설계과제 2 개요 : SSU-Repo

○ 개요
- 리눅스 시스템 상에서 사용자가 상태 관리를 원하는 디렉토리에 대해 경로를 추가 및 삭제하며 파일의 변경 사항을 추적하고, 추적된 파일들에 대해 백업하고 백업된 파일을 다시 복원하며 버전을 관리하는 프로그램

○ 목표
- 새로운 명령어를 시스템 함수를 사용하여 구현함으로써 쉘의 원리를 이해하고, 유닉스/리눅스 시스템에서 제공하는 여러 시스템 자료구조와 시스템 콜 및 라이브러리를 이용하여 프로세스를 생성하여 실행하는 프로그램을 작성함으로써 표준 입출력 및 파일 입출력에 대해 적응하고 이를 응용하여 디렉토리 구조를 링크드 리스트로 구현하며 시스템 프로그래밍 설계 및 응용 능력을 향상시킴

○ 팀 구성
- 개인별 프로젝트

○ 보고서 제출 방법
- 설계과제는 "#P설계과제번호_학번.zip" (예. #P1_20140000_V1.zip)형태로 압축하여 classroom.google.com에 제출해야 함.
- "#P설계과제번호_학번.zip" 내 보고서인 "#P설계과제번호.hwp" 와 헤더파일 및 소스코드 등 해당 디렉토리에서 컴파일과 실행이 가능하도록 모든 파일(makefile, obj, *.c, *.h 등 컴파일하고 실행하기 위한 파일들)을 포함시켜야 함. 단, 특정한 디렉토리에서 실행해야 할 경우는 예외.
- 구현보고서인 "#P설계과제번호.hwp"에는 1. 과제개요(명세에서 주어진 개요를 그대로 쓰면 안됨. 자기가 구현한 내용 요약) 2. 기능(구현한 기능 요약), 3. 상세설계(함수 및 모듈 구성, 순서도, 구현한 함수 프로토타입 등), 4. 실행결과(구현한 모든 기능 및 실행 결과 캡쳐)를 반드시 포함시켜야 함.
- 제출한 압축 파일을 풀었을 때 해당 디렉토리에서 컴파일 및 실행이 되어야 함(특정한 디렉토리에서 실행해야 하는 경우는 제외). 해당 디렉토리에서 컴파일이나 실행이 되지 않을 경우, 혹은 기본과제 및 설계과제 제출 방법(파일명, 디렉토리명, 컴파일 시에 포함되어야 할 파일 등)을 따르지 않으면 무조건 해당 과제 배점의 50% 감점
- 설계과제를 기한 내 새로 제출할 경우 기존 것은 삭제하지 않고 #P설계과제번호_학번_V1.zip 형태로 버전 번호를 붙이면 됨. 버전 이름은 대문자 V와 함께 integer를 1부터 incremental 증가시키면서 부여 (예. #P3_20140000_V1.zip, #P3_20140000_V2.zip) 하면 됨. 단, 처음 제출 시는 버전 번호를 붙이지 않아도 되며 두 번째부터 V1를 붙여 제출하면 됨. 기한 내에 여러 버전의 보고서를 제출할 수 있으나, 채점은 최종 버전만을 대상으로 함.

✓ 설계과제명세서와 강의계획서 상 배점 기준이 다를 경우 해당 설계과제명세서의 배점 기준이 우선 적용

✓ 보고서 #P설계과제번호.hwp (15점) : 개요 1점, 기능 1점, 상세설계 10점, 실행 결과 3점

✓ 소스코드 (85점) : 소스코드 주석 5점, 실행 여부 80점 (설계 요구에 따르지 않고 설계된 경우 소스코드 주석 및 실행 여부는 0점 부여. 설계 요구에 따라 설계된 경우 기능 미구현 부분을 설계명세서의 100점 기준에서 해당 기능 감점 후 이를 80점으로 환산)

✓ 각 설계과제의 완성 유무는 제출 여부로 판단하는 것이 아니라 주어진 과제에서 명시된 "필수기능요건" 의 구현으로 판단. (예. 특정 과제의 필수 기능 중 일부 기능만 구현했을 경우 해당 점수는 부여하나 과제는 미구현으로 판단하고 본 교과목 이수조건인 설계 과제 최소 구현 개수 2개에 포함시키지 않음)

- 기타 내용은 강의계획서 참고

○ 제출 기한
- 4월 28일(일) 오후 11시 59분 59초

○ 보고서 및 소스코드 배점
- 보고서는 다음과 같은 양식으로 작성(강의계획서 FAQ 참고)

1. 과제 개요 (1점) // 명세에 주어진 개요를 더 상세하게 작성
2. 구현 기능 (1점) // 함수 프로토타입 반드시 포함
3. 상세 설계 (10점) // 함수 기능별 흐름도(순서도) 반드시 포함
4. 실행결과 (3점) // 테스트 프로그램의 실행결과 캡쳐 및 분석

- 소스코드 및 실행 여부 (85점) // 주석 (5점), 실행 여부 (80점)

○ ssu_repo 프로그램 기본 사항
- 리눅스 시스템에서 생성한 자신의 사용자 아이디에 대해 사용자 홈 디렉토리를 확인 (예. 자신의 리눅스 시스템상 계정 아이디가 oslab일 경우 $HOME은 /home/oslab임. (%echo $HOME, 또는 %cat /etc/passwd 파일에서 해당 아이디의 홈 디렉토리 확인)

- ssu_repo 프로그램 실행 시 현재작업경로(PWD)에 레포 디렉토리(.repo)를 생성하고 하위에 커밋 로그(.commit.log)와 스테이징 구역(.staging.log)를 생성
- ssu_repo의 내장명령어 중 모든 명령어는 프로세스를 생성(fork())하고 이를 실행(exec()류 함수)시켜줌 (exit 명령어는 exec()류 함수 사용하지 않아도 됨)
- add, remove 내장명령어를 통해 스테이징 구역에 기록한 파일 및 디렉토리는 위에서부터 한줄씩 읽으며 포함 및 제거할 경로를 기록함
- commit 내장명령어 사용 시 인자로 받은 이름으로 버전 디렉토리를 생성하여 스테이징 구역에 포함되는 파일을 해당 버전 디렉토리에 백업함. 만약 해당 파일의 백업 파일이 존재하고 마지막 백업 파일과 내용이 동일할 경우 기록 및 백업하지 않음. 버전 디렉토리 및 하위 디렉토리의 경우 백업 파일이 없을 경우 디렉토리를 생성하지 않음
- revert 내장명령어 사용 시 인자로 받은 이름의 버전 디렉토리로 상태를 복원함. 버전 디렉토리를 복원할 때는 커밋 로그를 아래에서부터 읽으며 링크드 리스트를 이용하여 해당 버전 이후 변경내역에 대해 되돌아가며 복원을 시도함
- 하나의 프로세스(단일 쓰레드)가 한 개 파일 또는 한 개 디렉토리(하부 디렉토리 및 파일 모두)를 백업해야 함
- ssu_repo 프로그램 실행 시 레포 디렉토리에 백업된 파일 리스트와 스테이징 구역에 포함되어 변경 사항을 추적하는 경로들에 대해 링크드 리스트로 관리하며 내장명령어 실행 시 삽입, 제거 등 올바른 작동이 가능하게 해야 함
- 프로그램 실행 시 내장명령어(add, remove, status, commit, revert, log, help, exit)에 따라 해당 기능 실행
- ssu_repo 프로그램을 통해 입력 가능한 경로들은 작업경로 내 경로(작업경로 포함)여야 함
  ✓ 리눅스 상에서 파일 경로의 최대 크기는 4,096 바이트이며, 파일 이름의 최대 크기는 255 바이트임
- 백업 파일과 원본 파일에 대한 비교는 stat 구조체를 통해 사이즈를 비교 후 md5를 이용하여 해시값을 통해 비교
- 프로그램 전체에서 system() 절대 사용 불가. 사용 시 0점 처리
- getopt() 라이브러리를 사용하여 옵션 처리 권장
- ssu_repo은 foreground로만 수행되는 것으로 가정함(& background 수행은 안되는 것으로 함)

○ 설계 및 구현

1. ssu_repo

1) Usage : ssu_repo
- ssu_repo 실행 시 사용자 입력을 기다리는 프롬프트 출력

2) 인자 설명
- 프로그램 실행시 인자는 따로 받지 않음

3) 실행결과
- ssu_repo 프로그램 실행 시 레포 디렉토리에 백업된 파일들의 경로와 스테이징 구역에 포함되어 변경 사항을 추적하는 경로들에 대해 링크드 리스트를 관리
- ssu_repo의 실행결과는 "학번〉"이 표준 출력되고 내장명령어(add, remove, status, commit, revert, log, help, exit) 입력 대기. 학번이 20220000일 경우 "20220000〉" 형태로 출력
- 현재 작업경로에서 ssu_repo 프로그램을 처음 실행 시 현재작업경로에 레포 디렉토리(.repo)를 생성하고 하위에 커밋 로그(.commit.log)와 스테이징 구역(.staging.log)를 생성. 이미 존재힐 경우에는 생성하지 않음
- 내장명령어 실행 시 경로 하위에 레포 디렉토리가 포함된다면 무시하고 프로그램 실행. 만약 레포 디렉토리 포함, 레포 디렉토리 하위 경로를 입력할 경우 에러처리 후 프롬프트 재출력

예제 1. ssu_repo 실행결과

```
% ./ssu_repo
20220000〉
20220000〉 asd
Usage:
  〉 add 〈PATH〉 :  record path to staging area, path will tracking modification
  〉 remove 〈PATH〉 : record path to staging area, path will not tracking modification
  〉 status : show staging area status
  〉 commit 〈NAME〉 : backup staging area with commit name
  〉 revert 〈NAME〉 : recover commit version with commit name
  〉 log : show commit log
  〉 help : show commands for program
  〉 exit : exit program

% ./ssu_repo
20220000〉 exit

%
```

4) 예외처리(미 구현 시 아래 점수만큼 감점, 필수 기능 구현 여부 판단과는 상관 없음)
- 프롬프트 상에서 엔터만 입력 시 프롬프트 재출력(2점)

- 프롬프트 상에서 지정한 내장명령어 외 기타 명령어 입력 시 help 명령어의 결과를 출력 후 프롬프트 재출력(2점)

2. 내장명령어 1. add

1) Usage : add 〈PATH〉
- 스테이징 구역에 추가하여 변경내역을 추적할 경로(PATH)를 입력받아 해당 파일 또는 디렉토리 경로를 스테이징 구역에 추가

2) 인자 설명
- 첫 번째 인자 〈PATH〉는 스테이징 구역에 추가할 파일이나 디렉토리의 상대경로와 절대경로 모두 입력 가능해야 함

3) 실행결과
- 첫 번째 인자 〈PATH〉를 스테이징 구역(.staging.log)에 기록함. 스테이징 구역에 기록된 경로는 commit 내장명령어 실행 시 변경내용을 추적하여 백업을 진행할 수 있는 경로임
- 만약 첫 번째 인자 〈PATH〉가 스테이징 구역에 기록되어 변경내용을 추적하는 경로에 포함될 경우 ("현재작업경로에 대한 원본 상대경로" already added in staging area)의 형태로 출력
- 실행 성공 시 (add "현재작업경로에 대한 원본 상대경로")를 표준 출력하고, 스테이징 구역에 (add "원본 절대경로")의 형태로 기록함

예제 2-1. add 내장명령어 실행(현재 작업 디렉토리(pwd)가 "/home/oslab/lsp"일 경우)

```
% ./ssu_repo
20220000〉 add
ERROR: 〈PATH〉 is not include
Usage: add 〈PATH〉 :  record path to staging area, path will tracking modification

20220000〉 add asd.txt
ERROR: 'asd.txt' is wrong path

20220000〉 add a.txt
add "./a.txt"

20220000〉 add a.txt
"./a.txt" already exist in staging area

20220000〉 add ./a
add "./a"

20220000〉 add /home/oslab/lsp/a/b
"./a/b" already exist in staging area

20220000〉 add ./
add "."

20220000〉 remove a
remove "./a"

20220000〉 add /home/oslab/lsp/a/a.txt
add "./a/a.txt"

20220000〉 add ./a
add "./a"
```

그림 2-2. /home/oslab/lsp 디렉토리 트리 구조

```
/home/oslab/lsp
├ .repo/
|  ├ .commit.log
|  └ .staging.log
├ a/
|  ├ a.txt
|  ├ b/
|  |  └ c.txt
|  └ c.txt
├ a.txt
└ b/
```

예제 2-3. 스테이징 구역(.staging.log) 파일 내용

```
add "/home/oslab/lsp/a.txt"
add "/home/oslab/lsp/a"
add "/home/oslab/lsp"
remove "/home/oslab/lsp/a"
add "/home/oslab/lsp/a/a.txt"
add "/home/oslab/lsp/a"
```

4) 예외 처리(미 구현 시 아래 점수만큼 감점, 필수 기능 구현 여부 판단과는 상관 없음)
- 경로를 입력하지 않을 경우, add 명령어에 대한 에러 처리 및 Usage 출력 후 프롬프트 재출력(2점)
- 인자로 입력받은 경로가 길이 제한(4,096 Byte)를 넘거나 경로가 올바르지 않은 경우, 에러 처리 후 프롬프트 재출력(2점)
- 인자로 입력받은 경로가 현재작업경로를 벗어나거나 레포디렉토리를 포함한 하위 경로일 경우, 에러 처리 후 프롬프트 재출력(2점)
- 인자로 입력받은 경로가 일반 파일이나 디렉토리가 아니거나 해당 경로에 대해 접근권한이 없는 경우, 에러 처리 후 프로그램 프롬프트 재출력(2점)

3. 내장명령어 2. remove

1) Usage : remove 〈PATH〉
- 스테이징 구역에서 제거하여 변경내역을 추적하지 않을 경로(PATH)를 입력받아 해당 파일 또는 디렉토리 경로를 스테이징 구역에서 제거

2) 인자 설명
- 첫 번째 인자 〈PATH〉는 스테이징 구역에서 삭제할 파일이나 디렉토리의 상대경로와 절대경로 모두 입력 가능해야 함

3) 실행결과
- 첫 번째 인자 〈PATH〉를 스테이징 구역(.staging.log)에 기록함. 스테이징 구역에 기록된 경로는 commit 내장명령어 실행 시 변경내역을 추적하지 않아 백업을 진행하지 않는 경로임
- 만약 첫 번째 인자 〈PATH〉가 스테이징 구역에 기록되어 변경내용을 추적하지 않는 경로에 포함될 경우 ("현재작업경로에 대한 원본 상대경로" already removed from staging area")의 형태로 출력
- 실행 성공 시 (remove "현재작업경로에 대한 원본 상대경로")를 표준 출력하고, 스테이징 구역에 (remove "원본 절대경로")의 형태로 기록함

예제 3-1. remove 내장명령어 실행(현재 작업 디렉토리(pwd)가 "/home/oslab/lsp"일 경우)

```
% ./ssu_repo
20220000〉 add a
add "./a"

20220000〉 remove
ERROR: 〈PATH〉 is not include
Usage: remove 〈PATH〉 : record path to staging area, path will not tracking modification

20220000〉 remove asd.txt
ERROR: 'asd.txt' is wrong path

20220000〉 remove a.txt
remove "./a.txt"

20220000〉 remove a.txt
"./a.txt" already removed from staging area

20220000〉 remove ./a
remove "./a"

20220000〉 remove /home/oslab/lsp/a/b
"./a/b" already removed from staging area

20220000〉 remove ./
remove "."

20220000〉 add a
add "./a"

20220000〉 remove /home/oslab/lsp/a/a.txt
remove "./a/a.txt"

20220000〉 remove ./a
remove "./a"

20220000〉 add .
add "."
```

그림 3-2. /home/oslab/lsp 디렉토리 트리 구조

```
/home/oslab/lsp
├ .repo/
|  ├ .commit.log
|  └ .staging.log
├ a/
|  ├ a.txt
|  ├ b/
|  |  └ c.txt
|  └ c.txt
├ a.txt
└ b/
```

예제 3-3. 스테이징 구역(.staging.log) 파일 내용

```
add "/home/oslab/lsp/a"
remove "/home/oslab/lsp/a.txt"
remove "/home/oslab/lsp/a"
remove "/home/oslab/lsp"
add "/home/oslab/lsp/a"
remove "/home/oslab/lsp/a/a.txt"
remove "/home/oslab/lsp/a"
add "/home/oslab/lsp"
```

4) 예외 처리(미 구현 시 아래 점수만큼 감점, 필수 기능 구현 여부 판단과는 상관 없음)
- 경로를 입력하지 않을 경우, remove 명령어에 대한 에러 처리 및 Usage 출력 후 프롬프트 재출력(2점)
- 인자로 입력받은 경로가 길이 제한(4,096 Byte)를 넘거나 경로가 올바르지 않은 경우, 에러 처리 후 프롬프트 재출력(2점)
- 인자로 입력받은 경로가 현재작업경로를 벗어나거나 레포디렉토리를 포함한 하위 경로일 경우, 에러 처리 후 프롬프트 재출력(2점)
- 인자로 입력받은 경로가 일반 파일이나 디렉토리가 아니거나 해당 경로에 대해 접근권한이 없는 경우, 에러 처리 후 프로그램 프롬프트 재출력(2점)

4. 내장명령어 3. status

1) Usage : status

- 스테이징 구역에 추가된 경로들의 파일들에 대해 변경 내역을 추적하여 정보를 출력함

2) 인자 설명

- status 내장명령어는 인자를 추가로 받지 않음

3) 실행결과

- 스테이징 구역에 포함되어 있는 경로의 파일들에 대해 변경 내역을 추적하여 정보를 출력함. remove를 통해 제외된 경로에 대해서는 추적을 진행하지 않음
- 현재작업경로 내의 모든 파일들 중 스테이징 구역에 포함되어 있는 경로 내 파일들에 대하여 각자의 마지막 백업 파일에서 변경되거나 삭제, 새롭게 추가된 파일들에 대해 "Changes to be committed:" 다음 줄에 해당 파일의 상태와 경로를 출력함. 다음줄에 출력할 파일이 없다면 "Changes to be committed:"를 출력하지 않음
- 현재작업경로 내의 모든 파일들 중 스테이징 구역에 포함되어 있지 않고, 스테이징 구역에서 제거되지 않은 경로 내 파일들에 대하여 각자의 마지막 백업 파일에서 변경되거나 삭제, 새롭게 추가된 파일들에 대해 파일들에 대해 "Untracked files:" 다음 줄에 해당 파일의 상태와 경로를 출력함. 다음줄에 출력할 파일이 없다면 "Untracked files:"를 출력하지 않음
- 만약 마지막 백업 파일에서 변경되거나 삭제, 새롭게 추가된 파일이 없을 경우 "Nothing to commit" 출력 후 프롬프트 재출력
- 새로 추가된 파일의 경우 (new file: "현재작업경로에 대한 원본 상대경로"), 내용이 변경된 파일의 경우 (modified: "현재작업경로에 대한 원본 상대경로"), 삭제된 파일의 경우 (removed: "현재작업경로에 대한 원본 상대경로")를 출력. 이 때 파일의 상태와 경로를 출력하는 순서는 상위 경로로부터 BFS로 출력

예제 4-1. status 내장명령어 실행(현재 작업 디렉토리(pwd)가 "/home/oslab/lsp" 일 때)

```
// a.txt, a/a.txt는 직전에 "first commit"이라는 이름으로 백업되어 있음

% echo "edit file" 〉 ./a/b/c.txt

% echo "new file" 〉 ./a/c.txt

% ./ssu_repo
20220000〉 status
Changes to be committed:
  new file: "./a/b/c.txt"

Untracked files:
  new file: "./a/c.txt"

20220000〉 exit

% echo "hello world" 〉 a.txt

% rm ./a/a.txt

% ./ssu_repo
20220000〉 status
Changes to be committed:
  modified: "./a.txt"
  removed: "./a/a.txt"
  new file: "./a/b/c.txt"

Untracked files:
  new file: "./a/c.txt"

20220000〉
```

그림 4-2. /home/oslab/lsp 디렉토리 트리 구조

```
/home/oslab/lsp
├ .repo/
|  ├ .commit.log
|  ├ .staging.log
|  └ "first commit"
|      ├ .a/
|      |  └ a.txt
|      └ .a.txt
├ a/
|  ├ b/
|  |  └ c.txt
|  └ c.txt
├ a.txt
└ b/
```

예제 4-3. 스테이징 구역(.staging.log) 파일 내용

```
add "/home/oslab/lsp/a.txt"
add "/home/oslab/lsp/a/a.txt"
add "/home/oslab/lsp/a/b"
remove "/home/oslab/lsp/a/b/c.txt"
```

예제 4-4. 커밋 로그(.commit.log) 파일 내용

```
commit: "first commit" - new file: "/home/oslab/lsp/a.txt"
commit: "first commit" - new file: "/home/oslab/lsp/a/a.txt"
```

5. 내장명령어 4. commit

1) Usage : commit 〈NAME〉

- 커밋 이름(NAME)을 입력받아 해당 이름으로 버전 디렉토리를 생성하여 스테이징 구역에 포함되어 있는 경로 내 파일들에 대하여 각자의 마지막 백업 파일에서 변경되거나 삭제, 새롭게 추가된 파일들에 대해 백업을 진행

2) 인자 설명

- 첫 번째 인자 〈NAME〉은 버전 디렉토리 이름으로써 시스템상에서 지원하는 최대 파일 길이(255 byte)를 넘으면 안 됨

3) 실행결과

- 첫 번째 인자 〈NAME〉을 이름으로 하는 버전 디렉토리를 생성하여 스테이징 구역에 포함되어 있는 경로 내 파일들에 대하여 각자의 마지막 백업 파일에서 변경되거나 삭제, 새롭게 추가된 파일들에 대해 백업을 진행 및 커밋 로그 파일에 기록. 만약 기록할 파일이 없을 경우 버전 디렉토리를 생성하지 않고, "Nothing to commit"을 출력 후 프롬프트 재출력
- 스테이징 구역에 포함되어 있는 경로 내 파일들에 대하여 가장 마지막에 백업된 파일과 비교를 진행하여 백업 파일이 없을 시 "추가", 백업 파일이 있는데 내용이 변경된 경우 "변경", 백업 파일이 있는데 현재 작업경로에는 파일이 없을 경우 "삭제"로 취급. 그 외 파일은 백업 및 기록 대상이 아님
- commit 내장명령어 실행 성공 시 (commit to "〈NAME〉")을 출력 후 다음 줄에 몇 개의 파일이 변경되었는지, 몇 줄이 추가 및 삭제되었는지 (ex. 3 files changed, 17 insertions(+), 51 deletions(-))와 같은 형식으로 출력
- 다음 줄부터 백업 및 기록 대상인 파일들에 대해 "추가"의 경우 (new file: "현재작업경로에 대한 원본 상대경로"), "변경"의 경우 (modified: "현재작업경로에 대한 원본 상대경로"), "삭제"의 경우 (removed: "현재작업경로에 대한 원본 상대경로")의 형태로 출력. 이 때 파일의 상태와 경로를 출력하는 순서는 상위 경로로부터 BFS로 출력
- commit 내장명령어 실행 성공 시 커밋 로그 파일에 백업 및 기록 대상인 파일들에 대해 "추가"의 경우 (commit: "〈NAME〉" - new_file: "원본 절대경로"), "변경"의 경우 (commit: "〈NAME〉" - modified: "원본 절대경로"), "삭제"의 경우 (commit: "〈NAME〉" - removed: "원본 절대경로")의 형태로 기록. 이 때 파일의 상태와 경로를 기록하는 순서는 상위 경로로부터 BFS로 출력
- commit 내장명령어 실행 성공 시 생성되는 백업 파일의 경로는 현재작업경로 기준 동일한 경로의 파일을 버전 디렉토리 기준으로 생성(/home/oslab/lsp/**a/b/c.txt** -〉 /home/oslab/lsp/.repo/first_commit/**a/b/c.txt)**. 버전 디렉토리에 생성된 파일은 원본 파일과 stat값(파일모드, 시간, etc..)이 동일해야 함.
- commit 내장명령어 실행 성공 후에도 스테이징 구역은 유지

예제 5-1. commit 내장명령어 실행(현재 작업 디렉토리(pwd)가 "/home/oslab/lsp" 일 때)

```
// a.txt, a/a.txt는 직전에 "first commit"이라는 이름으로 백업되어 있음
// a.txt는 변경, a/a.txt는 현재 디렉토리에서 삭제됨

% ./ssu_repo
20220000〉 status
Changes to be committed:
  modified: "./a.txt"
  removed: "./a/a.txt"
  new file: "./a/b/c.txt"

Untracked files:
  new file: "./a/c.txt"

20220000〉 add ./a/b/c.txt
add "./a/b/c.txt"

20220000〉 commit
ERROR: 〈NAME〉 is not include
Usage: commit 〈NAME〉 : backup staging area with commit name

20220000〉 commit "second commit"
commit to "second commit"
3 files changed, 17 insertions(+), 51 deletions(-)
  modified: "./a.txt"
  removed: "./a/a.txt"
  new file: "./a/b/c.txt"

20220000〉 commit "first commit"
"first commit" is already exist in repo

20220000〉 commit "third commit"
Nothing to commit

20220000〉
```

그림 5-2. /home/oslab/lsp 디렉토리 트리 구조	예제 5-3. 스테이징 구역(.staging.log) 파일 내용
/home/oslab/lsp ├ .repo/ │ ├ .commit.log │ ├ .staging.log │ └ "first commit" │ ├ .a/ │ │ └ a.txt │ └ .a.txt ├ a/ │ ├ b/ │ │ └ c.txt │ └ c.txt ├ a.txt └ b/	add "/home/oslab/lsp/a.txt" add "/home/oslab/lsp/a/a.txt" add "/home/oslab/lsp/a/b" remove "/home/oslab/lsp/a/b/c.txt" add "/home/oslab/lsp/a/b/c.txt"

그림 5-4. commit 이후 /home/oslab/lsp 디렉토리 트리 구조	예제 5-5. 커밋 로그(.commit.log) 파일 내용
/home/oslab/lsp ├ .repo/ │ ├ .commit.log │ ├ .staging.log │ ├ "first commit" │ │ ├ .a/ │ │ │ └ a.txt │ │ └ .a.txt │ └ "second commit" │ ├ a/ │ │ └ b/ │ │ └ c.txt │ └ .a.txt ├ a/ │ ├ a.txt │ ├ b/ │ │ └ c.txt │ └ c.txt ├ a.txt └ b/	commit: "first commit" - new file: "/home/oslab/lsp/a.txt" commit: "first commit" - new file: "/home/oslab/lsp/a/a.txt" commit: "second commit" - modified: "/home/oslab/lsp/a.txt" commit: "second commit" - removed: "/home/oslab/lsp/a/a.txt" commit: "second commit" - new file: "/home/oslab/lsp/a/b/c.txt"

4) 예외 처리(미 구현 시 아래 점수만큼 감점, 필수 기능 구현 여부 판단과는 상관 없음)
- 첫 번째 인자 〈NAME〉을 입력하지 않을 경우, commit 내장명령어에 대한 Usage 출력 후 프롬프트 재출력(2점)
- 첫 번째 인자 〈NAME〉이 파일 길이 제한(255 Byte)를 넘거나 이름이 올바르지 않은 경우, 에러 처리 후 프롬프트 재출력(3점)
- 첫 번째 인자 〈NAME〉을 이름으로 하는 버전 디렉토리가 이미 존재한다면 에러 처리 후 프롬프트 재출력(3점)

6. 내장명령어 5. revert

1) Usage : revert 〈NAME〉
- 커밋 이름(NAME)을 입력받아 해당 이름의 버전으로 되돌아감

2) 인자 설명
- 첫 번째 인자 〈NAME〉은 버전 디렉토리 이름으로써 시스템상에서 지원하는 최대 파일 길이(255 byte)를 넘으면 안 됨

3) 실행결과
- 첫 번째 인자 〈NAME〉을 이름으로 하는 버전 디렉토리를 찾아 현재작업경로 내 파일들을 해당 버전을 백업했던 상태로 되돌림. 첫 번째 인자 〈NAME〉을 이름으로 하는 버전 디렉토리를 포함하여 해당 버전 이전에 commit을 진행한 모든 버전 디렉토리를 탐색하며 원본 경로가 같은 파일들에 대하여 각자의 마지막 백업 파일을 대상으로 복원을 진행함. 이 때, 삭제된 경우도 판단하여 파일의 유무 또한 판단해야 함. 만약 현재작업경로에는 파일이 있는데 해당 버전에서는 없던 파일일 경우 그대로 둠(현재작업경로의 파일을 삭제하지는 않음).
- revert 내장명령어 실행 성공 시 (revert to "〈NAME〉")을 출력 후, 다음 줄부터 복원된 파일들에 대해 (recover "현재작업경로에 대한 원본 상대경로" from "복구를 진행한 버전 디렉토리 이름")의 형태로 출력
- 해당 버전에서 존재하던 마지막 백업된 파일이 현재 파일과 내용이 같은 경우 복원을 진행하지 않음. 복원한 파일이 하나도 없다면 (nothing changed with "〈NAME〉")의 형태로 출력 후 프롬프트 재출력

예제 6-1. revert 내장명령어 실행(현재 작업 디렉토리(pwd)가 “/home/oslab/lsp” 일 때)

```
% ./ssu_repo
20220000〉 revert
ERROR: 〈NAME〉 is not include
Usage: revert 〈NAME〉 : recover commit version with commit name

20220000〉 revert “forth commit”
revert to “forth commit”
recover “./a/b/c.txt” from “third commit”

20220000〉 revert “second commit”
revert to “second commit”
recover “./a.txt” from “second commit”
recover “./a/b/c.txt” from “second commit” //modified

20220000〉 revert “second commit”
nothig changed with “second commit”

20220000〉 revert “tmp commit”
“tmp commit” is not exist in repo

20220000〉
```

그림 6-2. /home/oslab/lsp 디렉토리 트리 구조

```
/home/oslab/lsp
├ .repo/
|  ├ .commit.log
|  ├ .staging.log
|  ├ “first commit”
|  |  ├ a/
|  |  |  └ a.txt
|  |  └ a.txt
|  ├ “second commit”
|  |  ├ a/
|  |  |  ├ a.txt
|  |  |  └ b/
|  |  |     └ c.txt
|  |  └ a.txt
|  └ “third commit”
|     └ a/
|       └ b/
|         └ c.txt
└ b/
```

예제 6-3. 커밋 로그(.commit.log) 파일 내용

```
commit: “first commit” - new file: “/home/oslab/lsp/a.txt”
commit: “first commit” - new file: “/home/oslab/lsp/a/a.txt”
commit: “second commit” - modified: “/home/oslab/lsp/a.txt”
commit: “second commit” - removed: “/home/oslab/lsp/a/a.txt”
commit: “second commit” - new file: “/home/oslab/lsp/a/b/c.txt”
commit: “third commit” - modified: “/home/oslab/lsp/a/b/c.txt”
commit: “forth commit” - removed: “/home/oslab/lsp/a.txt”
…
```

그림 6-4. revert 후 /home/oslab/lsp 디렉토리 트리 구조

```
/home/oslab/lsp
├ .repo/
|  ├ .commit.log
|  ├ .staging.log
|  ├ “first commit”
|  |  ├ a/
|  |  |  └ a.txt
|  |  └ a.txt
|  ├ “second commit”
|  |  ├ a/
|  |  |  ├ a.txt
|  |  |  └ b/
|  |  |     └ c.txt
|  |  └ a.txt
|  └ “third commit”
|     └ a/
|       └ b/
|         └ c.txt
├ a/
|  └ b/
|     └ c.txt
├ a.txt
└ b/
```

4) 예외 처리(미 구현 시 아래 점수만큼 감점, 필수 기능 구현 여부 판단과는 상관 없음)

- 첫 번째 인자 〈NAME〉을 입력하지 않을 경우, revert 내장명령어에 대한 Usage 출력 후 프롬프트 재출력(2점)
- 첫 번째 인자 〈NAME〉이 파일 길이 제한(255 Byte)를 넘거나 이름이 올바르지 않은 경우, 에러 처리 후 프롬프트 재출력(3점)
- 첫 번째 인자 〈NAME〉으로 commit 명령어를 실행한 적이 없는 경우(버전 디렉토리가 존재하지 않고, 커밋 로그에도 해당 버전을 커밋한 흔적이 없는 경우) 에러 처리 후 프롬프트 재출력(3점)
- 해당 상태의 파일/디렉토리를 복원하려는데 똑같은 경로의 디렉토리/파일이 현재작업경로에 있을 경우, 에러 처리 후 프롬프트 재출력(3점)

7. 내장명령어 6. log

1) Usage : log [NAME]

- 커밋 기록에 대해 로그를 출력

2) 인자 설명

- 첫 번째 인자 [NAME] 입력 시 동일한 커밋 로그를 출력. 첫 번째 인자는 생략 가능하며, 생략 시 모든 커밋 로그를 출력

3) 실행결과

- commit한 기록에 대해 출력. (commit: “커밋 이름”) 형태로 출력하고 다음 줄부터 변경 사항을 출력함
- 출력하는 commit 기록의 순서는 commit 시간에 대해 오름차순으로 출력
- 첫 번째 인자 [NAME] 입력 시 동일한 커밋 로그에 대해서만 출력

예제 7-1. log 내장명령어 실행(현재 작업 디렉토리(pwd)가 "/home/oslab/lsp" 일 때)

```
% ./ssu_repo
20220000〉 log
commit: "first commit"
 - new file: "/home/oslab/lsp/a.txt"
 - new file: "/home/oslab/lsp/a/a.txt"

commit: "second commit"
 - modified: "/home/oslab/lsp/a.txt"
 - removed: "/home/oslab/lsp/a/a.txt"
 - new file: "/home/oslab/lsp/a/b/c.txt"

commit: "third commit"
 - modified: "/home/oslab/lsp/a/b/c.txt"

commit: "forth commit"
 - removed: "/home/oslab/lsp/a.txt"

…

20220000〉 log "second commit"
commit: "second commit"
 - modified: "/home/oslab/lsp/a.txt"
 - removed: "/home/oslab/lsp/a/a.txt"
 - new file: "/home/oslab/lsp/a/b/c.txt"

20220000〉
```

그림 7-2. /home/oslab/lsp 디렉토리 트리 구조

```
/home/oslab/lsp
├ .repo/
|  ├ .commit.log
|  ├ .staging.log
|  ├ "first commit"
|  |  ├ a/
|  |  |  └ a.txt
|  |  └ a.txt
|  ├ "second commit"
|  |  ├ a/
|  |  |  ├ a.txt
|  |  |  └ b/
|  |  |     └ c.txt
|  |  └ a.txt
|  └ "third commit"
|     └ a/
|       └ b/
|         └ c.txt
└ b/
```

예제 7-3. 커밋 로그(.commit.log) 파일 내용

```
commit: "first commit" - new file: "/home/oslab/lsp/a.txt"
commit: "first commit" - new file: "/home/oslab/lsp/a/a.txt"
commit: "second commit" - modified: "/home/oslab/lsp/a.txt"
commit: "second commit" - removed: "/home/oslab/lsp/a/a.txt"
commit: "second commit" - new file: "/home/oslab/lsp/a/b/c.txt"
commit: "third commit" - modified: "/home/oslab/lsp/a/b/c.txt"
commit: "forth commit" - removed: "/home/oslab/lsp/a.txt"
…
```

4) 예외 처리(미 구현 시 아래 점수만큼 감점, 필수 기능 구현 여부 판단과는 상관 없음)

- 첫 번째 인자 〈NAME〉입력 시 파일 길이 제한(255 Byte)를 넘거나 이름이 올바르지 않은 경우, 에러 처리 후 프롬프트 재출력(2점)
- 첫 번째 인자 〈NAME〉으로 commit 명령어를 실행한 적이 없는 경우(버전 디렉토리가 존재하지 않는 경우), 에러 처리 후 프롬프트 재출력(2점)
- 커밋 기록이 없다면 에러 처리 후 프롬프트 재출력(1점)

8. 내장명령어 7. help

1) Usage : help [COMMAND]

- 프로그램 내장명령어에 대한 설명(Usage) 출력

2) 인자 설명

- 첫 번째 인자 [COMMAND]에 대해 해당 내장명령어에 대한 설명(Usage)를 출력. 첫 번째 인자는 생략 가능하며, 생략 시 모든 내장명령어에 대한 설명(Usage) 출력

3) 실행결과

예제 8. help 내장명령어 실행

```
% ./ssu_repo
20220000〉 help
Usage:
  〉 add 〈PATH〉 : record path to staging area, path will tracking modification
  〉 remove 〈PATH〉 : record path to staging area, path will not tracking modification
  〉 status : show staging area status
  〉 commit 〈NAME〉 : backup staging area with commit name
  〉 revert 〈NAME〉 : recover commit version with commit name
  〉 log : show commit log
  〉 help : show commands for program
  〉 exit : exit program

20220000〉 help commit
Usage: commit 〈NAME〉 : backup staging area with commit name

20220000〉
```

4) 예외 처리(미 구현 시 아래 점수만큼 감점, 필수 기능 구현 여부 판단과는 상관 없음)

- 잘못된 내장명령어 입력 시 에러 처리 후 프롬프트 재출력(1점)

9. 내장명령어 8. exit

1) Usage : exit

- 현재 실행중인 ssu_repo 프로그램 종료

2) 실행결과

- 프로그램 종료

예제 9. exit 내장명령어 실행

```
% ./ssu_repo
20220000〉 exit

%
```

○ 과제 구현에 필요한 함수 (필수 아님)

- 1. getopt() : 프로그램 실행 시 입력한 인자를 처리하는 라이브러리 함수

```
#include 〈unistd.h〉
int getopt(int argc, char * const argv[], const char *optstring); //_POSIX_C_SOURCE

#include 〈getopt.h〉
int getopt_long(int argc, char * const argv[], const char *optstring, const struct option *longopts, int *longindex);
//_GNU_SOURCE
```

- 2. scandir : 디렉토리에 존재하는 파일 및 디렉토리 전체 목록 조회하는 라이브러리 함수

```
#include 〈dirent.h〉
int scandir(const char *dirp, struct dirent ***namelist, int (*filter)(const struct dirent *), int (*compar)(const struct
dirent **, const struct dirent **));

-1 : 오류가 발생, 상세한 오류 내용은 errno에 설정
0 이상 : 정상적으로 처리, namelist에 저장된 struct dirent *의 개수가 return
```

- 3. realpath : 상대경로를 절대경로로 변환하는 라이브러리 함수

```
#include 〈stdlib.h〉
char *realpath(const char *path, char *resolved_path);

NULL : 오류가 발생, 상세한 오류 내용은 errno 전역변수에 설정
NULL이 아닌 경우 : resolved_path가 NULL이 아니면, resolved_path를 return,
resolved_path가 NULL이면, malloc(3)으로 할당하여 real path를 저장한 후에 return
```

- 4. strtok : 특정 문자 기준으로 문자열을 분리하는 함수

```
#include 〈string.h〉
char *strtok(char *restrict str, const char *restrict delim);

 return a pointer to the next token, or NULL if there are no more tokens.
```

- 5. exec()류 함수 : 현재 프로세스 이미지를 새로운 프로세스 이미지로 대체하는 함수(라이브러리, 시스템콜)

```
#include 〈unistd.h〉
int execl(const char *pathname, const char *arg, .../* (char  *) NULL */);
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg, .../*, (char *) NULL, char *const envp[] */);
int execve(const har* pathname, char *const argv[], char *const ecnp[]);
int execlp(const char *file, const char *arg, .../* (char  *) NULL */);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],char *const envp[]);

 The exec() family of functions replaces the current process image with a new process image.
 https://man7.org/linux/man-pages/man3/exec.3.html  또는 교재 참고
```

- 6. MD5

✓ MD5 해시값을 구하기 위해 MD5(openssl/md5.h)를 사용

- Linux, Ubuntu :"sudo apt-get install libssl-dev"로 라이브러리 설치 필요
- Linux, Fedora :"sudo dnf-get install libssl-devel"로 라이브러리 설치 필요
- MacOS : homebrew (https://brew.sh/ 참고) 설치 -〉 % brew install openssl
- MD5 관련 함수를 사용하기 위해 컴파일 시 "-lcrypto" 옵션 필요
- md5() 사용법은 https://www.openssl.org/docs/man1.1.1/man3/MD5.html 및 https://github.com/Chronic-Dev/openssl/blob/master/crypto/md5/md5.c 참고

○ make와 Makefile

- make : 프로젝트 관리 유틸리티
  - ✓ 파일에 대한 반복 명령어를 자동화하고 수정된 소스 파일만 체크하여 재컴파일 후 종속된 부분만 재링크함
  - ✓ Makefile(규칙을 기술한 파일)에 기술된 대로 컴파일 명령 또는 쉘 명령을 순차적으로 실행함
- Makefile의 구성
  - ✓ Macro(매크로) : 자주 사용되는 문자열 또는 변수 정의 (컴파일러, 링크 옵션, 플래그 등)
  - ✓ Target(타겟) : 생성할 파일
  - ✓ Dependency(종속 항목) : 타겟을 만들기 위해 필요한 파일의 목록
  - ✓ Command(명령) : 타겟을 만들기 위해 필요한 명령(shell)

```
Macro

Target : Dependency1 Dependency2 ...
〈-Tab-〉Command 1
〈-Tab-〉Command 2
〈-Tab-〉...
```

- Makefile의 동작 순서
  - ✓ make 사용 시 타겟을 지정하지 않으면 제일 처음의 타겟을 수행
  - ✓ 타겟과 종속 항목들은 관습적으로 파일명을 명시
  - ✓ 명령 항목들이 충족되었을 때 타겟을 생성하기 위해 명령 (command 라인의 맨 위부터 순차적으로 수행)
  - ✓ 종속 항목의 마지막 수정 시간(st_mtime)을 비교 후 수행
- Makefile 작성 시 참고사항
  - ✓ 명령의 시작은 반드시 Tab으로 시작해야함
  - ✓ 한 줄 주석은 #, 여러 줄 주석은 자동 매크로 : 현재 타겟의 이름이나 종속 파일을 표현하는 매크로

매크로	설명
$?	타겟보다 최근에 변경된 종속 항목 리스트 (확장자 규칙에서 사용 불가능)
$^	현재 타겟의 종속 항목 (확장자 규칙에서 사용 불가능)
$<	타겟보다 최근에 변경된 종속 항목 리스트 (확장자 규칙에서만 사용 가능)
$*	타겟보다 최근에 변경된 종속 항목 리스트 (확장자 규칙에서만 사용 가능)
$@	현재 타겟의 이름

- Makefile 작성 예시

(예 1). Makefile

```
test : test.o add.o sub.o
    gcc test.o add.o sub.o -o test

test.o: test.c
    gcc -c test.c

add.o: add.c
    gcc -c add.c

sub.o: sub.c
    gcc -c sub.c

clean :
    rm test.o
    rm add.o
    rm sub.o
    rm test
```

(예 2). 매크로를 사용한 경우

```
OBJECTS = test.o add.o sub.o
TARGET = test
CC = gcc

$(TARGET) : $(OBJECTS)
    $(CC) -o $(TARGET) $(OBJECTS)

test.o: test.c
    $(CC) -c test.c

add.o: add.c
    $(CC) -c add.c

sub.o: sub.c
    $(CC) -c sub.c
```

(예 3). 자동 매크로를 사용한 경우

```
OBJECTS = test.o add.o sub.o
TARGET = test
CC = gcc

$(TARGET) : $(OBJECTS)
    $(CC) -o $@ $^

test.o: test.c
    $(CC) -c $^

add.o: add.c
    $(CC) -c $^

sub.o: sub.c
    $(CC) -c $^
```

- (예 4). Makefile 수행 예시

```
oslab@a-VirtualBox:~$ make
gcc -c test.c
gcc -c add.c
gcc -c sub.c
gcc test.o add.o sub.o -o test
oslab@a-VirtualBox:~$
```

```
oslab@a-VirtualBox:~$ make clean
rm test.o
rm add.o
rm sub.o
rm test
oslab@a-VirtualBox:~$
```

○ 보고서 제출 시 유의 사항

- 보고서 제출 마감은 제출일 11:59PM까지 (구글 서버시간)
- 지연 제출 시 감점 : 1일 지연 시 30% 감점, 2일 이후 미제출 처리
- 압축 오류, 파일 누락 관련 감점 syllabus 참고

- 필수구현 : 1. ssu_repo, 2. 내장명령어 add~~, 5. 내장명령어 commit, 6. 내장명령어 revert~~(각 옵션 및 예외처리는 필수 구현 아님. 단, 별도 감점 있음)
- 배점(100점 만점. 실행 여부 배점 80점으로 최종 환산하며, 보고서 15점과 소스코드 주석 5점은 별도, 강의계획서 확인)

1. ssu_repo - 10점
2. 내장명령어 1. add - 15점
3. 내장명령어 2. remove - 15점
4. 내장명령어 3. status - 5점
5. 내장명령어 4. commit - 20점
6. 내장명령어 5. revert - 20점
7. 내장명령어 6. log - 5점
8. 내장명령어 7. help - 3점
9. 내장명령어 8. exit - 2점

- makefile 작성(매크로 사용하지 않아도 됨) - 5점