**System Programming Project 2**

담당 교수 : 이영민

이름 : 김정환

학번 : 20211522

1. **개발 목표**

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

리눅스에서 사용자로부터 명령어를 입력 받아 수행하는 프로그램을 구현해보는 프로젝트이다. 이는 3가지의 phase로 나뉘어서 단계별로 구현하며, 이는 아래와 같다.

우선 phase 1은 하나의 명령을 foreground에서 수행하는 기능을 구현하는 단계이다. 해당 기능을 구현하기 위해서는 먼저 사용자의 입력을 띄어쓰기를 기준으로 나눠서 저장하는 과정이 필요하다. 이 후 fork를 통해서 Child process를 생성하고 종료 시에 Parent process로 돌아오도록 한다.

다음으로 phase 2는 |를 이용하여 명령어의 결과를 다음 명령어의 입력으로 사용할 수 있도록 하는 기능을 구현한다. 이 때는 File descriptor를 이용하여 읽기와 쓰기를 열고 닫아 전달하는 과정을 이용한다.

마지막으로 phase 3는 &가 끝에 있어 background에서 실행되는 명령을 구현하고, fg, bg, kill, jobs를 통해 프로세스를 foreground, background로 이동시키거나 signal로 조정하는 과정과 존재하는 background process들의 목록을 보는 기능을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

Phase 1은 shell의 기본적인 단일 명령어들이 Child process에서 실행되고 다시 Parent process로 돌아올 수 있도록 하는 것이 목적이다. 이 때 실행 결과는 기존의 shell과 동일한 결과를 보여야 한다. main 함수에서 while loop를 돌며 계속 입력을 받아 명령어를 출력하고 exit이나 quit를 입력 받으면 종료한다. 없는 명령어들에 대해서는 예외처리를 해주는 결과가 나오고 cd와 같은 exec으로 실행하지 못하는 명령어들에 대해서는 따로 처리해주어 실행되도록 구현한다.

1. Phase 2

Phase 2는 pipe를 통해서 명령어의 결과를 다음 명령어의 입력에 전달할 수 있는 기능을 구현하는 단계이다. 이 때 | 기호를 기준으로 나누고 다음 명령어로 결과를 전달하도록 하는데, 이 때 phase 1에서 구현한 단일 명령어들이 실행되는 것은 똑같이 실행되는 것은 유지되어야 한다. 또한 각각의 명령어에서 명령어에 옵션을 넣은 것도 모두 그에 맞게 동작해야 한다.

1. Phase 3

Phase 3에서는 background에서의 실행을 구현하여 최종적으로 myshell을 완성한다. 사용자의 명령어 뒤에 &를 붙여서 하는 경우나 띄어쓰기를 하고 넣는 경우 모두 background에서 명령을 실행한다. 다음으로 이전 phase에서 구현하지 않은 jobs, fg, bg, kill 명령어를 추가로 구현한다. jobs는 background에서 Running 중인 process와 stopped 된 process를 list로 나타낸다. fg는 background process를 foreground로 이동하여 실행한다. bg는 stopped인 background process를 running으로 바꿔준다. kill은 process를 종료한다. 또한 SIGINT, SIGSTP signal에 대해서 handler를 만들어서 Ctrl + C, Ctrl + Z로 각각의 신호를 주었을 때 종료, 중지의 동작을 수행하도록 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

fork() 함수는 기존의 process에서 child process를 생성하면서 parent process와 동일한 코드, 동일한 변수를 참조하도록 한다. 실행 시에 fork 함수의 return으로는 parent에는 child의 pid가 전달되고, child에는 0으로 리턴되므로 이를 이용하여 parent와 child의 동작에 차이를 둘 수 있다. 프로젝트 내에서는 exec을 이용하여 system call을 수행할 때 child에서 실행하도록 하므로 fork()의 리턴이 0일 때 실행하도록 하는 방식 등을 이용할 수 있다. Parent에서는 child를 reaping하여 zombie process를 방지하기 위해서 wait를 이용한다.

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

child process가 종료되면 SIGCHLD를 받는다. Phase 1에서는 foreground에서의 작업만 존재하여 parent process에서 작업을 하나씩만 처리하므로 waitpid를 이용하여 기다리는 방식으로 종료까지 기다렸다가 parent process가 진행하도록 하는 방식을 이용하였다. 이는 SIGCHLD에 대해서 signal handler를 만들어두고 signal(SIGCHLD, 만든 handler)로 처리하는 방법도 있다.

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

입력된 명령어를 우선 |을 기준으로 나누고 이를 각각 queue에 문자열로 저장하도록 하였다. 각각의 명령어를 실행하는 과정은 phase 1에서 사용한 방식과 동일하게 exec으로 system call을 child process에서 하는 것은 동일하나, |로 여러 개 연결된 경우에는 해당 명령어의 출력을 pipe로 연결해야 한다. 이 경우는 file descriptor를 dup2 함수를 이용하여 읽기와 쓰기를 활성화하는 것을 조절하는 방식을 통해서 다음 명령어의 입력으로 명령어의 출력값을 전달할 수 있도록 하였다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

입력된 명령어에서 |를 기준으로 나눌 때 나눠진 문자열의 개수를 다른 변수에 저장해둔다. 이 때 pipe의 개수는 나눠진 문자열보다 하나 적다고 할 수 있다. 이들은 따로 명령어를 저장하기 위한 구조체를 생성한 후에 이를 queue로 이용할 수 있도록 구현한 후, 저장하였다. 따라서 먼저 들어온 명령어부터 dequeue하여 얻을 수 있도록 하였고, 이는 저장해둔 문자열의 개수만큼 반복한다. 나눠진 문자열의 개수가 1개인 경우는 phase 1과 동일한 case이므로 phase 1에서 구현한 부분을 그대로 수행하도록 놔두었고, 나머지 case일 때 pipeline을 수행하도록 구현했다.

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

background에서 process를 실행하는 경우에는 parent process가 종료까지 기다리는 것이 아닌 다른 명령어도 받을 수 있다. 따라서 이를 구현하기 위해서는 signal handler를 이용하여 signal이 왔을 때 처리해줄 동작을 설정하는 것이 필요하다. 이 때 signal handler에서 waitpid를 사용해줌으로써 child process가 종료되는 것을 기다릴 수 있도록 해준다. Background process의 목록은 전역변수에 저장을 위한 구조체의 array를 생성하여 관리하였다. 이는 jobs 명령어가 입력되었을 때 상태에 맞도록 출력하는 역할을 수행하고, bg, fg, kill이 입력되면 프로세스를 적절히 목록에서 삭제하는 등의 역할을 수행한다. 명령어에 &가 마지막에 들어갔는지는 기존의 명령어를 띄어쓰기를 기준으로 나누는 parseline 함수에서 확인하고, &가 마지막에 들어가 있으면 bg를 1로 리턴하여 표시한다.

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

main 함수에서는 while 문을 이용하여 사용자에게 계속 입력을 받고 해당 명령어를 eval 함수에 전달하여 명령어가 원하는 실행을 확인한 후에 그에 맞게 실행하는 기능을 수행한다. Phase 3에서는 signal handler를 사용하므로 while문 시작하기 전에 SIGTSTP, SIGINT, SIGCHLD에 대해서 signal 함수로 handler를 연결해준다. eval 함수는 인자로 전달받은 명령어 한 줄을 분석하는 함수이다. 이는 phase 1에서는 우선 명령어를 parseline 함수로 띄어쓰기에 따라서 나누고 argv 배열에 문자열을 순서대로 저장해둔다. 이를 builtin\_command 함수에서 execvp 함수에서 실행되지 않는 exit, cd, quit에 대해서는 실행하고 아닌 경우에는 0을 리턴하여 fork가 실행되도록 한다. builtin\_command에서 cd를 구현할 때는 뒤의 문자열이 .이나 ~만 존재시는 home 디렉토리로 이동하는 것이므로 chdir(getenv(“HOME”))을 이용하여 구현했다. 이외에는 argv[1]에 있는 문자열의 directory로 이동하도록 했고, 0이 아닌 값을 리턴하는 경우는 오류가 발생한 것이므로 표시하도록 하였다. fork 실행 후에 pid가 0, 즉 child process인 경우에는 execvp를 실행하여 system call을 수행한다. Parent process는 해당 부분의 뒤에서 waitpid를 통해서 child process의 종료를 대기한다.

Phase 2에서 pipeline을 구현하기 위해서는 우선 각각의 명령어를 저장할 방법을 만들어야 했다. 이에 따라서 |를 기준으로 나눈 명령어들을 저장하기 위한 queue를 만들기로 하였다. 이는 명령어를 저장하기 위한 char 타입의 MAXLINE만큼의 크기를 가진 array인 command와 다음 노드를 연결하기 위한 next를 가진 Node 구조체를 만들었다. 다음으로 queue를 완성하기 위해서 첫 Node를 저장할 front와 마지막 Node를 저장할 rear와 Node의 개수를 저장할 cnt를 넣은 Queue 구조체를 선언했다. 이 후에는 queue를 초기화하는 initQueue, 비어있는지 확인하는 isEmpty 함수를 넣었고, element들을 넣고 뺄 수 있는 enqueue, dequeue와 동작 확인시 사용할 printQueue, queue를 삭제할 때 사용할 freeQueue를 만들었다. Pipeline는 eval 함수 내에서 동작하는 것으로 구현하였다. 우선 queue를 초기화하고 strtok을 이용하여 |를 기준으로 명령어 문자열을 나누도록 하였다. 이 때 나눌 때마다 queue에 enqueue 함수를 이용하여 넣어주었다. tot 변수에는 |를 기준으로 나눈 후에 queue의 element 개수를 넣어주었다. 이 때 tot가 1인 경우는 |가 없는 경우이므로 phase 1의 동작을 그대로 실행한다. |가 없을 때는 queue를 사용하지 않으므로 명령 수행전에 freeQueue를 하도록 하였다. Pipe를 구현하기 위해서 file descriptor를 저장할 배열과 pid를 저장할 배열을 선언해두고 queue가 빌 때까지 for문으로 반복하면서 우선 dequeue로 command를 받아온다. 이를 parseline 함수로 띄어쓰기를 기준으로 구분하였다. 이 후에 pid\_arr에 순서대로 fork를 해준다. 다음으로 자식 프로세스인 경우에는 각각 첫 명령어인지 마지막 명령어인지, 그 외의 경우인지에 따라서 file descriptor를 설정한다. 첫 번째 명령어의 경우에는 명령어의 출력을 pipe의 쓰기 descriptor로 바꿔야 하므로 읽기 descriptor는 close로 닫아주고, 쓰기 descriptor는 dup2 함수로 STDOUT\_FILENO로 바꿔준다. 중간에 있는 명령어는 이전 명령어의 읽기 descriptor를 STDIN\_FILENO로 바꿔 입력으로 들어오도록 하고, 해당 명령어의 쓰기 descriptor를 STDOUT\_FILENO로 바꿔 다음 명령어에 전달한다. 마지막 명령어는 입력만 이전 명령어에서 받아오면 되므로 이전 명령어의 읽기 descriptor를 STDIN\_FILENO로 바꿔주었다. 모든 descriptor는 사용 완료 후에 close로 닫아주었다. 이 후에 execvp로 명령어를 실행한다. Parent process는 waitpid로 종료를 기다린다. Queue를 모두 돌았을 테지만 그래도 마지막에 freeQueue로 비우는 과정은 넣어두었다.

Phase 3에서 background를 구현하기 위해서는 우선 process들을 저장할 array를 만들기로 하였다. 이는 job\_id, pid, command, state를 저장하는 구조체를 만들어서 array로 선언하였다. state는 int 타입 변수로 -1일 때는 stopped, 0일 때는 foreground, 1이면 background, -5면 삭제된 process, -7은 정상종료된 process를 나타내도록 정해두었다. 또한 저장할 index를 관리하기 위해서 pid\_cnt를 선언하였다. Handler를 만든 signal은 SIGSTP, SIGINT, SIGCHLD가 있다. SIGSTP는 foreground에서 동작하는 process에 SIGTSTP를 kill 명령어로 보내도록 하고 해당 job의 state를 -1로 수정하도록 하였다. SIGCHLD는 child process의 종료, 즉 명령어가 정상적으로 실행된 경우이므로 job에서 state를 -7로 바꿔 주도록 하였다. SIGINT의 경우에는 종료시키는 signal이므로 job에서 state를 -5로 바꾸고, kill 함수로 SIGINT를 전달하도록 하였다. 이들은 main 함수에서 signal 함수로 handler로써 연결되도록 하였다. 또한 handler들은 모두 동작 전에 signal을 mask 처리로 동작 중에 영향 받지 않도록 하였다. 명령어가 뒤에 &가 붙었는지는 parseline 함수에서 확인한다. 이전에 phase1, 2에서 구현했을 때에서 수정하여 맨 뒤에 &가 있는지 확인한 후에 제일 마지막이 &이면 bg를 1로 하고 제거한 후에 띄어쓰기 기준으로 parsing하도록 하였다. Background process가 실질적으로 작동하는 것은 eval 함수이고 해당 경우에 foreground이면 기존처럼 waitpid로 기다리도록 하였고, background인 경우에는 job\_arr에 job을 추가하도록 하였다. 이 떄 추가될 때의 pid\_cnt가 job\_id가 되도록 정했다. 다음으로 추가해야 하는 명령어인 jobs, bg, fg, kill은 execvp로 실행하는 명령어가 아니므로 builtin\_command에서 구현하였다. Background process들을 상태와 함께 출력하는 jobs는 우선 명령을 실행하는 동안에는 signal을 받지 않도록 하고, state가 -5로 저장된, 즉 이미 종료된 process들을 job\_arr에서 제거한 후, state가 1, 즉 실행 중인 background process는 Running으로 출력한다. -1은 SIGSTP를 받은 상태이므로 Stopped를 출력한다. -7은 정상적으로 종료된 것이므로 Done을 출력하고 -5로 state를 바꿔 다음에 jobs 실행시에는 삭제되도록 한다. 다음으로 kill은 job\_arr을 순회하며 job\_id가 같은 것을 찾고 이를 array에서 삭제하며 SIGKILL signal을 해당 process에 보낸다. 명령어 bg는 중지된 background process를 다시 running으로 바꾼다. 따라서 job\_arr에서 찾고 SIGCONT signal을 보내 다시 running으로 바꾸고 state를 1로 바꿔 표시한다. 명령어 fg는 중지된 background process를 foreground에서 실행하도록 한다. 따라서 kill 함수로 SIGCONT signal을 보낸 후에 waitpid로 종료를 기다려야 한다. 이 후에 완료된 후에는 job\_arr에서 삭제하는 과정을 거친 후 종료된다.

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)**

**텍스트, 도표, 라인, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

1. **Phase 2 (pipeline)**

**도표, 텍스트, 평면도, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

1. **Phase 3 (background)**

**도표, 스케치, 평면도, 그림이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**