**Multicore Programming Project 1**

담당 교수 : 박성용

이름 : 이건화

학번 : 20191616

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

**해당 프로젝트는 쉘을 구현하는 것이 목적이다. 쉘은 운영체제와 사용자 사이에서 명령어를 텍스트형식으로 입력받고 입력받은 명령어를 운영체제가 처리할 수 있게 하고, 명령어에 대한 출력을 텍스트형식으로 사용자에게 제공한다.**

**쉘 자체에 내장되어 있는 명령어와 실행파일을 실행할 수 있어야 한다.**

**여러 명령어를 하나의 명령어가 다른 명령어의 입력값으로 들어가게 하는 pipe 기능을 통해 특정 기능을 여러 개의 명령어를 통해 다채롭게 구현할 수도 있다. 이를 통해 프로세스 간의 통신에 대한 이해를 향상시킬 수 있다.**

**또한 쉘은 명령어를 실행하는 프로세스를 여러 개를 형성하여 background에서 실행하게할 수도 있다. signal handler를 통해 signal를 처리하며 여러 프로세스를 질서 있게 처리하며 프로세스에 대한 이해를 향상 시킬 수 있다.**

**이번 프로젝트에서 위와 같은 기능을 구현함으로써 프로그래밍 능력을 향상시킬 뿐만 아니라 시스템에 대한 이해를 높일 수 있다.**

**이러한 주요기능 외에도 쉘에서 사용자의 편의를 위해 제공하는 다양한 예외처리를 이번 프로젝트를 통해 구현함으로써 프로그래밍 능력을 향상시킨다.**

* **(MyShell을 만드는 전체적인 개요에 대해서 작성하면 됨.)**

Phase 1에서는 실행파일이나 쉘에 내장되어 있는 명령어 뿐만 아니라 내장되어 있지 않은 명령어 또한 실행할 수 있도록 구현한다. 쉘에 내장되어 있지 않은 명령어는 builtin\_command 함수 안에서 구현한다.

Phase 2에서는 pipe 기능을 구현하는데 여러 명령어를 pipe를 통해서 상호작용하며 동작할 수 있게 한다. 이는 dup2()함수를 사용하여 파일 출력과 입력 디스크립터를 변경해줌으로써 구현한다.

Phase 3에서는 여러 프로세스를 background에서 실행할 수 있는 기능을 구현한다. 이를 구현하기 위해 signal에 대한 이해를 완성한 후, 해당 signal을 처리하는 handler를 구현하며 안정적으로 background 실행될 수 있게 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1
   * Phase1을 구현하고 나서는 ls와 같은 명령어를 쳤을 때 ls에 해당하는 결과값이 출력된다. 출력되고 나서는 myShell이 종료되지 않고 새로운 프롬프트 창이 출력되면서 다음 명령어를 대기한다.
   * builtin\_command에 있는 history 명령어를 실행시켰을 때는 지금까지 입력한 명령어들이 차례대로 출력된다. 연속으로 똑같은 명령어를 입력했을 때는 history 파일에 저장되지 않는다. cd를 활용하여 다른 디렉토리로 이동할 수 있다.
   * !!기능을 활용하여 직전 명령어를 재활용할 수 있다. !# 기능을 활용하여 history파일에서 해당 번호의 명령어를 다시 실행할 수 있다.
2. Phase 2
   * pipe 기능을 구현하고 나면 ls | grep a 같은 명령여를 실행하면 ls를 통해 나오는 파일들 중 a가 들어가있는 파일들만 출력해준다. 이는 ls의 출력값들이 grep의 입력값으로 들어감에 따라 ls의 결과값 중 a가 들어가있는 파일들만 출력하기 때문이다.
   * ls | grep a | grep b와 같이 실행할 경우 ls의 출력들 중 a가 들어가있는 출력들만 또 그 중에 b만들어가 있는 출력값들만 출력하게 된다.
3. Phase 3
   * 이전 단계에서는 ctrl+C나 ctrl+Z가 누를 경우 myShell자체가 종료됐었다. 이는 signal에 대한 handling이 따로 없었기 때문이다. phase3구현 후에는 ctrl+C를 통해 SIGINT 신호를 입력받더라도 종료되지 않고 현재 내가 foreground로 실행중인 프로세스만 종료되게 된다. ctrl+Z를 통해 SIGTSTP 신호를 입력 받았을 때는 foreground로 동작중이던 프로세스는 멈추게 되고 background로 가게 된다.
   * fg %n를 실행하면 background에 있던 n번 째 프로세스를 foreground로 가져온 후 run 시킨다.
   * bg %n를 실행하면 n번 째 background에 있던 프로세스를 background로 run 시킨다.
   * jobs를 실행하면 헌재 background에 있는 프로세스들의 상태를 보여주며 나열한다..
   * kill을 %n을 실행하면 n번째의 프로세스를 종료시킨다.
   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
    - Phase1에서는 ls와 같은 내장명령어들은 execvp를 사용하여 실행할 수 있는데 execvp함수는 명령어를 실행 후 자동으로 exit()이 됨으로 부모(메인) 프로세스에서 실행할 경우 프로그램 자체가 종료되게 된다. 이를 해결하기 위해 child process를 생성하여 해당 child process에서 execvp를 호출하여 해당 명령어를 실행 후 종료되게 한다.
  + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow
    - Child process가 종료되면 부모 프로세스에 signal이 오고 부모 프로세스에서는 waitpid(pid, &status, 0)를 통해 해당 pid의 child process의가 종료되기를 기다리며 종료된 후에는 종료된 결과값은 status에 저장하고 child process를 reaping 해준다.
* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)
    - recursive로 구현하였다. 우선 parsing 과정에서 몇 개의 pipe가 있는지 파악하고 개수에 따라 recursive의 횟수를 정한다. char \*\*argv에서 pipe의 자리에는 NULL이 들어가면서 recursive가 구현가능해진다. pipe(fd)를 통해 fd[1]에 출력이 들어오면 fd[0]을 통해 읽을 수 있다. 부모프로세스에서 recursive가 일어난다. 부모프로세스가 생성한 자식프로세스는 출력을 fd[1]로 하고 부모 프로세스에서는 recursive를 호출하기전에 fd[0]에서 입력받게하여 재귀로 호출당한 부모프로세스는 호출했던 부모프로세스가 생성한 자식프로세스의 출력을 받을 수 있게된다. 마지막 호출된 부모프로세스는 따로 자식프로세스를 생성하지 않고 부모프로세스 자체에서 execvp를 실행하여 명령어를 출력하고 종료되게 한다. 가장 마지막 명령어부터 recursive가 풀리기 때문에 이에 따라 모든 프로세스들이 정상적으로 reaping될 수 있다.
  + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명
    - A | B | C 와 같이 명령어가 들어왔다고 하자. A의 부모프로세스가 B를 재귀로 호출하기 전에 A의 자식프로세스의 출력디스크립터에 파이프의 출력 디스크립터 복사하고 부모프로세스의 입력을 해당 파이프의 입력 디스크립터와 연결한다. 이 덕분에 B는 A의 자식프로세스의 출력으로부터 입력받게 된다. 이러한 알고리즘에 기반하여 개수와 상관없이 계속 재귀로 동작할 수 있다. 마지막 프로세스는 자식프로세스를 따로 생성할 필요없이 execvp를 통해 출력을 터미널에 출력하고 종료되며 recursive를 연쇄적으로 탈출하게 된다. 이 과정에서 reaping도 모두 정상적으로 이뤄질 수 있다.
* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명
    - &
      * parsing과정에서 &의 유무를 확인하여 &가 있을 경우 bg가 1로 set된다. bg가 1이 아니라면, 즉 foreground process라면 child\_handler에서 해당 pid가 reap할 때까지 기다린다. 이전 phase1에서는 waitpid를 사용하였지만 지금은 while문과 sigsuspend, sigblockmask, 그리고 child handler를 대신 사용한다. 이를 통해 더욱 안정적인 프로세스 관리를 할 수 있다.
    - fg
      * fg는 background에 있는 process에게 SIGCONT signal을 보내주고, 해당 process의 status를 변경한다. foreground process의 처리와 마찬가지로 해당 process가 child\_handler에게 reaping될 때까지 기다린다. 기다릴 때는 foreground process 때의 방법과 같은 방법을 사용한다.
    - bg
      * bg는 background에 있는 process에게 SIGCONT signal을 보내주고 해당 process의 status를 변경한다.
    - jobs
      * 저장한 자료구조를 통해 명령어들을 출력한다.
    - kill
      * 해당 프로세스의 pid에 SIGINT signal을 보낸다.
    - TSTP\_handler
      * 현재 동작 중인 foreground에 process에 SIGTSTP 신호를 보내고 현재 동작 중인 foreground가 없음을 나타내고 이를 통해 foreground의 reaping을 기다리던 while문을 탈출할 수 있게 된다.
    - INT\_handler
      * 현재 동작 중인 foregorund에 process를 SIGKILL을 보낸다.
    - CHLD\_handler
      * 좀비프로세스가 된 process들을 모두 reaping하고 reaping한 pid에 따라 자료구조를 최신화한다.
      * CHLD\_handler을 통해 위의 명령어들 중 SIGKILL을 보내는 기능들은 CHLD\_handler에서 자료구조까지 다 정리해주게 된다.
  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 (fork & signal)**
  + **main**
    - **현재 경로를 구한 후, 현재 경로에 history.txt가 생길 수 있도록 history파일의 경로를 생성해서 변수로 저장해놓는다.**
  + replace\_exclam
    - 우선 !!, !#에 대한 처리를 해준다. !로 시작되는 위치를 찾고 !다음에 !가 한번 더 나온다면 history.txt의 마지막 줄을 !!와 치환한다. 숫자가 나오는 경우는 atoi를 활용하여 숫자로 바꾼 후, 해당 줄을 !#와 치환한다. 재귀로 구현하여 여러 개의 !!, !#가 나와도 모두 처리 가능하다.
  + eval
    - builtin함수가 아닐 경우 fork로 child process를 사용하여 execvp함수를 사용하여 명령어를 실행한다. execvp를 사용할 경우, execve와는 다르게 직접 생성한 실행파일도 더 수월하게 실행할 수 있다. 또한, 환경변수를 넘겨줄 필요도 없다. execvp를 통해 명령어를 실행한 후 종료되면 부모프로세스로 돌아오게 된다.
  + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow
    - execvp를 통해 명령어가 실행되고 난 후 childp process가 종료되면 부모 프로세스에 signal이 온다. 그러면 child process에 대한 자원을 정리해주기 위해 reaping을 해줘야 한다. 이는 이전에 부모 프로세스에서 fork를 통해 생성된 pid에 waitpid(pid, &status, 0)을 사용하여 child process에 대한 reaping을 진행할 수 있다. pid의 child process가 종료되기를 기다리며 종료된 후에는 종료된 결과값은 status에 저장하고 child process를 reaping 해준다.
  + write\_to\_history
    - main에서 구해놓은 history.txt파일에 지금까지 명령어를 기록하여 나중에 기록해놓은 명령어를 확인할 수 있게 한다. 이를 통해 histroy 명령어는 물론, !!, !#도 구현할 수 있게 된다. write\_to\_history는 빈 개행문자는 저장하지 않고 마지막 명령어와 다른 명령어라면 모두 history.txt파일에 추가한다.
    - !!, !#
      * history\_path로부터 !!는 가장 마지막 command로 치환하고, !#는 #에 해당하는 command로 치환한다. 이는 replace\_exclam에 구현되어 있다. 치환된 후 다른 명령어들과 마찬가지로 history.txt의 마지막 줄과 다르다면 추가한다.
    - cd
      * cd는 builtin에 구현되어있는데 chdir(경로)를 사용하여 해당 directory로 이동할 수 있다. 이동할 directory앞에 현재 경로를 의미하는 ./를 붙여줘서 바르게 이동할 수 있다.
      * home으로 이동하는 경우 getenv(“HOME”)를 사용하여 홈디렉토리로 이동할 수 있다.
* **Phase2 (pipelining)**
  + eval
    - parse\_line에 들어가기 전에 | 에 공백이 붙지 않더라도 바르게 parsing이 될 수 있도록 | 앞 뒤에 공백을 넣어주고 parsing하게 한다.
    - | 로 구분된 명령어들의 개수를 구한다. 구한 명령어들의 개수에 따라 recursive가 진행된다. | 가 있던 argv의 자리에는 NULL이 들어감으로써 recursive의 준비를 마친다.
  + parsline
    - 공백을 기준으로 parsing하며 마지막에 &가 붙어있으면 return 값은 1이되고 background process임을 의미한다. 마지막 &는 제거한다.
  + pipe\_recursive
    - recursive로 구현하였다. 우선 parsing 과정에서 몇 개의 pipe가 있는지 파악하고 개수에 따라 recursive의 횟수를 정한다. char \*\*argv에서 pipe의 자리에는 NULL이 들어가면서 recursive가 구현가능해진다. pipe(fd)를 통해 fd[1]에 출력이 들어오면 fd[0]을 통해 읽을 수 있다. 부모프로세스에서 recursive가 일어난다. 부모프로세스가 생성한 자식프로세스는 출력을 fd[1]로 하고 recursive를 호출하기전에 fd[0]에서 입력받게하여 재귀로 호출당한 부모프로세스는 호출했던 부모프로세스가 생성한 자식프로세스의 출력을 받을 수 있게된다. 마지막 호출된 부모프로세스는 따로 자식프로세스를 생성하지 않고 부모프로세스 자체에서 execvp를 실행하여 명령어를 출력하고 종료되게 한다. 마지막 명령어가 종료되면 끝에서부터 recursive가 풀리기 때문에 이에 따라 모든 프로세스들이 정상적으로 reaping될 수 있다. pipe의 discriptor와 현재 프로세스의 discripter를 연결하기 위해 dup2(fd[0], STDIN\_FILENO)처럼 사용할 수 있다. STDIN\_FILENO는 이제 fd[0]의 파일 discriptor로 변경되는 것이다.
    - 명령어의 개수가 많아진다면 pipe\_recursive의 ins\_n인자 값이 커질 뿐이다. current\_idx와 ins\_n를 비교하며 recursion을 통제한다.
* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명
    - 자료구조
      * 연결리스트를 사용하였다. 연결리스트는 job들을 저장하고 있다. 각각의 노드는 해당 job의 pid, status, name, num을 가지는데 pid는 process의 pid를 나타내고, status는 RUNNING인지 SUSPENDED인지를 나타낸다. name은 프로세스의 command를 parsing하여 저장하고 있고, num은 몇 번째 job인지를 저장하고 있다. link는 다음 프로세스를 가리키고 있다. fg\_job은 job들 중 foreground인 프로세스를 가리키고 있다. fg\_job이 0인 경우 현재 foreground로 동작하고 있는 프로세스가 없음을 나타낸다. 연결리스트에 job이 추가되는 경우는 fork할 때이다. add\_job 중에 signal이 발생하여 자료구조에 오류가 생기는 것을 방지하기 위해 sigprocmask를 통해 signal을 막아준다. 또한 이때 아주 중요한 것으로는 setpgid(0,0)을 통해 생성한 자식프로세스를 부모 프로세스와 다른 그룹에 넣어줘야 한다는 점이다. kill signal이 갈 때 같은 그룹에 있는 프로세스들 모두 signal을 수신하기 때문에 다른 그룹에 넣어줘야 signal을 받았을 때 영향받지 않을 수 있다. 연결리스트에 job이 삭제되는 경우는 child\_handler에서밖에 없다. 이로 인해 안정적인 자료구조를 유지할 수 있다. child\_handler에서 reaping이 이뤄지면 reaping 한 pid를 return 받아 해당 pid의 노드를 삭제해준다. 또한, 삭제한 노드가 foreground일 경우 fg\_job = 0을 통해 현재 foreground로 동작하고 있는 process가 없음을 나타낸다.
    - &
      * parsing과정에서 &의 유무를 확인하여 &가 있을 경우 bg가 1로 set된다. bg가 1이 아니라면, 즉 foreground process라면 child\_handler에서 해당 pid가 reap할 때까지 기다린다. 기다릴 때는 child\_handler에서 이미 waitpid를 하기 때문에 waitpid를 활용하지 않는다. while를 이용하여 child\_handler가 foreground의 process를 reaping하기를 기다리는데 while문을 도는 동안 signal이 발생하여 놓칠 수 있기 때문에 sigprocmask를 사용하여 SIGCHILD 신호를 안정적으로 받을 수 있게한다. child\_handler에서 foreground의 process를 reaping하고 fg\_job = 0이 된다면 while문을 탈출하게 되고 다시 명령어를 받을 수 있게된다.
    - fg
      * fg는 background에 있는 process에게 SIGCONT signal을 보내주고, 해당 process가 마찬가지로 child\_handler에게 reaping될 때까지 기다린다. 현재 다시 동작 중인 프로세스가 foreground임을 나타내기 위해 fg\_job은 현재 job을 가리키게 된다. 이로 인해 eval에서 처음 &를 처리할 때처럼 foreground의 reaping을 기다릴 수 있게 된다.
    - bg
      * bg는 background에 있는 process에게 SIGCONT signal을 보내주고 해당 process의 status를 변경한다. SIGCONT를 받은 프로세스는 다시 RUNNING으로 변하게 된다.
    - jobs
      * 저장한 자료구조를 통해 명령어들을 상태에 맞게 형식에 맞춰 출력한다.
    - kill
      * 해당 프로세스의 pid에 SIGINT signal을 보낸다. 그럼 해당 프로세스는 죽게 되고 child\_handler에서 처리할 수 있게 된다.
    - TSTP\_handler
      * 현재 동작 중인 foreground에 process에 SIGTSTP 신호를 보내고 자료구조를 최신화한다. TSTP\_handler에서는 프로세스에 SIGTSTP신호를 보내기 때문에 프로세스가 child\_handler에서 reap되지 않는다. 때문에 child\_handler에서처럼 foreground process가 reaping되기를 기다리는 while문의 조건을 직접 해방시켜주어야 한다. 이를 위해서 fg\_job = 0을 통해 while문을 탈출시킬 수 있다. 그러나 이를 위해서는 현재 foreground로 동작하고 있는 process가 있는지 확인해주어야 한다.
    - INT\_handler
      * 현재 동작 중인 foregorund process에 SIGKILL을 보낸다. SIGKILL을 받은 job은 종료되게 되고 child\_handler에서 이를 reaping 해주고 자료구조에서 삭제해준다. 때문에 TSPT\_handler에서처럼 while문을 탈출 시켜줄 필요가 없다. 그러나 역시 foreground process가 존재하는지 확인해주어야 한다.
    - CHLD\_handler
      * 좀비프로세스가 된 process들을 모두 reaping하고 reaping한 pid에 따라 해당 노드를 삭제해준다. 이를 통해 job리스트에는 현재 동작 중이거나 정지된 프로세스만 저장될 수 있다.
      * CHLD\_handler을 통해 위의 명령어들 중 SIGKILL을 보내는 기능들은 CHLD\_handler에서 자료구조까지 다 정리해주게 된다.

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**