**Multicore Programming Project 1**

담당 교수 : 최재승 교수님

이름 : 이주헌

학번 : 20191629

1. **개발 목표**

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

이 프로젝트에서는 MyShell이라는 이름의 간단한 셸을 작성한다. 셸은 그래픽 환경이 존재하지 않을 때 사용하는 텍스트 기반의 대화형 인터페이스로, 사용자의 명령을 입력받아 입력값을 분석, 실행한 뒤, 프로그램의 출력값을 보여주고 다시 명령을 기다리는 REPL(Read, Evaluate, Print, Loop) 구조를 가지고 있다.

특히, UNIX 계열 운영체제에서의 셸은 여러 프로그램의 입출력을 서로 잇는 파이프(piping) 기능을 지원한다. 이는 첫째, 모든 프로그램은 단 하나의 기능을 잘 수행해야 하고 둘째, 텍스트야말로 모든 프로그램이 서로 통신할 수 있게 해 주는 “공용어”라는 UNIX 정신에서 유래한다. 셸 사용자는 기능 한 가지를 수행하는 프로그램의 텍스트 입출력을 파이프로 연결하여 필요한 작업을 수행할 수 있다.

이번 MyShell 프로젝트에서는 프로그램 실행, 파이프, 백그라운드 실행을 지원하는 간단한 셸을 3개의 페이즈에 걸쳐 구현한다. 첫 페이즈에서는 프로그램을 실행하는 기본적인 기능을, 두 번째로 파이프를 구현하고, 마지막으로 간단한 병렬 작업을 위한 백그라운드 실행 기능을 추가한다. 이 프로젝트를 통해 POSIX 운영체제, 특히 Linux에서 프로세스와 병렬화를 처리하는 내부적인 구조를 익힐 수 있다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1
   * 명령어 한 줄을 읽어들이고, 명령어의 프로그램 이름과 인자를 분리하여 저장한다.
   * 입력된 명령어는 argv[0]의 내용에 따라 execvp() 함수에 의해 PATH 내에 존재하는 프로그램을 실행한다.
   * 각 명령어는 우선 셸 프로세스를 fork한 후, 자식 프로세스에서 exec\* 계열 함수를 통해 프로세스를 교환한다.
   * 셸 빌트인을 사용해서 명령어 사용 기록을 조회하거나 (history), 현재 디렉토리를 바꾸거나 (cd), 셸 사용을 종료할 수 있다 (quit, exit).
   * Bash와 비슷한 방식으로 history의 내용을 조회할 수 있다. (!! substitution, !n substitution)
2. Phase 2
   * 파이프 기능을 이용해서 두 명령어의 입출력을 연결할 수 있다.
     + (예시) echo hello, world | tee hello.txt 🡪 echo 명령어의 표준 출력이 tee 명령어의 표준 입력과 연결되어 “hello, world”라는 값이 hello.txt와 표준 출력에 저장된다.
   * 파이프 기능은 두 개 이상의 명령어가 주어질 때도 올바르게 동작해야 한다.
     + (예시) find . -type f | grep hello | sort -r 🡪 find 명령어의 표준 출력이 grep 명령어의 표준 입력에, grep 명령어의 표준 출력이 sort 명령어의 표준 입력과 연결된다. 즉, 현재 디렉토리에 있는 모든 파일 중 파일 경로에 hello라는 문자열이 있는 파일을 추린 뒤 역순으로 정렬하여 표준 출력에 저장된다.
   * 파이프 연결은 pipe(2) 함수를 사용하여 파이프를 생성한 뒤, dup(2) 함수를 사용하여 기존에 존재하는 표준 입출력 장치를 대체한다.
3. Phase 3
   * 작업 제어 기능을 사용해서 오랜 시간 동작하는 프로그램을 병렬적으로 실행할 수 있다.
   * 셸 빌트인을 사용해서 작업 목록을 조회하거나 (jobs), 실행 중인 작업을 종료하거나 (kill), 정지되어 있거나 후방에서 실행 중인 작업을 전방으로 끌어오거나 (fg), 정지되어 있는 작업을 후방에서 재개시킬 수 있다 (bg).
   * Ctrl + C를 입력하면 현재 전방에서 실행 중인 작업이 SIGINT 신호를 받고 종료된다.
   * Ctrl + Z를 입력하면 현재 전방에서 실행 중인 작업이 SIGTSTP 신호를 받고 정지된다.
   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
  + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow

POSIX 운영체제에서는 프로그램 프로세스를 바로 생성하는 방법이 존재하지 않는다. 따라서 이러한 운영체제에서는 먼저 부모 프로세스가 자기 자신을 복제한 뒤, 새로 생겨난 복사본이 실행할 내용을 다른 프로그램으로 대체하는 패턴을 사용한다. 먼저 fork(2) syscall을 사용해서 부모 프로세스에서 자식 프로세스를 복사하고, exec(3) 함수군을 사용해서 자식 프로세스의 내용을 새로운 프로그램으로 대체한다. 여기서 exec(3) 함수**군**이라고 표현한 이유는 여러 종류의 프로그램이 존재하기 때문인데, 특히 페이즈 1에서 사용한 함수는 execvp 함수이다. 이 함수는 프로그램 이름과 프로그램 인자의 벡터(v)를 받고, 프로그램 이름을 PATH 환경변수에서 찾아(p) 실행하는 exec(3) 함수이다.

이 페이즈에서의 목표는 실행한 명령어가 종료될 때까지 기다렸다가 다음 명령어를 입력받는 것인데, 이렇게 작성하면 자식 프로세스가 사용자의 명령을 실행하는 도중 부모 프로세스가 미리 사용자 입력을 기다리게 된다. 이 상황을 방지하기 위해 자식 프로세스를 복제한 이후 부모 프로세스는 waitpid 함수를 호출하여 자식 프로세스가 종료될 때까지 기다린다. 현재 페이즈에서는 전방에서 동작하는 프로세스만 고려하므로, 특별한 신호 처리 루틴은 필요하지 않다.

* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)
  + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

파이프를 구현하기 위해서는 기존에 만든 명령어 분석 루틴을 먼저 수정할 필요가 있다. 파이프 문자( | )는 사실상 한 줄의 명령어를 두 개의 명령어로 나누는 역할을 수행하기 때문에, 여러 개의 명령어를 저장할 수 있도록 바꿀 필요가 있다. 또한 파이프 문자는 위에서 보인 예시와 같이 두 개 이상 한번에 사용할 수 있으므로, 파이프 문자를 인식하더라도 그 후 파이프 문자가 다시 등장할 가능성이 있음을 고려한 디자인이 필요하다. 이는 입력된 명령줄 문자열을 간단한 추상 구문 트리(Abstract Syntax Tree) 구조로 파싱하는 방향으로 구현했다.

명령어 파싱이 종료되면 명령어를 실행할 수 있게 되는데, 만약 현재 실행 중인 트리 노드가 파이프 노드라면 따로 표준 입출력을 변경할 수 있게 조치하였다. 먼저 pipe(2) 함수를 사용해서 파이프 파일 서술자 2개를 생성하고, fork(2) syscall을 이용해서 출력 프로그램과 입력 프로그램 프로세스를 만든다. 다음, 각 프로세스에서는 명령어 노드를 실행하기 전 dup2 함수를 사용해서 표준 입출력 장치를 역할에 맞게 파이프의 쓰기 서술자나 읽기 서술자로 설정한다. 이 작업이 완료되면 부모 프로세스에서는 파이프 서술자를 모두 닫고, phase 1에서와 같이 waitpid 함수를 사용해서 두 자식 프로세스가 종료되는 것을 기다릴 수 있다.

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

먼저 명령어 파싱 단계에서 명령어 마지막에 & 문자가 존재하는지 확인하고, 그에 따라 트리 노드 구조체에 후방에서 동작하는 명령인지 명시하는 필드를 추가한다. 명령어가 후방에서 실행되는 경우, 기존 구현 방식과는 달리 셸 프로세스가 자식 프로세스를 기다리지 않고 바로 사용자 입력으로 넘어가야 하며, 자식 프로세스가 종료되면 그 신호를 받아 자식 프로세스를 정리하는 작업을 해 주어야 한다. 여기서 자식 프로세스의 상태가 변경되었을 때 송신되는 SIGCHLD 메시지를 활용하면 자식 프로세스가 중단되거나 종료되었을 때를 알 수 있다.

또한, 작업 관리 기능을 위해 전후방 관계없이 모든 프로세스는 각자 자신의 작업 ID를 할당받는다. 이를 통해 Ctrl+Z를 눌러 사용자가 전방에서 실행 중인 프로세스를 중단했을 때 손쉽게 전방 프로세스를 확인하고 중단시킬 수 있다. 이 중단된 작업은 fg 또는 bg 빌트인 명령어를 통해 전방 또는 후방에서 재개시킬 수 있다.

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

본 프로젝트는 예시로 주어진 shellex.c 파일을 기반으로 하여 많은 수정을 가하는 방식으로 개발하였다. 또한, 각 phase는 이전 phase의 소스코드로부터 수정을 하였으며, 필요한 경우 이후 phase의 수정사항을 이전 phase로 backport하였다.

1. Phase 1

Phase 1에서는 주로 eval 함수와 parse 함수에 대한 수정이 이루어졌다. 먼저 strtok(3) 기반의 파싱을 수행해던 기존 구현체를 버리고, 문자 단위로 파싱하는 알고리즘을 새로 개발하였다. 문자 파싱을 수행하면서, 셸 명령어에서 따옴표를 사용해서 공백을 포함한 인수를 작성하거나, !! 및 !N 치환 기능을 구현할 수 있게 되었다.

eval 함수에서는 기존에 execve 함수만을 사용하여 자기 자신의 프로세스를 대체하던 루틴에서 fork syscall을 추가하여 자식 프로세스에서 다른 프로그램을 실행할 수 있게 바꾸었다. 또, PATH에 존재하는 프로그램 실행을 위해 execve 함수를 execvp 함수로 바꾸었으며, 자식 프로세스가 끝나는 것을 기다리기 위해 waitpid 함수도 함께 사용하였다.

예시 파일에서는 builtin\_command 함수를 호출하여 각종 빌트인 명령어를 처리하는데, 이 프로젝트에서는 builtin\_command 함수를 일종의 dispatcher로 취급하여 builtin\_\* 함수를 각각 실행하는 역할로 바꾸었다. 따라서, 명령 기록을 보여 주는 history 빌트인 명령은 builtin\_history 함수에 구현되었다. 명령 사용 기록은 HOME 아래 .myshell\_pN\_history라는 파일에 텍스트 형식으로 저장된다. 파일명에서 N은 phase 숫자를 의미한다.

이후 phase 2 구현 과정에서 !!, !N 치환 기능에 존재하던 버그를 수정하여 backport하여 기존 구현체와는 명령어 파싱 방식이 달라졌다.

1. Phase 2

Phase 2에서는 파이프라는 새로운 문법이 도입되면서, 한 줄에 여러 명령이 포함될 가능성이 생겼다. 따라서, 기존에 있던 배열 기반 파싱 알고리즘이 AST 기반 파싱 알고리즘으로 대체되었다. 따라서, 하나의 명령은 아래 구조체의 인스턴스 하나이다.

struct node {

enum {

NODE\_TYPE\_EMPTY,

NODE\_TYPE\_COMMAND,

NODE\_TYPE\_PIPE,

} type;

union {

struct {

int argc;

char\* argv[MAXLINE];

bool background;

} command;

struct {

struct node\* left;

struct node\* right;

} pipe;

} data;

};

이후 eval 함수는 위에서 주어진 node를 실행할 수 있는 eval\_node를 호출하며, eval\_node는 현재 노드가 명령 노드(NODE\_TYPE\_COMMAND)이면 명령을 수행(execvp)하고, 파이프 노드(NODE\_TYPE\_PIPE)라면 파이프와 표준 입출력을 설정한 뒤, 프로세스 2개를 생성하여 각 프로세스에서 eval\_node를 재귀적으로 호출한다. 파서 구현체의 특성상 왼쪽 노드는 항상 명령 노드이다. 즉, left | mid | right 명령은 (pipe (command left) (pipe (command mid) (command right)))의 트리 형태로 파싱된다. 따라서, 파이프 개수가 많더라도 시스템 리소스가 허용하는 한 명령을 파이프로 연결할 수 있다.

1. Phase 3

Phase 3에서는 & 문자를 사용하여 명령을 후방에서 실행시킬 수 있는 기능이 추가되었다. 이미 phase 2 구현 시점에 해당 기능을 파서에 추가했기에 파서에는 더 추가할 기능이 없다.

작업 관리 기능을 구현하기 위해 eval 함수를 통해 실행하는 모든 명령어를 전역 변수 배열에 저장하기로 하였다. 작업 하나를 나타내는 구조체는 아래와 같다. 각 작업은 작업 ID, 프로세스 ID, 실행한 명령, 작업 상태를 가질 수 있으며, eval 함수에서 파싱이 끝난 뒤 전역 배열에 저장된다.

typedef int jid\_t;

struct job {

jid\_t jid;

pid\_t pid;

char cmdline[MAXLINE];

enum job\_state {

JOB\_STATE\_INVALID = 0,

JOB\_STATE\_FOREGROUND,

JOB\_STATE\_BACKGROUND,

JOB\_STATE\_STOPPED,

} state;

};

또한, 자식 프로세스가 종료되면 부모 프로세스에서 reaping을 해 줘야 한다. 이를 위해 엔트리포인트인 main 함수에서 signal(2) 함수를 사용해서 SIGCHLD 신호가 들어온 경우 handle\_sigchld 함수를 호출하도록 등록했다. 이 핸들러 함수에서는 SIGCHLD가 들어온 이유에 따라 프로세스를 분류하여 적절하게 처리한다. 프로세스가 종료된 경우 작업 목록에서 삭제하고, 프로세스가 정지된 경우 작업 목록에서 해당 프로세스를 찾아 정지된 상태(JOB\_STATE\_STOPPED)로 바꾼다.

또한, Ctrl + Z를 눌렀을 때, 셸은 종료되지 않고 전방에 있는 프로세스가 정지되어야 하기 때문에 SIGTSTP 핸들러도 설치한다. 이 핸들러에서는 전방에 나와 있다고 표시된 작업을 찾아서 그 프로세스에만 SIGSTOP 신호를 보내고, 자기 자신에게는 아무 작업도 하지 않는다. 해당 자식 프로세스는 다시 부모 프로세스에게 SIGCHLD 신호를 보내서, 위에서 작성한 핸들러가 작업 목록을 관리할 수 있도록 한다.

마지막으로, Ctrl + C를 눌렀을 때, 셸이 종료되지 않고 전방에 있는 프로세스가 정지되어야 한다. 따라서 SIGTSTP 핸들러와 마찬가지로 전방에 있는 프로세스를 찾은 뒤 그 프로세스에만 SIGINT 신호를 보낸다.

다만, SIGTSTP와 SIGINT가 후방에서 실행 중인 작업에게도 전달되는 경우가 발생하여 eval\_node에서 자식 프로세스를 시작할 때 signal(SIGTSTP, SIG\_IGN) 함수를 호출하여 SIGTSTP는 셸 프로세스만이 핸들링하도록 설정하였다.

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)**

Diagram

Description automatically generated

1. **Phase 2 (pipeline)**

**Diagram

Description automatically generated**

1. **Phase 3 (background)**

**Diagram

Description automatically generated**