컴퓨터학부 20202920 조민혁

1.과제 개요

ssu\_repo는 함수를 실행하여 add명령어를 통해 staging에 올려서 파일이나 디렉토리를 추적하여, 그 변화를 감지하여 변화(modified), 삭제(removed), 추가(new file)을 확인하는 함수입니다. 이때 staging되는 것들에 대해서는 로그파일을 통해서 관리하게 됩니다. Staging에 올려둔 파일이나 디렉토리 들은 remove명령어를 통해서 staging에서 내려올 수 있습니다. 그렇게 되는 경우 더 이상 추적하지 않습니다. 또한 exit명령어를 사용하여 프로그램을 끝낼 수 있고 help 명령어를 사용하여 내장명령어의 사용법에 대해서 출력할 수 있습니다. Commit명령어를 이용하여 백업된 것들과 staging된 것들의 내용을 비교하여 백업을 진행하고 이에 대한 결과를 출력하며 .commit.log파일에 작성하여 관리하는 프로그램입니다.

2.기능

add : add명령어는 파일이나 디렉토리를 staging에 올려주는 기능을 합니다. 파일이나 디렉토리가 존재하지 않는 경우나 이미 staging에 올라가 있는 파일에 대해서는 예외처리를 진행해줍니다. add하게 되어 staging에 올라가게 된 경우는 현재 작업디렉토리에 생긴 .repo디렉토리 밑의 .staging.log에 해당 내용을 작성하여 줍ㄴ디ㅏ.

remove : remove명령어는 staging에 올라가 있는 파일이나 명령어를 staging에서 내려주는 함수입니다. 파일이나 디렉토리가 존재하지 않는 경우에도 staging에 올라가 있으면 삭제를 진행해주고, staging에도 없고 실제로 존재하지 않는 파일에 대한 처리와 잘못된 경로에 대한 예외처리도 진행해줍니다.

commit : commit명령어는 staging에 올라간 파일에 대해 처음 commit하는 경우 디렉토리의 이름을 입력받아 .repo디렉토리 밑의 해당디렉토리에 staging에 있는 파일들과 디렉토리 안의 파일들을 계층 구조를 유지한 채로 백입시켜 줍니다. 또한 commit 명령어는 .repo디렉토리안의 이미 존재하는 디렉토리의 이름에 대해서는 예외처리를 진행해주고, 바뀐 내용이 없을 경우에 대해서도 예외처리를 해줍니다. 이때 바뀐 내용이란 파일 내부의 데이터가 바뀐 것뿐만이 아니라 변화(modified), 삭제(removed), 추가(new file)의 바뀜을 의미합니다. modified의 경우에는 백업 파일과 비교하여 바뀐 파일이 몇 줄 삭제되었고 몇 줄이 추가되었는지에 대한 정보도 출력하여 줍니다.

help : help명령어는 내장명령어에 대한 사용법을 출력하고 help commit 과 같이 입력하면 commit에 대한 사용법을 따로 출력하며 잘못된 명령어나 “help 이상한 문자열” 과 같은 경우에는 전체 명령어에 대한 설명이 출력되게 됩니다. 또한 다른 명령어를 실행 중에 잘못된 입력이 들어온 경우에 그 명령어에 대한 설명이 출력되게 만들어줍니다.

exit : exit명령어 사용 시 프로그램을 안전하게 종료할 수 있습니다.

3.상세설계

<함수 및 모듈 구성>

우선 Init에서는 staging로그를 기준으로 스테이지 된 것에 대한 링크드리스트를 생성해줍니다. 로그를 읽어서 add의 경우에는 파일과 디렉토리를 나눠서 각각의 링크드리스트로 만들어주고, 디렉토리의 경우 하위 디렉토리와 디렉토리 내부의 파일들을 각각의 디렉토리 노드에 집어넣습니다. 디렉토리 링크드 리스트의 경우 bfs로 추가하기 때문에 링크드리스트로 계층관계를 유지할 수 있습니다.

그 후 로그를 읽어 remove가 적힌 경우에는 앞서 만들어진 링크드 리스트를 돌면서 해당하는 파일들을 스테이징을 위한 링크드 리스트에서 제거시켜줍니다. 또한 remove명령어 같은 경우에는 삭제되었지만 staging에 남아있는 파일에 대해서도 예외처리를 해주었습니다. 파일의 경우 첫 째로 staging에 올라간 파일들에 대한 링크드리스트에서 조사하고 그 후에 디렉토리 링크드 리스트 안에 있는 파일들에 대해서 조사합니다.

commit 명령어의 경우에는 만들어진 staging의 리스트들을 통해서 첫 번째 commit의 경우 staging리스트에 올라와있는 모든 파일들에 대해서 계층구조를 유지한 채 입력 받은 디렉토리 이름으로 생성하여 .repo디렉토리 밑에 백업파일들을 생성해줍니다. 두 번째 commit의 경우에는 현재 staging에 올라온 파일들에 한해서 전의 commit으로 백업된 파일들의 최신 버전을 찾아서 비교해줍니다. 이때 비교하였을 때는 파일이 그대로 존재하지만 변경된 경우 modified로 몇 줄이 추가되었고 몇 줄이 삭제되었는지도 출력합니다. 파일이 백업 안되어있지만 staging에 새로 올라온 경우에는 new file로 새롭게 추가된 파일로 적용되어서 백업시켜 줍니다. 백업된 파일에는 존재하지만 staging에는 있고 현재 작업디렉토리 상에서 삭제된 파일의 경우에는 removed가 작성되고 이 removed를 staging에서 내려주지 않는 이상 commit할 때마다 로그에 계속 작성되게 됩니다. 그리고 또한 변화가 없는 경우에는 “Nothing to commit”을 출력하는데 이 경우에는 staging에 올라온 파일들과 백업된 파일들 간의 차이가 없기에 디렉토리를 생성하여 백업하는 과정을 진행하지 않습니다. 또한 여러 개의 staging file 중에서 modified되지 않은 파일에 대해서는 백업을 진행하여 주지 않습니다. 왜냐하면 modified되지 않은 경우에는 commit 로그에 작성되지 않는데 로그를 기준으로 최신버전의 백업과 비교하는 것인데 modfied되지 않았는데 파일을 백업할 경우에는 그 파일들을 접근하지 않는데 메모리를 낭비하는 것이기 때문입니다.

help를 사용하게 되는 경우에 내장명령어에 대해서 출력할 수 있습니다. “help commit” 과 같이 commit에 대한 설명만을 출력할 수 있고 잘못된 명령어를 입력한 경우나 help뒤에 잘못된 명령어가 입력된 경우에는 전체 내장명령어를 출력합니다.

exit 명령어를 통해서는 프로그램을 안전하게 종료할 수 있습니다.

아래는 헤더파일에 작성되어 있는 구조체에 대한 설명입니다.

//파일의 변경된 부분들을 계산하기 위한 구조체입니다.

typedef struct change\_File {

int insertLine;

int deleteLine;

int fileCount;

} changeFile;

//exec해줄 때 넘겨주는 인자에 대한 구조체입니다.

typedef struct command\_parameter {

char \*command;

char \*filename;

char \*tmpname;

char \*argv[10];

} command\_parameter;

//path를 구해줄 때 쓰기 위한 구조체이다.

typedef struct pathList\_ {

struct pathList\_ \*next;

struct pathList\_ \*prev;

char path[NAMEMAX];

} pathList;

//스테이지에 올라온 것들을 링크드리스트로 관리하기 위해 사용하는 구조체입니다.

typedef struct stagingList\_ {

struct stagingList\_ \*next;

struct stagingList\_ \*head;

int level;

char dirPath[PATHMAX];

char file\_name[FILE\_MAX];

char path[PATHMAX];

}stagingList;

//디렉토리를 스테이징에서 관리하기 위한 구조체입니다.

typedef struct dirList\_{

struct dirList\_ \*next;

struct dirList\_ \*head;

//디렉토리 리스트의 헤드를 설정해주는 것이다.

struct stagingList\_ \*stageHead;

//stagingList\_의 헤더를 저장해서 넣어줘야한다.

int level;

char parentDir[PATHMAX];

//부모 디렉토리를 적어둬서 얘의 계층구조를 이용할 수 있도록 해준다.

char dirPath[PATHMAX];

char dirName[STRMAX];

int file\_cnt;

//파일의 개수가 필요할 수 있기 때문에 저장해준다.

}dirList;

//queue를 만들기 위한 구조체입니다.

typedef struct queue{

dirList \*first;

dirList \*last;

}queue;

//전역변수로 선언하여 링크드리스트를 관리할 때 유용하게 만든다.

stagingList \*useStage;

dirList \*useDirList;

stagingList \*removeFileList;

dirList \*removeList;

위의 전역변수에서 총 4개를 만듭니다. 2개는 삭제된 애들 remove되었던 애를 다시 remove하는 경우에 대한 already removed를 출력하기 위해서 사용하고 나머지 2개의 경우는 add명령어를 통해 staging에 올려두기 위해 사용합니다.

위의 stagingList 구조체를 통해 파일들에 대한 add와 remove 명령어를 읽고 링크드리스트를 생성하는 것입니다.  
그리고 dirList 구조체를 통해 디렉토리에 대한 add와 remove 명령어를 읽고 링크드리스트를 생성합니다.

commit에 사용되는 리스트 또한 위의 구조체인 stagingList를 활용하여 staging에 해당하는 백업파일들의 리스트를 구성합니다

더 자세한 함수 모듈의 작동 과정은 아래의 함수 프로토콜을 통해서 같이 설명하겠습니다.

<함수 프로토타입>

//\*\*\*헤더에 포함되어 있는 함수들\*\*\*

\*\*아래의 큐와 관련된 함수들은 BFS를 사용하기 위해서 만들졌습니다.

//queue를 연결하는 함수입니다.

void enqueue(queue\* Queue, char\* dirPath)

//큐에서 맨앞에 들어온 node를 빼는 함수이다.

dirList\* dequeue(queue\* Queue)

//파일의 이름을 뽑아내는 함수입니다.

char \*GetFileName(char file\_path[])

//스테이지에 올라간 파일의 트리 구조상 level을 구하는 함수입니다.

void getLevelForStage(stagingList \*stageNode)

//스테이지에 올라간 디렉토리의 트리 구조상 level을 구하는 함수입니다.

void getLevelForDir(dirList \*dirNode)

level을 구해서 상위 경로로부터 BFS로 출력되게끔 설정해줍니다. Level이 낮은 것이 앞으로 오게끔 뒤에 등장하게 되는 함수에서 사용합니다

//stagingList를 추가하는 함수이다.

stagingList\* addStagingList(stagingList \*head, char\* newPath,char \*dirPath)

위의 함수를 사용하여 stageList, add명령어를 통해 스테이지에 올라간 파일들을 링크드리스트의 형태로 만들어서 추가하는 역할을 합니다.

//파일들로 이루어진 링크드리스트를 입력받아 노드를 삭제해주는 함수입니다.

stagingList \*deleteStagingList(stagingList \*head, char\* deletePath)

stagingList의 링크드리스트에서 원하는 경로를 입력받아 해당 경로를 포함하고 있는 노드를 제거해줍니다.

//원하는 파일을 찾는 함수입니다.

int findingStagingNode(stagingList \*head, char\* findingPath)

//스테이징에 올릴 디렉토리에 대한 노드를 연결해서 추가해주는 함수입니다.

dirList \*addDirListNode(dirList \*head, char \*newPath,stagingList \*stageHead,int fileCount)

dirList라는 구조체를 사용하여 staging에 올라오는 디렉토리를 이어주는 링크드리스트를 만들어줍니다.

//스테이징에 올릴 디렉토리에 대한 노드를 재귀함수를 이용한 BFS로 구현하여 찾는 함수입니다.

void addDirList(queue\* Queue,dirList \*head, char\* newPath, stagingList \*stageHead)

재귀 함수를 사용하는 BFS를 통해서 위의 addDirListNode함수를 내부에서 호출하며 staging된 디렉토리 리스트를 관리하게 됩니다.

//디렉토리 링크드리스트 안에서 해당하는 노드를 찾는 함수입니다.

int searchingNode(dirList\* head, char \* findPath, int option)

option에 따라 찾는 것이 디렉토리인지 파일인지를 결정합니다.

//디렉토리 안에 있는 파일 링크드 리스트에서 해당하는 것을 찾아 삭제시켜주는 함수입니다.

void deleteNodeInDir(dirList \*head, char \*findPath)

//디렉토리 링크드 리스트에서 디렉토리를 찾아 삭제해주는 함수입니다.

dirList \*deleteDirNode(dirList \*head, char\*deletePath)

//디렉토리의 트리 구조대로 삭제하기 위해서 상위디렉토리가 삭제되면 하위 디렉토리도 삭제되게끔 구현한 함수입니다.

void deleteDir(dirList \*head, char \*findPath,int option)

위의 deleteDirNode함수를 내부에서 호출하며 재귀적으로 option을 사용하여 부모 디렉토리 경로로 가지고 있느냐로 따져서 부모 디렉토리 경로로 삭제될 디렉토리의 경로를 가지고 있다면 그 하위 디렉토리도 삭제되끔 만들어두었습니다. 삭제할 디렉토리의 하위 구조의 모든 디렉토리가 삭제될 수 있습니다.

//해시 값을 비교하기 위해서 md5를 사용하는 것입니다.

int md5(char \*target\_path, char \*hash\_result)

//특정 구분자를 문자열을 분리하는 함수입니다.

char \*Tokenize(char \*str, char \*del)

//특정한 문자열 기준으로 토큰화하여 잘라주는 함수입니다.

char \*\*GetSubstring(char \*str, int \*cnt, char \*del)

//절대 경로로 변환시켜주는 함수입니다.

int ConvertPath(char\* origin, char\* resolved)

//절대경로를 상대경로로 바꿔주는 함수입니다.

int realpath\_2\_relative(char \*realpath, char \*returnPath)

출력해야하는 것들이 상대경로가 많기 때문에 상대경로로 바꾸어주는 함수입니다.

//directory에 접근가능한지 체크하여 접근가능하지 않으면 하위 디렉토리부터 생성해주는 함수입니다.

char \*accessDirCheck(char \*newPath)

경로를 입력받으면 해당경로의 디렉토리에 접근가능한지 확인하고 접근이 가능하지 않으면 재귀적으로 하나씩 access하여 생성할 수 있는지 확인하여 mkdir로 없는 경우에 디렉토리를 생성하는 함수입니다.

//링크드리스트를 level을 기준으로 정렬해주는 함수입니다.

void sortList(dirList \*\*head)

앞에서 구해두었던 level을 통해 디렉토리의 리스트를 level별로 다시금 링크드리스트를 정렬합니다.

//스테이지에 올라간 함수와 디렉토리로 올라간 것들을 총합하여 하나의 디렉토리 리스트로 만들어주는 함수입니다.

void collectDir(dirList \*dirHead, stagingList \*stageHead)

commit명령어를 쉽게 사용하기 위해서 stageList에 파일들과 dirList의 디렉토리안의 파일들을 다르게 보고 두 개의 링크드리스트에서 확인하는 과정을 앞선 add와 remove 명령어에서는 진행했었는데, 하나의 디렉토리 링크드리스트로 만들어서 파일들을 관리하기 쉽게 만들었습니다.

// 디렉토리 안에 있는 모든 파일들을 BFS로 순회하여 stagingList로 만드는 함수입니다.

//option이 0일 경우 repo함수를 포함해서 하는 경우입니다.

void traverseDirectory(const char \*dirPath, stagingList \*\*head, char \*unchangeName,int option)

commit된 파일들의 링크드리스트를 생성하기 위해서 해당하는 디렉토리의 전체 파일들을 하나의 stagList 즉, 파일의 링크드리스트로 만들어주는 함수입니다.

//두 파일의 삭제된 문장과 추가된 문장을 찾는 함수입니다.

void compareFile(char \*oldPath, char \*newPath,changeFile \*changeCollect)

파일 두 개를 비교하여 commit하였을 때 몇 문장이 삭제되었고 몇 문장이 추가되었는지 출력하기 위해서 만들어진 함수입니다.

//파일에 있는 문자의 수를 세는 함수입니다.

void countFileLine(changeFile \*changeFile,char\* filePath){

new file로 추가되는 파일에 대해서도 추가되는 문장이기 때문에 갯수를 세줍니다.

\*\*\*ssu\_repo에 포함되어 있는 함수들\*\*\*

//commit이 입력되었을 때 실행되는 함수입니다.

int CommitCommand(command\_parameter \*parameter)

commit 명령어가 입력되었을 때 fork() exec()되어서 실행되는 함수입니다. 이 함수는 첫 번째 commit의 경우에는 모든 파일들을 백업해준 후 두 번째 시행이후부터는 최신 버진의 백업된 것을 기준으로 하여 추가, 삭제, 변형을 판단하여 이에 대해서 출력하여 줍니다. 내부에는 최신 버전의 백업파일들의 링크드를 만드는 함수와 그를 통해서 현재 staging에 있는데 백업에 없는 경우에는 추가, 현재 스테이지에 없는데 백업에 있는 경우 삭제, 현재와 백업에 같이 존재하는데 내용이 변경된 파일에 대해서는 변형과 관련된 결과를 출력하여 줍니다.

//add명령어를 입력하였을 때 실행되는 함수입니다.

int AddCommand(command\_parameter \*parameter)

이미 staging에 올라와있는지 아닌지 판단하여 있는 경우에 대한 예외처리와 없는 경우에 대한 정상적인 staging추가를 진행시켜줍니다.

//remove명령어를 시행했을 경우입니다.

int RemoveCommand(command\_parameter \*parameter)

removed리스트에 있는지 판단하여 이미 removed된 것인지 아닌지 판단하여 줍니다.

//help명령어가 시행되었을 때 실행되는 함수입니다.

int HelpCommand(command\_parameter \*parameter)

필요한 명령어에 대한 검색을 가능하게 해주는 함수입니다.

//fork()되었을 때 어떤 함수를 실행할지 설정해주는 함수입니다.

void CommandFun(char \*\*arglist)

exec되어 넘어왔을 때 실행되는 함수로 parameter를 넣은 채로 해당하는 함수를 실행시켜줍니다.

//fork()시켜주는 함수입니다.

void CommandExec(command\_parameter parameter)

해당하는 내장 명령어와 같은 parameter들을 담아서 fork후 exec해주는 것입니다.

//helpExec으로 잘못된 명령어가 들어왔을 때 help명령어를 수행하기 위해서 따로 fork()해주는 함수입니다.

void HelpExec()

//parameter를 초기화해주는 함수입니다.

void ParameterInit(command\_parameter \*parameter)

//인자로 받은 것에 대한 처리를 진행해주는 함수입니다.

int ParameterProcessing(int argcnt, char \*\*arglist, int command, command\_parameter \*parameter)

인자로 받은 것들을 어떻게 넣어줄 것인지 먼저 설정해주는 함수입니다.

//계속 출력되어 나오는 프롬프트를 설정하는 함수이다.

int Prompt()

입력되는 문자열이 어떤 것인지에 대한 설정을 할 수 있고 fork한 후에 exec할 때 parameter로 넣어줄 값들을 설정할 수 있게 해주는 함수입니다. 여러 가지의 내장명령어에 대해서 분기하고 ParameterProcessing을 내부에서 호출한 후 exec하기 위해서 CommandExec함수도 내부에서 호출시켜줍니다.

//처음 프로그램이 실행될 때 기본 설정을 해주는 함수다.

void Init()

로그를 기준으로 add와 remove를 읽고 해당하는 removed링크드리스트랑 add해서 staging에 올라간 링크드 리스트 총 4가지를 생성해줍니다. 프로그램이 실행될 때마다 가장 먼저 실행되는 함수로 기본적인 설정을 이곳에서 진행해줍니다.

\*\*\*help.c에 포함되어 있는 함수들\*\*\*

아래의 함수들을 통해서 각각의 help에 대한 것들을 출력하여 줍니다.

//help.c에 적혀 있는 함수들의 함수원형들입니다.

void printAll();

void printAdd();

void printRemove();

void printStatus();

void printCommit();

void printRevert();

void printLog();

void printHelp();

void printExit();

<순서도>

도표, 평면도, 기술 도면, 개략도이(가) 표시된 사진

자동 생성된 설명

4.실행화면

add 명령어를 수행하였을 때 :

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

remove 명령어를 수행하였을 때

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

commit 명령어를 수행하였을 때 :

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

그때의 commit log입니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

그때의 staging log입니다.

help 명령어를 수행하였을 때 :

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

exit 명령어를 수행하였을 때 :

텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명