**Multicore Programming Project 1**

담당 교수 : 박성용

이름 : 고동헌

학번 : 20191564

1. **개발 목표**

* 이번 프로젝트는 Unix Shell 프로그램을 만드는 것이다. 총 3개의 Phase로 구성돼 있으며, 각 단계마다 이전 Phase의 기능을 확장 시켜 나가는 것이 목표다.

Phase1에선 linux 의 기본적인 명령어들인 ls, touch, history, exit등을 fork 와 exec를 이용해 구현해야 한다. 이때 built-in 명령인 cd, history는 /bin에 있는 내장 명령어들을 사용하는 것이 아닌, shell 프로그램 내부에서 처리해야 한다.

Phase2에선 Pipe를 구현해야 한다. Pipe는 데이터가 한 프로세스에서 다른 프로세스로 전달되도록 하며, 이로 인해 프로세스들이 서로 연결된다. pipe(), dup2(), dup() 함수를 이용해서 recursive하게 구현할 수 있으며, pipe의 개수는 제한이 없다.

process들을 foreground에서만 실행시켰던 Phase1, Phase2와는 달리, Phase3에서는 Process를 Background 에서도 실행할 수 있어야 한다. myshell은 이러한 Background job들에 대해 몇 가지 기능을 제공한다. kill, bg, fg, jobs 이 이 예시들인데, 각각은 선택된 job을 종료시키거나, Background에서 running 상태로 만들거나, 해당 job을 foreground로 만들거나, background 에 있는 running, stopped 상태의 job들의 리스트를 보여준다. Fork()를 통해 생성한 process가 종료되길 wait함수로 기다렸던 위 phase들 과는 달리, phase3 에선 sigchld handler를 install해서 background job이 종료되면 reaping 될 수 있도록 해야한다. 이 기능들은 SIGINT, SIGCONT, SIGTSTP 등의 signal 이 발생했을 시 이 signal들을 적절히 catch 하고, kill system call을 이용해 child process한테 signal을 전달하는 방식으로 구현된다.

모든 phase에 공통적으로 적용되는 사항은 SIGINT, SIGTSTP signal이 발생했을 때 shell program이 종료되거나 stop되면 안된다는 점이다. 이를 위해 이 signal에 대한 handler를 만들어 shell을 종료, stop 시키지 않고 다른 특정한 일을 수행하도록 만들어야 한다.

**개발 범위 및 내용**

* 1. **개발 범위**

1. **Phase 1**

* Phase1에선 linux의 기본 명령어들을 사용할 수 있다. 명세서에 나와있는 명령어 들을 예로 들면 ls, mkdir, rmdir, touch, cat, echo, history, exit 등이 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

myshell을 실행한 예시는 위의 그림과 같다. 각 기본 명령어들을 실행했을 때, 실제 shell과 동일하게 동작하는 것을 확인할 수 있다.

1. **Phase 2**

* Phase2에선 Phase1의 기능들과 더불어, Pipe 기능까지 사용할 수 있다. pipe를 사용하면 하나의 pipe에 대해 앞쪽 process의 out이 뒤쪽 process의 in 으로 들어가게 된다. 기존의 shell이 pipe의 개수에 대한 제한이 없는 것처럼, myshell에서도 pipe의 개수에 대한 제한이 없다.

텍스트이(가) 표시된 사진

자동 생성된 설명

차트, 텍스트이(가) 표시된 사진

자동 생성된 설명

Phase2 명세서에 있는 명령어를 실행한 결과는 위와 같다. 이것을 보면 pipe의 개수에 상관없이 pipe의 앞쪽에 있는 process의 out이 그 다음 process의 in으로 가서, 최종적으로 stdout으로 정상적으로 출력되는 것을 확인할 수 있다.

1. **Phase 3**

* Phase3 에선 process들을 background 에서도 실행시킬 수 있어야 한다. 또한, background process들에 대한 여러 기능들도 함께 제공해야 한다. ‘jobs’ 명령어를 통해 현재 background 에 있는 job들의 list를 알 수 있어야 하고, ‘bg’, ‘fg’ 명령어 들을 통해 이 job들을 running 상태로 바꾸거나 foreground로 실행시킬 수 있어야 한다. 또한, kill 명령어를 통해 해당 process group 을 종료 시킬 수 있어야 한다. myshell에서의 실행 결과는 다음과 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

*\* a.out 은 무한 루프를 도는 실행 파일이다.*

* 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

Phase1에선 built-in 명령어가 아닐 경우 fork()를 호출해 child process를 만든 뒤, exec 함수를 실행시켜 process의 이미지를 바꿔서 해당 명령어를 실행한다. 이 부분은 소스 코드의 eval 함수에 구현돼 있다.

eval 함수를 보면, fork()를 호출한 뒤 리턴값이 0이면 child process 에서 실행되는 부분이므로, 여기서 필요한 작업을 수행한 뒤 exec 를 호출한다. 이 때 필요한 작업 중 첫 번째는 SIGINT, SIGTSTP Handler를 Default Handler로 변경하는 것이다. main program인 shell 은 앞의 Signal에 의해 종료되면 안되므로 User defined Handler를 install 해놓은 상태인데, child process의 경우 이 handler를 사용하면 안되므로 다시 Default Handler 로 설정해 주어야 한다. fork()의 경우 csapp.c 에 정의된 wrapper함수를 이용하였다.

두 번째는 Setpgid(0,0) 을 호출해 process group을 생성하는 것이다. SIGINT, SIGTSTP Signal 등의 처리를 위해, child process와 parent process는 서로 다른 process group에 속해야 한다. 이유는 다음과 같다. 만약 SIGINT Signal이 발생한 해당 process group에 SIGINT signal을 보내게 되는데, 만약 parent 와 child가 같은 group 에 속하면 parent process 도 종료되기 때문이다. 이 때 ‘less’명령어는 정상적인 실행을 위해 parent 와 같은 group 에 속해야 한다. 따라서 이를 제외하곤 별도의 process group을 만든다.

마지막은 exec() 를 호출하는 것이다. 이 함수는 fork()를 통해 생성된 child process의 이미지를 parameter의 argv[0] 실행 파일의 이미지로 바꾸어 준다. 이를 통해 우리가 원하는 프로그램을 실행시킬 수 있다. exec 함수는 여러 종류가 있는데, 내가 사용한 함수는 execvp() 이다. 절대 경로를 주어야 실행되는 execve 함수와 다르게, 이 함수는 파일명을 주면 환경 변수에 등록된 PATH에 있는 파일들을 참조한다. /bin, /usr/bin 에 있는 명령어들뿐만 아니라 별도로 만든 a.out 같은 실행 파일도 shell에서 수행할 수 있도록 하기 위해 이 함수를 사용하였다.

child process 가 connection 을 종료할 경우 zombie 상태가 되고, SIGCHLD signal 이 발생한다. 이 signal은 SIGCHLD handler를 별도로 install해서 handling 할 수도 있다. 하지만 phase 1 같은 경우, 오직 foreground process 만 생성하기 때문에 별도로 SIGCHLD Handler를 install 하지 않고, process 를 생성한 뒤 Waitpid를 호출해 reaping 해주었다. 이 함수는 child process 의 connection이 종료될 때까지 호출한 process 를 정지시키고, connection이 종료되면 reaping을 해준다. 이 때 종료되지 않고 stop된 process 까지 처리하기 위해 waitpid 의 parameter 로 ‘WUNTRACED’를 주었다.

앞에서 SIGINT, SIGTSTP Handler 에서 이러한 signal을 받은 child process를 종료시키거나, stop 시키기 위해 shell program 은 항상 ‘foregroundGroup’ 이라는 변수에 현재 foregroundGroup process의 pgid를 저장하고 있다. eval 의 waitpid 앞 뒤로 있는 부분은 이 변수에 대한 처리를 위한 부분이다.

**Phase2 (pipelining)**

* + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)
  + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명
* pipeline 명령어에 대한 처리는 treatPipeLine 함수에서 한다. 대략적인 pipeline 처리 flow는 다음과 같다.
  + 1. pipeline 명령어인지 파악(parseline 에서 isPipe변수에 set)
    2. pipeline 명령어 내부에 !!, !# 등이 있을 시 replaceMark 함수에서 argv에 대해 처리
    3. treatPipeLine 호출
    4. 반복문에서 pipeline 의 마지막 명령어인지에 대한 flag set, 각 명령어에 대한 개별적인 subArgv 구성, argv에서 다음 명령어 위치 저장.(nxt\_idx 변수)
    5. *if 마지막 명령어라면*
       - fork(), exec() 를 통해 해당 명령어 수행 후 종료

*if 마지막 명령어가 아니라면*

* + - * pipe() 함수를 호출해 pipe 라인을 만들고, 해당 process의 output을 만든 pipeline의 writable end로 연결해 줌. 그 후 treatPipeLine 함수를 재귀적으로 호출하며 parameter로 만든 pipeline의 readable end, nxt\_idx를 넘겨줌.

treatPipeLine 함수에는 parameter로 readableEnd를 넘겨준다. 이는 재귀적으로 호출하는 과정에서, 바로 전 단계에 만든 pipe에 대한 readable End이다. 해당 단계에서 만든 process에 대해 자신의 STDIN\_FILENO을 readableEnd로 바꾸면 입력을 stdin이 아닌, 전에 생성한 pipeline을 통해 받을 수 있다. 또한, pipeline의 마지막 명령어가 아닌 경우엔 자신의 output을 생성한 pipe line에 내보내야 한다. 따라서 STDOUT\_FILENO을, 만든 pipe의 writable end로 바꿔주었다. file Descriptor값 변경은 dup2() 함수를 사용하였고, pipeline의 생성은 pipe() 함수를 사용하였다.

myshell에서 pipe의 개수는 제한이 없다. 이에 대한 처리는 앞에서 설명한 것처럼 treatPipeLine 함수의 재귀적인 호출을 통해 handling 하였다. 함수의 초반에 argv[idx] 부터 하나씩 보면서, 아직 pipeline이 남아있을 경우 다시 이 함수를 호출한다. pipeline이 없다면 명령어를 실행하고 종료한다.

* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명
* foreground process의 경우 fork()를 한 뒤, wait 함수를 호출해 이 process 가 종료될 때까지 기다린 후 reaping 해주었다. 반면 background process의 경우 foreground는 myshell이어야 하기 때문에, wait함수를 호출하지 않는다. 종료되는 process에 대한 reaping은 SIGCHLD handler를 install함으로써 해결하였다.

background process를 생성해야 하는 경우, fork() 와 exec() 함수를 호출하는 것은 똑같지만 foreground와 달리 signal에 대한 explicit waiting 을 하지 않는다. 단지 이 process를 jobList에 추가해주고, shell은 다음 command를 받는다. 대략적인 flow는 다음과 같다.

(i) background process(&) 인지 파악

(ii) fork(), exec() 호출을 통해 process 생성

(iii) explicit waiting을 하지 않고, jobList에 해당 job 추가 후 다시 shell 입력대기

(iv) background process 종료 시 SIGCHLD Handler에 의해 자동으로 reaping 후 jobList에서 해당 job 제거

(+) jobs, bg, fg, kill 등의 명령어 입력 시 해당 background job에 대한 처리

(i) 항목의 구현은 parseline 함수에 돼있다. 함수의 마지막 부분에서 명령어의 끝에 ‘&’ 가 있는지 확인하며, 있다면 bg flag를 1로 set 해준다. (ii) 항목은 foreground process 생성과 동일하다. eval 함수에서 fork() 와 exec() 함수를 호출해서 구현하였다. (iii)도 eval 함수 내부에 구현돼있다. if-else 부분을 보면 foreground process의 경우 Sigsuspend 함수를 호출해 explicit waiting을 하고 있지만, background process의 경우 waiting 하는 부분 없이 addjob 함수를 호출해 jobList에 추가만 해주고 있다.

(iv) 의 경우 child\_handler 함수에서 처리한다. 이 handler는 SIGCHLD signal이 발생했을 때 SIGCHLD signal 이 block 돼있지 않은 경우 호출되는데, 함수 내부에서 waitpid 함수를 통해 종료된 background process 를 reaping 해준다. 또한, reaping 할 때 deletejob 함수를 호출해 해당 process를 jobList에서 제거해준다.

(+) 항목은 job구조체 array를 통해 job들을 관리하고, 각각의 명령어들에 대응되는 함수를 정의해 해결하였다. process를 생성할 때, foreground process, background process 구분없이 이를 jobList에 추가해준다. foreground process 까지 jobList에 추가하는 것은, 추후 이 process가 signal을 받아 background job으로 변경될 수도 있기 때문이다. 명령어들을 수행하기 위해 특정 job number나 pid 에 대응되는 job이 필요한 경우가 있는데, 이는 getBgJob, getJob함수에 구현 돼있다.

또한, getJobNumberFromQuery 함수는 bg %4, fg %5 등과 같은 명령어가 입력 됐을 때 뒤의 숫자를 계산해 저장하고, 해당 숫자가 옳은 숫자인지 판단해주는 함수이다.

* 1. **개발 방법**

**1. Phase1 --> Phase2**

- 우선, phase1에서 phase2 확장 시키면서 추가된 함수는 void treatPipeLine(), int replaceMark(), int parseSubArgv() 함수이다. treatPipeLine() 함수는 phase1 에서 처리하지 않은 pipeline에 대한 처리를 해주는 함수이다. pipeline이 끝날 때까지 재귀적으로 호출되면서 주어진 pipeline 을 처리한다. replaceMark() 함수는 pipeline 명령어 내부에 !!, !# 등이 있을 때 argv를 실제 historyLog 에 있는 명령어로 바꿔주는 함수이다.

parseSubArgv() 함수는 pipeline 에서 subArgv 구성 시 사용되는 함수이다. replaceMark() 함수에서 대체된 명령어는 ‘cat historyLog.txt’ 등과 같이 내부에 공백이 있는 문장도 하나의 argv에 넣어준다. 하지만 실제 명령어를 실행할 땐 cat, historyLog.txt 가 subArgv에 하나씩 저장돼 있어야 한다. 이 작업을 해주는 함수가 parseSubArgv 함수이다.

추가된 변수는 int isPipe, int saveStdin 이다. parseline함수에서 cmdline을 parsing 하면서 해당 명령어가 pipeline인 경우 isPipe 변수를 set 해준다. saveStdin 변수는 각 process의 in, out 을 redirection 하는 과정에서 필요한 변수이다. treatPipeLine 함수에서, parent process의 경우 argument로 들어온 readableEnd, 새롭게 만든 pipe의 readable end는 필요하지 않기 때문에 이를 close 한다. 하지만 처음 함수 호출 시의 readableEnd argument는 stdin file descriptor 이므로, treatPipeLine을 호출하기 전에 STDIN\_FILENO을 저장해 두어야 한다. 이 값의 저장을 위한 변수가 saveStdin 이다.

수정된 함수는 void eval, int parseline 함수이다. parseline 함수엔 입력된 명령어가 pipeline 명령어인지 확인하는 부분이 추가됐다. 그리고 eval 함수 같은 경우 pipeline 명령어와, pipeline이 아닌 명령어를 처리하는 두 부분으로 분리됐다.

**2. Phase2 --> Phase3**

- phase3로 확장 시키면서 추가된 부분 및 수정된 부분이 많다. 우선 추가된 매크로, 함수 및 변수를 정리하면 다음과 같다.

**(i) 매크로**

- BG, FG, ST, TT, MAXJOBS

**(ii) 변수 및 구조체**

* 1. job structure
* 2. struct job jobList[MAXJOBS]
* 3. int jobSize, int sequence
* 4. volatile sig\_atomic\_t hpid

**(iii) 함수**

- 1. void child\_handler(int sig)

- 2. void addjob(pid\_t pid, char\* cmdline, int state)

- 3. void deletejob(pid\_t pid)

- 4. void mybg(char \*\*argv)

- 5. void myfg(char \*\*argv)

- 6. void mykill(char \*\* argv)

- 7. void printJobs(char\*\* argv)

- 8. struct job\* getJob(pid\_t pid)

- 9. struct job\* getBgJob(int jobNumber)

- 10. int getJobNumberFromQuery(char \*\*argv, char \* cmdline)

**(i) 매크로**

- 매크로 작업에 추가된 값들은 BG, FG, ST, TT, MAXJOBS가 있다. 앞의 4개들은 jobList에 들어온 job의 현재 상태를 나타낸다. MAXJOBS는 jobList 에 들어올 수 있는 job 의 최대 개수이다. 만약 더 많은 job이 필요하다면, 이 값을 수정하면 된다. 관리할 수 있는 job의 개수엔 제한이 있지만, background process의 생성개수에는 제한이 없다.

**(ii) 변수 및 구조체**

* job structure 의 구조는 다음과 같다.

*struct job{*

*int sequence;*

*pid\_t pid;*

*int state;*

*char cmdline[MAXLINE];*

*}*

구조체 member인 sequence는 background job인 경우만 양수의 값을 갖는다. foreground job의 경우 jobList에 추가될 때 0값을 갖게 된다. pid\_t 는 해당 job의 pgid이다. state는 해당 job의 state이다. 위의 BG, FG, ST, TT 매크로 값을 가질 수 있다. cmdline은 해당 background process를 실행했을 때의 명령어를 저장하고 있다.

struct job jobList[MAXJOBS] 는 모든 job들을 저장하고 있는 array 이다. jobSize는 현재 jobList의 크기를 저장하고 있다. sequence는 background job이 새롭게 들어올 경우 부여할 job의 번호이다. 이 값은 새로운 background job이 들어올 경우에만 증가한다.

hpid 같은 경우 foreground process 들을 reaping 할 때 사용한다. sigchld handler에서 정상적으로 종료된 process를 reaping 하는 경우, hpid 가 해당 process의 pid 값으로 변경 돼 while loop를 나갈 수 있다.

**(iii) 함수**

* child\_handler 함수는 SIGCHLD signal 이 발생했을 때, 종료된 process를 reaping 하고 해당 job을 jobList에서 제거해준다. background process를 정상적으로 reaping하기 위해 추가됐으며, phase3 에서는 foreground process도 이 handler를 통해 reaping 된다.

addjob 함수는 process를 생성할 때마다 해당 process를 jobList 에 추가해주는 함수이다. argument에 따라 각 job에 맞는 sequence와 state 부여하고, jobList에 추가해준다. deletejob 함수는 jobList에서 pid가 argument pid 와 같은 job을 제거해주는 함수이다. jobList 가 array로 구현 돼있으므로 직접 제거하지 않고, state 만 TT로 바꿔준다.

mybg, mykill 함수는 ‘bg’, ‘kill’ 명령어를 입력할 때 주어진 jobNumber 에 해당하는 background process 를 running 상태로 만들거나, terminate시키는 함수이다. getJobNumberFromQuery 함수를 통해 jobNumber를 알아낸 뒤, getBgJob 함수를 통해 해당 jobNumber에 해당하는 job의 주소를 얻는다. 그 후 job pid 에 SIGCONT, SIGKILL signal을 줌으로써 명령을 수행한다.

myfg 함수도 위의 함수와 비슷하지만, 이 함수는 해당 job을 foreground로 만들어야 한다. 따라서 해당 job에게 SIGCONT signal을 보낸 뒤, foreground group의 pgid를 저장하고 있는 foregroundGroup 변수의 값을 해당 job의 pid 로 변경 하고 waitpid 함수를 호출해 해당 job이 종료되거나 끝날 때까지 shell program을 wait 시킨다.

printJobs 는 ‘jobs’ 명령어를 구현한 함수이다. jobList를 처음부터 하나씩 보면서 state 가 BG, ST인 job들을 하나씩 출력해준다.

getJob 함수는 argument로 넘어온 pid에 맞는 job의 주소를 return 해주고, getBgJob 함수는 jobNumber 에 맞는 job의 주소를 return 해준다.

getJobNumberFromQuery 함수는 bg, fg, kill 명령어와 함께 주어진 jobNumber를 저장하고, 해당 jobNumber 가 옳은 값인지 판단한 뒤 이 값을 return 해주는 함수이다.

**(iv) 수정된 부분**

* 우선, foreground process들을 reaping 하는 부분이 모두 수정됐다. foreground process 인 경우, 기존에는 shell 에서 waitpid 함수를 호출하였다. 이젠 모든 reaping이 child\_handler 에서 일어난다. foreground process 의 경우 shell 에서 해당 process 가 reaping 될 때까지 wait 해주어야 하는데, 이는 sigsuspend 와 hpid 를 이용해서 해결했다. hpid=0 이 돼서 while loop에 들어가고, process 가 reaping 되면 hpid 가 0이 아닌 다른 값으로 변경 돼 종료조건을 만족한다. 그 후 foregroudGroup 의 값을 0으로 변경한다. 이 때, addjob 함수를 호출하거나 foregroudGroup 변수의 값을 변경하는 경우 항상 block 을 해줘 race condition이나 handler의 interrupt로 인한 예기치 못한 상황이 발생하지 못하도록 하였다.

‘jobs’, ‘bg’, ‘fg’, ‘kill’ 등의 built-in 명령어들이 새로 생겼기 때문에, builtin\_command 함수에서 이를 처리할 수 있도록 수정하였다. 또한, treatPipeLine 함수도 수정됐다. child process 의 reaping 방식이 변경됐고, job을 추가하는 부분이 생겼기 때문이다.

sigint\_handler 함수도 변경됐다. sigint\_handler 의 경우 특정 foreground process 를 종료시키고 내부에서 foregroundGroup의 값을 0으로 변경했다.(이 값이 0이면 foreground process 가 shell 이라는 의미)

phase1, phase2에서의 sigtstp\_handler는 단지 child process에게 SIGTSTP signal 보냈었다. 하지만 phase3에선, foreground process 가 stop 될 경우 해당 process job의 state를 바꾸고, 이를 jobList에 새롭게 추가해야 한다. 이러한 작업을 해주는 부분이 추가적으로 구현돼있다.

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

**도표이(가) 표시된 사진

자동 생성된 설명**

1. **Phase 2 (pipeline)**

도표이(가) 표시된 사진

자동 생성된 설명

1. **Phase 3 (background)**

**도표이(가) 표시된 사진

자동 생성된 설명**