**Multicore Programming Project 1**

담당 교수 : 박성용 교수님

이름 : 이수빈

학번 : 20200422

1. **개발 목표**

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

사용자로부터 키보드 상으로 명령을 입력 받고 이를 실행해주는 shell 프로그램을 일부 구현한다. 기초적인 명령어부터, pipelining, signal handling, background process와 관련된 명령어 등을 구현하는 것을 목표로 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

기초적인 shell 명령어들을 구현한다. 예시로는 cd, ls, mkdir, rmdir, touch, cat, echo, history, !!, !#, exit 등이 있다. Builtin commands의 경우에는 직접 작성한 코드를 통해 실행되며, extern functions의 경우에는 부모 프로세스의 fork를 통해 생성된 자식 프로세스에 의해 실행된다.

1. Phase 2

2개 이상의 명령어를 pipe(‘|’)와 함께 사용하면, 앞 명령어의 output을 다음 명령어의 input으로 사용할 수 있도록 한다. 이는 부모 프로세스가 pipeline의 각 명령에 대하여 자식 프로세스를 생성하고, 각 자식 프로세스는 dup2()라는 system call을 통해 입출력이 연결 및 전달되면서 수행된다.

1. Phase 3

명령어에 ‘&’가 붙으면 background로 실행한다. Foreground job, background job 간의 이동과 RUNNING, STOPPED, TERMINATED 등의 프로세스 상태를 제어할 수 있는 명령어인 jobs, bg, fg, kill을 구현한다. 또한, background process가 종료되었을 때 이를 reaping할 수 있어야 한다.

* 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

사용자로부터 키보드 상으로 입력받은 명령어를 실행할 때, builtin command의 경우 직접 구현한 함수를 바로 실행한다. 이와 달리 extern function의 경우 부모 프로세스의 fork를 통해 생성한 자식 프로세스에서 명령을 실행한다. 해당 명령어가 builtin command인지, 혹은 extern function인지는 shell에서 which 명령어로 확인할 수 있다. Fork의 반환값으로 부모 프로세스는 자식 프로세스의 pid를, 자식 프로세스는 0을 반환받는데, 이를 통해 부모와 자식 프로세스를 구분할 수 있으며, 부모 프로세스는 자식 프로세스가 끝날 때까지 기다려 reaping한다.

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

connection을 종료할 때 자식 프로세스는 부모 프로세스에게 SIGCHLD라는 signal을 보낸다. 이를 handling하기 위해서는 부모 프로세스에 SIGCHLD handler를 등록해놓고 그 이후에 fork를 진행한다. SIGCHLD handler에서는 waitpid 함수를 이용해 자식 프로세스가 종료됐을 때 reaping할 수 있도록 하며, 이 작업이 끝나면 signal flag를 통해 끝났음을 알리고 부모 프로세스의 suspend 상태를 멈춘다.

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

크게 첫 파이프와 마지막 파이프, 그리고 그 사이의 파이프들을 처리하는 세 함수를 구현하였고, 각 프로세스는 fork를 하여 순차적으로 실행하였다. 각 파이프 간에는 dup2 system call을 이용해 이전 프로세스의 output이 다음 프로세스의 input으로 실행되도록 하였다. 파이프라인을 모두 실행한 후 마지막에 wait 함수를 이용해 수행이 끝난 child process를 모두 reaping하였다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

Pipeline 개수에 따라 handling을 달리 하지 않고, 앞에서 설명한 바와 같이 동일하게 작업을 수행하였다. 다만, 명령어를 parsing하는 과정에서 pipe의 개수를 세어 pipe\_count라는 변수에 저장해두고, 명령어 내에 파이프가 존재하는 경우와 존재하지 않는 경우를 나누었다. 파이프가 존재하지 않을 때에는 phase1과 동일하게 수행하였으며, 파이프가 존재하는 경우에는 앞서 설명했던 파이프 동작 구현 방식에 따라 수행하였다.

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

job\_entry\_t라는 구조체를 만들어 job을 관리하였으며, 구조체 내에 background process에 부여되는 jid(job id)와 각 프로세스의 state를 저장하여 foreground와 background 간의 전환과 running, stopped, terminated 등 프로세스의 state 변화를 구현하였다. foreground process는 프로세스가 끝날 때까지 부모 프로세스에서 wait 후 reaping하였으며, background process 또한 프로세스가 끝난 후 reaping을 수행해야 하기 때문에 명령어를 받기 직전에 waitpid를 통해 zombie process를 처리하였다.

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

**Phase1**

1. myshell.c

Main 함수가 있는 파일로, (char \*)cmdline에 명령어를 입력받는다.

History를 저장하기 위한 history.txt를 열고 닫는 작업 또한 한다.

2. evaluator.c

순차적으로 parser와 executer를 실행한다.

3. parser.c

(char \*)cmdline을 (char \*\*)argv에 옮겨담는다. 그 과정에서 명령어를 실행하기 좋게 정제한다. 고려사항은 다음과 같다. 첫째, 마지막에 입력되는 개행문자는 제거한다. 둘째, ‘|’와 ‘&’ 문자는 이후 Phase2와 Phase3에서의 확장을 위해 사이에 공백을 삽입해 식별한다. 셋째, 공백은 개수와 상관없이 무시한다. 넷째, 따옴표를 처리한다. 짝이 맞는 따옴표는 무시하고, 짝이 맞지 않는 따옴표는 해당 따옴표가 입력될 때까지 사용자로부터 추가적으로 명령어를 입력하도록 한 후, 마지막에 이를 실행한다. 예를 들어, “ls” 혹은 ‘l’s는 따옴표의 짝이 맞기 때문에 ls와 동일한 수행을 보이지만, “l”s’와 같은 경우에는 ‘가 다시 입력될 때까지 추가적으로 사용자에게 입력을 요구한다. 다만 “l’s”와 같이 짝이 맞는 따옴표 사이에 짝이 맞지 않는 따옴표가 존재하면, 짝이 맞는 따옴표만 무시되고 추가적인 입력은 받지 않은 상태로 실행한다. 즉, 해당 예시는 l’s와 동일한 수행을 보인다.

4. executer.c

해당 명령어가 builtin command이면 직접 작성한 코드를 통해 수행한다. builtin command가 수행되면 1을 반환하고, 그렇지 않으면 0이 반환되며, 따라서 0이 반환되었을 때 extern funtion으로 취급하여 fork를 실행한다. 그 다음, PATH라는 환경변수에서 ‘:’를 기준으로 구분되어 있는 경로들을 하나하나 명령어에 붙여가며 작동할 때까지 execve 함수를 실행해본다. 만약 모든 경로에서 실패하면 해당 명령어는 존재하지 않는 명령어로 취급하여 에러 메시지를 띄우고 종료한다.

**Phase2**

5. pipe.c

명령어 내의 pipe 개수를 세어 pipe\_count에 저장한 후, pipe\_count가 0이면 noPipe라는 함수로 Phase1과 동일하게 수행한다. 만약 pipe\_count가 0보다 크면 isPipe라는 함수로 파이프라인을 수행한다. 먼저, (int \*\*)fd라는 배열을 만든다. 이는 이후 사용하지 않는 fd를 close할 때와 dup2를 실행할 때 사용된다. 하나의 pipeline을 기준으로 fd[0]의 두 칸은 각각 앞 프로세스의 input fd와 output fd이며, fd[1]의 두 칸은 각각 뒤 프로세스의 input fd와 output fd이다. 단일 파이프이든 다중 파이프이든 동작은 동일하게 각 프로세스가 fork를 통해 작동을 시작하며, pipeline을 기준으로 앞 프로세스의 output이 다음 프로세스의 input으로 연결된다. 이렇게 시행되고 끝난 child process들은 가장 마지막에 wait 함수를 이용해 (pipe\_count + 1)개만큼 한꺼번에 reaping되도록 한다.

**Phase3**

6. job\_entry\_t

Phase3에서 구현해야 하는 동작들을 위한 구조체로, 각 job의 pid(process id), jid(job id), state(UNDEF, BG, FG, ST, DONE), cmdline이 저장될 계획이며, next로 서로 연결되어 있는 형태이다. Job\_entry pointer 형식을 가지는 job\_front와 job\_last를 별개로 선언하여 사용할 예정이며, job\_count 변수에 job의 개수를 저장할 것이다.

7. executer.c

앞서 Phase1에서 설명한 바에 추가하여, jobs, bg, fg, kill이라는 builtin commands가 추가적으로 구현된다.

- jobs 명령어는 지금까지 job entry에 등록된 job들을 나열하여 출력한다.

- bg 명령어는 인자가 없을 경우 가장 마지막으로 stopped되어있는 job을 찾아 SIGCONT 시그널을 보낸다. 인자가 있을 경우에는 주어진 job id에 해당하는 job의 job id, process id, command line을 출력하고 SIGCONT를 보낸다.

- fg 명령어의 동작 형식은 bg와 매우 유사하고, 동작 방식은 bg와 반대이다. stopped된 job, 혹은 background 중 가장 최근 job을 찾아 foreground로 작업을 돌려준다.

- kill 명령어는 인자로 pid를 입력하거나, %pid를 입력할 수 있다. 만약 이를 통해 job이 stopped되면 job id와 process id, command line을 출력하고 수행을 종료한다.

8. job.c

위 명령어 구현을 위해 job을 제어하는 함수들을 작성한 파일이다. 새로운 job을 job entry에 추가하는 addJob, 상태를 업데이트해주는 updateJob, process id로 해당 job을 찾아주는 getJob, 기존 job을 job entry에서 제거하는 deleteJob, jobs 명령어 실행 시 job entry 항목 나열 및 출력을 위한 printJob이 있다.

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