**Multicore Programming Project 1**

담당 교수 : 최재승

이름 : 김상현

학번 : 20181605

1. **개발 목표**

구현 목표는 user command를 수행할 수 있는 shell program을 개발하는 것이다. Shell program은 내장된 명령어들 외에도 shell 이외에도 작성된 프로그램들의 binary를 실행한다. 이를 위해서는 input command parsing, forking, piping, background, foreground, kill, 등의 명령어들을 수행할 수 있어야 한다. 그리고 추가로 shell은 이전에 수행했던 명령을 따로 저장한다. Bash shell의 경우 .bash\_history에 저장된다.

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

1단계 specification에 명시된 명령어들을 shell이 수행할 수 있어야 한다.

* + - cd: 명령어 다음 인자로 들어온 path로 이동하여 디렉토리를 이동할 수 있는 기능을 제공한다. 다음 인자가 없을 경우 $HOME 디렉토리로 이동한다.
    - ls: 명령어 다음 인자로 들어온 옵션과 마지막 인자로 들어온 디렉토리 path를 이용하여 옵션에 따라 해당 디렉토리의 내용을 shell에 출력한다.
    - mkdir: 명령어 다음 인자로 들어온 디렉토리를 생성한다.
    - rmdir: 명령어 다음 인자로 들어온 디렉토리를 삭제한다.
    - touch: 명령어 다음 인자로 들어온 이름의 파일을 생성한다. 파일 이름이 path로 주어진 경우 해당 경로에 주어진 이름으로 파일을 생성한다.
    - cat: 명령어 다음 인자로 들어온 이름의 파일의 내용을 shell에 출력한다. 파일 이름이 path로 주어진 경우 해당 경로에 주어진 이름의 파일의 내용을 출력한다.
    - echo: 명령어 다음 인자로 들어온 텍스트를 shell에 출력한다. 인자로 variable을 받을 수도 있다.
    - history: shell이 실행한 명령어들을 기록한다. 명령을 수행할 때마다 .shell\_history 파일에 저장한다. 추가로 ‘!!’과 ‘!#’ 명령어를 사용하여 이전에 수행한 명령을 다시 수행하게 한다. 동일한 명령은 수행되지 않는다.

1. Phase 2

C library의 pipe 함수를 사용하여 실행되는 process들의 입력과 출력을 redirect할 것이다. Pipe의 최대길이가 제한되어 있는지 확인해볼 필요가 있다.

**command1 | command2 | … | commandn**

1. Phase 3

프로세스를 foreground, background에서 수행할지 선택할 수 있는 기능과 실행되고 있는 프로세스를 중단하고 재시작 하는 기능을 구현한다. 그리고 현재 실행되고 있는 job들에 대한 정보를 출력하는 기능을 제공해야 한다.

* + - Jobs: 실행 중인 job과 suspend된 job들을 상태와 함께 출력한다
    - bg: 다음 인자로 받은 번호에 해당되는 background job을 중단 상태에서 재시작한다.
    - fg: 다음 인자로 받은 번호에 해당되는 실행 상태의 background job을 foreground job으로 바꾼다.
    - kill: 다음 인자로 받은 번호에 해당되는 job의 process를 끝낸다. Pipe 명령도 수행이 가능하다.
  1. **개발 내용**
* **Phase1 (fork & signal)**

fork를 하여 child process를 생성할 때 child process는 parent process의 모든 내용을 다른 메모리 물리적 공간에 동일하게 같게 된다. Fork 함수는 parent process에게 child process의 pid, child process에게는 0을 반환한다. 이를 이용하여 pid의 0 유무에 따른 if-else 구문을 통해 0인 경우에는 parsing된 명령어를 통해 실행파일의 위치와 이외의 argument들을 execve 함수에 넘겨 프로그램을 child process에서 실행한다. Parent process는 else문을 따라가서 waitpid 함수를 사용해 child process를 기다린다. Phase1은 background process가 구현되지 않았기 때문에 항상 child process를 기다린다. waitpid는 parent process가 SIGCHLD 신호를 받았을 때 인자로 주어진 기다리고 있는 child process가 보낸 signal이 맞을 경우 SIGCHLD 신호를 보낸 process의 pid를 반환한다.

* **Phase2 (pipelining)**

Shell에 입력된 명령어를 parsing하면서 ‘|’ 기호가 몇 개 있는지 확인한다. 그리고 그 개수에 2를 곱한 만큼의 int array를 할당한다. pipe 함수는 int array의 포인터를 받아 포인터 위치와 그 다음 위치에 FD pipe를 활성화하기 때문이다. STDIN=0, STDOUT=1과 유사하게 포인터 위치에 해당하는 FD에서 읽고 그 다음 위치에 쓸 수 있다. 아래의 예시를 통해 pipe라인이 어떻게 형성되어 있는지 알 수 있다.

[1] [0] [3] [2] [5] [4] [7] [6] [9] [8]

process1 | process2 | process2 | process3 | process4 | process5

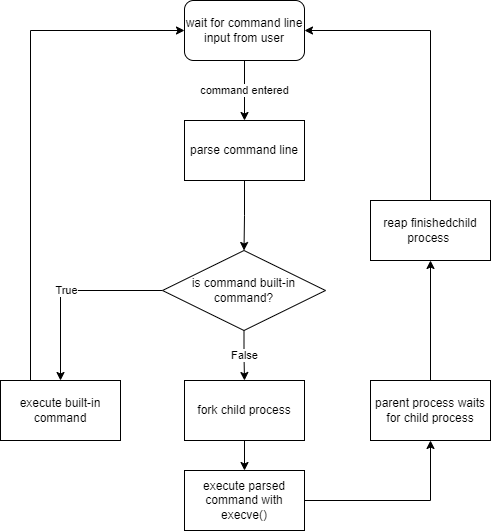
Right side가 process의 출력, left side가 process가 입력할 file descriptor이다. Parent process의 FD는 수정하지 않아도 되기 때문에 구현이 안전하다고 볼 수 있다.

* **Phase3 (background process)**

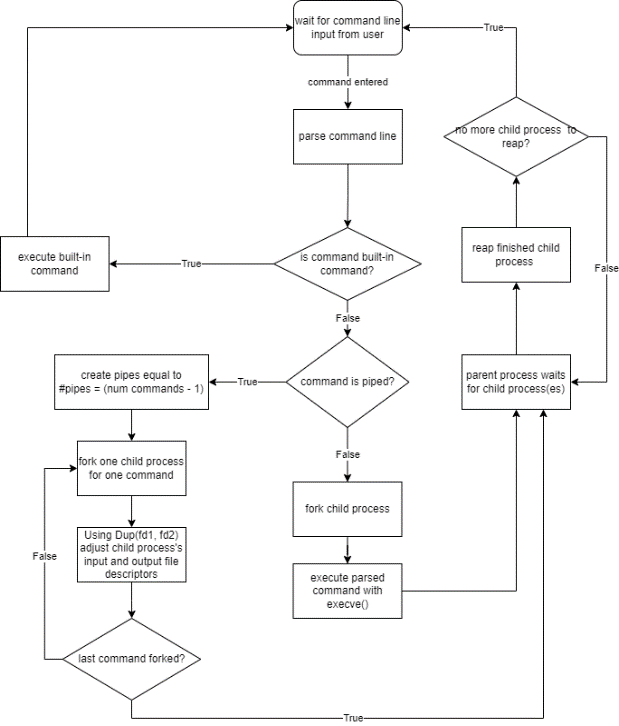
Background process를 수행하기 위해서는 shell이 다른 기능을 수행하면서 background process의 signal을 받을 수 있어야 한다. 이를 위해 parent process인 shell이 자신의 child process에 signal를 어떻게 처리할지 정하는 sigchld\_handler를 직접 구현해야 한다. Build-in 명령어를 제외한 shell에 입력된 명령어에 &가 붙어있을 경우 parsing을 통해 이를 확인한다. 확인한 이후 child process에서 수행되는데 이때 shell은 parent process에서 바로 waitpid를 통해 한정없이 기다리지 않고 바로 다음 명령어를 받아 수행할 수 있다. 다만 자신의 child process 중 하나가 SIGCHLD 신호를 보냈을 때 이에 대해 reaping을 수행하며 job의 상태를 바꾼다. 그리고 shell이 새로운 명령을 수행하는 중에 SIGCHLD 신호에 방해를 받지 않도록 Sigprocmask 함수를 사용하여 해당 구간에서는 SIGCHLD에 반응하지 않도록 했다.

* 1. **개발 방법**
     + History를 구현하기 위해 HIST\_ENTRY 구조체를 만든다. 또한 specification에 따라 history는 무한히 늘어날 수 있기 때문에 linked list 형태로 구현하도록 한다. 따라서 field로 history element에 저장할 명령어와 다음 history element를 가리키는 pointer가 필요하다. 그리고 global variable로 history linked list의 head와 tail을 저장한다. 그리고 쉘마다 각자 실행된 directory에 history 파일을 저장해야 하기 때문에 해당 디렉토리의 이름을 따로 저장한다. History를 구현하기 위해 필요한 함수들을 아래와 같다.
       1. void open\_shell\_history(void): shell history 파일을 열어 global 파일 스트림에 지정하는 함수
       2. void save\_shell\_history(void): 메모리에 있는 history를 저장하는 함수
       3. void add\_command\_to\_history(char \*cmdline): 해당 커맨드를 메모리에 저장
       4. void remove\_command\_from\_history(char \*cmd): 메모리에 있는 history 중 cmd에 대응하는 것 삭제
       5. int history\_command(char\* extension, char\* cmdline): 입력된 history 명령어를 수행하는 함수
     + Piping을 구현하기 위해서는 pipe된 command들의 수만큼 child process를 생성해야 한다. 따라서 필요한 함수들은 아래와 같다.
       1. char\*\* parse\_pipe\_command(char\*\* command): 서로 pipe를 해야 하는 command들을 parsing 하는 함수이다.
       2. void run\_pipes(char\*\* piped\_commands, int num\_commands): Parsing이 끝난 command들을 각자의 child process로 실행하는 함수이다. 해당 함수에서는 inter-child-process communication을 위해 pipe를 함수에서 초기화하고 process들의 input과 output file descriptor들을 서로 연결한다. 각 child의 input과 output도 redirect해야하기 때문에 Dup2(fd1, fd2) 함수도 사용해야 한다.
     + 마지막 phase에서는 child process들의 상태를 fg, bg, kill 함수, 그리고 [CTRL+Z]를 통해 바꿀 수 있어야 한다. 따라서 signal을 다룰 때 필요한 함수들(Signal, Sigprocmask, etc.)을 비롯해 child process의 signal을 받았을 때 이에 대응하는 signal handler 함수들을 만들어야 한다. Wait 혹은 Waitpid 함수를 통해 child의 status를 retrieve 할 수 있고 status를 확인하는 macro 함수들을 통해 해당 child의 상태를 어떻게 바꿀지 결정할 수 있다. 마지막으로 생성된 job들의 상태를 실시간으로 저장하는 table을 관리하는 함수가 필요하다. 이에 따라 만들어질 함수들은 아래와 같다.
       1. void fg(int jobID), void bg(int jobID), void killjob(int jobID): jobID를 받아 해당 job을 각각 순서대로 foreground, background, 그리고 kill하는 함수들이다.
       2. deleteJob(int jobID): job table에서 해당 job을 삭제한다.
       3. addjob(int jobID, enum state): 새로운 job의 상태와 함께 table에 저장한다.

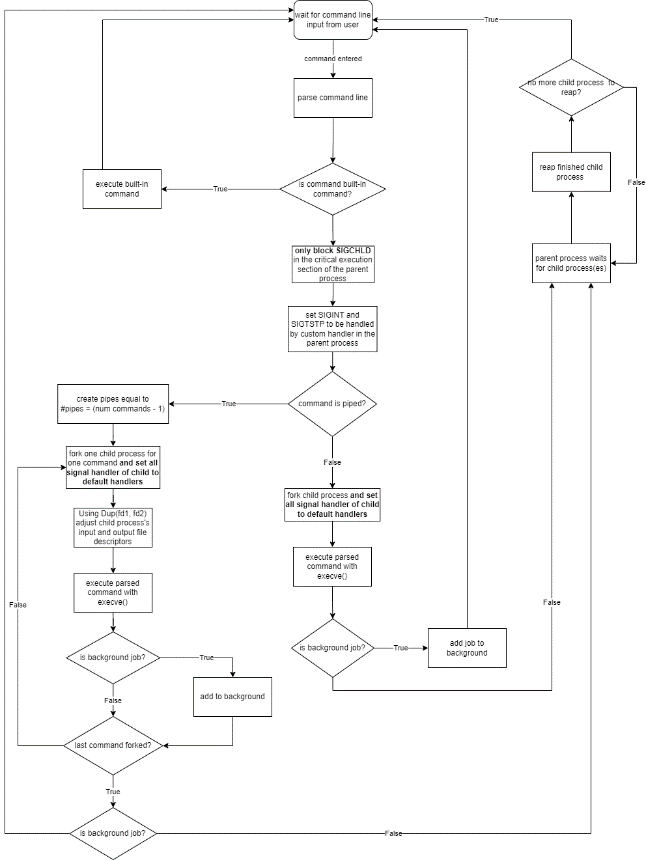
1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**



1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****