**Multicore Programming Project 1**

담당 교수 :박성용

이름 :백상욱

학번 :20190388

1. **개발 목표**

간단한 리눅스 쉘을 만들면서 여러 프로세스를 제어, 시그널링, 프로세스 간의 커뮤니케이션, 프로세스와 job을 관리하는 방법에 대해 학습한다. Phase 1에서는 cd, ls, mkdir, echo, exit, history, !!, !# 등의 기본적인 쉘 명령어를 구현한다. Phase 2에서는 Phase 1의 구현 내용을 확장하여 명령어 인자 개수 제한이 없는 명령어 파이프를 구현한다. Phase 3에서는 백그라운드의 프로세스 운영을 구현한다. 사용자가 fore그라운드와 백그라운드 사이에서 job들을 자유로이 움직일 수 있도록 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

Pwd, cd, ls, mkdir , rmdir, echo, exit, history, !!, !# 등의 기본적인 쉘 명령어를 수행하도록 한다. 각 명령어의 실행 결과는 bash shell 명령어와 동일한 동작을 수행한다. 그리고 history에서 출력하는 cmd 리스트는 각 폴더의 history.txt에 저장된다.

1. Phase 2

Phase 1에서 구현한 쉘의 명령어들이 파이프에서 구현되도록 한다. 파이프라인은 왼쪽 명령어의 결과가 오른쪽 명령어의 입력으로 들어가는 기능이다. 아래와 같이 실행된다. Ls의 결과가 grep history의 입력이 되었고 cat example.txt의 결과가 grep – v “abc”의 입력이 된다. 그리고 다시 한번 grep의 결과가 sort – r의 입력이 되도록 한다.

1. Phase 3

백그라운드 프로세스를 구현한다. 명령어 끝 부분에 &를 붙이면 백그라운드 프로세스에 해당 명령을 돌려 running 상태가 된다. Jobs를 입력하면 실행 중 혹은 실행이 중단된 jobs 목록을 job 번호, 상태, 명령어 순으로 출력한다.

* 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

명령어를 입력받으면 그 명령어를 parseline을 통해 파싱하고 명령어를 얻어낸 후에 builtin\_command인지 확인한다. Builtin\_command는 fork를 한 후 수행하는 것이 아니므로 builtin\_command에서 실행하고 builtin함수가 아닌 경우 fork를 실행한다. 그리고 fork의 반환값에 따라 parent process와 child process를 구분합니다. 만약 fork 반환값이 음수라면 fork error를 출력한다. 반환값이 0인 경우 자식 프로세스인데 이 때 execvp함수를 호출하여 ./bin에 저장된 것 중 현재 명령어와 같은 명령을 실행한다. 이때 부모 프로세스의 모든것을 복사하지만 execvp에서 호출된 함수 내용을 overwrite하고 실행한다. 이후 exit을하여 자식 process를 종료한다. 반환값이 양수이면 부모 프로세스인데 정확히는 child의 process id를 받는다. 부모 프로세스의 역할은 자식 프로세스의 종료와 관련되므로 다음 문항에서 다룬다.

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

부모 프로세스는 waitpid함수를 실행하고 fork의 결과로 받은 pid를 인자로 준다. 이때 wait은 인자로 받은 pid의 process, 즉 child process가 종료되었다는 신호를 받을때 까지 멈춰있다가 해당 신호를를 받으면 child process를 reaping한다.

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

Pipe line 구현을 위해 우선 현재 입력받은 명령어에서 pipeline(‘|’)이 있는지를 확인한다. 왜냐하면 pipeline의 input으로 들어가는 명령어는 built\_in함수라도 fork해야하기 때문이며 또한 pipeline의 input과 outputd의 file descriptor를 바꿔야하기 때문이다. Pipeline이 있는 것으로 확인되었다면 가장 먼저 input, output에 대한 file descriptor를 바꿔줘야하는데 이를 위해서 pipe()함수를 사용한다. Pipe() 함수는 fork시 생성되는 child와 parent간 상호작용을 용이하게 해준다. Pipe를 사용하면 2개의 file descriptor를 할당하게 되는데 이를 저장하기 위해서 배열을 사용한다. 가령 fd[2]라는 배열을 사용하고 pipe(fd)를 사용하면 fd안에 parent, child 각각의 File descriptor에 접근할 수 있게된다. 파이프라인을 구현할 때, input과 output 모두에 대해 pipe를 실행한다. 그렇다면 pipe를 통해 file descriptor를 저장할 fd 배열은 size가 4가 되어야한다. 이랬을 때, fd[0]은 첫 번째 명령어에 대한 read를 fd[1]은 첫 번째 명령어의 write을 fd[2]는 두 번째 명령어의 read를 fd[3]은 두번째 명령어의 write을 의미한다. 이랬을 때 dup2(fd[1],1)로 첫 번째 명령어의 write을 stdout과 연결하고 dup(fd[2],0)으로 두 번째 명령어의 read를 stdin으로 연결한다 이러면 첫 번째 명령어의 출력을 두번 째 명령어의 입력으로 연결할 수 있다. 연결 후 input과 output에 해당하는 커맨드를 모두 처리해줘야하는데 우선 input의 명령어를 builtin 함수여부와 상관없이 일단 fork를 한다. 그리고 child process라면 builtin command인지 아닌지 확인하고 built\_in command라면 해당 함수를 수행, 아니라면 execvp함수에 인자로 주어 실행한다. 그리고 parent process는 모든 fd를 닫아주며 모든 Child를 reaping한다.

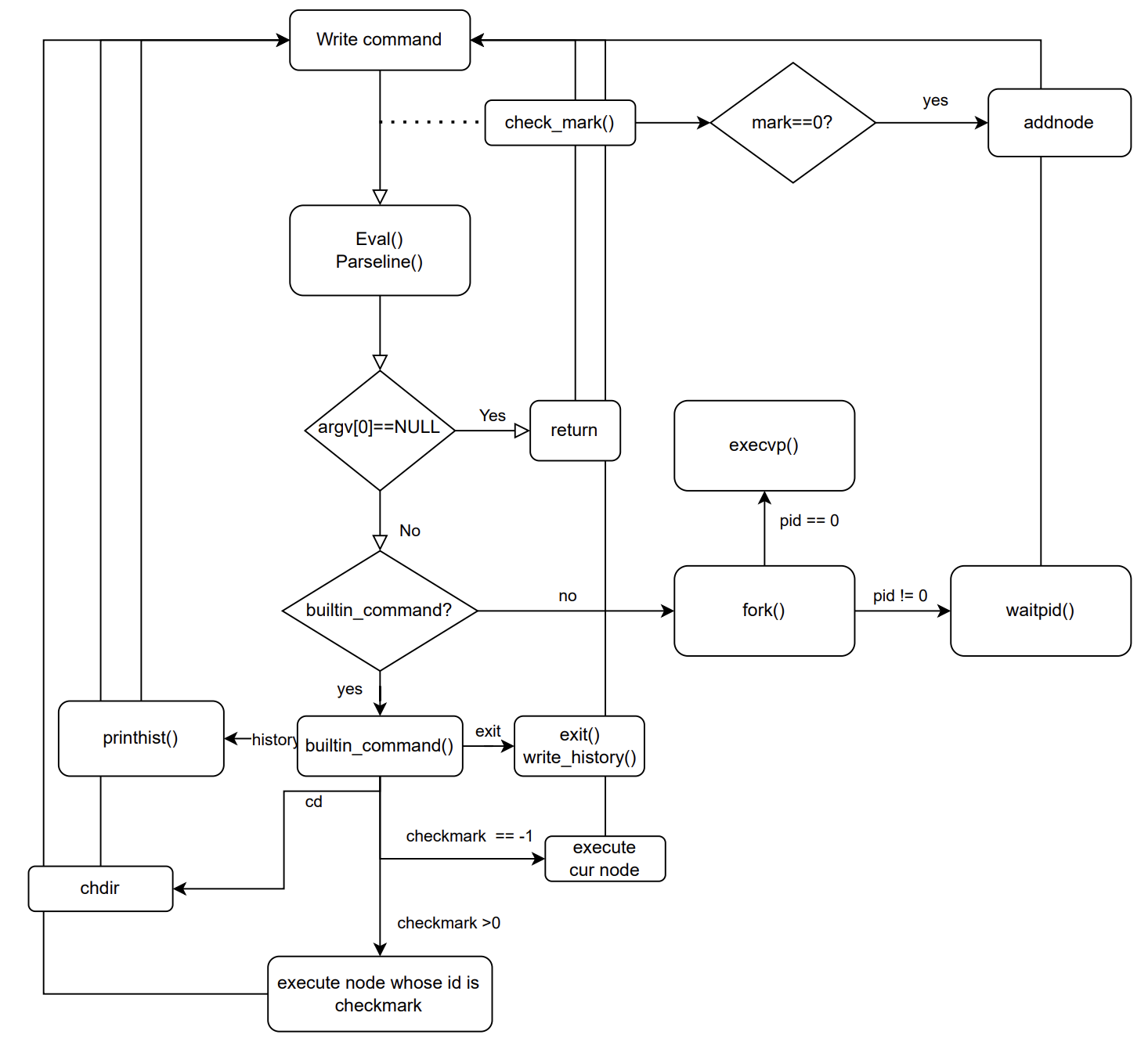
* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

일단 입력 커맨드를 파싱하여 pipeline의 숫자를 알아낸다. 그리고 파이프라인의 숫자 +1만큼 fork를하고 pipe를 수행한다. 만약 파이프라인의 숫자가 2라면 fd를 2\*(2+1)만큼 만들고 fd[0],fd[2],fd[4]의 위치에서 파이프를 수행한다. 그리고 pipe+1만큼 반복문을 돌고, 첫 번째 입력과 마지막 입력인 경우를 특별히 예외처리를 한다. 첫 번째 입력은 입력에 대해서 filedescriptor 중 stdin을 건드리지 않아야하고 마지막 입력은 file descriptor 중 stdout을 건드릴 필요가 없으므로 이들에 대해서 예외를 둔다. 나머지는 반복문을 통해 동일하게 구현한다.

* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명
  1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**
* phase1  
  명령어를 parseline 함수에 넣은 뒤 얻어진 argv를 builtin\_command함수에 집어넣는다. 이때 builtin\_command가 아닌 경우라면 fork를 수행한다. 그리고 child인 경우와 child가 아닌 경우로 나누는데 child가 아닌 경우라면 waitpid함수를 사용하여 인자에 child process의 pid를 주고 이를 reaping한다. Child process인 경우라면 이를 execvp함수를 활용하여 수행한다. Execvp함수는 인자로 준 함수가 있는지를 폴더 및 파일에서 찾아서 있다면 이를 실행하고 없다면 음수를 반환한다. 이를 통해 builtin command가 아닌 경우를 처리한다. 만약 builtin\_command라면 builtin\_command에서 처리해야하는데 이번 과제에서는 exit, history, cd ,!!, !#의 구현에 대해서 신경썼다. 우선 history의 경우 !! 및 !#를 제외한 입력에 대해서 기록을 가지고 있어야하는데 이를 bidirectional linked list를 통해 구현한다. node라는 구조체를 사용하며 node는 int id, char \*cmd, node\* next, prev를 변수로 가지고 있다. Id는 history에서 각 커맨드에 대해 부여하는 숫자를 의미하며 cmd는 입력된 커맨드, next와 prev은 앞뒤 node를 가리키는 pointer이다. 그리고 shell이 꺼지더라도 history를 저장하기 위해 history.txt파일에 모든 node의 cmd 내용을 입력해둔다. 그래서 shell을 처음 실행하면 main함수에서 history.txt를 열고 모든 cmd를 읽고 순서에 맞춰 id를 부여하여 node에 저장한다. 그리고 history builtin\_command를 입력하면 이 linked list를 처음부터 끝까지 출력한다. 모든 linedlist를 출력하는 것은 printhist함수에서 구현한다. 이 linkedlist에 노드를 추가하는 방식은 명령어가 입력받아지면 이를 eval 함수에 집어넣고 parseline을 수행하는데 parseline에 집어넣기 전에 마지막 node로 추가한다. 노드 추가는 addnode함수에서 구현한다. addnode는 현재 node위치에(마지막 부분)새로운 노드를 추가하는 함수로 원소를 추가하고 cur을 이 원소로 이동하고 pointer로 Link한다. !!와 !# 의 구현을 위해서 check\_mark라는 함수를 추가했다. 이는 커맨드에 !!가 오면 -1을 반환하고 !#가 오면 #값을 반환하고 그 외의 경우는 0을 반환하는 함수다. 이 결과에 따라 !!와 !#을 구분한다. !!인 경우는 현재 링크드리스트 상 가장 마지막 원소의 cmd를 eval함수에 넣어준다. 이러면 다시 해당 cmd를 수행하는 것과 같이 동작하며 history에도 넣어줄 수 있다. !#는 #번이 id로 나올때까지 뒤로 이동하고 #번이 나오면 이를 eval에 넣어준다. Cd는 chdir함수를 사용한다. Chdir 함수는 인자로 넣은 곳으로 디렉토리를 이동한다. 그리고 cd 뒤에 ~, NULL, $HOME이 나온 경우 HOME directory로 이동하게 한다. cd .. 는 상위 폴더로 이동하게 해준다. 그런데 cd로 다른 폴더에 이동한 후 exit하게되면 history를 그 directory에 저장하게 되므로 exit하고 history를 history.txt에 저장할 경우 최초에 시작한 디렉토리 주소로 Chidir한 후에 저장한다. 최초 시작 디렉토리 주소는 main이 실행되자마자 cwd에 저장한다. exit의 구현은 argv[0]이 "exit"인 경우 exit(0)을 수행하는데, 수행 하기 전 history의 내용을 history.txt파일에 모두 입력한다. 입력하는 함수는 writehistory이다. writehistory는 head부터 tail까지 linkedlist의 모든 원소를 인자로 들어온 file pointer에 fprintf한다.
* phase2  
  pipeline을 구현하기 위해서 parsepipe라는 함수를 추가한다. parsepipe함수의 주 목적은 pipiline('|')를 만났을 때 입력으로 받은 명령어(인자로 받은 buf에 저장 중)를 인자로 받은 pipe\_buf에 수정하여 저장하고 pipe의 개수인 pipe\_num을 반환하는 것이다. parsepipe는 pipeline이 0을 반환한다. Pipieline이 있다면 pipeline자리에 '\0'을 삽입하고 그 앞에 '\n'을 삽입한다. 가령 ls | grep history가 입력된 명령어(buf)라면 parsepipe의 return은 1이고 pipe\_buf에 ("ls\n\0grep history)가 저장된다. 그리고 pipe\_num에 따라서 pipe가 있는 경우와 없는 경우로 나눠서 eval을 수행한다. 그리고 pipe\_num을 알아냈다면 pipe의 입력들에 대한 file descriptor를 저장할 fd에게 2\*(pipe\_num+1)만큼 메모리를 할당하고, fd[0],fd[2],...fd[pipe\_num\*2]에 대해서 Pipe를 모두 수행해 file descriptor를 만들어준다. 위 경우에서는 ls\n에 대한 file descriptor가 fd[0](read),fd[1](write), grep history에 대한 file descriptor가 fd[2](read),fd[3](write)에 저장된다. pipe가 없는 경우는 기존과 같으므로 생략한다. pipe가 있다면 '\0'을 기준으로 cmd를 자르고 반복한다. 위의 경우에서 'ls\n'를 가지고 한번 수행하고 'grep history\n'을 기준으로 한번 수행하는 식이다. 여기서 수행한다는 것은 해당 커맨드를 처리하는 것을 말한다. 이를 구현하기 위해서 tmp\_pointer라는 char pointer를 pipe\_buf에 가리키게 한다. tmp\_pointer는 \0을 기준으로 나눠진 char pointer에 각각 접근하도록 한다. 이를 위해 tmp\_cmd와 tmp\_len 변수를 선언하는데 tmp\_cmd는 '\0'을 기준으로 잘려진 명령어를 저장한다. tmp\_len은 잘려진 명령어의 길이를 의미한다. 보통 strlen(tmp\_pointer)로 계산한다. 가령 위의 경우에서 strncpy(tmp\_cmd, tmp\_pointer, tmp\_len)를 하게 되면 tmp\_cmd에는 "ls\n"이 저장된다. 그리고 tmp\_pointer = tmp\_pointer + tmp\_len + 1로 '\0'다음으로 pointer를 옮긴다(tmp\_pointer는 g를 가리키게된다). 그리고 위의 과정을 반복하면 tmp\_cmd는 ''grep history\n"를 저장하게 될 것이다. 이런 과정으로 \0을 기준으로 나눠진 각 명령에 접근한다. 각 접근된 명령은 parseline에 넘겨주고 그것을 이후 수행에 활용한다. 이때 pipe의 input, output은 builtin Function과 무관하게 fork해야하므로 fork를 한다. child인 경우는 첫 번째 현재 명령어가 파이프 상 마지막인지 확인한다. 위의 경우에서 ls는 마지막이 아니다. 이 경우 파이프상 다음 child의 stdin을 현재 descriptor 1로 복사한다.(dup2(fd[tmp\_fd+1],1)) tmp\_fd는 fd 참조에 사용되는 index로 현재 pipe상 몇번째 입력인지를 확인하고 2씩 증가한다. ls라면 tmp\_fd는 0이된다. 즉 위의 코드를 분석하자면 1(stdout)을 fd[1]에 복사한다. 그래서 ls의 결과는 stdout으로 연결된다. 다음으로 현재 명령어가 파이프상 첫 번째인지 확인한다. ls는 첫번째 이므로 이 부분을 넘어간다. 이후 child의 명령어를 수행하기 전에 fd를 모두 닫아준다. fork되어있는 상태이기 때문에 현재의 fd는 모두 닫아주고 다음 명령어로 넘어가야하기 때문이다. 그래서 close해준 후 builtin\_command 혹은 execvp로 명령어를 수행한다. builtin\_command의 경우 명시적으로 exit을 수행한다. 이후 다음 명령어로 넘어가 grep history를 처리한다. grep history는 마지막 명령이므로 ls가 수행한 .(dup(fd[tmp\_fd+1],1))는 수행하지 않는다. 다만 이전 명령어의 stdin을 현재 명령어의 stdin으로 연결해줘야한다. dup2(fd[tmp\_fd-2],0) 현재 tmp\_fd는 2 이므로 dup(fd[0],0)이 되고 fd[0]의 read를 현재 stdin으로 연결해준다. 결과적으로 이전 명령어(ls)의 입력은 현재 명령어(grep history)의 입력이 된다. pipe수가 여러개라면 반복문을 돌며 위의 과정을 반복하는데 첫번째 명령어는 이전 명령어의 입력을 연결해줄 필요가 없고 마지막 명령어는 다음 명령어로 출력을 연결할 필요가 없으므로 두개에 대해서 예외를 준다. child process가 아닌 경우는 반복문이 모두 돌아간 후 fd를 모두 close하고 (while ((pid= wait(NULL))>0));를 써서 모든 child가 reaping되도록 한다.

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)  
   **

**2. Phase 2 (pipeline)  
도표이(가) 표시된 사진

자동 생성된 설명**

**3. Phase 3 (background)**