통해 처리를 해주고, pipe가 아니라면 execvp함수를 통해 argv를 바로 처리한 후 child process를 종료한다. parent process에서는 command가 background job인지 아닌지 구분한다. background job일 경우, 해당 job을 bg\_job[]에 저장하고, child process를 기다리지 않은 채로 eval함수를 종료한다. foreground job일 경우, fg\_pid와 fg\_cmd에 해당 job의 pid와 command를 저장한 뒤 child process가 작업을 마칠 때까지 기다린다. 이후 child process가 종료 signal을 보내면 eval함수를 종료한다.

(2) int builtin\_command(char \*\*argv)

이 함수는 built-in command 여부를 판별하고 이를 처리하는 함수이다.

argv는 command가 ‘ ‘를 기준으로 parse되어 분리된 각 단어들이 들어있는 변수이다. argv[0]가 cd, exit, jobs, bg, fg, kill, quit, &인 경우 해당 command는 built-in command이고, 각 경우에 대한 작업을 처리한 후 1을 반환한다. 그 외의 command들에 대해서는 0을 반환한다.

(3) int parseline(char \*buf, char \*\*argv)

이 함수는 command를 ‘ ‘를 기준으로 parse하는 함수이다.

buf는 parse할 command이고, argv는 parse한 buf의 각 결과들을 저장한다. parse가 완료되면 buf가 background job인지 아닌지에 대한 값을 반환한다. background job인 경우 1이 반환되고, 그렇지 않은 경우 0이 반환된다.

(4) int parsepipe(char \*buf, char \*tmpCMD, char \*\*argv)

이 함수는 commands를 ‘|’를 기준으로 parse하는 함수이다.

buf는 parse할 commands이다. argv는 buf에서 가장 처음에 등장하는 ‘|’를 기준으로 앞에 해당하는 command를 ‘ ‘로 parse한 단어들이 저장된다. tmpCMD는 가장 처음에 등장하는 ‘|’를 기준으로 뒤에 해당하는 나머지 commands가 저장된다. 예를 들어, buf가 a | b | c | d라고 할 때, argv에는 a를 ‘ ‘로 parse한 단어들이 저장되며, tmpCMD에는 b | c | d가 저장된다.

(5) void makepipe(char \*\*argv, char \*buf)

이 함수는 파이프가 포함된 command를 처리하는 재귀함수이다.

argv에는 현재 처리할 command를 ‘ ‘를 기준으로 parse한 단어들이 저장되어있다. buf에는 앞으로 처리할 나머지 commands들이 저장되어 있다.

우선 pipe함수를 통해 파이프를 생성한 다음, fork를 한다. child process에서는 argv에 대한 처리를 한다. parent process에서는 child process의 작업이 끝날 때까지 기다린다. 이 때 각 pipe는 이전 pipe의 output을 input으로 받고, 자신의 output을 다음 pipe의 input으로 넘긴다. 이후 parsepipe함수를 호출하여 buf를 또다시 ‘|’를 기준으로 자른 뒤 makepipe를 호출하여 재귀적으로 처리한다. 만약 마지막 pipe일 경우, 더 이상 makepipe를 호출하지 않고 곧바로 해당 command를 처리한 뒤 eval함수로 돌아간다.

(6) void check\_jobs()

이 함수는 bg\_job에서 terminate된 job이 있는지 체크하고, 있다면 해당 job을 bg\_job에서 delete해주는 함수이다.

(7) int CMD\_fgbgkill(char \*\*argv, int bg)

이 함수는 fg, bg, kill이라는 built-in command들에 대해 <job>이 무엇을 가리키는지 판별해주는 함수이다.

argv는 해당 built-in command를 ‘ ‘에 대해 parse한 단어들이 저장되어있다. bg는 어떤 command에 해당하는지 나타내는 변수이다. 0이면 fg, 1이면 bg, -1이면 kill command이다. fg, bg, kill은 <job>과 함께 입력받는다. <job>은 %숫자, 또는 그냥 숫자 형태로 입력 받을 수 있다. <job>이 %숫자인 경우, 해당 숫자는 bg\_job의 index에 해당한다. <job>이 그냥 숫자인 경우, 해당 숫자는 pid에 해당한다. 이후 updateJob함수를 호출하여 해당 pid에 대한 처리를 수행한다.

(8) int updateJob(int pid, int bg)

이 함수는 fg, bg, kill이라는 built-in command들에 대한 실행함수이다. pid는 처리해야할 job의 process id를 가리킨다. bg는 처리해야할 command가 무엇인지 나타내는 것으로, -1이면 kill, 0이면 fg, 1이면 bg command에 대한 작업을 처리할 수 있도록 한다.

우선, bg\_job에서 parameter로 받은 pid의 idx를 찾는다. 이 때 주의할 점은, fg, bg, kill command는 bg\_job에 저장되어 있는 background job들을 대상으로 처리된다. 따라서 bg\_job에 없는 pid를 입력 받은 경우, -1을 반환하며, 이 경우 해당 command는 오류이다. 만약 bg\_job에 있는 pid에 접근하는 경우, 각 command에 따른 작업을 처리해준다.

bg == 0인 경우, 이 command는 fg이다. fg는 bg\_job[idx]가 stopped, sleeping상태인 경우, kill(pid, SIGCONT)를 호출하고, waitpid함수를 통해 해당 command가 foreground에서 running되도록 한다.

bg == 1인 경우, 이 command는 bg이다. bg는 bg\_job[idx]가 stopped 상태인 경우, kill(pid, SIGCONT)를 호출하여 background에서 running 되도록 한다.

bg == -1인 경우, 이 command는 kill이다. kill은 kill(pid, SIGKILL)를 호출하여 해당 process를 죽인 뒤, deleteJob()함수를 호출하여 해당 job을 bg\_job에서 제거한다.

(9) void listJobs()

이 함수는 job command에 대한 실행함수이다. bg\_job을 탐색하면서 유효한 background jobs를 화면 상에 출력한다.

(10) void addJob(int pid, char \*cmdline)

이 함수는 bg\_job에 background process를 추가하는 함수이다. pid는 bg\_job에 추가할 background process의 process id이고, cmdline은 해당 pid의 command이다.

(11) void deleteJob(int pid, char type)

이 함수는 bg\_job에서 process를 제거하는 함수이다. pid는 제거할 process의 id이고, type은 “T”인 경우 terminated, “K”인 경우 killed되었음을 나타내기 위한 변수이다.

(12) int getStatus(int pid)

이 함수는 pid에 해당하는 process의 현재 상태를 얻기 위한 함수이다. 현재 상태가 stopped인 경우 0을 반환, sleeping인 경우 1을 반환, running인 경우 2를 반환, zombie인 경우 3을 반환하며, 그 외에는 -1을 반환한다.

(13) int getJobIdx(int pid)

이 함수는 pid에 해당하는 process를 bg\_job에서 찾은 뒤, 그 index를 반환해주는 함수이다. 만약 pid에 해당하는 process가 bg\_job에 없는 경우, -1을 반환한다.

(14) void handler(int signo)

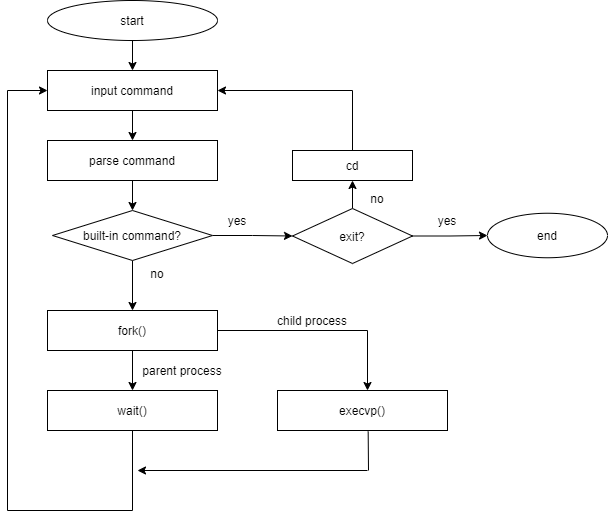
이 함수는 signal handling 함수로, SIGINT, SIGTSTP, SIGCHLD에 대한 signal들을 처리한다.

SIGINT의 경우, 현재 돌고 있는 foreground process를 종료시킨 뒤, deleteJob함수를 호출하여 해당 process를 bg\_job에서 제거한다.

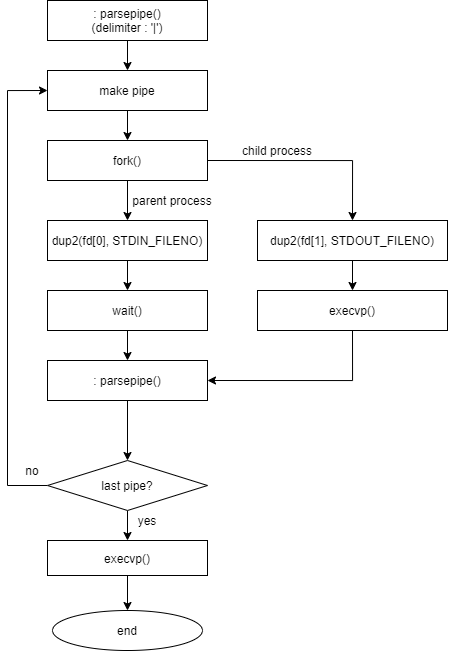
SIGTSTP의 경우, 현재 돌고 있는 foreground process는 stopped시킨 뒤 addJob함수를 호출하여 해당 process를 bg\_job에 추가한다.

SIGCHLD의 경우, 현재 zombie상태의 process들을 reaping해준다.

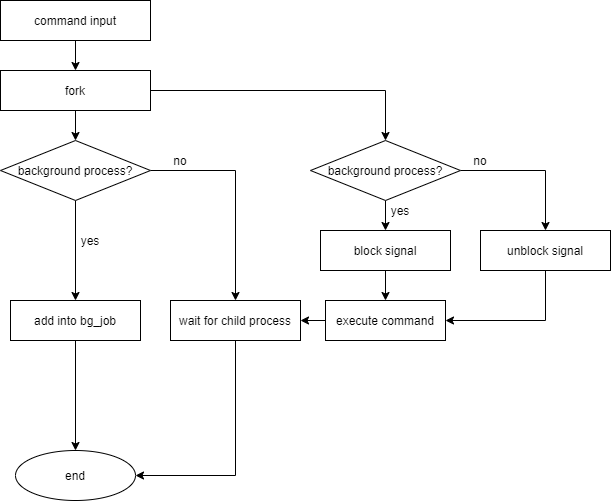
1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****