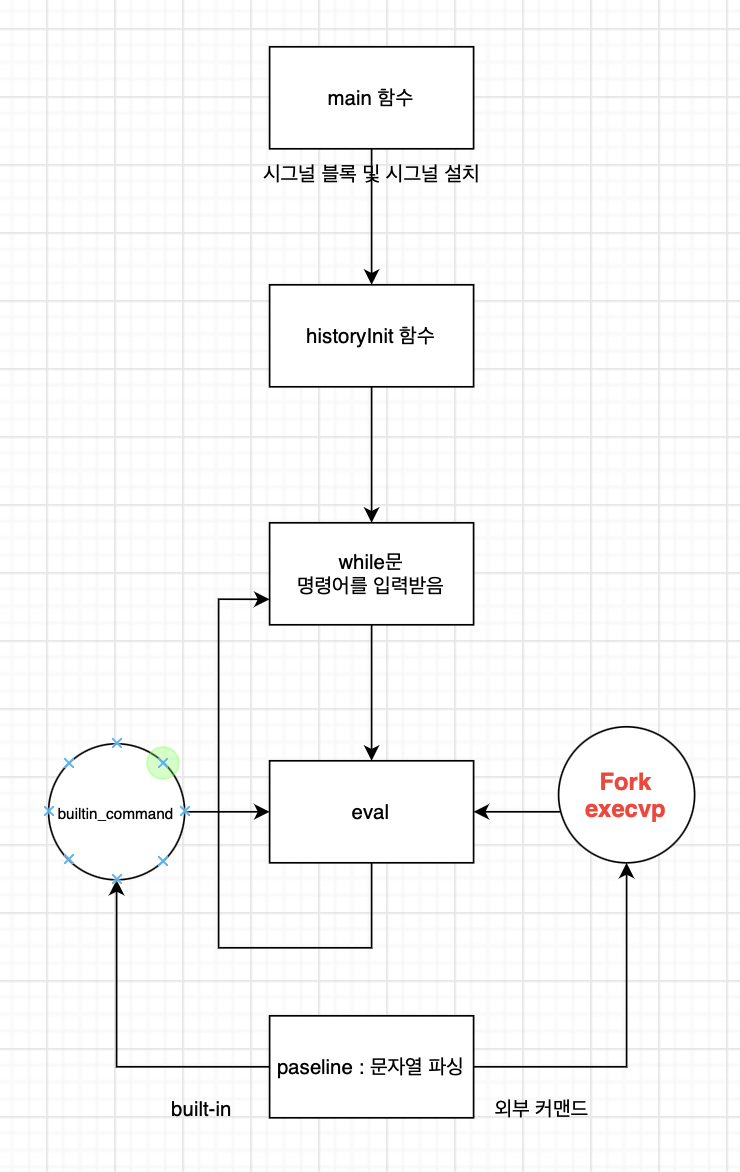
**Multicore Programming Project 1**

담당 교수 : 최재승 교수님

이름 : 나호영

학번 : 20211531

1. 개발목표  
   phase1 : cd, ls, mkdir, history 등의 명령어를 받아 실행하는 기본적인 mini shell을 구현한다.  
   phase2 : 파이프가 포함된 명령어를 실행하도록 구현한다.  
   phase3 : 백그라운드로 명령어를 실행하도록 구현한다.
2. 개발범위및내용
   1. 개발 범위
3. Phase 1  
   ls, mkdir, rmdir, touch, cat, echo 명령어는 fork 함수를 이용하여 자식 프로세스를 생성해 외부 프로그램을 실행하게 하였다.  
   cd, history, exit은 built-in 명령어로 따로 쉘 프로그램의 코드 내부에서 구현하였다. cd 명령어는 chdir 함수를 이용하여 디렉토리를 이동할 수 있게 하였고 exit 명령어는 exit 함수를 이용해 쉞 프로그램이 종료되게 하였다. history와 !!, !# 명령어는 구현 내용이 많았는데 간략하게 설명하자면 따로 함수를 작성하고 외부 파일인 .history에 히스토리를 저장하는 방식을 이용해 명령어 history를 볼 수 있게 하였고 history를 참조해 !!, !#와 같은 명령어들로 특정 라인을 실행할 수 있게 하였다.
4. Phase 2  
   | 를 이용하여 명령어들을 구분해 파이프 명령을 구현하였다. 이전 명령어의 출력이 다음 명령어의 입력으로 들어가 최종적으로 마지막 명령어의 출력은 터미널에 출력된다. 외부로 연결된 명령을 포함해 built-in으로 구현된 명령들도 파이프를 이용하여 실행될 수 있게 하였다.
5. Phase 3  
   & 를 이용하여 백그라운드 명령을 구분해 실행하도록 구현하였다. 백그라운드 명령은 부모 프로세스가 fork를 이용해 생성한 자식 프로세스를 wait나 waitpid로 기다리지 않고 독립적으로 쉘을 계속해서 사용할 수 있도록 하게 만들었다.
   1. 개발 내용  
        
      Phase1 (fork & signal)
   * fork를 통해서 child process를 생성하는 부분에 대해서 설명  
       
     built-in 명령어를 제외한 명령어들은 모두 fork를 통하여 구현하였다. fork를 이용하여 실행 중인 프로세스(부모 프로세스)를 복제하고 새로운 자식 프로세스를 생성한다. 이 후에는 부모 프로세스와 자식 프로세스가 동시에 실행되고 동일한 메모리, 코드 등을 복사하지만 pid가 다르다. 자식 프로세스의 생성을 확인하기 위해 fork 호출 후의 pid를 반환받는데 이 때 0인 경우 자식 프로세스이고 부모 프로세스는 자식 프로세스의 pid를 받는다. 이후에 이 자식 프로세스에서 exec 함수를 이용하여 명령을 실행한다.
   * connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow   
       
     - sigint / sigquit  
     control + C 를 키보드로 입력하면 인터럽트로 인해 sigint가 프로세스에 도착한다. 이 시그널은 실행을 중지하고 종료하는 시그널이다. control + |를 입력하면 sigquit가 도착한다. 이 시그널은 프로세스를 종료시키고 코어를 덤프하는 역할을 수행한다. 하지만 phase1 명세서에는 foreground job으로 작동하는 명령어가 존재하지 않는다. 따라서 이 경우를 생략하고 쉘 프로그램 자체가 sigint나 sigquit로 종료되는 것을 방지하기 위해 해당 시그널을 부모 프로세스에서는 블락하고 fork로 자식 프로세스가 생성될 때는 다시 unblock하게 하였다.  
       
     - sigchld  
     main 함수에서 부모 프로세스가 waitpid 함수를 통해 자식 프로세스의 종료를 기다리게 되면, 해당 시간동안 프로세스가 블락된다. 따라서 sigchld 핸들러 함수에서 waitpid 함수 옵션에 WHOHANG을 주어 자식 프로세스를 reaping하는 방법을 생각해볼 수 있다. 이 방법을 이용하여 부모 프로세스가 자식 프로세스가 종료될 때까지 기다리지 않으면서 작업을 수행하며 지속적으로 자식 프로세스를 모니터링 할 수 있으나 시그널 핸들링 함수는 비동기적으로 작동하기 때문에 쉘 프로그램이 자식 프로세스가 끝날 때까지 기다리지 않으면 동기화 문제가 발생하여 명령어 수행이 완료되기도 전에 쉘 입력창이 나타나는 문제가 있었다. 따라서 eval 함수에서 waitpid 함수로 자식 프로세스의 종료를 기다리게 하여 동기적으로 작동하게 하였고 sigchld 핸들러 함수를 이용해 자식 프로세스의 시그널을 부모 프로세스가 받았을 때 상태를 확인할 수 있게 하였다. 이 때 시그널 핸들러에서 printf와 같은 standard IO 함수를 호출하면 async-signal-safe 하지 않기 때문에 unix IO 함수를 호출했다.  
       
     Phase2 (pipelining)
   * Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)  
     먼저 parseline 함수에서 파이프가 있는 명령인지 확인하고 적절히 파싱하였다. 리턴값이 0이면 파이프 명령이 아닌 명령이고 리턴값이 1이면 파이프 명령, 리턴값이 -1이면 유효하지 않은 파이프 명령으로 구분하였다. 파이프 명령일시 파이프 커맨드를 수행하는 함수를 호출하였다. 이 함수에서는 파이프의 개수를 100개로 제한하고 argv에서 |가 존재하는 개수를 세고 해당 인덱스를 pipeIndex에 저장한 뒤 argv 인덱스를 NULL값으로 바꾸었다. 그 후 파이프를 만드는데 필요한 만큼 파일 디스크립터 배열 fd 이차원 배열을 만들었다. 이 후 for문을 순환하며 파이프를 생성하고 fork를 이용해 자식 프로세스들을 생성한 뒤 연결하였다. 이 때 첫번 째 명령은 dup2함수를 이용하여 출력을 파이프 출력에 연결하고 중간 명령들은 이전 파이프 파일 디스크립터 입력에 입력을 연결하고 다음 파이프 출력에 출력을 연결하였다. 마지막 명령은 이전 파이프 입력을 입력에 연결하였다. 부모 프로세스에서는 다 사용한 파일 디스크립터들을 닫았고 각 자식 프로세스들을 waitpid를 이용해 reaping하였다.
   * Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명  
       
     일단 파이프가 존재하고 해당 명령이 유효한 파이프 명령이라면 해당 명령을 실행하는 mypipe 함수가 호출된다. 이 때 해당 명령에서 파이프 개수에 제한을 두었다. 정해진 수의 pid 배열과 파일 디스크립터 fid 배열을 정적 할당해야 하기 때문이다. 파이프 개수가 초과되면 리턴되게 만들었다. 이 상황이 아니라면 파이프 개수 + 1 만큼 자식 프로세스를 생성하여 파이프로 연결해 최종적으로 마지막에는 터미널에 출력되게 만들었다.  
       
     Phase3 (background process)
   * Background (’&’) process를 구현한 부분에 대해서 간략히 설명  
       
     백그라운드인지 포어그라운드인지 &가 끝에 붙은 명령을 받으면 구분할 수 있도록 명령어를 파싱하였다. 이 때 &가 마지막 argv에 붙어있더라도 백그라운드 명령어로 구분할 수 있도록 하였다. 백그라운드 명령 실행으로 구분되면 일반적으로 fork 이후 부모 프로세스가 자식 프로세스를 기다리게 하는 waitpid를 호출하지 않게 하였다. 또한 백그라운드 프로세스를 관리하기 위한 jobs, bg, fg, kill 과 같은 명령어들을 built-in 명령으로 구현하였다. 이 명령들을 구현하기 위해 SIGINT, SIGSTP, SIGCONT와 같은 시그널들을 핸들링했다.
   1. 개발 방법  
        
      phase1 : 외부로 연결되어 실행되는 명령어들은 fork를 이용해 자식 프로세스를 생성해 execvp 함수로 실행되게 하였다. 이 때 부모 프로세스가 waitpid를 이용하여 해당 작업이 완료될 때까지 대기한다. 외부 프로그램으로 실행되지 않는 명령어들인 built-in 명령어들은 builtin\_command 함수에서 위와 다르게 자식 프로세스에서 실행되는 것이 아니라 부모프로세스에서 직접 실행된다. quit, exit은 exit 함수를 통해 쉘이 종료되게 하였다. cd 명령어는 chdir함수를 이용해 디렉토리를 이동하도록 구현하였다. 마지막으로 history 와 !!, !# 명령은 구현 과정이 복잡하였다. 프로그램이 실행되었을 때 이전 history 기록을 읽어올 수 있도록 미리 정의되어 있는 히스토리 저장 기록 파일인 historyInit 함수를 호출해 histfilePath 파일을 읽어 기록들을 카운트하고 읽어올 수 있게 만들었다. 그리고 명령어가 입력될 때마다 history\_add 함수를 호출하여 histfilePath를 열어 끝에 추가되도록 만들었다. 이전 명령어들을 출력해주는 history, !!, !# 명령어는 history\_command 함수를 호출해 각각 파라미터 type을 이용해 0, -1, 나머지로 구분하여 해당 함수에서 처리를 진행한다. history는 histfilePath를 FILE 포인터를 이용해 open하여 순차적으로 line들을 읽어서 터미널에 출력한다. !!는 history와 마찬가지로 histfilePath를 FILE 포인터를 이용해 읽어온 뒤 가장 마지막 명령어를 읽어들이고 해당 명령어를 인자로 eval 함수를 호출하여 실행되게 하였다. !#도 읽어온 뒤 해당하는 라인의 명령어를 출력하고 eval 함수를 호출하여 실행되게 하였다.  
        
      phase2 : 먼저 |가 들어간 명령어들을 올바르게 파싱하기 위해 parseline 함수에서 argv에 명령어들을 파싱할 때 |를 기준으로도 파싱되도록 코드를 추가하였다. 이렇게 파싱된 파이프 명령이 유효한지 확인하는 checkPipeCommand 함수를 호출하여 명령이 유효한지 검사하였다. 그 후 유효한 파이프 명령은 구분되어 eval 함수에서 파이프 명령을 실행하는 mypipe 함수가 호출되어 해당 함수에서 실행된다. 이 함수에서 명령어에서 파이프의 개수와 파이프의 위치를 저장하고 해당 파이프 개수+1 만큼 반복문을 돌면서 각각 명령어들을 순차적으로 pipe 함수를 통해 파이프를 생성해 dup2 함수를 사용해 입력과 출력을 연결하였고 마지막 명령에서 결과적으로 터미널에 출력되도록 구현하였다.  
        
      phase3 : 백그라운드 프로세스를 확인하기 위한 jobs 명령어 구현 과정 중에 백그라운드 작업 중인 job들을 저장하기 위한 구조체인 job을 선언하고 해당 구조체에 pid, status, command 멤버들을 선언하였다. job 구조체 배열인 jobs를 전역변수로 생성하고 각각 생기는 job들을 해당 배열에 추가하였고 전체 index를 관리하기 위한 integer 전역 변수 num\_jobs를 이용하였다. 명령어를 입력받아 실행하는 부분에서 백그라운드 실행이면 waitpid를 호출하지않고 쉘이 독립적으로 계속 입력을 받을 수 있도록 하였다. 또한 SIGTSTP 시그널을 받았을 때 포어그라운드 프로세스만 중지시키고 백그라운드 작업으로 변경되도록 시그널 핸들링을 하였다. 이 때 SIGTSTP 시그널 핸들러에서 부모 프로세스가 현재 실행중인 포어그라운드 프로세스의 pid로 kill을 이용해 SIGTSTP을 보내고 jobs 배열에 추가한다. 백그라운드 작업을 관리하기 위해 fg, bg, kill 를 모두 built-in 명령으로 구현하였는데 모두 % 이후의 숫자를 인덱스로 가지는 jobs 배열의 job을 대상으로 작동하게 구현하였다.

1. 구현결과
   1. Flow Chart
2. Phase 1 (fork)  
   빌트인 명령어가 아닌 명령어들은 외부로 연결되어 실행되게 된다. 이 때 fork를 이용해 자식 프로세스를 생성하고 execvp를 호출하여 명령을 실행한다.  
   
3. Phase 2 (pipeline)  
   파이프 명령어들을 파싱하여 유효 검사를 진행한 다음 mypipe 함수에서 결과적으로 실행되게 된다. 이 때 mypipe 함수에서 파이프의 개수 + 1만큼 반복문을 돌려 구분된 명령어들을 순차적으로 pipe와 dup2 함수를 이용하여 입력, 출력을 연결해서 결과적으로 마지막 명령어의 출력이 터미널에 출력되게 한다.  
   pasted-image.tiff
4. Phase 3 (background)  
   백그라운드 명령어를 입력받으면 해당 명령어를 백/포어그라운드인지 구분하여 백그라운드 명령이면 waitpid함수를 이용해 기다리지 않고 실행되게 구현하였다.

