**Multicore Programming Project 1**

담당 교수 : 박성용

이름 : 임형준

학번 : 20191638

1. **개발 목표**

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

이번 프로젝트는 실제 linux shell을 제한적으로 구현하는 것이다. 지금까지 배운 unix 명령어와 signal control들을 이용하여 실행시키는 것이 목표이다.

총 세 가지 phase로 이루어져 있는데, 첫 번째 단계를 shell의 기본적인 명령 과정인 read – evaluation 을 구현하는 것이다. 두 번째 단계를 IPC를 구현하는 파이프라인을 구현하는 것이다. 세 번째 단계는 shell에서 process를 background로 실행시키는 것을 구현하는 것이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

linux shell의 기본적인 flow인 read – evaluation 을 구현하는 단계이다. execvp에서 지원하는 내장 함수들인 ls, ps, mkdir, touch 들과 cd, history 같은 built-in 명령어들의 실행이 보장된다. 또한 다양한 옵션들을 명령어 뒤에 추가 입력하여 실행시킬 수 있으며, 올바르지 않은 명령어 입력은 error message를 출력한다.

1. Phase 2

linux shell의 IPC 기능을 수행하는 파이프라인 기능을 구현하는 단계이다. 사용되는 파이프의 개수에는 제한이 없으며 파이프에 공백 유무에도 상관없이 실제 shell에서 작동하는 것처럼 실행이 보장된다.

1. Phase 3  
   signal handling을 토대로 background process를 구현하는 단계이다. &을 명령어 뒤에 입력하면 background로 인식해 작동시키게 되며, jobs 명령어로 현재 background process들의 cmdline, state 등을 알 수 있다. 또한 bg, fg, kill 명령어의 구현을 통해 background의 상태를 자유롭게 변경할 수 있으며, ctrl+z , ctrl+c를 입력했을 때, foreground process를 종료 시키거나 suspended 시킬 수 있도록 구현하였다.
   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
* fork를 하기 전에 입력된 문자열인 cmdline을 parseline에서 argv로 재배열한다.
* argv가 built-in-command인지를 확인하여, built-in일 경우에 fork를 호출하지 않고 별도의 subroutine을 호출하여 과정을 거쳐 결과를 출력하고 main에서 다시 명령어를 입력 받는다.
* 하지만 built-in-command가 아닐 경우, fork를 하여 자식 프로세스를 생성한다.
* 자식 프로세스에서는 execvp()를 실행하여 명령어를 실행하고 exit()로 종료한다.
* 부모 프로세스에서는 wait() 함수를 통해 자식 프로세스의 종료를 확인하고 reaping하는 과정을 거친다.
  + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow

우선 ctrl +c와 ctrl +z로 foreground process를 종료 혹은 일시정지 하기 위해, user-signal-handler를 사용한다. 프로세스가 생성되면 배열에 pid와 state, 그리고 cmdline등을 저장하고 있다가. shell에서 ctrl+c 혹은 ctrl+z가 입력됐을 때, handler에서 SIGINT 혹은 SIGTSTP을 catch하여 일련의 처리 과정을 수행하게 되고 프로세스가 종료 되거나 background에서 suspended된 상태로 변하게 된다.

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

main에서 ls -al | grep ‘.c’라는 명령어가 입력되었다고 가정하자. parseline 함수에서 pipe의 개수를 계산하는 과정이 있으며 이 값을 통해 loop를 돌면서 특정 파이프 앞에 위치한 명령어들을 다시 parsing한다. cmdline [0]에는 ls와 -al가 저장되고 cmdline[1]에는 grep과 ‘.c’이 저장된다. 그 이후의 과정은 phase1의 과정과 비슷하며 built-in command인지를 확인한 후, built-in command라면 이름이 일치하는 명령어를 찾아 실행하고 아니라면 execvp를 통해 실행한다. pipeline의 핵심은 결과를 바로 출력하는 것이 아니라 다음 명령어의 입력으로 사용해야 한다는 것이다. 이를 구현하기 위해 file descriptor를 pipenum만큼 동적 할당하였고 pipe(), dup2() 함수들을 이용해 파이프를 연결해주었다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

exec\_pipe() 함수에서 바로 명령어를 실행하는 것이 아니라 argv를 파이프를 기준으로 다시 parsing하는 과정이 있다. local variable인 cmdline을 살펴보면 삼차원 배열인 것을 알 수 있는데 맨 앞 인덱스는 특정 파이프의 인덱스이고 특정 파이프 이전의 명령어들을 저장하게 된다. parsing이 끝나고 나면 이러한 명령어들을 순서에 맞게 실행시키면서 파일 디스크럽터에 명령어의 결과를 읽기 혹은 쓰기를 진행한다. 맨 첫 번째 명령어는 파이프에 쓰기만 하게 되고, 마지막 명령어는 파이프에서 읽기만 한다는 것을 주의해야 한다.

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

background process의 정의를 다시 살펴보자. shell에서 process를 생성할 때 foreground로 생성을 하게 되는데, shell은 이 process가 완전히 종료된 이후에 다시 입력값을 받고 작업을 하게 된다.

background process는 이러한 과정없이 쉘은 계속해서 작동되어 입력값을 받고 작업을 수행하며 background process들은 이와 별개로 작업을 수행한 뒤 종료된다. 생성에 대한 부분을 살펴보면 parseline에서 명령어의 끝에 &가 있다면 bg의 값을 1로 변경한다.

이를 통해 우리는 해당 명령어를 background로 실행시켜야 함을 알 수 있다. built-in command인지 확인 한 후 아닐 때, bg가 true라면 background로 실행시켜야 하므로 부모 프로세스에서 wait routine을 하지 않는다.

이렇게 하면 해당 자식 프로세스를 reaping하기 위해 wait를 하지 않으므로 부모 프로세스는 다시 main으로 돌아가 새로운 명령어를 받게 될 것이고, 자식 프로세스는 작업이 끝나면 exit를 통해 자신의 작업이 끝났고, 종료될 것임을 SIGCHLD를 보내면서 알리게 된다. 이를 처리해주는 default handler들도 존재하지만 이 쉘에서는 background process의 정보들을 별도의 자료구조로 관리하기 때문에 user-signal-handler를 구현한 뒤 설치하여 process를 관리하였다. 또한 built-in command 함수에 bg, fg, kill 같은 명령어를 구현하여 이러한 명령어가 작동하도록 하였다.

* 1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

history에서 이전에 입력된 적 있는 명령어들로 치환하는 명령어인 !! 과 !n을 구현하기 위해 file stream을 통해 이전에 입력된 명령어들을 저장하는 파일인 command.log를 생성하고 연결하였다. eval 함수가 호출되면 fopen을 통해 해당 파일을 생성 혹은 실행하게 되고. cmdline을 for문을 통해 분석하게 된다. cmdline에 !!나 !n이 입력되었다면 file stream을 통해 파일의 포인터를 이동해가며, 파일 마지막 줄에 위치한 명령어를 읽거나 n번째 줄에 있는 명령어를 읽어 temp\_str에 저장한다. 모든 명령어가 치환되면 temp에 치환된 명령어들을 저장하고 일련의 과정이 모두 끝나게 되면 temp에는 이전의 기록에서 읽은 명령어들로 치환된 명령어가 저장되게 된다. 이를 다시 command.log에 저장하여 log를 관리하였으며, 예외 처리 사항인 직전 명령어가 현재 log file에 쓸려는 명령어와 일치한다면 이를 쓰지 않도록 문자열 비교 함수인 strcmp가 사용되었다.

파이프라인을 구현하기 위해 exec\_pipe에서는 file descriptor를 저장하기 위한 이차원 배열 fd를 사용하고 있다. 이 변수는 pipe의 개수만큼 동적 할당을 받는다.

pipe함수는 int[2]인 배열을 인자로 받는데, fd[2]를 넘겨줬다면, fd[0]은 읽기, fd[1]은 쓰기에 해당하는 fd값이 저장된다. 이를 통해 dup2()로 stdin과 stdout인 0 과 1에 pipe의 fd을 복사하여 명령어의 출력이 파이프 다음에 위치한 명령어의 입력 값으로 들어가도록 한다. 이러한 일련의 과정을 통해 파이프라인이 구현된다.

process를 관리하기 위한 변수는

pid 와 fpid로 나뉘며 pid는 background, fpid는 foreground의 정보를 저장하는 배열 혹은 변수이다.

pid를 예로 들어보면 pid\_arr[]는 bg process의 pid값을 저장하며, pid\_cnt는 다음 들어갈 프로세스의 인덱스 값을 의미한다. pid\_com은 char 이차원 배열로 각 pid에 해당하는 명령어를 저장한다. 이러한 변수를 통해 fg, bg, kill 명령어와 ctrl+z, ctrl+c을 통한 상태 변화 타이핑이 적절하게 상태를 변경하도록 한다.

handle\_sigint는 SIGINT를 처리하는 signal handler이다.

로직을 통해 값을 처리하기 전에, 다른 signal의 간섭을 막기 위해 sigprocmask로 모든 signal를 block하였다.

이 작업은 모든 handler에 동일하게 적용되었다.

ctrl+c가 입력되어 handle\_sigint가 실행되면 모든 foreground process가 종료되며 이를 구현하기 위해 kill 함수를 사용하였다. 그 이후 fpid\_arr의 값을 -1로 변경하여 종료된 process임을 pid 값을 통해 알 수 있게 하였다.

handle\_sigtstp는 SIGTSTP를 처리하는 signal handler이다.

ctrl+z가 입력되어 handle\_sigtstp가 실행되면 모든 foreground process가 일시중지된 상태(suspended)가 되어 background에서 실행되도록 변경한다. 즉 foreground에서 background로 이동시켜 중지시킨다. 이를 구분하기 위해 fpid에 저장되어 있던 값들을 pid인 변수들에 저장하고 fpid\_arr의 값을 -1로 변경하여 foreground에 더 이상 없는 process임을 알 수 있도록 하였다.

handle\_sigchld는 SIGCHLD를 처리하는 signal handler이다.

background process가 종료되면 handle\_sigchld가 호출되며 wait 함수를 통해 reaping해주는 과정을 거친다. while((pid=waitpid(-1,NULL,WNOHANG))>0)을 통해 handler 실행 도중에 SIGCHLD이 더 들어온다면 discard 하지 않도록 추가적으로 waitpid 함수가 실행되어 모두 reaping 될 수 있도록 하였다. waitpid가 정상적으로 실행되면 pid\_arr[]의 값을 -1로 변경하여 background에서 더 이상 실행되고 있지 않은 process임을 알 수 있게 하였다.

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)**

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