**System Programming Project 2**

담당 교수 : 김영재

이름 : 이도건

학번 : 20180411

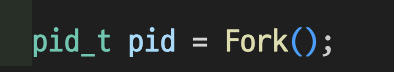
1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* **(MyShell을 만드는 전체적인 개요에 대해서 작성하면 됨.)**이 프로젝트의 개발 목표는 리눅스 쉘의 주요 개념인 시스템 수준의 프로세스 제어, 프로세스 시그널링, 프로세스 간 통신, 그리고 백그라운드에서 프로세스와 작업 실행에 대해 이해하고, 이를 성공적으로 개발하는 것이다. 위 기능을 모두 지원하는 맞춤형 리눅스 쉘을 직접 프로그래밍해야 한다.
* .**개발 범위 및 내용**
  1. **개발 범위**
* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1
2. Phase 2
3. Phase 3
   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
    - 우선 command line 을 받아서 parseline 함수로 파싱하여 띄어쓰기를 기준으로 argv[] 에 넣어 준다. 이 argv 는 command\_ptr 이라고 하는 구조체의 char\*\* argv 필드에 넣어 주고, fork and handle command 항목으로 복사한다. 그리고 fork() 를 실행해 자식 프로세스를 하나 만들고, 프로세스 id 를 pid 변수에 저장한다. 흐름 제어는 If 문을 이용하여 하였다. 우선 현재 프로세스가 부모 프로세스라면 (if pid > 0) waitpid(pid,&status,0) 함수를 실행해 chlid process 가 실행된 이후 끝날 때 까지 기다린다. 현재 프로세스가 자식 프로세스라면 (if pid == 0)  
      run\_command (char\*\* argv) 함수를 실행한다. 인자로는 파싱한 argv 배열 (command\_ptr 구조체에 넣어서) 을 넘겨준다. 넘겨준 인자를 통해 만약 “./” 로 시작하는 커맨드이면 파일을 실행하고 싶다는 것이니 execv 함수를 통해 실행시켜준다. 그게 아니라면 bin 과 usr bin path 를 각각 확인해 해당하는 위치에 파일이 있다면 파일을 execvp 를 이용해 실행시켜준다.
  + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow
    - Child process 는 죽을 때 parent process 에게 SIGCHLD 를 보낸다. 하지만 기본적으로 SIGCHLD 를 받았을 때 프로세스는 아무것도 하지 않는다. 또한 parent process 에서 fork 한 이후 바로 waitpid 의 실행을 통해 자식 process가 죽을때까지 기다리고, reaping 을 한 이후 진행하므로 따로 SIGCHLD 에 대한 signal handler 를 install 하거나 할 필요 없이 waitpid 함수만 쓰면 된다. 그렇다면 waitpid 내부의 flow 를 설명하겠다.  
      Waitpid 함수 내부에서 죽은 자식을 감지했을 때 (sigchld 신호) 자식이 죽은 것을 reap 하고, 현재 프로세스 (부모 프로세스) 의 pending bit vetor 에서 SIGCHLD 의 부분을 다시 0으로 만들어준다. Waitpid 함수가 끝난 이후 parent 프로세스의 flow 는 다시 main 함수로 돌아가 while 문 내부를 한번 더 돌게 된다.
* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)
    - 우선 위에 설명한 command\_ptr->argv 를 활용해 총 몇 개의 커맨드가 있는지를 확인하였다. 이를 활용해 command 의 개수보다 한 개 적은 양의 파이프를 생성하고, pipe 함수를 통해 초기화해주었다. 그리고 For 문을 활용하여 fork 이후의 과정을 커맨드의 개수만큼 반복하였다. 내부의 Parent Process 를 핸들하는 부분에서는 현재 커맨드의 인덱스 (몇 번째인지에 대한 정보 활용)에 따라 해당하는 파이프의 read/write end 를 적절히 막아주었다. 그리고 child 에서는 커맨드의 인덱스에 따라 dup2를 이용해 이전 커맨드의 출력이 파이프를 활용, 이후 커맨드의 입력으로 들어갈 수 있게 설정해주었다. 그리고 set\_up\_current\_command\_args 를 통해 현재 실행할 커맨드가 정확히 무엇인지에 대한 정보를 얻어 cur\_command\_args 에 넣어주었다., 따옴표를 제거한 뒤 run\_command 를 실행하여 커맨드를 실행하였다. 이 과정을 for loop 만큼 반복하였다.
  + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명
    - 파이프라인의 개수, 즉 커맨드의 개수(-1)가 n 개일때의 경우까지 모두 구현하였다. 우선 부모 프로세스에서는, 현재의 커맨드가 첫 번째 커맨드가 아닌 경우 자식 프로세스에서 read 를 끝내려면 파이프의 읽기 부분을 닫아줘야 하므로 해당 파이프의 read 부분을 close 하였다. 또한 현재의 커맨드가 마지막 커맨드가 아닌 경우 자식 프로세스가 다음 프로세스에 전해주는것을 마치기 위해 해당 파이프의 write 부분을 close 하였다. 또한 현재의 커맨드가 마지막인 경우 cur\_command\_args 내부의 메모리를 정리해주었다. 자식 프로세스에서는 현재의 커맨드가 첫 번째 커맨드가 아닌 경우 dup2 를 활용해 이전 파이프에 적힌 내용을 읽어올 수 있도록 한다. 현재의 커맨드가 마지막 커맨드가 아닌 경우 dup2를 이용해 현재 프로세스가 출력하는 내용을 다음 프로세스의 파이프에 쓸 수 있게 한다.
* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명
    - 우선 fork 를 하기 전에 parseline 에서 해당 커맨드가 백그라운드 커맨드인지 확인한다. 백그라운드 커맨드라면 fork 하고 나서 부모 프로세스의 반응이 달라진다.   
      우선 부모 프로세스가 reap 을 해주지 않고, signal handler 에 의존한다. 따라서 부모 프로세스는 백그라운드 자식 프로세스가 죽지 않아도 막힘 없이 계속 실행할 수 있다.  
      또한 부모 프로세스는 background job 의 정보들과 pid 를 받아서 job list 내부에 저장한다.   
      자식 프로세스는 다른 프로세스들과 동일하게 실행한다.   
      이 프로세스가 죽게 되면 부모에게 SIGCHLD 신호가 가는데, 이 신호를 받으면 핸들러를 통해 job\_list 내부에 있는 자식 프로세스 엔트리를 지워 준다.
  1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**
* **프로세스의 hierarchy**
* **텍스트, 스크린샷, 폰트이(가) 표시된 사진

  자동 생성된 설명**
* Hierarchy 에 대한 설명
  + Defs.h, defs.c : 프로그램에서 사용할 구조체를 정의하고, 프로그램 전반에서 사용되는 편의용 함수 (log function, error function, 문자열 파싱 function 등) 을 정의한다.
  + Phase3.c, Phase3.h : Phase 3의 background job, signal handling 과 연관된 함수들을 구현하였다.
  + Runner\_handler.h,Runner\_handler.c : fork 에 필요한 함수들과 eval 내부 fork flow 를 구현한 fork\_handle\_command 함수를 정의하였다.
* **PHASE 1**
  + **텍스트, 스크린샷, 폰트이(가) 표시된 사진

    자동 생성된 설명**구조체이다. 현재 유저가 입력한 Command 의 정보를 쉽게 넘겨주기 위해 만들었다. 파싱한 argv 문자열 배열, background process 를 실행하는 커맨드인지를 알려주는 bg, 유저가 입력한 command line 전체를 저장한 command, 그리고 job\_list 에 해당 커맨드를 추가했는지를 알려주는 already added 필드들이 존재한다.
  + 
    - Fork 와 reaping 을 담당하는 함수이다. 내부에서 fork 도 하고 리핑도 해준다.
    -   
      Fork 를 통해 pid 에 값을 넣고 부모 자식 프로세스로 갈린다.  
      텍스트, 스크린샷, 폰트이(가) 표시된 사진

      자동 생성된 설명부모 프로세스에서는 위와 같은 코드로 자식 프로세스의 실행이 종료될때까지 기다린 후 reap 을 해주고 실행을 계속한다.  
      텍스트, 스크린샷, 폰트이(가) 표시된 사진

      자동 생성된 설명child process 에서는 run\_command 함수에 넘겨줄 command line 을 세팅해주고, 세팅한 것에서 따옴표들을 모두 지워준 뒤, Run\_command 에 넘겨준다 (함수 추후 설명 예정)
  + 
    - Cur\_command\_args 에다가 현재 실행해야 하는 커맨드들만 넣어주는 함수이다. 예를 들면 [“ls”,”-al”] 이런식으로 말이다.
    - 자세한 기능은 Phase 2 에서 설명한다. 현재 알아야 하는 것은,   
      텍스트, 스크린샷, 폰트이(가) 표시된 사진

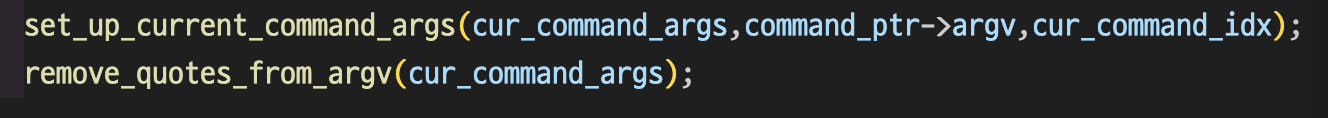
      자동 생성된 설명  
      argv 의 커맨드들중 필요한 것을 cur\_command\_args 에 각각 공간을 할당하여 붙여넣기 한다는 것이다.
  + 
    - 이 함수는 argv 배열로 받은 커맨드의 각 부분들을 합쳐 하나의 커맨드로 실행하는 함수이다.  
       폰트, 텍스트, 스크린샷, 그래픽이(가) 표시된 사진

      자동 생성된 설명
    - 텍스트, 스크린샷, 폰트이(가) 표시된 사진

      자동 생성된 설명   
      Phase1 에서 요구하는 실행 파일들은 모두 bin/ , usr/bin 내부에 있다. 따라서 두 개의 경로를 만들어 준 후, 두 번 execvp 를 실행한다. 둘 중 하나의 경로만 맞고, 하나의 execvp 가 실패하면 프로그램이 뻗지 않고 다시 돌아오니 두 개를 연달아 놔도 상관이 없다. Execvp 가 제대로 실행이 되어서 현재 프로세스가 변하지 않았다면 command 가 유효하지 않은 것이니 이를 출력해준다.
* PHASE 2
  + 
    - 추가된 기능들이 있다.
    -   
      우선 num\_of\_pipes 함수를 이용해 함수에 커맨드가 몇 개가 있는지를 알아온다.
    - 텍스트, 스크린샷, 폰트이(가) 표시된 사진

      자동 생성된 설명  
      그리고 pipe 를 세팅해준다. 파이프는 커맨드 개수보다 한 개가 적게 필요하다. 예를 들어 커맨드가 세 개이면 1,2를 이어주는 파이프 하나, 2,3을 이어주는 파이프 하나 이렇게 두 개가 필요하다. 또한 하단에
    - init\_pipe() 함수를 통해 반복문을 돌며 파이프의 개수만큼 pipe() 를 실행해주어 파이프로 만든다.  
      또한 cur\_command\_args 배열인데, 이 배열은 현재 실행해야 하는 args 를 저장하는 문자열 배열이다. 예를 들어, “cat csapp.h | grep open”의 첫 번째 커맨드를 실행해야 한다면 “cat” “csapp.h” 가 저장된다.
    -   
      이번 Phase 의 핵심이 되는 for 문이다. 이 포문을 커맨드의 개수만큼 돌며 각각 커맨드를 처리한다.
    - 텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

      자동 생성된 설명  
      이 사진은 포문 내부에서 fork 를 한 이후 (이 바로 위에 fork()가 있고 pid 에 child pid 를 저장한다.)의 모습이다. 파이프를 관리하는 핵심적인 알고리즘이 들어가 있다. 현재의 커맨드가 첫 번째 커맨드가 아닌 경우 이전 파이프의 read end 에서 child process 가 읽어오는 것을 끝마쳐주기 위해 이전 파이프의 read end 를 닫아 준다. 현재의 커맨드가 마지막 커맨드가 아닌 경우 자식 프로세스가 현재의 파이프에 내용을 전해주는것을 마치기 위해 해당 파이프의 write 부분을 close 하였다. 다음의 커맨드가 이 파이프에서 읽어갈 것이다. 또한 command 가 마지막 커맨드인 경우 cur\_command\_args 가 할당한 메모리를 모두 초기화하였다.
    - 텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

      자동 생성된 설명  
      이 사진은 for 문 내부에서 child process 일때 dup 을 통해 fd 를 관리해주는 부분이다. 현재의 커맨드가 첫 번째 커맨드가 아닌 경우 dup2 를 활용해 프로세스가 stdin 으로 읽어오지 않고 이전 파이프의 read end 로부터 읽어오게 한다. 현재의 커맨드가 마지막 커맨드가 아닌 경우 dup2를 이용해 현재 프로세스가 출력하는 내용을 stdout 에 출력하는 것이 아닌 현재 파이프의 write end 에 출력하여 다음 프로세스가 access 할 수 있게 한다.
    - 따라서 이를 활용하면 n 개까지의 pipeline command 를 실행할 수 있다.
  + 
    - 자잘한 문자열 관련 함수들이다. 우선 위에 함수를 보면 텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

      자동 생성된 설명  
      argv 내부를 순회하며 어디서부터 기록을 시작해야할지를 위의 for 문에서 설정해준다. 그리고 밑에 if 문에서 기록할 때가 오는 것을 체크하고, 기록할 순간이 오면 더 이상 기록을 하지 않아도 될 때까지 기록을 한다. 그리고 이 바로 밑에 cur\_command\_args[j] = NULL 이 있는데, 문자열 배열이 null terminated 되어야 하므로 넣었다.  
        
      다음 remove quotes from argv 함수를 보면   
      텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

      자동 생성된 설명  
      매우 단순한 함수이다. Argv 내부의 각각의 string 들에서 따옴표를 모두 제거한 후 다시 argv 에 넣어 준다.
  + 텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

    자동 생성된 설명
    - log\_force 와 동일한 기능을 가진 함수이다. Piping 때문에 printf 함수를 써도 terminal 에 출력하지 못하는 사태가 발생하는데, 이 함수 내부에서 직접 버퍼를 만들어서 바로 write 함수를 통해 터미널에 직접 출력하는 함수를 만들었다. (로그 찍는 용)
* **PHASE 3 (구조체와 함수 설명)**
  + 텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

    자동 생성된 설명
    - 이 구조체는 job list 에 들어가는 구조체이다. Job list 출력 시 맨 앞에 표시되는 job\_id, job 의 pid, job 의 command 등을 포함하고 있고, doubly linked list 로 구현하였다. Recency 는 텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

      자동 생성된 설명  
      jobs 를 이용하여 출력하였을 때 앞에 +,- 가 출력되는 것을 구현하기 위한 Enum 이고, is\_bg 는 이 커맨드가 bg 인지 알려주는 boolean 값, 그리고 process state 는 “RUNNING” 혹은 “STOPPED” 로 나뉜다.  
      Job\_list\_front 는 이 doubly linked list 의 맨 앞 원소이고, id\_for\_next\_job 는 다음에 추가되는 job 의 job id 를 계산하는데에 도움을 주는 변수이다.
  + 텍스트, 스크린샷, 폰트이(가) 표시된 사진

    자동 생성된 설명우선 SIGCHLD 에 대한 시그널 핸들러를 설치하였다. 이 핸들러를 실행하는 동안에는 SIGCHLD 신호가 blocked 되며, system call 하는 중에 방해를 받으면 다시 시작한다.
  + 텍스트, 스크린샷, 폰트이(가) 표시된 사진

    자동 생성된 설명
    - Block signal 함수를 편의를 위해 정의하였다. 이 함수 내부에서 빈 마스크에 signum 으로 들어온 값만 마스킹한 후, 원래의 마스크와 합쳐서 블로킹 (SIG\_BLOCK) 옵션을 달아 현재 프로세스의 비트 벡터를 마스킹해준다. Unblock signal 함수도 위 기능과 유사하다.
  + 텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

    자동 생성된 설명부모에게 설치하는 SIGCHLD 핸들러이다. 우선 wait pid 함수를 연속적으로 돌아 죽거나 좀비가 된 자식 프로세스들이 있는지 확인하고, 이 자식 프로세스들을 모두 reap 하고 job\_list 에서 삭제하는 것이 전부이다. 이때 WNOHANG 옵션을 걸어서 죽은 child 가 없다면 바로 0을 반환하고 waitpid 함수를 탈출하도록 설정해주었다. 따라서 죽은 child 가 없는 상태여도 흐름에 지장이 생기지 않는다.
  + job\_list 에 job 를 추가하는 함수이다.   
    텍스트, 스크린샷, 폰트이(가) 표시된 사진

    자동 생성된 설명  
    이렇게 위에서 설명한 pid, job\_id, recency command 등을 설정해준다. 그리고 doubly linked list 에 insert 하는 과정을 통해 리스트에 추가하게 된다. 이는 기초적인 과정이니 생략하고, 특이한 점은   
    텍스트, 스크린샷, 폰트이(가) 표시된 사진

    자동 생성된 설명recency 를 설정해주는 방식이다. Job list 를 순회하면서, job 들의 recency 를 한 단계씩 낮춰준다. 왜냐하면 새로운 프로세스가 CURRENT 가 될 것이고, 이전에 CURRENT 였던 process 는 PREVIOUS 가 되어야 하며, 이전에 PREVIOUS 였던 프로세스는 NORMAL 이 되어야 하기 때문이다.
  + 폰트, 그래픽, 스크린샷, 그래픽 디자인이(가) 표시된 사진

    자동 생성된 설명
    - 여기서 jobs list 를 출력하게 된다. Option (jobs 커맨드 뒤에 붙는 옵션) 을 받아서 그 옵션대로 하게 된다.   
      “-l” 을 받으면 job 들의 pid 까지 출력을 하고, 받지 않으면 그냥 이름들과 상태만 출력한다. 터미널의 jobs 커맨드를 출력하는 결과랑 똑같다.   
      함수 내부를 보면 recency char 를 설정해주는 것을 볼 수 있는데, recency 가 CURRENT 면 +, PREVIOUS 면 -, 그 외에는 빈칸을 설정해주는 것을 볼 수 있다. 그리고 jobs 는 background process 만 출력해야 하므로 job\_t 구조체 내부의 is bg 필드를 사용하여 구별한다. 그리고 job 를 추가하게 되면 id\_for\_next\_job 를 하나 증가시켜준다. 이 id\_for\_next\_job 는 항상 잡 리스트 내의 job id 가 가장 큰 잡의 job id 보다 1 큰 값을 나타내야 한다.
  + 
    - Job id 를 받아서 그에 해당하는 jobs 의 pid 를 반환해준다. 단순히 링크드 리스트를 순회하면서, job\_id 가 일치하는 jobs 를 찾아 그 job 의 pid 를 반환한다.
  + 
    - Job list 에서 pid 에 해당하는 job 을 삭제하는 함수이다. 더블리 링크드 리스트를 순회하면서 pid 가 같은 job 을 찾아서 그 앞의 노드와 뒤의 노드를 서로 이어주고, 해당 노드는 free 하는 작업을 거친다. 여기서 찾은 노드의 next가 null 이면, 즉 job id 가 가장 큰 노드를 삭제해야 하는 상황이면 다음에 들어오는 job id 가 현재의 노드의 job id 를 대체해야 하므로 (예를 들어, 1,2,4 가 있는데 4를 지우면, 다음 job 은 5가 아닌 4가 되어야함) id\_for\_next\_job 를 1 감소시킨다.
* **PHASE 3 (플로우와 기능 설명)**
  + 
    - 텍스트, 스크린샷, 폰트이(가) 표시된 사진

      자동 생성된 설명  
      for 문에 들어가자 마자 fork 하기 전에 sigchld 신호를 블락해버린다.  
      텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

      자동 생성된 설명  
      그리고 포크를 하기 전에 만약 커맨드가 fg 라면 job list 에서 해당 job 을 제거하고, wait 함수를 통해 job 이 forground 에서 돌아가도록 (부모 프로세스가 이 자식 프로세스가 끝날때까지 더이상 실행을 하지 않도록) 한다.
    - 텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

      자동 생성된 설명  
      이 이프문은 부모 프로세스에서 실행된다.커맨드가 백그라운드 커맨드가 아니라면, 이것은 포그라운드 커맨드이다. 즉 프로세스끼리의 실행이 겹치는 것(부모가 먼저 terminal 에서 CSE3010….을 출력하는 경우)을 막기 위해 기다렸다가 직접 reap 을 해줘야 한다. 따라서 기다렷다가 reap 을 해주고, reap 을 해준 이후 SIGCHLD 신호의 블락을 풀어서 SIGCHLD 핸들러가 그 이후에는 작동할 수 있게 한다
    - 텍스트, 스크린샷, 폰트이(가) 표시된 사진

      자동 생성된 설명  
      이 이프문 역시 부모 프로세스 내부에서 자식 프로세스가 백그라운드 프로세스인 경우 실행된다. 백그라운드 프로세스 이므로 add\_job 를 활용해 job list 에 현재 프로세스를 넣고, 현재 커맨드 포인터에 이미 이 커맨드와 연관된 잡은 잡 리스트에 넣어주었다고 표시해준다.  
      Child process 인 경우 그냥 계속 실행해주면 된다.

1. **구현 결과**
   1. **Flow Chart**
2. **도표, 라인, 폰트이(가) 표시된 사진

   자동 생성된 설명Phase 1 (fork)  
     
     
     
     
     
     
     
     
     
     
     
     
   terminated 이후에는 계속 CSE…> 로 가서 다음 커맨드를 읽을 준비를 한다.**
3. **텍스트, 폰트, 도표, 스크린샷이(가) 표시된 사진

   자동 생성된 설명Phase 2 (pipeline)**
4. **텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

   자동 생성된 설명텍스트, 폰트, 스크린샷, 도표이(가) 표시된 사진

   자동 생성된 설명Phase 3 (background)**스케치, 그림, 친필, 디자인이(가) 표시된 사진

   자동 생성된 설명