**2 Project Phase I: Building and Testing Your Shell (Points: 30)**

이 단계에서 첫 번째 작업은 간단한 셸을 작성하는 것이며 프로세스를 시작하는 것이 리눅스 셸의 주요 기능입니다.  
셸을 작성한다는 것은 프로세스에서 어떤 일이 일어나고 있으며 프로세스가 어떻게 시작되는지 정확히 알아야 한다는 것을 의미합니다.  
셸은 다음과 같은 기본 내부 셸 명령을 실행할 수 있어야 합니다,  
• cd: 셸의 디렉토리를 탐색합니다  
• ls: 디렉터리 내용 나열  
• mkdir, rmdir: 셸을 사용하여 디렉터리를 만들고 제거합니다  
• touch, cat, echo: 파일의 내용 만들기, 읽기 및 인쇄  
• history: 셸이 시작된 이후 실행된 셸 명령을 추적합니다(아래 사양 참조)  
• exit: 모든 하위 프로세스를 종료하고 셸을 종료합니다  
그림 1과 같이 명령은 cd와 history를 제외한 부모 프로세스에서 forking을 통해 생성된 자식 프로세스에 의해 실행되어야 합니다.

힌트: 셸은 주로 fork( ) 및 exec( ) 시스템 호출에 의존합니다. 이 두 가지 시스템 호출은 실제로 대부분의 프로그램이 Linux에서 실행되는 방식의 구성 요소입니다. 첫째, 기존 프로세스는 두 개의 개별 프로세스로 분할됩니다. 그런 다음 자녀는 exec()을 사용하여 자신을 새 프로그램으로 바꿉니다.

셸은 내부에 세 가지 기능이 포함된 지속적으로 루프를 실행합니다;

do{

// 셸 프롬프트: 프롬프트 인쇄

print “CSE4100-MP-P1>”

// Reading(읽기): 표준 입력에서 명령을 읽습니다.

input = myshell readinput ();

// 구문 분석: 입력 문자열을 명령줄 인수로 변환합니다.

args = myshell parseinput (input);

// 실행 중: 자식 프로세스를 포킹하여 명령을 실행하고 부모 프로세스로 돌아갑니다.

myshell execute(args);

} while (true);

참고: fork(), exec(), wait() 및 기타 관련 시스템 호출은 man 페이지를 참조하십시오.

history: history 명령은 기본 제공 명령으로 실행됩니다. history 명령은 셸이 실행된 이후 실행된 명령을 추적해야 합니다. 기본 셸의 history 명령은 많은 기능을 제공하지만 이 프로젝트에서는 두 가지 기능만 구현하면 됩니다.  
• ! : 최근에 실행된 명령어를 출력합니다. 그런 다음 명령을 실행합니다. (!! 명령은 기록 로그를 업데이트하지 않습니다.)  
• !# : # 라인에 명령어를 출력합니다. 그런 다음 명령을 실행합니다. (예: !12)  
다음은 history 명령의 예입니다.

기록 로그는 셸이 종료된 후에도 유지 관리되어야 합니다. 즉, 셸을 다시 실행하고 history 명령을 실행하면 이전과 동일한 history 로그가 출력되어야 합니다. 또한 셸 기록 로그는 무한히 증가할 수 있습니다.  
평가: 셸은 위의 작업 사양에 설명된 모든 기능을 수행해야 합니다.

**3 Project Phase II: Redirection and Pipe (Points: 30)**

프로젝트 두 번째 단계에서는, 이전에 프로그래밍한 간단한 쉘 예제의 기능성을 확장할 것입니다. 이를 위해서는 각 파이프라인 명령에 대해 새 프로세스를 생성하고 부모 프로세스가 마지막 명령을 기다리도록 해야합니다. 이를 통해 "ls -al | grep filename"과 같은 간단한 명령을 실행할 수 있습니다.

핵심 아이디어는 한 프로세스의 출력을 다른 프로세스의 입력으로 전달하는 것입니다. 여러 파이프 라인 체인을 명령 줄 인수로 가질 수 있다는 것을 유의하십시오. 파이프는 Linux에서 두 개 이상의 명령을 사용하여 한 명령의 출력이 다음 명령의 입력으로 작용하도록 하는 명령입니다. 즉, 각 프로세스의 출력이 다음 프로세스의 입력으로 작용하는 것입니다. 명령 줄 인수에 여러 파이프가 있는 경우 재귀 함수를 사용하여 구문 분석할 수 있습니다.

참고: dup() 및 dup2() 시스템 콜에 대한 매뉴얼 페이지를 참조하십시오.

참고: 이 단계에서는 리디렉션을 구현할 필요가 없습니다. 그러나 리디렉션은 파이프와 유사한 방식으로 작동합니다. 차이점은 리디렉션이 파일 관련 명령을 처리한다는 것입니다.

다음과 같은 파이프 명령을 실행할 수 있습니다.

• ls | grep filename

• cat filename | less

• cat filename | grep -i "abc" | sort -r

**4 Project Phase III: Run Processes in Background (Points: 40)**

이것은 MyShell 프로젝트의 마지막 단계로, 셸이 백그라운드에서 프로세스를 실행할 수 있도록하는 단계입니다. Linux 셸은 작업 제어라는 기능을 지원하여 사용자가 작업의 프로세스 상태 (실행 중지 또는 종료)를 변경하고 백그라운드와 포그라운드 작업 간에 작업을 이동할 수 있습니다.셸은 명령 줄 인수에 ‘&’가있는 경우 명령을 백그라운드에서 시작해야합니다. 또한 셸은 작업 제어를 지원하는 다양한 내장 명령도 제공해야합니다.

파이프를 사용한 다음 셸 명령을 평가 할 수 있습니다.

• jobs : 백그라운드 작업 목록을 표시합니다.

• bg ⟨job⟩: 중지된 백그라운드 작업을 실행중인 백그라운드 작업으로 변경합니다.

• fg ⟨job⟩: 중지된 또는 실행중인 백그라운드 작업을 포그라운드에서 실행중인 작업으로 변경합니다.

• kill ⟨job⟩: 작업을 종료합니다.

Note : 명령과 ‘&’를 공백으로 구분하도록 요구하지 않아야합니다. 예를 들어 ‘sort foo.txt &’ 및 ‘sort foo.txt&’ 및 ‘sort foo.txt &’ (앰퍼샌드 뒤에 공백)는 모두 유효합니다.

힌트 : ctrl-c를 누르면 SIGINT 신호가 포그라운드 작업의 각 프로세스에 전달됩니다. SIGINT의 기본 동작은 (foreground) 프로세스를 종료하는 것입니다. 마찬가지로 ctrl-z를 누르면 SIGTSTP 신호가 포그라운드 작업의 각 프로세스에 전달됩니다. SIGTSTP의 기본 동작은 (foreground) 프로세스를 중지 상태로 놓는 것이며, SIGCONT 신호를 수신하여 깨어날 때까지 중지 상태로 유지됩니다.

참고 :이 프로젝트 단계를 완료하려면 8 장에서 시그널 핸들러 등록을 참조하십시오.SIGINT, SIGSTP 및 SIGCONT는 필수 항목입니다.

이 단계에 대해 다음 셸 명령을 평가 할 수 있습니다.

• 프로젝트 제2 단계에서 모든 명령은 명령 줄 끝에 ‘&’를 붙여서 제공 할 수 있습니다.

5 제출 지침  
Hand-In Specifications: 제출에는 파일, 메이크 파일(모두 소문자) 및 리드미 파일을 포함한 소스 코드 파일만 포함되어야 합니다. 실행 프로그램을 포함해서는 안 됩니다. 당신의 프로젝트의 등급을 매기는 일을 담당하는 TA는 제공된 소스 코드로부터 당신의 셸 프로그램을 자동으로 재구축할 것입니다. 아래의 제출 지침을 엄격하게 따르십시오. 그렇지 않을 경우 전체 프로젝트 점수에서 페널티를 받을 수 있습니다.  
1. 아카이브 파일 하나를 제출해야 하며, 제출할 아카이브 파일(.tar.gz)의 이름은 prj1 학생 ID.tar.gz로 지정해야 합니다.  
2. 파일을 보관 해제할 경우 tar 파일의 루트 디렉터리 이름이 학생 ID여야 합니다.  
3. "phase 1/2/3"이라는 이름으로 각 단계에 대한 디렉토리를 생성해야 합니다  
4. 각 단계의 각 디렉터리에는 Makefile, README.md 및 소스 코드가 포함되어야 합니다. 당신은 또한 3단계에 걸쳐 1개의 문서를 제출해야 합니다.  
첨부 파일:  
• 1단계 폴더(소스 코드(myshell.c, myshell.h), 파일 만들기, 읽기)(30점)  
• 2단계 폴더(소스 코드(myshell.c, myshell.h), 파일 만들기, 읽기)(30점)  
• 3단계 폴더(소스 코드(myshell.c, myshell.h), 파일 만들기, 읽기)(30점)  
• 문서망(단계 1, 2, 3)

<unix 단축키>

