**System Programming Project 2**

담당 교수 : 박성용 교수님

이름 : 이민석

학번 : 20201614

1. **개발 목표**

이번 “MyShell” 프로젝트는 일반적인 shell, 현재 cspro에서 작동하는 bash의 기능을 간단하게 구현하는 프로젝트이다. C언어를 이용하여 시스템 프로그래밍의 지식들을 바탕으로 사용자 정의 리눅스 쉘을 모방하게 되는데, 우선 user command line을 실행하고 다른 프로그램을 실행하는 interactive한 터미널 프로그램인 shell을 이해해야 한다.

Phase 1의 경우, ‘fork()’와 ‘exec()’ 같은 system call을 사용하여 프로세스를 실행하고 명령을 실행하는 과정들을 직접 구현한다. /bin파일에 있는 내부 쉘 명령어(‘ls’, ‘rmdir’, ‘touch’, ‘cat’ .. etc) 들을 실행하게끔 하거나, ‘cd’나 ‘exit’과 같은 명령어들도 실행할 수 있도록 한다. 또한 ‘wait()’을 통해 자식 프로세스를 적절히 관리해야 한다.

Phase 2의 경우, 간단한 파이프라인 명령들을 처리할 수 있도록 이전 Phase 1의 쉘 기능을 확장하게 된다. 파이프라인을 통해 이전 명령의 출력을 다음 명령의 입력으로 전달할 수 있도록, 적절히 조절해야 한다.

Phase 3의 경우, shell의 백그라운드 명령 실행 기능 ‘command &’를 구현하여 user가 적절히 process를 background 혹은 foreground에서 실행될 수 있도록 한다. 이 단계에서 프로세스 신호 처리가 매우 중요하며, ‘SIGINT’, ‘SIGSTP’, ‘SIGCONT’와 같은 중요한 시그널을 적절히 처리하는 과정이 필요하다. ‘jobs’, ‘bg’, ‘fg’, ‘kill’ 과 같은 작업 제어 명령 또한 구현해야 한다.

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

우선 user의 input 명령들을 실행하기 위해 command line을 입력받아. argument들을 명령어와 옵션 등으로 적절히 나누는 parsing작업을 하게 된다. Parsing을 바탕으로 내부 쉘 명령어들은 ‘exec()’ system call을 통해 실행하게 되는데 ‘fork()’를 통해 자식 프로세스를 생성하여 실행한다. ‘exit’, ‘cd’와 같은 명령들은 부모 프로세스에서 직접 처리할 수 있도록 한다. 또한 shell과 같이 명령을 실행했다면, 반복적으로 command line을 입력 받을 수 있도록 ‘MyShell’ 이 작동하게 된다.

1. Phase 2

파이프라인을 처리하기 위해 Phase 1의 parsing에 (‘ | ‘)을 기준으로 분리하는 parsing 작업을 추가한다. 결과로 분리된 각 명령어는 파이프라인 개수에 맞게 프로세스를 실행하도록 하며, 한 프로세스의 출력을 다른 프로세스의 입력으로 전달하게 된다. 이 때 file descriptor를 초기화 하고 적절히 관리하여 각 프로세스의 입력 또는 출력이 다른 프로세스와 연결되도록 한다.

1. Phase 3

Phase 1, Phase 2에서 구현한 내용들을 바탕으로 프로세스를 실행할 때, 이를 background / foreground에서 실행하는지 확인한 후, 부모 프로세스에서 적절히 자식 프로세스를 control 하여 구현하게 된다. 또한 ‘jobs’, ‘bg’, ‘fg’, ‘kill’ 과 같은 쉘 내장 명령을 통해 현재 실행 중인 프로세스들의 상태를 관리할 수 있다. ‘SIGINT’, ‘SIGSTP’, ‘SIGCONT’와 같은 signal들을 처리할 때, 각각의 signal handler 함수들을 구현하였으므로 “MyShell”에서 직접 프로세스를 관리할 수 있게 된다.

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

‘fork()’ 함수는 새로운 자식 프로세스를 생성하는 데 사용된다. 이 때 현재 실행중인 프로세스(부모 프로세스)의 복사본을 생성하고, “Copy-On-Write”를 거치게 된다. fork()의 호출은 부모 프로세스에게 자식 프로세스의 PID, 자식프로세스에게는 0을 반환하고, 만약 실패할 경우 -1을 반환하게 된다. 따라서 if문의 condition에 fork()의 반환 값이 0인 경우(자식 프로세스)와 아닌 경우를 나누어 자식 프로세스는 명령어를 적절히 실행할 수 있도록 한다.

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

Linux는 Signal을 사용하여 프로세스 간 통신을 하게 되는데, 이 Signal 종류 중 하나인 ‘SIGCHLD’는 자식 프로세스가 종료되거나 status가 변경 시 부모 프로세스에게 알리는데 사용된다. 만약 자식 프로세스가 종료되었음을 감지한다면, 부모 프로세스는 waitpid(pid,&status,NULL)을 호출하여 자식 프로세스를 reaping하게 된다.

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

사용자의 입력에서 파이프(‘|’) 문자를 기준으로 명령을 분리하는 parsing 작업을 거쳐야 한다. 이는 parsepipe(char \*cmdline)에서 하게 된다. Parsing한 command들을 바탕으로 pipe\_eval함수에서 실제 파이프라인을 구성하고 명령어들을 실행하게 된다. 이 때 ‘pipe()’ 를 사용하여 각 명령 사이의 통신을 위한 파이프를 생성한 후, ‘dup2()’로 각 프로세스의 입출력을 원하는 file descriptor 또는 stdin(0), stdout(1)로 연결하게 된다. 또한 사용하지 않는 file descriptor는 모두 ‘close()’함수로 닫아주었다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

Command line의 초기 parsing 작업을 통해 파이프 개수를 파악하게 된다. 이를 ‘pipe\_eval()’의 인자로 넘겨주어, Pipe\_eval()에서 파이프 개수만큼 각 파이프에 대한 file descriptor를 할당하게 된다.

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

Background process는 eval()함수 또는 pipe\_eval()함수와 builtin\_command()함수에서 처리된다.

만약 user가 command line 뒤에 ‘&’를 붙여 background process 실행을 입력한다면, eval() 함수 또는 pipe\_eval()함수 내에서 부모 프로세스는 즉시 프롬프트로 돌아가 user에게 다음 명령을 받을 준비를 한다. 이는 parseline()함수에서 background 실행 유무를 확인하여 변수 ‘bg’에 값을 넣어 알 수 있다. 만약 ‘bg’ 가 0이라면(foreground 실행) 자식 프로세스의 종류를 기다리며, ‘bg’ 가 1이라면(background 실행) 해당 command는 background에서 실행된다.

Builitin\_command()함수는 fg, bg, kill 등 background process와 관련된 Linux 명령들을 실행하게끔 한다. ‘fg %(job\_id)’를 통해 background process를 foreground에서 실행하도록 하거나, ‘bg %(job\_id)’를 통해 ‘STOPPED’된 background process를 다시 ‘RUNNING’상태로 변경한다.

또한 직접 구현한 ‘SIGINT’, ‘SIGTSTP’, ‘SIGCHLD’의 handler 함수인 ‘sigint\_handler()’, ‘sigtstp\_handler()’, 그리고 ‘sigchld\_handler()’ 를 통해 시그널을 적절히 처리하여 시스템의 상태를 업데이트 한다.

* 1. **개발 방법**

1. Phase 1

‘builtin\_command()’함수는 ‘strcmp()’함수를 사용하여 어떤 명령어가 입력되었는지 확인한 후, 실행할 수 있도록 한다. 종료 명령인 ‘exit’ 또는 ‘quit’은 exit(0)를 통해 프로그램을 종료하고, 디렉토리 변경 명령인 ‘cd’는 대상 디렉토리 경로를 얻은 후, ‘chdir()’함수를 사용하여 디렉토리를 변경한다. ‘/bin/’ 파일에 있는 함수들은 ‘eval()’함수에서 ‘execve()’함수를 통해 작동할 수 있도록 문자열을 조작한다.

‘eval()’함수는 입력된 command line을 해석하고 실행한다. 이 때 ‘fork()’함수를 사용하여 부모 프로세스 및 자식 프로세스의 동작을 구분하고 관리한다. ‘if ~ else ~’ 구문을 이용하여 pid = 0 인 자식 프로세스는 ‘execve()’함수를 통해 명령어를 실행하게 된다. pid != 0인 부모 프로세스는 ‘waitpid()’함수를 통해 자식 프로세스의 종료를 동기적으로 기다린다. 이를 통해 user가 다음 명령을 입력하기 전에 현재 명령의 실행 완료를 보장하게 된다. 또한 background 에서 실행되는지, 아닌지 확인하는 기능도 하게 된다.

‘del\_char()’함수는 입력된 command line의 큰따옴표(“), 작은따옴표(‘) 들을 공백으로 처리하는 기능을 하며, 이는 parsing 작업에 도움을 주게 된다.

1. Phase 2

‘parsepipe()’함수는 주어진 command line에서 파이프 (‘|’)를 기준으로 각각의 명령어를 분리하고, 분리된 명령어들을 ‘commands[MAXARGS]’에 저장한다. 이 때 ‘strtok()’함수를 사용하여 (‘|’) 문자를 기준으로 토큰을 분리하며, 파이프의 개수를 계산하게 된다.

‘Pipe\_eval()’함수는 ‘parsepipe()’함수에서 얻은 ‘commands[MAXARGS]’를 실행하도록 한다. 이 때 각각의 command를 argument들로 나누기 위해 동적으로 메모리를 할당한다. 파이프 개수만큼 file descriptor 배열인 ‘fds[]’를 생성하고 초기화 한다. 각 파이프는 두 개의 file descriptor를 가질 수 있도록 ‘pipe()’함수를 이용하였다. 파이프 연결은 ‘dup2()’함수를 통해 해결하는데, 이는 이전 명령의 출력이 다음 명령의 입력으로 전달될 수 있도록 하며, 적절히 file descriptor를 연결해야 한다. 사용하지 않는 file descriptor들은 ‘close()’함수를 통해 닫아주어 입출력이 올바르지 않은 곳에 되지 않도록 관리하였다. 나머지 명령의 실행은 ‘eval()’함수와 동일하다.

1. Phase 3

‘main()’함수에서 ‘Signal()’함수를 통해 ‘SIGINT’, ‘SIGTSTP’, ‘SIGCHLD’를 새로 구현한 sig\_handler가 작동하도록 한다. 또한 ‘ctrl + c’, ‘ctrl + z’ 의 키가 프로그램 실행 중과 아닐 때의 프롬프트 창이 다르게 떠야하므로 이에 관련된 ‘command\_flag’변수를 volatile int 로 선언하였다.

User의 command line을 읽어들여 이를 실행하는 ‘Pipe\_eval()’함수와 ‘eval()’함수에선 때에 적절하게 Sigprocmask를 통하여 특정 Signal을 blocking하는 작업이 필요하다. 이 때, 부모 프로세스와 자식 프로세스의 blocking해야하는 signal이 다르기에 서로 다른 두 개의 sigset\_t 변수를 설정하였다. 또한 parsing 작업 중에 background 실행 유무를 파악하여, 만약 foreground process라면, 모든 Signal을 우선 blocking한 후, 전역변수들의 작업을 처리한다. 이 때, while()문과 Sigsuspend를 통해 race condition을 방지한다. Background 작업일 경우 마찬가지로 모든 Signal을 blocking한 후, 전역변수 작업 후에, wait을 하지 않으며, 프롬프트 창을 띄우도록 한다. ‘fg’, ‘bg’에 맞는 작업을 하였다면, 다시 이전의 ‘prev\_mask’로 돌아가도록 bit setting을 한다.

모든 프로세스의 상태를 저장해야 하기에 필요한 변수들을 모아 ‘jobs’ 구조체를 선언하였다. ‘pid’는 해당 job의 pid, ‘id’는 해당 job의 index, ‘cmd'는 해당 프로세스의 command line, ‘state’는 해당 프로세스의 상태(RUNNING\_FG, RUNNING\_BG, SUSPENDED)를 나타낸다.

‘builtin\_command()’함수에서 ‘jobs’, ‘fg’, ‘bg’, ‘kill’과 같은 명령어를 실행할 수 있도록 구현하였다. ‘jobs’는 index가 0부터 시작한다. background에서 실행되고 있는 프로세스들의 목록을 확인할 수 있다. ‘print\_jobs()’를 통해 모든 프로세스의 리스트인 jobs\_list[]를 출력한다. ‘bg’의 경우 background에서 ‘SUSPEND’된 함수를 다시 ‘RUNNING’상태로 변경될 수 있도록 ‘SIGCONT’ Signal을 주게 된다. ‘fg’의 경우, background에서 멈춤 상태이거나 실행중인 프로세스를 foreground에서 실행되도록 하는 명령이다. ‘kill’의 경우, jobs의 list index를 통해 해당 프로세스를 KILL하는 명령이다.

‘sigint\_handler()’는 ‘ctrl + c’가 입력될 경우 실행되는 sig\_handler함수이다. 이는 foreground에서 돌아가는 프로세스를 SIGKILL signal을 주어 종료시킨다. 후에 jobs\_list에서 이를 clear한다. ‘sigtstp\_handler()’는 ‘ctrl + z’가 입력될 경우 실행되는 sig\_handler함수이다. 이는 foreground에서 돌아가는 프로세스를 SIGSTOP signal을 주어 멈춤시킨다. 해당 프로세스의 state를 ‘SUSPENDED’로 변경한다. ‘sigchld\_handler()’는 자식 프로세스에서 시그널이 발생할 경우 실행되는 sig\_handler함수이다. Waitpid를 통해 자식프로세스를 정상적으로 reap하게 되며, 만약 정상적으로 종료하거나 해당 작업이 파이프라인의 프로세스 일부일 경우, 이를 체크하여 jobs\_list에서 clear하게 된다. 이 3개의 handler를 다룰 때 Signal을 때에 맞게 BLOCK, SET\_MASK해야 한다.

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

   자동 생성된 설명**Phase 1 (fork)**
3. **텍스트, 도표, 평면도, 라인이(가) 표시된 사진

   자동 생성된 설명Phase 2 (pipeline)**

**텍스트, 도표, 기술 도면, 평면도이(가) 표시된 사진

자동 생성된 설명**

1. **텍스트, 도표, 스크린샷, 평면도이(가) 표시된 사진

   자동 생성된 설명Phase 3 (background)**