**Multicore Programming Project 1**

담당 교수 : 최재승

이름 : 김동현

학번 : 20190345

1. **개발 목표**

* 기본적인 shell의 구현을 목표로 하며, 다음과 같은 기능이 수행되도록 한다. Shell을 통해 명령어 입력 받아 유저 정의 함수 혹은 파일을 호출하여 실행한다.

Shell을 통해 파이프라인으로 연결된 명령어 입력 받아 실행한다.

Shell을 통해 입력 받은 명령어를 foreground, background 상태로 실행 및 control한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Phase 1

Shell의 기본적인 명령어를 호출하여 실행한다. History, cd, quit과 같은 user-define function을 구현한다. 이외의 shell 함수에 대해서는 기존의 파일을 읽어와 실행한다.

1. Phase 2

파이프라인으로 연결되어 입력된 명령어를 실행한다. 파이프라인으로 들어온 명령어를 parsing하여 구분한뒤, 각 명령어를 실행한다. 각각의 프로세스의 표준입출력을 control하여 한 프로세스의 출력이 다른 프로세스의 입력으로 전달되도록 한다.

1. Phase 3

앞서 구현한 명령어들에 대해서 Foreground에서 뿐만 아니라 background에서도 실행되도록 한다. 입력으로 명령어를 parsing하여 포그라운드, 백그라운드를 결정한다. Signal handler 및 user-define function 정의하여, 작업 목록 조회, 작업의 실행위치(포그라운드 혹은 백그라운드) 변경, 작업의 실행, 정지, 종료에 대한 명령을 가능하도록 한다.

* 1. **개발 내용**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명

: 입력받은 명령어가 builtin\_command가 아닌 경우, 리눅스에 있는 파일을 실행하여 명령어를 실행한다. 이를 위해 Fork() 함수를 실행하여 자식 프로세스를 만들게 된다. 이를 통해 자식 프로세스(pid==0)와 부모 프로세스(pid==child process’s pid)로 나뉘게 되며, 자식 프로세스에서는 execvp함수를 호출하여 명령어에 대한 파일을 호출하여 실행한다.

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

: 자식 프로세스는 명령어를 처리하고 난 뒤 혹은 해당 명령어를 호출하지 못하고 에러 메세지를 출력하고 난 뒤 exit() 함수를 호출하여 terminate되게 된다. 이때 커널은 부모 프로세스에 SIGCHLD 시그널을 보내 자식 프로세스의 종료를 알린다. 이후 부모 프로세스에서 SIGCHLD 시그널을 받아 Waitpid()함수에서 자식 프로세스에 대한 reaping을 진행한다.

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

: parseline()을 통해 파이프 라인이 존재하는지 여부 및 파이프 라인 및 빈칸에 따른 문자열 파싱을 진행하였다. 파이프 라인의 개수를 나타내는 전역 변수(pipeline\_num)을 참고하여 파이프 라인이 있다면, command\_pipe() 함수를 실행한다. 파이프라인 개수에 따른 파일 디스크립터를 생성한 뒤, pipe함수를 통해 파이프 라인을 만들어 준다. Fork() 함수를 실행하여 프로세스를 나누어준 뒤, dup2() 함수를 통해 명령어의 위치에 따른 파일디스크립터의 입출력 방식을 변경한다. 이후 builtin\_command, execvp 함수를 통해 명령어를 실행한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

: 몇개의 파이프라인이 들어올지 모르는 상황이기에 파이프라인의 개수를 나타내는 전역변수와 반복문을 사용하여 파이프라인을 handling하였다. Parseline() 함수에서 문자열을 파싱하며 파이프 라인의 개수를 파악한다. 이후 command\_pipe() 함수에서 해당 파이프라인 개수만큼 pipefds라는 파일디스크립터를 만든 뒤, 반복문을 이용하여 파이프라인 개수만큼 pipe명령어를 호출하여 파이프를 생성한다. 또다시 반복문을 사용하여 파이프라인으로 나누어진 명령어의 개수만큼 fork를 실행한다. For문의 변수를 활용하여 해당 명령어가 가장 처음 명령어인지, 가장 마지막 명령어인지, 혹은 중간에 있는 명령어인지를 구분하여 dup2() 함수를 활용하였다. 마지막 명령어가 아닌 경우에는 STDOUT을 바꾸어 주었으며, 첫번째 명령어가 아닌 경우에는 STDIN을 바꾸어 주었다. 이후 모든 파이프라인을 닫아준 뒤, 명령어를 실행한다. 모든 명령어가 실행되고 난 뒤, 또 다시 모든 파이프라인을 닫아주고, 부모 프로세스에서는 wait()함수를 통해 terminate 된 자식 프로세스를 reaping해 준다.

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

: 입력으로 들어온 커멘드에 대한 parsing 과정에서 ‘&’ 여부를 확인한다. Bg값을 통해 백그라운드로 들어온 경우라면, sigsuspend 함수를 호출하지 않고, 백그라운드 실행여부를 알려준 뒤, 다시 커멘드를 입력받도록 한다.

* 1. **개발 방법**
* **Phase1**
  + Eval() 수정

: history 기능을 위해 입력 받은 명령어를 저장하는 Insert\_node() 함수를 추가한다.

Builtin\_command에서 user-define가 아닌 경우, fork()를 통해 자식 프로세스를 생성한다. 자식 프로세스에서는 execvp함수를 호출하여 현재 path에 있는 실행파일 혹은 /bin/ 디렉토리에 있는 실행파일을 실행하도록 한다. 부모 프로세스에서는 waitpid() 함수를 통해 자식 프로세스의 terminate를 기다리도록 한다.

* + Builtin\_command() 수정

: exit 명령어가 입력될 경우 quit 명령어와 같은 기능이 수행되도록 한다. 이때 쉘을 종료하기 전 write\_history() 함수를 호출하여 명령어 히스토리를 파일에 저장하도록 한다.

“Cd”명령어가 입력될 경우 command\_cd() 함수를 호출하도록, “history”, “!%”이 입력된 경우 command\_history() 함수가 호출되도록 한다.

* + Parseline() 수정

: 따옴표(“)로 묶인 문자열을 하나의 인자로 처리하기 위해 parsing 방법을 변경하였다. 따옴표로 시작하는 경우, 따옴표로 끝나는 경우, 따옴표 안이 아니면서 빈칸(‘ ‘)을 만난 경우로 나누어 처리한다.

* + History command 구현

: 히스토리 기능을 구현하기 위해 \_comm\_node 구조체를 선언하였다. 해당 구조체에는 연결리스트의 노드로 사용되며, 명령어 번호, 명령어의 길이, 명령어 문자열, 연결리스트를 위한 노드 포인터로 구성되어 있다. 이외에도 연결리스트 구현을 위한 노드 포인터 head, tail, curr과 노드의 개수를 나타내는 line\_num 변수, 히스토리 파일의 저장위치를 저장하는 문자 배열을 전역 변수로 선언하였다.

History() 함수에서는 “history” 명령어가 입력된 경우 연결리스트를 순회하며, 명령어 기록을 출력한다. “!!” 명령어가 입력된 경우 연결리스트의 마지막 노드에 접근하여 해당 명령어를 eval함수의 인자로 넘겨 호출한다. “!#” 명령어가 입력된 경우 #부분의 문자열을 파싱하여 인덱스 값을 구한다. 이때 형식에 맞지 않거나 범위를 벗어나게 되면 함수를 종료한다. 이후 해당 인덱스 노드에 접근하여 저장된 명령어를 eval함수의 인자로 넘겨 호출한다.

Read\_history() 함수는 파일(history.txt) 파일에 저장된 데이터를 읽어와 연결리스트에 저장한다. 이때 저장된 데이터는 이전에 실행한 쉘의 명령어 기록이다.

Write\_history() 함수는 쉘을 정상적으로 종료하기 전에 호출되며, 연결리스트에 저장된 명령어 기록을 파일에 저장하며 해당 노드를 deallocate시킨다.

Insert\_node()와 delete\_node() 함수는 연결리스트에 노드를 저장 및 삭제 시키는 함수이다. 히스토리 기능에서 노드의 삽입은 가장 마지막 노드에, 삭제는 가장 첫번째 노드부터 실행된다.

* + Cd command 구현

: dirName 문자 배열에 현재 디렉토리를 나타내는 “./”를 저장한 뒤, 인자로 들어온 경로를 이어붙인다. 만약 인자가 없는 경우 home 디렉토리를 구하는 함수 getenv(“HOME”)을 호출하여 배열에 복사한다. Chdir함수를 호출하여 현재 디렉토리를 변경한다. 만약 반환값이 -1인 경우에는 디렉토리로 이동하지 못한 경우이므로 오류 메세지를 출력한다.

* **Phase2**
  + Eval() 함수 변경

: 파이프라인의 개수를 나타내는 pipeline\_num 변수를 확인하여 command에 파이프라인 존재 여부를 확인한다. 0이 아닌 경우 파이프 라인이 존재하므로 command\_pipe 함수를 호출하고, 그렇지 않은 경우 기존 구현한 flow로 진행한다.

* + Parseline() 함수 변경

: 기존 빈칸(‘ ‘)과 따옴표(‘”’)으로만 파싱하던 방법에 파이프라인(‘|’)을 기준으로 추가하였다. 문자열을 파이프라인을 기준으로 파싱하여 문자포인터 배열 parsePipe에 나누어 저장한다. 이후 하나의 원소(파이프라인으로 나누어진 명령어 하나)를 기존의 방법으로 파싱하여 argv 배열에 저장한다. 이때, 파이프라인으로 구분되는 인덱스를 pipeIdx 배열에 저장한다.

* + Pipeline 구현

: 파이프라인 구현을 위해 전역변수로 pipeline\_num과 pipeIdx[MAXARGS]를 선언하였다. Pipeline\_num은 파이프라인(‘|’)의 개수로 파이프라인이 포함된 명령어인지를 확인하고, 파이프 라인을 기준으로 몇개의 명령어로 나누어졌는지 확인할 수 있다. pipeIdx 배열은 argv배열에 저장된 인자값들이 어디서 파이프라인을 통해 나누어졌는지를 알려주는 배열이다.

Command\_pipe() 함수를 통해 파이프라인으로 이어진 명령어를 실행한다. 기존의 일차원 문자열 포인터 배열인 argv를 pipeIdx 배열을 참고하여 파이프로 나누어진 명령어의 집합을 하나의 배열로 하는 2차원 문자열 포인터 배열 newArgv에 저장한다. 이후 파이프라인의 개수의 2배(하나의 파이프에 입력부분과 출력부분이 필요하기 떄문에) 크기의 파일 디스크립터 배열 pipefds를 선언한다. 파이프의 개수만큼 반복문을 통해 pipe() 함수를 호출하여 파이프를 생성한다. 파이프로 나누어진 명령어의 개수만큼 반복문을 실행한다. Fork()를 통해 자식 프로세스를 생성한다. 자식 프로세스는 부모 프로세스의 파일 디스크립터 역시 복사해오는 점을 이용한다. Dup2함수를 통해 마지막 명령어가 아닌 경우, 표준 출력에 현재 파이프의 출력 부분을 연결한다. 또한 첫번째 명령어가 아닌 경우, 표준 입력에 이전 파이프의 입력 부분을 연결한다. 이후 모든 파이프를 닫은 뒤, 명령어를 실행한다. 부모 프로세스에서는 자식 프로세스의 Terminate를 기다리고, 자식 프로세스에 대한 reaping을 wait함수를 통해 실시한다.

* **Phase3**
  + Job 리스트의 구현

: 실행되는 job들의 정보를 관리하기 위한 구조체 \_job\_node를 설정하였다. 구조체의 원소로 job의 번호와 상태(RUNNING, SUSPENDED), 실행위치(FOREGROUND, BACKGROUND), pid, ground pid, 커멘드의 정보를 담고 있다. 이러한 작업은 리스트에 저장되어 관리된다. Numofjobs는 리스트에 저장되어 있는 작업의 개수를 저장하고 있는 변수이다. Shell이 시작되면, initjob() 함수를 호출하여 모든 작업의 리스트를 빈 상태(EMPTY)로 초기화하고, numofjobs도 0으로 초기화 한다. 이후 명령어가 실행될 때 마다 addjob() 함수를 호출하여 해당 작업을 리스트에 넣는다. 또한 job이 terminate되거나 kill되는 경우 deletejob()을 호출하여 해당 작업을 리스트에서 제거한다.

* + Signal handler의 구현

: 총 3개의 시그널 핸들러를 구현하였다. SIGCHLD 시그널에 대한 핸들러는 waitpid를 호출하여 프로세스의 reaping을 진행한다. 이때, SIGTSTP 시그널에 대해서도 SIGCHLD 핸들러가 호출되기 때문에 이를 예외처리 해주어야 한다. 만약 SIGTSTP이라면 job\_list의 내용을 바꿔준다. 자식 프로세스가 종료되는 경우에는 모든 시그널을 막고 job\_list에서 해당 job을 제거해 준다. SIGINT 시그널에 대한 핸들러는 포그라운드에서 돌아가고 있는 작업을 찾아 Kill 함수를 사용해 해당 그룹 프로세스에 SIGINT 시그널을 보낸다. SIGTSTP 시그널에 대한 핸들러는 포그라운드에서 실행되는 프로세스 그룹에 Kill 함수를 통해 SIGTSTP 시그널을 보낸다.

* + Jobs 함수 구현

: command\_jobs() 함수는 현재 실행되거나 멈춰있는 백그라운드 작업을 보여주는 함수이다. 사용자가 jobs 커멘드를 입력했을 경우 실행되며 job\_list를 반복문을 통해 순회하며 EMPTY 상태가 아닌 작업을 출력한다. 출력 양식은 작업 번호, 작업의 상태(RUNNING, SUSPENDED), 커멘드를 나타낸다.

* + Bg <job> 함수 구현

: command\_bg\_job() 함수를 통해 백그라운드에 멈춰있는 작업을 백그라운드에서 실행하도록 한다. 우선 함수 내에서 올바른 입력이 들어왔는지 확인한다. bg %# 형식인지를 확인하며, #은 작업의 번호이다. 만약 그렇지 않은 경우 잘못된 입력임을 출력하고 함수를 종료한다. 올바른 입력이 들어온 경우 반복문을 통해 job\_list를 순회하며 해당 작업 번호를 찾는다. 해당 프로세스 그룹에 SIGCONT 시그널을 Kill 함수를 통해 전달하고, job\_list의 내용은 백그라운드에서 실행중인 상태로 변경한다.

* + Fg <job> 함수 구현

: command\_fg\_job() 함수를 통해 백그라운드에 멈춰있거나 실행 중인 작업을 포그라운드에서 실행하도록 한다. 우선 함수 내에서 올바른 입력이 들어왔는지 확인한다. fg %# 형식인지를 확인하며, #은 작업의 번호이다. 만약 그렇지 않은 경우 잘못된 입력임을 출력하고 함수를 종료한다. 올바른 입력이 들어온 경우 반복문을 통해 job\_list를 순회하며 해당 작업 번호를 찾는다. 해당 프로세스 그룹에 SIGCONT 시그널을 Kill 함수를 통해 전달하고, job\_list의 내용은 포그라운드에서 실행중인 상태로 변경한다.

* + Kill <job> 함수 구현

: command\_kill\_job() 함수를 통해 백그라운드에 멈춰있는 작업을 백그라운드에서 실행하도록 한다. 우선 함수 내에서 올바른 입력이 들어왔는지 확인한다. kill %# 형식인지를 확인하며, #은 작업의 번호이다. 만약 그렇지 않은 경우 잘못된 입력임을 출력하고 함수를 종료한다. 올바른 입력이 들어온 경우 반복문을 통해 job\_list를 순회하며 해당 작업 번호를 찾는다. 해당 프로세스 그룹에 SIGKILL 시그널을 Kill 함수를 통해 전달하고, SIGCHLD 시그널을 통해 job\_list에서 해당 작업이 제거된다.

* + Main 함수 수정

: Signal() 함수를 사용하여, SIGINT, SIGCHLD, SIGTSTP 시그널에 대한 핸들러 함수를 설정한다.

* + Eval 함수 수정

: SIGCHLD 시그널을 블로킹하여 fork() 이후 작업을 job\_list에 더하기 전, parent process에서 작업을 리스트에 더하기 전 SIGCHLD 핸들러가 실행되어 job\_list에서 작업을 제거하지 못하도록 한다. 또한 setpgid함수를 통해 그룹 프로세스를 세로 설정하여 새로운 그룹으로 설정한다. 또한 sigsuspend() 함수를 통해 자식 프로세스가 terminate되는 것을 기다리는 과정을 atomic하게 실행될 수 있도록 한다.

* + Builtin\_command 함수 수정

: 커멘드에 들어온 입력이 “jobs”, “bg”, “fg”, “kill” 인 경우에 해당 커맨드에 대한 함수를 호출하도록 추가하였다.

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

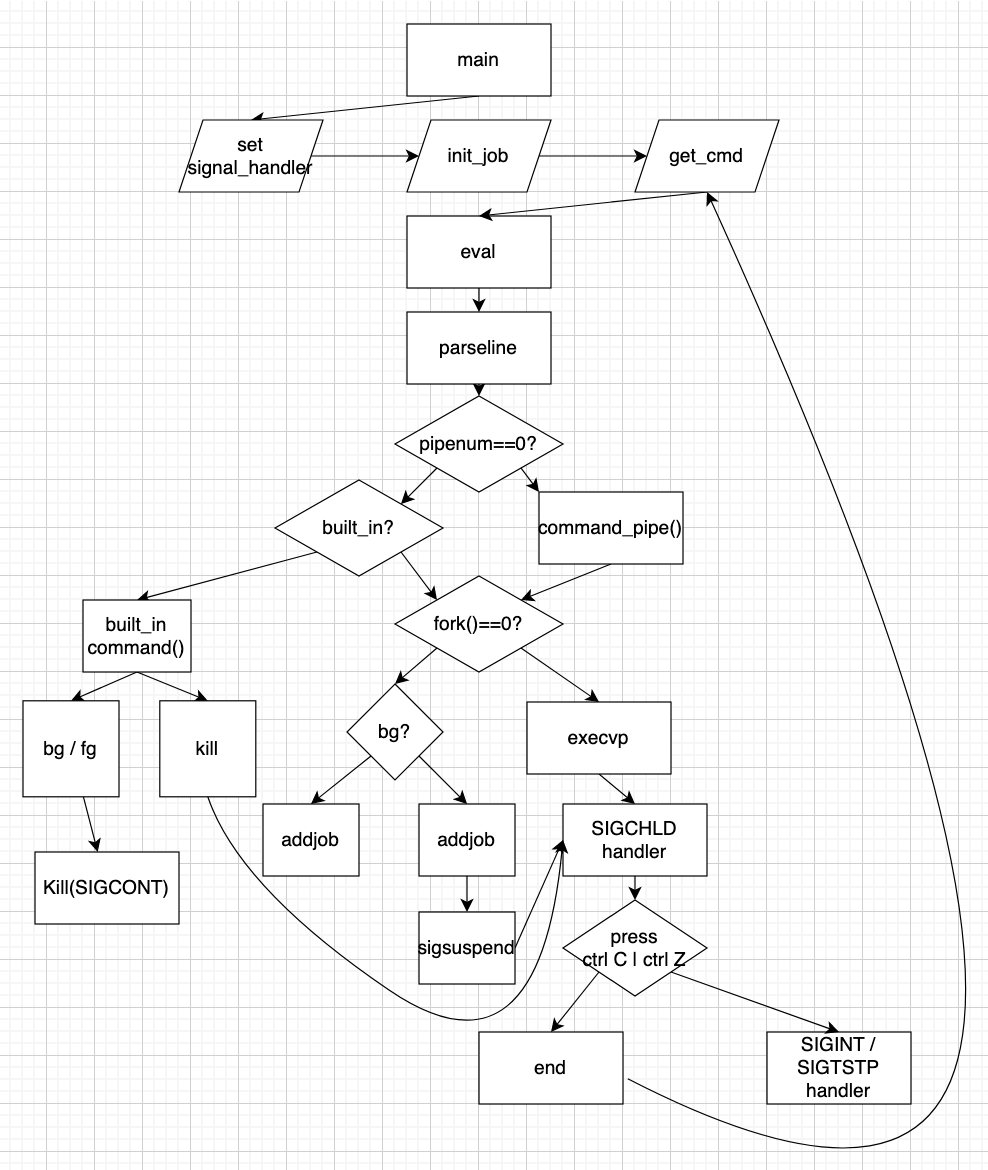
자동 생성된 설명

1. **Phase 2 (pipeline)**

**도표이(가) 표시된 사진

자동 생성된 설명**

1. **Phase 3 (background)**

****