**System Programming Project 2**

담당 교수 : 박성용 교수님

이름 : 김규빈

학번 : 20200152

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
  + 이 프로젝트의 목표는 linux 환경에서 사용되는 쉘을 모방한 MyShell을 개발하는 것이다. 이 쉘은 사용자로부터 텍스트 형식의 명령어를 받아 운영 체제가 처리할 수 있도록 하며, 처리 결과를 텍스트로 사용자에게 제공한다. 기본적인 내장 명령어 실행 능력과 함께, 실행 파일을 호출하는 기능도 포함할 예정이다.
* **(MyShell을 만드는 전체적인 개요에 대해서 작성하면 됨.)**
  + Phase 1: 기본적인 명령어 실행 기능을 구현한다. 내장 명령어는 builtin\_command 함수 내에서 처리하고, 내장되지 않은 명령어도 실행할 수 있게 한다.
  + Phase 2: 파이프 기능을 구현하여 한 명령어의 출력을 다른 명령어의 입력으로 사용할 수 있습니다. 이는 dup2() 함수를 사용해 파일의 출력과 입력 디스크립터를 변경하여 가능하게 한다.
  + Phase 3: 백그라운드 프로세스 실행 기능을 추가한다. 이를 위해 시그널 처리에 대한 이해를 바탕으로 하여 적절한 시그널 핸들러를 구현하고, 프로세스를 안정적으로 백그라운드에서 실행할 수 있도록 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1
   * 단일 명령어를 shell에 입력하였을 때 해당 명령어에 맞는 출력결과가 나올 수 있도록 한다. 명령어의 출력이 끝나고 나면 다음 명령어를 입력받을 수 있는 입력창이 출력이되고 이 과정이 반복되도록 하고 exit명령어를 입력할 때에 shell을 종료하도록 구현한다.
2. Phase 2
   * 두 가지 이상의 명령어를 pipe 기호를 이용하여 입력하였을 때, 명령어 차례로 출력값이 다음 명령어의 입력값으로 들어가며 순차적으로 명령어를 실행하여 나오는 결과 값을 출력할 수 있도록 구현한다.
3. Phase 3
   * 명령어를 입력할 때 foreground로 process를 진행할 것인지 background로 process를 진행할 것인지 & 문자 입력으로 여부를 판단하고 그에 맞게 process를 진행하도록 한다. jobs 명령어를 입력시 background의 process들을 출력하도록 하고 bg <job> 명령어를 입력시 멈춰있는 백그라운드의 해당 job을 다시 실행하도록 하고 fg <job> 명령어를 입력시 백그라운드의 해당 job을 foreground로 실행하도록 한다. kill <job> 명령어를 입력시 해당 job을 종료시킬 수 있도록 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
    - eval 함수에서, 사용자로부터 입력 받은 명령을 처리
    - fork를 호출하여 자식 프로세스를 생성
    - fork의 반환 값이 0이면, 현재 프로세스가 자식 프로세스임을 의미, 자식 프로세스는 execvp를 호출하여 사용자 명령을 실행합
    - execvp 호출이 실패한 경우, 해당 명령을 찾을 수 없다는 메시지를 출력하고 프로세스를 종료
    - 부모 프로세스에서는 fork가 0이 아닌 PID(프로세스 ID)를 반환합니다. 부모 프로세스는 자식 프로세스가 종료될 때까지 대기
  + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow
    - 자식 프로세스가 종료되면, 커널은 부모 프로세스에 시그널을 보냄
    - 부모 프로세스는 waitpid(pid, &status, 0)를 호출하여 특정 PID를 가진 자식 프로세스의 종료를 대기
    - 자식 프로세스가 종료되면, 그의 종료 상태는 status 변수에 저장
    - waitpid() 함수의 호출이 성공적으로 반환되면, 자식 프로세스의 프로세스 엔트리와 관련 리소스들이 시스템에서 정리
* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)
    - for 루프를 사용하여 각 명령을 순회
    - 마지막 명령이 아닌 경우, pipe 시스템 콜을 사용하여 파이프를 생성
    - 각 명령은 fork를 통해 생성된 자식 프로세스에서 실행
    - 자식 프로세스에서는 dup2를 사용하여 표준 입력과 표준 출력을 적절한 파이프 파일 디스크립터에 연결
    - 마지막 명령 후, 모든 자식 프로세스가 종료될 때까지 부모 프로세스는 wait을 호출하여 대기
  + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명
    - eval 함수에서 사용자로부터 입력된 전체 명령 문자열을 분석하여 파이프라인을 사용하는 명령의 시작 인덱스를 기록
    - 파이프라인이 없는 경우(단일 명령), 일반적인 방식(Fork와 execvp)으로 명령을 실행
    - 하나 이상의 파이프라인이 있는 경우, execute\_commands 함수를 호출하여 복잡한 파이프라인 로직을 처리, 이때 각 파이프라인 사이의 데이터 흐름을 관리
    - 각 파이프라인 명령은 별도의 자식 프로세스에서 실행되며, 파이프를 통해 이전 명령의 출력이 다음 명령의 입력으로 전달됨
* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명
    - &
      * 명령어 끝에 '&'가 있으면 해당 프로세스가 백그라운드에서 실행되어야 함을 나타냄. 파싱 과정에서 '&'를 찾으면 bg 변수가 1로 설정됨
      * 백그라운드로 설정되지 않은 포어그라운드 프로세스의 경우, 프로세스 관리자는 schld\_handler를 통해 자식 프로세스의 상태 변화를 기다림. 이는 단순히 waitpid를 사용하는 것보다 더 안정적인 프로세스 관리를 가능하게 함
    - fg
      * fg는 백그라운드 프로세스에게 SIGCONT 시그널을 보내어 포어그라운드로 다시 실행하게 함
      * 프로세스의 상태를 포어그라운드로 변경하고 종료될 때까지 기다린다. 이는 초기에 포어그라운드 프로세스를 실행할 때 사용하는 대기 방법과 동일
    - bg
      * bg는 백그라운드 프로세스에게 SIGCONT 시그널을 보내어 실행을 계속하게 하지만, 프로세스가 종료될 때까지 기다리지는 않는다.
      * 프로세스의 상태를 백그라운드로 변경하여 독립적으로 계속 실행할 수 있게 한다.
    - jobs
      * 모든 작업을 나열하며, 작업 제어 데이터 구조를 사용하여 각 작업의 상태와 커맨드 라인을 보여줌
    - kill
      * 작업 ID로 식별된 프로세스에게 SIGKILL 시그널을 보내어 강제로 종료
    - ststp\_handler
      * 현재 포어그라운드 프로세스에 SIGTSTP 시그널을 보내어 중지시키고, 해당 상태를 표시하여 프로세스 관리자가 프로세스의 종료를 기다리는 것을 멈추게 함
    - sint\_handler
      * 현재 포어그라운드 프로세스에 SIGKILL을 보내어 즉시 종료
    - schld\_handler
      * 모든 좀비 프로세스를 정리하고, 작업 제어 데이터 구조를 그에 따라 업데이트
      * SIGKILL을 보내는 모든 명령어는 schld\_handler에 의해 작업 제어 데이터에서 정리된다.
  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)**
  + 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.
* **Phase2 (pipelining)**
  + main
    - 메인 함수는 무한 루프를 통해 사용자로부터 명령어를 입력 받는다. 입력된 명령어는 eval 함수에 의해 처리된다. 사용자가 EOF (End-of-File)를 입력하면 프로그램은 종료된다.
  + eval
    - eval 함수는 입력받은 명령어 라인을 처리한다. 이 함수는 명령어를 파싱하고, 내장 명령어인지 확인한 후, 해당 명령어를 실행한다. 파이프 (|)가 있는 경우, execute\_commands 함수를 통해 여러 명령어를 파이프라인으로 연결하여 실행한다. 백그라운드 (&)가 명시된 경우 명령어는 백그라운드에서 실행된다.
  + parseline
    - parseline 함수는 입력된 명령어 라인을 개별 명령어와 인자로 나눕니다. 공백을 구분자로 사용하여 명령어를 분리하며, 명령어 끝에 '&'가 붙어있는 경우 이는 백그라운드 작업을 의미하며, 이 문자를 제거하고 해당 정보를 반환한다.
  + builtin\_command
    - 내장 명령어 처리 함수는 몇 가지 기본적인 명령어를 처리합니다. 예를 들어, exit는 프로그램을 종료시키고, cd는 디렉터리를 변경한다. 이 함수는 명령어가 내장 명령어일 경우 해당 명령어를 실행하고 true를 반환한다.
  + execute\_commands
    - execute\_commands 함수는 파이프라인을 통해 여러 명령어를 연결하고 실행한다. 각 명령어는 자식 프로세스에서 실행되며, 각 프로세스의 입력과 출력은 파이프를 통해 연결된다. 마지막 명령어를 제외한 각 명령어 실행 후, 출력을 다음 명령어의 입력으로 전달한다.
* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명
    - 자료구조
      * 연결 리스트를 활용하여 백그라운드 작업들을 관리한다. 각 작업은 구조체(struct job\_t)를 통해 관리되며, 구조체는 작업의 PID, 상태(RUNNING 또는 SUSPENDED), 명령어 문자열, 작업 번호를 저장한다.
      * addjob 함수에서는 새로운 작업을 리스트에 추가하며, 신호로 인한 데이터 경쟁을 방지하기 위해 sigprocmask를 사용하여 신호를 차단한다.
      * deletejob 함수에서는 작업을 리스트에서 삭제하며, 해당 작업이 포어그라운드에서 실행 중이었다면, 관련 상태를 업데이트한다.
    - &
      * 커맨드 라인에서 &의 존재 유무를 확인하여 작업이 백그라운드에서 실행될지 결정합니다. &가 존재하면 해당 작업을 백그라운드로 설정한다.
      * 백그라운드 작업의 경우, 포어그라운드 작업처럼 해당 작업의 종료를 기다리지 않고 즉시 새로운 명령어 입력을 받는다.
    - fg
      * fg 명령은 백그라운드 작업을 포어그라운드로 전환시키고, 해당 작업의 종료까지 대기한다. 이 과정에서 SIGCONT 신호를 보내 작업을 재개시킨다.
    - bg
      * bg 명령은 중지된 백그라운드 작업에 SIGCONT 신호를 보내어 작업을 재개시키며, 작업의 상태를 RUNNING으로 업데이트한다.
    - jobs
      * listjobs 함수를 통해 현재 작업 리스트의 상태를 출력한다. 각 작업의 실행 상태(RUNNING, SUSPENDED)와 명령어를 함께 출력한다.
    - kill
      * kill 명령을 통해 특정 작업에 SIGKILL 신호를 보내 작업을 종료시킨다. 작업은 종료되며 관련 프로세스는 시스템에 의해 수거된다.
    - ststp\_handler
      * 이 핸들러는 현재 실행 중인 포어그라운드 프로세스에 SIGTSTP 신호를 보내어 프로세스를 일시 중단시키고 작업 목록을 업데이트해야 한다. SIGTSTP 신호를 보내는 것은 프로세스가 schld\_handler에 의해 회수되지 않기 때문에, 핸들러는 포어그라운드 프로세스가 종료될 때까지 기다리는 루프를 수동으로 종료할 필요가 있다. 이를 위해 fg\_job = 0과 같은 플래그를 설정하여 루프에서 벗어날 수 있다. 그러나 이 작업을 하기 전에 실행 중인 포어그라운드 프로세스가 있는지 확인해야 한다.
    - sint\_handler
      * 이 핸들러는 현재 실행 중인 포어그라운드 프로세스에 SIGINT 신호를 보낸다. SIGINT 신호는 일반적으로 프로세스를 종료시키며, 이후 프로세스는 schld\_handler에 의해 회수되어야 한다. 종료된 프로세스는 schld\_handler에서 작업 목록에서 제거된다. 따라서 ststp\_handler와 달리 수동으로 대기 루프를 종료할 필요가 없다. 이 핸들러도 신호를 보내기 전에 실행 중인 포어그라운드 프로세스가 있는지 확인해야 한다.
    - schld\_handler
      * schld\_handler: 이 핸들러는 모든 좀비 프로세스(실행을 마쳤지만 프로세스 테이블에 여전히 항목이 있는 프로세스)를 회수한다. 회수 후, 회수된 PID에 따라 해당 작업을 작업 목록에서 제거해야 한다. 이를 통해 작업 목록에는 현재 실행 중이거나 중단된 프로세스만 포함되도록 한다. 또한, SIGKILL 신호를 보내 종료된 프로세스들에 대해서도 schld\_handler에서 작업 목록을 관리한다.

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)**

**도표, 기술 도면, 라인, 평면도이(가) 표시된 사진

자동 생성된 설명**

1. **Phase 2 (pipeline)**

**도표, 평면도, 기술 도면, 라인이(가) 표시된 사진

자동 생성된 설명**

1. **Phase 3 (background)**

**도표, 라인, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명**