**System Programming Project 2**

담당 교수 :김영재

이름 :정희선

학번 :20231609

1. **개발 목표**

이번 프로젝트는 리눅스 쉘과 유사하게 동작하는 쉘을 만들어 사용자가 명령어를 입력하고 실행할 수 있도록 하는 것입니다.

프로젝트를 세 단계로 나누어, 단계적으로 Myshell을 구현합니다.

Myshell은 fork, signal, pipeline, background process등 시스템 프로그래밍 개념을 기반으로 합니다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Phase 1
   * + 외부 명령어 실행 및 fork()를 통한 자식 프로세스 생성
3. Phase 2
   * + 파이프라인 기호 |를 통한 여러 명령어 연결 실행 구현
     + 다중 파이프라인 처리 및 프로세스 간 파일 디스크립터 핸들링 구현
4. Phase 3
   * + 백그라운드 명령어(&) 실행 기능 구현
     + 실행 중인 백그라운드 작업 관리 (jobs, fg, bg, kill 등)
   1. **개발 내용**

* **Phase1 (fork & signal)**
  + Fork()를 통해 자식 프로세스를 생성하고, 자식은 execve()로 명령어를 실행합니다. 부모는 waitpid()를 사용해 자식 종료를 감지하며, 각 프로세스는 setpgid()로 별도 그룹으로 설정되어 job control이 가능하게 구성했습니다. (해당 부분은 phase3을 위한 기반을 마련합니다.)
  + 자식 종료 시 SIGCHLD 시그널을 통해 부모는 handler에서 waitpid로 수거합니다. SIGINT, SIGSTP 시그널은 실행 중인 foreground process group에 전달되며, 셀 자체는 유지되고 프롬프트만 재출력 되도록 구현합니다. (이 부분도 이후 단계와 통합되었습니다.)
* **Phase2 (pipelining)**
  + | 기호를 기준으로 명령어른 나누고, 각 명령어 사이에 pipe()를 생성하여 stdin과 stdout을 연결합니다. 각 명령어는 fork()로 실행되고, dup2()로 파이프 입출력을 재지정하고 실행되도록 구성하였습니다.
  + 파이프 개수 (n\_cmds-1)에 따라 필요한 만큼 pipe fd 배열을 생성하고, 각 프로세스마다 적절히 입출력을 redirection하였습니다. 사용하지 않는 pipe는 close()하여 리소스를 정리하고, 마지막에 모든 자식 프로세스를 wait()로 수거하였습니다.
* **Phase3 (background process)**
  + 명령어 끝의 &를 통해 background 실행 여부를 판단합니다. 해당 프로세스는 wait하지 않고 job 리스트에 등록합니다. 이후 jobs, fg, bg, kill 명령을 통해 백그라운드 작업을 조회하고 재현할 수 있도록 구현합니다.
  1. **개발 방법**
  + **Phase 1**

사용자가 명령어를 입력하면, 셸은 먼저 이를 파싱하고 myshell\_execute() 함수로 전달합니다. 이 함수에서 fork()를 호출하여 부모와 자식 프로세스로 분기합니다. 자식 프로세스는 다음 순서로 외부 명령을 실행합니다.

1. setpgid(0, 0)을 호출하여 자신을 새로운 프로세스 그룹으로 등록
2. execve()를 통해 명령어 실행
3. 실패할 경우 오류 메시지를 출력하고 exit(1)로 종료

부모 프로세스는 실행된 프로세스가 foreground인지 background인지에 따라 다음을 처리합니다.

* + - foreground인 경우
      * setpgid(pid, pid)로 자식의 PGID를 설정
      * tcsetpgrp(STDIN\_FILENO, pid)를 통해 터미널 제어를 자식에게 넘김
      * waitpid()로 종료 또는 중지를 대기
      * tcsetpgrp()를 통해 터미널 제어를 다시 myshell에게 복구
    - background인 경우
      * waitpid()를 호출하지 않음
      * job 리스트에 정보를 등록하고 바로 프롬프트로 복귀

시그널 처리는 main에서 다음과 같이 등록합니다.

*signal(SIGCHLD, sigchld\_handler);*

*signal(SIGINT, sigint\_handler);*

*signal(SIGTSTP, sigtstp\_handler);*

핸들러 내부에서는 waitpid(-1, &status, WNOHANG | WUNTRACED)를 통해 자식의 상태를 수거하거나 업데이트하며, job 상태를 “DONE” 또는 “STOPPED”로 변경합니다.

* + **Phase 2**

myshell\_execute\_pipeline() 함수는 | 기호를 기준으로 명령어를 분리하고, 각 명령어 구간을 개별적으로 처리합니다. 다음 흐름으로 구성됩니다.

1. strtok(cmdline, "|")으로 명령어 분리
2. 각 명령어마다 fork()를 호출하여 자식 생성
3. 파이프가 필요한 경우 pipe(fd) 호출
4. 자식에서 dup2()를 사용해 stdin/stdout을 파이프에 연결
5. execve()로 명령어 실행

예를 하나 들자면, 중간 명령어의 경우 다음의 입출력 redirection을 수행합니다.

*dup2(prev\_fd, STDIN\_FILENO); // 이전 명령어의 출력이 나의 입력*

*dup2(pipefd[1], STDOUT\_FILENO); // 나의 출력이 다음 명령어의 입력*

모든 명령어 실행 후, 부모 프로세스는 다음을 수행합니다.

* + - * 파이프를 닫아 리소스 누수 방지
      * 각 자식 프로세스의 pid를 저장하여 waitpid()로 수거

또한 pipeline\_pids[] 배열을 사용해 각 프로세스의 pid를 추적하며, pipeline 중 SIGINT 등 시그널 전달이 필요할 경우 process group 전체에 전파할 수 있도록 준비합니다.

* + **Phase 3**

사용자가 입력한 명령어의 끝에 &가 있는 경우, 셸은 해당 명령을 백그라운드로 실행해야 한다고 판단합니다. 따라서 다음의 절차가 수행됩니다.

1. 입력 라인 파싱 단계에서 &를 감지하고 is\_background 플래그를 설정
2. fork()로 자식 프로세스를 생성하고, 자식은 execve()로 명령 실행
3. 부모는 waitpid()를 호출하지 않고, job 정보를 구조체에 저장하여 관리

*typedef struct {*

*int job\_id;*

*pid\_t pgid;*

*char cmdline[MAXLINE];*

*job\_state state;*

*} job\_t;*

job 상태는 RUNNING, STOPPED, DONE 중 하나이며, 시그널 핸들러(SIGCHLD, SIGTSTP)에서 업데이트됩니다. 사용자는 다음과 같은 명령어로 job을 제어할 수 있습니다.

* + - * jobs: 실행 중인 job 목록 출력
      * fg %n: 지정한 job을 foreground로 가져와 제어권 전달 및 waitpid() 수행
      * bg %n: 중지된 job을 백그라운드로 재개 (kill(SIGCONT))
      * kill %n: 해당 job에 SIGKILL 전달

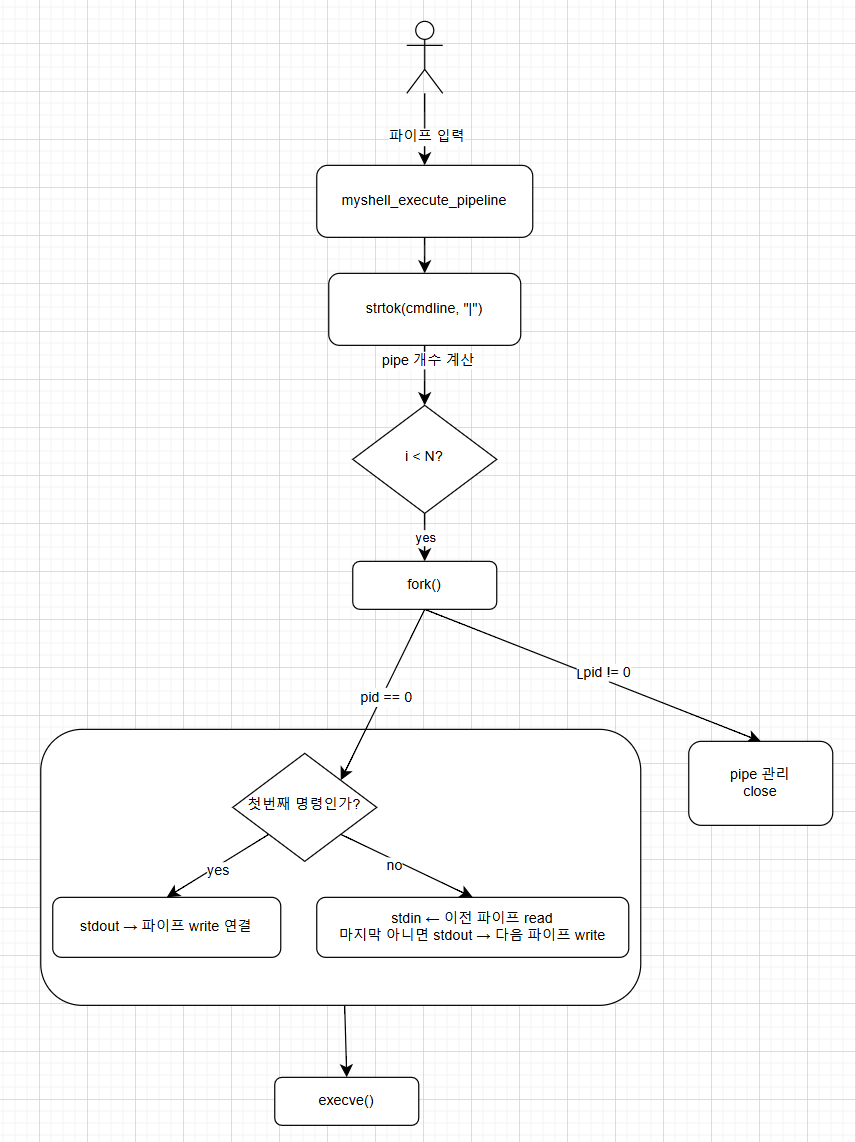
이들 명령은 각각 builtin\_fg(), builtin\_bg(), builtin\_jobs() 등의 함수에서 처리되며, 내부적으로 job 리스트를 순회하고 PGID 기준으로 프로세스를 제어합니다.

1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

**도표, 기술 도면, 라인, 평면도이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

**도표, 텍스트, 라인, 평면도이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**