컴퓨터학부, 201905111, 배준형

# ssu\_backup 프로그램 개요

|  |  |
| --- | --- |
| |  | | --- | | SSU-backup 프로그램의 핵심은 md5, sha1 해시알고리즘을 백업파일을 만들고 복원하는 프로그램이다.  다른 백업 알고리즘과 다른 가장 큰 핵심은 md5와 sha1을 이용한다는 점이다. 해시 알고리즘을 쓰면 개각각 다른 길이의 파일들을 해시 특유의 성질인 동일한 크기의 비트를 생성할 수 있다는 장점이 있다. 또 다른 장점으로는, 내용이 조금만 달라지더라도 해시값이 엄청나게 바뀌기 때문에 문자열 비교에서도 큰 장점을 보인다.  예를 들면, md5 기준으로  "The quick brown fox jumps over the lazy dog" 를 해시로 돌리면 9E107D9D372BB6826BD81D3542A419D6  "The quick brown fox jumps over the lazy dog." 를 해시로 돌리면 E4D909C290D0FB1CA068FFADDF22CBD0 가 된다.  단지 '.' 하나가 붙냐 안붙냐 차이로 해시값은 천차만별이 되어 딱봐도 달라지게 된다. 이러한 두 장점 덕분에 두 파일의 해시 비교비용이 매우 줄어들게 된다는 엄청난 장점이 있다. 해시값으로 두 값을 비교하는 과정이 효율적이다는 실제 사례 알고리즘은 **Rabin-Karp 알고리즘**을 보면 알 수 있을 것이다. Rabin-Karp 알고리즘은 두 문자열을 비교문자열(Pattern) 만큼을 서로 해싱하여 해시값이 같은지 비교하여 서브스트링(SubString)이 있는지 확인하는 알고리즘이다. 이 알고리즘을 사용하면 시간 복잡도가 O(N)으로 줄일 수 있다.  브루트포스 (Brute-Force) 방식이 O(N\*N) 시간 복잡도를 가질 때 O(N) 시간복잡도로 줄일 수 있다는 점을 보면 얼마나 획기적인 방식인지 알 수 있을 것이다.  해시 함수에 대해 설명을 해보자면,  md5 (Message-Digest algorithm 5) 란 임의의 길이값을 받아 128비트 길이의 해시값을 출력하는 알고리즘이다. SHA1 (Secure Hash Algorithm1) 란 임의의 길이값을 받아 160 비트 해시값을 사용한다.  간결하게, 두 알고리즘들은 단방향 암호화 알고리즘으로서 원문데이터를 해시계산과정에서 비트 단위로 잘게 부셔버리기 때문에 반대로 복호화 하기 힘들게 만드는 알고리즘이다. 덕분에 **과거에** 패스워드 같은 데이터베이스에서 많이 활용하게 되었다. 하지만 기술력이 발달함에 따라 장비가 좋아져 같은 해시값을 가지는 문자열을 찾는 과정 "충돌(Collision)" 을 찾기가 쉬워져 md5 알고리즘은 현재 사용하지 않고 있고 SHA1, SHA2, SHA3 이라는 이름으로 SHA는 많은 분야 (인터넷 뱅킹, 비트코인 등) 에서 서용중이다.    SSU-backup 에서 MD5 와 SHA-1 알고리즘은 파일 비교과정에서 사용된다.  백업파일을 만든다고 가정해보자, 백업파일에 이미 내용이 같은 백업파일을 하나 더 만든다면 메모리 낭비가 심할 것이다. 그럼 백업 전 파일 내용의 비교가 중요하다. 그런데 파일길이가 굉장히 길다고 생각해보자. DRAM(메인메모리) 공간은 한정적인데 디스크에서 두 파일을 한꺼번에 읽어들여서 버퍼에 저장해서 하나씩 서로 비교하는 과정은 비효율적이다. 그리고 해당 내용들을 저장해놓기도 매우 곤란하다. 아마 heap이던 stack 공간이던 메모리공간이 박살이 날 것이라고 강한 추론을 한다. 다수의 파일들을 서로 비교할 때 가장 좋은 점은 무엇일까?  많은 정답 중 하나는 파일들을 각각 해싱하여 해시값을 저장해놓으면 된다. 파일의 길이가 얼마나 길던 해싱하여 나온 문자열의 길이는 고정된 길이의 문자열로 특정 비트 (긴 SHA-1 기준으로 160비트 == 16진수로 40글자) 로 표현 가능해진다. 이러한 특징으로 인하여 파일마다 고정된 길이의 해시값을 가지게하는 구조체를 만들 수 있다는 생각을 할 수 있다. 덤으로 해시의 두 번째 특징인 조금만 바뀌더라도 해시값이 엄청나게 달라짐을 이용한다면 문자열 비교과정도 굉장히 짧은 시간에 끝날 것이다. | |

# ssu\_backup 구현된 기능 소개 및 결과물

* 우선 뒤에 ‘실행결과’를 보면 확인할 수 있는 부분은 캡쳐사진을 생략하였다.

## ***ssu\_backup 전체적인 기능 구현***

### **<ssu\_backup 전체적인 기능 원리/및 구현>**

|  |
| --- |
| ssu\_backup 에서 파일관리 방식은 모두 **연결리스트**로 관리된다. ssu\_backup에서 각 명령어를 입력한다면 우선적으로 Mlist라는 독특한 연결리스트가 dictionary / regualar 파일을 관리한다. 특히 백업 경로에 있는 파일들 같은 경우에는 Mlist 의 same\_next 포인터를 활용하여 같은 파일로 묶은 구조를 사용하였다. 이것은  “- ssu\_backup에 내장된 명령어들은 링크드 리스트로 파일 리스트를 관리해야 함” 이라는 명세에 부합하기 위해 체택되었다. 그리고 단순 연결리스트구조를 넘어 체이닝 연결리스트 구조 형태로 제작되었다. 예를들어, ssu\_manage.c\_230327151003, ssu\_manage.c\_230327151326 같이 파일 명이 동일한 파일은 같은 연결리스트 선상에 연결될 수 있도록 또 다른 포인터인 same\_next 포인터를 두어 구현하여 더욱 진화한 형태로 모든 파일을 **연결리스트로 구성하기 위해 제작하였다.**  add, recover 에서 생성되는 파일들 역시 Mlist 에서 갱신되도록 설정되어있다. 이 Mlist는 내부 함수 print\_mlist() 호출을 통해 mlist 의 내부 모습을 다음과 같이 볼 수 있다.  ssu\_backup 메커니즘은 파일리스트를 전체적으로 관리하는 Mlist와 rlist (오리지널 경로의 파일리스트로 구현되어있다.  그리고 추가적으로 **1GB** 가 넘는 파일에 관해서는 hashing을 구하는 과정이 너무 지나치게 많이 소요되어 Pass 하도록 설계되어있는데, 이것을 변경하고 싶으면 MAX\_FILE\_SIZE 값을 변경함으로써 해결할 수 있다.    [Mlist 가 파일리스트를 관리하는 모습] |

## ***ssu\_backup 기능 구현***

### <기본 명세 구현>

|  |
| --- |
| * ACTUAL\_PATH : 입력받는 파일을 기준으로 /home/<username> 을 알아내는 전역변수 * BACKUP\_PATH : getenv(“HOME”) 을 사용하여 $HOME 을 기준으로 현재 자신의 $HOME 경로를 파악후 백업 디렉토리 $HOME/backup 디렉토리 생성 * **만약 $HOME 경로가 /root 인 경우 pwd 를 통해 현재 자신 경로를 기준으로 backup 경로 생성.** * BACKUP\_PATH 를 바탕으로 mkdir()를 호출하여 **백업디렉토리 생성** * 해당 프로그램은 단일 프로세스를 이용하며 멀티스레드를 이용하지않음 * 프로그램의 파일리스트 관리는 단순연결리스트 rlist와 포인터를 두개 가지고 있어 백업 경로의 파일 중 a.file\_<time> 중 time만 다른 파일은 same\_next 포인터에 연결되도록 하는 완전 연결리스트 체이닝 기법을 사용함. * 본 프로그램은 system() 호출이 없음. 대신 ls, vim같이 필요한 경우에는 exec() 를 사용. * **상대경로 . , ~를 지원함, 단 상대경로 . 는 현재 파일이 위차한 디렉토리를 기준으로 설정됨, 그리고 ~는 새로생성된 아이디 /user/<생성된 아아디>를 $HOME으로 간주하여 해당경로 생성됨** * **(중요), add, recover, remove 등에서 인자가 2개만 들어온 경우, -lcrypto 와 같이 -로 시작하는 파일등이 존재하기 때문에, 명령어인지 파일명인지 판단이 힘듦. 그래서 명령어 바로 뒤의 첫 번째 인자는 옵션이 아닌 <FILENAME> 으로 무조건 인식하게 설정함. 만약 해당 경로에 -가 들어간 FILENAME 이 없으면 자동으로 예외처리 해주도록 설정** |

### < ssu\_backup 명세 구현>

|  |
| --- |
| * 프롬포트 20190511> 구현 * BACKUP\_PATH 를 바탕으로 mkdir()를 호출하여 **백업디렉토리 생성** (ssu\_add.c, ssu\_remove.c, ssu\_recover.c 생성 시에도 생성 가능) * ssu 같은 정의되지 않은 명령어 입력 시 ssu\_help.c 호출 구현 * 만약 해당경로에 디렉토리가 있으면 생성되지 않겠지만, 해당 경로에 백업파일이랑 이름이 동일한데 일반 파일같이 디렉토리가 아닌 다른 파일이 존재하면 즉시 프로그램 종료,       [/home/junhyeongBKS/backup 파일이 일반파일로 되어있음]   * help 입력 시 help.c 파일 호출 * exit 를 호출하여 종료될 땐, ssu\_add.c, ssu\_remove.c, ssu\_recover.c, 의 실행파일들이 모두 삭제되고 해당 명령어들이 fork 되고 실행되기 직전에 fork()+exec() 를 이용하여 컴파일하여 실행파일 생성   + 해당 명령어들이 최신상태를 유지할 수 있도록 하는 일종의 테크닉 |

### < ssu\_backup 예외처리 구현>

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | * 첫 번째 인자가 없거나 md5/sh1 이 아닌 경우 Usage 출력 후 종료  |  |  | | --- | --- | | 예시 |  |  * 프롬포트 상에서 엔터만 입력 시 프롬포트 재출력 구현 * 프롬포트 상에서 저장한 명령어 외 기타 명령어 입력 시 help 출력 후 prompt 재출력 구현  |  |  | | --- | --- | | 예시 |  |  * ssu\_add, ssu\_recover, ssu\_remove, ssu\_help 실행 파일 없을 시 해당 파일 실행 전 gcc로 컴파일 하여 실행파일 생성 구현 * exit 종료 시 자동으로 ssu\_add, ssu\_recover, ssu\_remove, ssu\_help 삭제 구현 * <filename> 가 **/home/<사용자 아이디>/backup** 경로 혹은 /home 단독으로 파일 이름이 오면 예외처리하였음 | |

## ***add 기능 구현***

### <add 명세 구현>

|  |  |
| --- | --- |
| |  | | --- | | * add만 입력 시 add Usage 출력 구현 * add 상대경로 단일파일 구현 * add 절대경로 단일파일 구현 * add -d 옵션 없이 <FILENAME> 이 디렉토리인 경우 에러처리 구현 * 경로가 백업디렉토리를 포함할 경우 예외처리 구현 * add <FILENAME: 디렉토리> -d 구현 * add <FILENAME: 레귤러파일> -d 인 경우 d옵션이 없는것 처럼 구현 * 백업파일의 해시값이 같은 경우 백업 진행 X 구현 * add 백업 시 현재시간을 백업시간을 붙여 백업하는 기능 구현 * add 뒷 이상한 옵션 에러처리 구현 * <filename> 가 **/home/<사용자 아이디>/backup** 경로 혹은 /home 단독으로 파일 이름이 오면 예외처리하였음 | |

### <add 예외처리 구현>

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | * add 첫 인자 입력이 없거나 올바르지 않은 경로 입력 시 Usage 출력 후 프롬포트 재출력  |  |  | | --- | --- | | 예시 | >> -d 가 파일명으로 인식하는 것은 -d가 실제로 <FILENAME>으로도 해석되어질 수 있어 -가 붙는다고 옵션이라고 간주하지 않도록 설계 |  * 첫 번째 인자로 입력받은 경로(절대경로) 가 길이 제한(4,096 Byte)을  넘거나 해당 경로가 일반 파일이나 디렉토리가 아닌 경우 에러처리 후 프롬포트 재출력   + 해당 access 나 file\_size\_check 함수 등에서 ssu\_backup 단계에서 미리 에러처리 후 프롬포트 재출력 * 첫번째 인자로 받은 경로가 백업 디렉토리를 벗어난 경우 + 백업디렉토리및 하부파일을 포함한 경우 에러처리  |  |  | | --- | --- | | 예시 |  |  * 두 번째 인자로 올바르지 않은 옵션이 들어왔을 경우 에러처리 구현  |  |  | | --- | --- | | 예시 |  |  * 첫 번째 인자가 디렉토리일 경우 -d 옵션을 처리하지 않았다면 에러처리 후 재출력 구현 * 백업파일의 기존 경로가 같고 해시값이 같은 백업파일이 이미 있다면 $(백업경로) is already backuped 출력 후 종료 구현  |  |  | | --- | --- | | 예시 |  | | |

### <add 추가 구현>

|  |  |
| --- | --- |
| |  | | --- | | * add 실행파일이 실행될 시 생성되었다가 exit 시 삭제 * 환경변수 ~ 를 사용한 경로로 <FILENAME> 이 입력으로 올 시 /home/사용자이름 으로 변환해주는 알고리즘 적용 | |

## ***remove 기능 구현***

### <remove 명세 구현>

|  |
| --- |
| * remove 만 홀로 들어간 경우Usage 출력 후 종료 구현 * 동일한 파일이 2개 있는 경우 -> 프롬포트 해시값 입력받아 삭제기능 구현 (결과출력에서 확인가능) * remove에서 백업경로에는 존재하지만 원래경로에는 존재하지 않는 경우 동일하게 삭제 기능 구현 * remove -a 옵션 시 디렉토리가 <FILENAME>으로 오면, 일치하는 모든 파일 하위 디렉토리 삭제 기능 추가 * remove -a 옵션 시 일반 파일이 <FILENAME>으로 오면 -a 옵션 없는 것 처럼 구현 * remove -c 옵션 시 백업 경로의 모든 파일/디렉토리 삭제 backup 경로에 디렉토리나 레귤러 파일이 하나라도 존재하면 backup directory cleared(3 regular files and 3 subdirectories totally). 출력 후 종료시키고, 하나도 존재하지 않는 경우 no file(s) in the backup 출력후 재프롬포트 구현 |

|  |
| --- |
|  |

### <remove 예외처리 구현>

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| * 첫 번쨰 인자 입력이 없거나 해당 백업파일이나 디렉토리 없을 경우 Usage 출력 후 프롬포트 재출력 구현  |  |  | | --- | --- | | 예시 |  |  * 첫 번째 인자로 입력받은 경로(절대경로) 가 길이 제한(4,096 Byte)을  넘거나 해당 경로가 일반 파일이나 디렉토리가 아닌 경우 에러처리 후 프롬포트 재출력   -> 해당 access 나 file\_size\_check 함수 등에서 에러처리 후 프롬포트 재출력   * 첫번째 인자로 받은 경로가 백업 디렉토리를 벗어난 경우 + 백업디렉토리및 하부파일을 포함한 경우 에러처리  |  |  | | --- | --- | | 예시 |  |  * 두번째 인자가 디렉토리인데 -a 옵션을 사용하지 않았다면 예외처리 후 프롬포트 재출력  |  |  | | --- | --- | | 예시 |  |  * remove -a -c 혼용 사용 시 에러처리 구현  |  |  | | --- | --- | | 예시 |  |  * remove -c 옵션 사용 시 다른 인자를 입력하면 Usage 출력 후 프롬포트 재출력  |  |  | | --- | --- | | 예시 |  |  * remove -c 옵션 사용 시 백업 디렉토리에 아무파일도 없을 경우 no file(s) in the backup 출력 후 프롬포트재출력   -> 뒤의 실행사진에 존재   |  |  | | --- | --- | | 예시 |  | |

|  |
| --- |
|  |

### <remove 추가 구현>

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | * 파일 크기가 너무 크면 해시값을 구하는데 시간이 너무 많이 걸려 파일 사이즈 제한을 걸어두었다. (1GB 제한)  |  |  | | --- | --- | | 예시 |  | | |

## ***recover 기능 구현***

### <recover 명세 구현>

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | * recover <FILENAME> 없이 용 시 Usage 출력 후 프롬포트 재출력  |  |  | | --- | --- | | 예시 |  |  * recover 상대경로 입력 시 구현   + 파일 사이즈를 3글자마다 , 붙여서 출력  |  |  | | --- | --- | | 예시 |  |  * 백업경로에만 있는 파일 복구  |  |  | | --- | --- | | 예시 |  |  * <filename> 가 **/home/<사용자 아이디>/backup** 경로 혹은 /home 단독으로 파일 이름이 오면 예외처리하였음 * recover -d 옵션 사용시 (백업경로에 2개있는경우) 구현   + 파일 사이즈를 3글자마다 , 붙여서 출력  |  |  | | --- | --- | | 예시 |  |  * recover -n <NEWNAME> 옵션 뒤에 <NEWNAME> 없는 경우 USAGE 출력 후 프롬포트 재출력  |  |  | | --- | --- | | 예시 |  |  * recover -n <NEWNAME> 단일 옵션 구현  |  |  | | --- | --- | | 예시 |  |  * recover -n 옵션 시 여러개의 파일이 있는경우 -> 프롬포트에 무엇을 복구할 것인지 입력받기 구현  |  |  | | --- | --- | | 예시 |  | | |

### <recover 예외처리 구현>

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| * 첫 번째 인자 입력 시 인자가 없거나 올바르지 않은 인자가 들어온 경우 Usage 입력 후 프롬포트 재출력  |  |  | | --- | --- | | 예시 |  |  * 첫 번째 인자로 입력받은 경로(절대경로) 가 길이 제한(4,096 Byte)을  넘거나 해당 경로가 일반 파일이나 디렉토리가 아닌 경우 에러처리 후 프롬포트 재출력   + 해당 access 나 file\_size\_check 함수 등에서 에러처리 후 프롬포트 재출력 * NEWFILE 이 home디렉토리를 벗어나거나 백업디렉토리를 포함하는 경우 can't be backuped 출력 후 프롬포트 재출력  |  |  | | --- | --- | | 예시 |  |  * 첫 번째 인자가 디렉토리일 경우 -d 옵션을 사용하지 않은 경우 에러처리 후 프롬포트 재출력  |  |  | | --- | --- | | 예시 |  |  * 두 번째 인자로 올바르지 않은 옵션이 들어왔을 경우 Usage 출력 후 프롬포트 재출력  |  |  | | --- | --- | | 예시 |  |  * 백업파일의 기존 경로가 <FILENAME>과 같고 해당 파일의 해시값이 같다면 백업복구를 진행하지 않음.  |  |  | | --- | --- | | 예시 |  |      * 기존 경로 파일이 없다면 백업파일 상대로 생성 : 3번 예시와 동일 |

### <recover 추가 구현>

|  |
| --- |
| * <NEWNAME> 에 들어갈 경로에 환경변수 ~ 가 들어간다면 /home/<파일명 기준> 으로 처리하였음. |

## ***ls 기능 구현***

### <명세 구현>

|  |  |  |
| --- | --- | --- |
| * ls 뒤에 인자값 넘겨주는 방식 구현  |  |  | | --- | --- | | 예시 |  |  * vi/vim, ls 에 인자를 넘겨 실행하는 방식 구현 |

## ***vim 기능 구현***

### <명세 구현>

|  |  |  |
| --- | --- | --- |
| * Vim 명령어에 사용되는 추가적인 옵션이나 경로처리 등을 구현하였음.   + Ex> vim ~/test.txt 등과 같이 환경변수를 사용하여도 vim 명령어 사용이 가능하다.  |  |  | | --- | --- | | 예시 |  | |

## ***help 기능 구현***

### <명세 구현>

|  |  |  |
| --- | --- | --- |
| * Help 나 다른 명령어 혹은 알 수 없는 명령어나 인자전달 시 출력되는 프로그램이다.  |  |  | | --- | --- | | 예시 |  | |

## Makefile 작성

### <작성방법>

|  |
| --- |
| * makefile의 경우, linux 커널 환경에서 openssl 의 libcrypto.a 의 경로를 찾지못하여 find 오류가 발생하는 문제점이 존재하였다. 그래서 /usr 디렉토리에서 동적 라이브러리를 직접 link하여 compile중 발생하는 경로 문제를 해결하려고 시도하였다. 현재 리눅스 환경에서 openssl 패키지의 경로는 **/usr/local/openssl/lib64** 경로에 libcrypto.a이 있으므로 makefile의 LDFLAG를 다음과 같이 설정하였다. LDFLAGS= -L/usr/local/openssl/lib64 -lcrypto  그리고 실제로 openssl 함수들을 사용하는 파일은 3개로 ssu\_add.c, ssu\_remove.c, ssu\_recover.c 와 동적링킹하여 makefile를 구성하였다 |

# 3. 프로그램 내부 파일들의 관계와 역할.

## ***용어 정리***

|  |
| --- |
| 본 내용에 들어가기 전,  해당 소스코드는 다음과 같이 분할 되어있다. ssu\_backup.c, ssu\_add.c, ssu\_remove.c, ssu\_recover.c, ssu\_exec.h, ssu\_shared.h, ssu\_help.c 로 분할 되어있다.  다음과 같이 이해와 설명을 쉽게하기 위해 용어를 아래와같이 정의하였다.  ※ **컨테이너 파일**: 다른 파일들을 fork 하여 execute를 해줄 수 있고 첫 실행단계에서 발생될 수 있는 어느 정도의 에러들을 걸러주는 파일이다. 컨테이너 파일들은 여러 개의 캡슐파일들을 실행해주는 디렉터와 같은 역할을 해준다. 캡슐파일들을 여러 개 포함하고 각 캡슐들의 내용물에 직접 접근하지 않기 때문에 컨테이너 파일이라고 정의하였다.  ※ **캡슐 파일**: 각 헤더에 정의된 함수들을 사용하여 옵션에 맞는 기능들을 수행해주는 파일이다. 마치 알약의 캡슐에서 껍데기와 같아서 캡슐파일이라고 정의하였다. 캡슐파일은 헤더파일 (내용물)을 사용하며 작동한다. |

## ***ssu\_backup 파일의 역할***

|  |
| --- |
| ssu\_backup.c 파일은 **컨테이너 파일**이다.   * ssu\_backup 실행 시 해시 값이 md5, sha1를 제대로 입력하였는지 체크해준다. * 잘못된 명령어가 입력되었을 때 help를 띄우고 다시입력 받게 해준다. * 프로그램 종료기능이 있다. (exit) * prompt (20190511>) 을 띄워준다. * add, remove, recover, ls, vi, vim 등을 지원해준다. 기본적으로 fork 되어  새 프로세스로 인식된다. * prompt 뒤에 항상 붙는 <FILENAME> 을 상대경로/절대경로를  모두 절대경로로 바꿔준다.  (환경변수 ~ 포함 + 그리고 길이도 체크해준다.) * BACKUP 경로를 생성한다. |

## ***ssu\_add 파일의 역할***

|  |
| --- |
| * ssu\_add : 내부 add연산을 수행해주는 캡슐파일이다. * add 뒤에 오는 옵션 ("", "-d") 를 처리해주는 기능이 있다. * 해당 <FILENAME> 이 실제로 접근가능한 경로인지 미리 판단을 해준다. * -d 옵션사용 시 please wait 를 출력한다. (파일이 많다면 시간이 오래 걸릴 수 있기 때문) * 옵션 이외의 내용을 사용할 경우 에러 메시지를 출력하고 종료한다. * 파일 크기가 ssu\_exec.h 에 정의된 MAX\_FILE\_SIZE (100MB) 를 넘지 않도록 한다.  (너무 큰 파일은 해싱과정이 너무 오래걸리기 때문.) 이는 사용자에 의해 변경할 수 있다. |

## ***ssu\_recover 파일의 역할***

|  |
| --- |
| ssu\_recover: 내부 recover연산을 수행해주는 **캡슐파일**이다.   * recover 뒤에 오는 옵션("", "-d", "-n") 를 처리해주는 기능이 있다. * -d 옵션사용 시 please wait 를 출력한다. (파일이 많다면 시간이 오래 걸릴 수 있기 때문)   옵션 이외의 내용을 사용할 경우 에러 메시지를 출력하고 종료한다. |

|  |
| --- |
|  |

## ***ssu\_remove 파일의 역할***

|  |
| --- |
| ssu\_remove: 내부 ssu\_remove연산을 수행해주는 **캡슐파일**이다.   * remove 뒤에 오는 옵션("", "-d", "-c") 를 처리해주는 기능이 있다. * 해당 <FILENAME> 이 실제로 접근가능한 경로인지 미리 판단을 해준다. * -a, -c 옵션사용 시 please wait 를 출력한다. (파일이 많다면 시간이 오래 걸릴 수 있기 때문) * -a, -c 옵션이 동시에 들어오면 에러메시지를 출력하고 종료한다 * 옵션 이외의 내용을 사용할 경우 에러 메시지를 출력하고 종료한다. |

|  |
| --- |
|  |

## ***ssu\_help 파일의 역할***

|  |
| --- |
| ssu\_help: help 메시지를 출력해주는 캡슐파일이다. |

|  |
| --- |
|  |

## ***ssu\_exec 파일의 역할***

|  |
| --- |
| ssu\_exec.h: add, recover, remove 파일을 실행할 때 필요한 함수들의 선언과 구현이 담긴 파일이다.   * ssu\_add, ssu\_recover, ssu\_remove 에서 사용될 함수들을 모아둔 헤더파일이다. * 해당 파일에는 많은 함수들이 구현되어있고 프로그램 구현 방식에서 자세히 설명할 예정이다. |

## **ssu\_shared.h 파일의 역할**

|  |
| --- |
| 기본적으로 컨테이너 파일인 ssu\_backup.c 와 다른 명령어들은 안정성을 위해 ssu\_exec.h헤더파일을 공유하지 않도록 설계하고 싶었으나, 프롬포트로 명령을 받은 <filename> 을 절대경로화+길이체크를 해주는 file\_size\_check 함수나, ACTUAL\_PATH, BACKUP\_PATH의 경로 지정 등을 함수를 다시 만들어서 제작하는 것은 비효율적이라 판단하여 ssu\_exec.h에서 사용될 전역변수, 등과 같이 필수적으로 ssu\_backup.c과 공유하는 파일들을 가지고 있다. |

|  |
| --- |
|  |

## ***프로그램 관계도***

|  |
| --- |
|  |

|  |
| --- |
|  |

# [ssu\_backup 파일들이 관리되는 방식과 구현 기본 원리 (프레임)](#_ssu_exec.h:__add,)

## ***파일 구조체 : Filenode, Mnode***

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| * Filenode: 프로그램 실행 중 탐색되는 파일들의 정보들은 filenode 라는 구조체에 저장된다.  file node 구조체는 다음과 같이 구성되어 있다.   typedef struct filenode {  // 초기화하면서 설정되는 값      char path\_name [MAXPATHLEN];     // 파일 경로      char file\_name [MAXFILELEN];     // 순수 파일 이름      char actual\_path [MAXPATHLEN];   // 백업파일 / 원래경로를 제외한 실질적 경로.      char inverse\_path [MAXPATHLEN];  // 백업경로 ↔ 원래경로 상대적인 경로      char hash[HASH\_LEN];           // 해당 파일 내용의 해시값 저장 (해시길이40:md5보다긴 sha기준)      struct stat file\_stat;         // 파일의 stat 저장 (바이트 크기 등에 사용.)  char back\_up\_time [TIME\_TYPE]; // 백업시간 저장    //나중에 설정할 값.      struct filenode\* next;   // 연결리스트 사용을 위한 다음 구조체 포인터 저장.  }Filenode;  Filenode 구조체는 Filenode\* new\_filenodes (char\* filename, int opt, int f\_opt); 라는 함수에 의해 초기화될 수 있다. Filenode 를 객체로 본다면 마치 생성자와 같은 역할을 한다. new\_filenodes 는 filenode 의 모든 멤버변수를 NULL로 초기화시켜주는 new\_filenode() 를 내부적으로 선언하면서 시작된다.  new\_filenodes의 파라미터는 아래와 같다   |  |  |  | | --- | --- | --- | |  | **0** | **1** | | **opt** | filename이 백업을 시킬 경로인 경우 | filename이 백업 경로인 경우 | | **f\_opt** | MD5 해시 옵션 | SHA-1 해시 옵션 |     예시를 들자면 아래와 같다.  (\* 백업경로에 있는 KMP.c 와 원래 경로에 있는 KMP.c 는 같은 파일이다.  하지만, 해시값이 다른데 이는 f\_opt를 다르게 호출한 모습을 보여준다. MD5, SHA-1 해시값의 차이)   |  | | --- | | ex> Filenode\* filename = new\_filenodes("/home/junhyeong/go2/KMP.c", 0, 1);   (\*: f\_opt==0 이므로 MD5 해싱으로 변수를 할당하였다.)  해당 filename 노드의 성분들을 출력해보면 아래와 같다.  path name is /home/junhyeong/go2/KMP.c  file name is KMP.c  actual path is /go2/KMP.c  inverse path is /home/junhyeong/backup/go2/KMP.c\_230311151809  hash is 52df55bb942f7e134dc39f3b08a9541edeb53cc9  /home/junhyeong/go2/KMP.c is Regular file    ex> Filenode\* filename = new\_filenodes("/home/junhyeong/backup/go2/KMP.c\_230311112507", 1, 0); (\*: f\_opt==1 이므로 SHA-1 해싱으로 변수를 할당하였다.)  path name is /home/junhyeong/backup/go2/KMP.c\_230311112507  file name is KMP.c  actual path is /go2/KMP.c\_230311112507  inverse path is /home/junhyeong/go2/KMP.c  hash is 90704eb4eb85366c535fddbf97f44267  /home/junhyeong/backup/go2/KMP.c is Regular file   * **Mnode 는 mlist에서 사용되는 파일구조체(managenode)로 filenode를 멤버변수로 가지지만, same\_next 포인터와 이와관련 포인터를 추가한 좀 더 무거운 구조체이다.**   [Mnode 구조 그림]  typedef struct managenode {      Filenode\* filenode;      //filenode의 next와 동일함 (다른 파일들을 연결할 포인터).      struct managenode\* next;  // 같은 파일 , 다른 백업시간의 파일들을 연결하는 포인터.      struct managenode\* same\_next;  // same\_next에 연결된 노드 중 가장 마지막 노드, 삽입시간을 O(1) 로 하기위함. 없으면 NULL      struct managenode\* same\_next\_tail;  // 자기 자신 노드를 포함하고 same\_next에 연결된 파일 개수  int same\_cnt;    }Mnode;   * 예를들어 파일\_<time> 에서 time부분만 다른 파일의경우에는 same\_next에 연결되며 기본적으로 큐 형태로 구현된다 (탐색을 용이하게 하기위함) 그렇기 때문에 O(1) 시간만에 same\_next에 연결될 수 있게 하기 위해 Mnode의 가장 첫 노드에 가장 마지막 노드의 정보인 same\_next\_tail 을 가지게된다. * 본의아니게, mlist의 tail과 Mnode의 tail은 반대적인 개념이다. | |

## ***파일관리 리스트 : Mlist, Rlist***

|  |
| --- |
| Filenode 구조체를 연결하는 연결리스트는manage list, r리스트, m리스트 이렇게 2가지 형태가 존재한다.  가장 먼저, mlist는 체이닝 연결리스트를 특화시킨 형태의 연결리스트로써, mappend()를 통해 전체적인 파일리스트를 연결리스트를 관리하는 리스트이다. 그리고rlist는 단순 연결리스트 구조체로서, rappend() 하여 단순하게 연결리스트에 연결할 수 있도록 설계되었다. 예를들어, 원래경로 ~/go2/file.c 가 있을 때 백업경로에는 ~/backup/go2/file\_<time>.c, ~/backup/go2/file\_<time2>.c, ~/backup/go2/file\_<time3>.c .. 와 같이 같은 파일이지만 여러개의 백업시간을 가진 파일들이 존재할 수 있다. 이를 그냥 rlist로 연결하게 된다면 remove나 recover 를 처리할 때, 하나의 파일에 대해 O(n) 번 탐색해야되는 시간낭비가 발생하게된다 이를 줄이기 위해서 고안해낸 방식이 Chaning 기법을 착안하여 제작한 체이닝 연결리스트 mlist (Chaining LinkedList) 를 설계하였다.  > 즉, ~/backup/go2/file\_<time>.c 과같은 node들을 효율적으로 정리하기 위해서 다음과 같은 규칙을 적용하였다.  1. 디렉토리와 파일 부분을 stat() 함수 , stat 구조체의 st\_mode 를 이용하여 딕셔너리/파일로 분류한 뒤 file\_tail, dir\_tail 를 시작으로 연결리스트로 연결한다. .  2. file 리스트에서 위와 같이 동일한 파일들이 여러 개 존재한다면 해당 파일은 Mnode\* 의 멤버변수의 next 를 이용하여 연결해주는 체이닝(Chaining) 을 한다.  3. file\_array에 저장된 파일들의 개수는 file\_cnt\_table로 관리하며, 아래의 flist 의 멤버변수를 이용하여 관리한다.  4. 해당 과정은 append() 함수를 호출함으로서 자동화시킨다.  대표적으로 mlist구조체의 내부 모습은 다음과 같다.  }Mnode;  typedef struct mlist{      Mnode\* dir\_head; // 딕셔너리 연결리스트를 바로 연결하기 위한 구조. O(1)      Mnode\* file\_head; // 파일 연결리스트를 바로 연결하기 위한 구조. O(1)      Mnode\* dir\_tail;      // 딕셔너리 리스트의 가장 시작지점 노드      Mnode\* file\_tail; // 파일리스트의 가장 시작지점 노드      int file\_cnt;                       // 파일 총 개수      int dir\_cnt;                        // 디렉토리 총 개수      //★file\_rear\_table, file\_cnt\_table 는 max\_file\_cnt를 따라감.  }Mlist;  내용을 정리하면, ssu\_backup 에서의 파일은 2가지 구조체 (연결리스트기반) 로 관리된다.    기본적으로 원래 경로에 있는 파일들을 rlist를 통해 단순하게 경로를 뽑아온 뒤 append\_samefile() 함수를 호출함으로써 상대적인 경로에 있는 (backup 경로) 딕셔너리/파일(같은파일) 를 가져오게 해주는 리스트가 mlist이다.  모든 명령어는 rlist와 mlist가 서로 상호보완적으로 존재하며 전체적인 파일리스트 관리는 mlist가 더 자세하게 다루고있다.  <해당 방식을 채택함으로서의 장점>  해당 방식을 사용함으로써 Rlist에서 ssu\_processor 의 같은 이름을 가지는 백업경로의 ssu\_processor.c 를 한꺼번에 찾을 수 있게되었고 이는 나중에 같은 이름의 파일을 가지지만 백업 시간이 다른 파일들을 선택하여 복구시키거나 삭제시키는 remove, recover 과정에서 편하게 활용될 수 있었다.  특히 세부적으로, mlist는 디렉토리파일을 분리시키는 테크닉을 통해 recover의 -n 인자가 들어왔을 경우 새로운 디렉토리를 생성해줘서 그 안으로 파일을 넣어주는 과정에서 쉽게 디렉토리 경로들을 확인할 수 있게 해준다는 장점 또한 존재하였다. 그리고 remove 안에서 -c 옵션을 넣는 경우 모든 디렉토리 + 세부 파일들을 삭제해줘야하는데 이때 file\_array 테이블의 요소들을 먼저 삭제하고, dir\_array 요소들을 삭제해주면 파일 삭제 과정이 간단한 작업으로 끝낼 수 있도록 해주는 일등공신같은 연결리스트 체인 형태의 구조가 되었다. |

# 각 파일들의 진행구조와 사용된 함수들의 프로토타입 및 로직 (Algorithm)

## ***ssu\_backup***

### ssu\_backup 내부 구현 로직

|  |
| --- |
| ssu\_backup.c  1. ssu\_backup 파일은 우선 들어온 argv 값에 MD5 와 SHA-1 가 있는지 확인한다. 없다면 Usage 출력 후 프로그램을 종료한다.  2. 그 다음 20190511> 이라는 프롬포트를 출력하고 해당 프롬포트 값의 첫 번째 인자가 옵션이 존재하는지 체크를 한다.  3. 백업 파일 $HOME/backup 경로에 디렉토리가 있는지 확인 후 없으면 생성한다.  4. 입력받은 옵션이 존재하지 않는다면 help함수를 호출하고 다시프롬포트를 재출력해준다.  5. 해당 프롬포트에 add, recover, remove 가 들어온다면 ssu\_add, ssu\_recover, ssu\_remove 의 실행파일이 존재하는지 물어본다.  6. 실행파일이 존재하지 않는다면 fork()->exec() 를 통해 /usr/bin/gcc 로 컴파일을 해주어 해당 옵션의 실행파일을 만들어주고 해당 옵션을 실행시킨다.  7. 실행을 종료하고 다시 프롬포트로 돌아온다. 이 과정을 계속 반복한다.  8. 모든 실행이 종료된 후, ssu\_remove, ssu\_recover, ssu\_add, ssu\_help 의 실행파일을 지워준다.  이는 후에 해당 파일들이 업데이트되었거나, 실행파일이 없어 exec 오류를 발생시키는 것을 방직하기 위함이다.  ★ 중요한 점은 exec() 를 실행하여 add, remove, recover 를 실행할 때 argument의 마지막 인덱스값에는 해시 종류를 붙여서 출력해준다. 예를들어, md5를 기반으로 "add . -d" 라는 prompt 값을 ssu\_add에 넘겨준다고 가정한다면 ssu\_backup에서 내부적으로 가공/처리되어 "/home/junhyeong/go2/ssu\_add /home/junhyeong go2 -d md5" 라고 넘겨주게 된다.    [ssu\_backup.c 의 순서도] |

### ssu\_backup 내부 함수 프로토타입 및 로직(Alogrithm)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [함수와 매크로 설명]   * PROMPT 는 Prompt를 띄워주는 매크로다. (20190511> 을 띄워주는 매크로다) * MAXPATHLEN 과 MAXPROMPT LEN은 각각 input 받은 파일경로의 최대길이, 프롬포트 라인 최대 글자수를 의미한다.   #define PROMPT  printf("20190511> ");  #define MAXPATHLEN      256  #define MAXPROMPTLEN    1024    void main\_help(); //20190511> help 를 입력했을 때 발생되는 메시지 출력함수  char\*\* prompt\_token(char\* char\_prompt, int\* prompt\_argc, char\* hash);   * return 값은 fork 후 exec 함수에서 사용될 argv 들이다. * 20190511> "입력된 문자열" 중 입력된 문자열을 ' ' 개행을 기준으로 분리시켜 exec() 함수 argv로 넘겨줄 수 있는 형태로 만드는 함수.  |  |  | | --- | --- | | 작동 방식 | 1. 프롬포트에 그냥 개행처리된 (\n) 경우는 바로 리턴한다.  2. " "띄워쓰기 단위로 토큰들을 분리시켜준다.  3. 분리된 토큰 중 가장 첫 토큰이 add, remove, remove 중 하나라면 절대경로로 변환하여 ssu\_???? 형태로 바꿔준다.  ex> 현재 위치가 /home/junhyeong/go2 고 첫 번째 인자가 add 가 왔다면 -> /home/junhyeong/go2/ssu\_add 로 변환  4. remove에서는 hash값을 추가할 필요가 없으므로 remove를 제외한 나머값들은 hash 방식 (md5/sha) 를 argv의 마지막 인자로 넘겨준다. |   int access\_check(char\* file\_name);   * file\_name (상대/절대) 경로의 접근 가능 여부를 확인하는 함수 * file\_name 에 /가 붙어있는지 확인한 후 절대경로로 변환한 후 파일 길이가 벗어나는지 체크.   void clear\_func();   * ssu\_help/add/recover/remove 실행파일을 싹 다 지워주는 함수이다. * 각 파일 내용이 업데이트 되거나 파일이 실행파일이 불어나 더러워보여서 삭제.  |  |  | | --- | --- | | 작동방식 | 1. ssu\_help/add/recover/remove 를 realpath 를 통해 실행파일 위치를 얻어낸다.  2. 해당 파일들이 존재하는지 확인한다.  3. 존재한다면 remove 함수를 호출하여 실행파일을 싹 다 지운다. | |

## ssu\_add

### ssu\_add 내부 구현 로직

|  |
| --- |
| ssu\_add.c   * ssu\_add 파일은 캡슐파일으로써 argument 개수에 대한 에러처리 및 ssu\_exec.h 에 정의된 함수들을 세부 옵션 (-d, "") 를 처리하여 알맞게 실행시켜주는 파일이다. 해당 파일은 우선적으로 getopt() 를 이용하여 ssu\_backup으로부터 받은 인자들을 처리해준다. 처리 옵션이 존재하지 않는 옵션이라면 help 함수를 호출하고 실행을 종료한다. 처리 중간에 만약 Filename이 Backup경로와 동일하다면 예외처리 후 실행을 종료한다.     [ssu\_add.c 의 순서도] |

## ***ssu\_recover***

### ssu\_recover 내부 구현 로직

|  |
| --- |
| ssu\_recover.c  ssu\_recover 파일은 **캡슐파일**으로써 argument 개수에 대한 에러처리 및 ssu\_exec.h 에 정의된 함수들을 세부 옵션 (-d, -n, "") 를 처리하여 알맞게 실행시켜주는 파일이다. 해당 파일은 우선적으로 getopt() 를 이용하여 ssu\_backup으로부터 받은 인자들을 처리해준다. 처리 옵션이 존재하지 않는 옵션이라면 help 함수를 호출하고 실행을 종료한다. getopt 함수 내부에서 <NEWNAME> 에 해당하는 부분을 상대경로/절대경로 에서 절대경로로 바꿔주는 작업을 해준다. 이후 각 플래그에 맞는 인자수가 맞는지 확인 후 ssu\_recover 함수를 실해아여 recover를 진행시켜준다. ssu\_recover 함수가 내부적으로 실행되는 과정은 아래에 자세하게 서술하겠다.    [ssu\_recover.c 의 순서도] |

## ***ssu\_remove***

### ssu\_remove 내부 구현 로직

|  |
| --- |
| * ssu\_remove 파일은 캡슐파일으로써 argument 개수에 대한 에러처리 및 ssu\_exec.h 에 정의된 함수들을 세부 옵션 (-a, -c, "") 를 처리하여 알맞게 실행시켜주는 파일이다. 해당 파일은 우선적으로 getopt() 를 이용하여 ssu\_backup으로부터 받은 인자들을 처리해준다. 만약 <FILENAME> 이 해당 경로에 파일이 없다면 에러메시지를 출력하고 실행을 종료한다. * 그리고 -a 옵션과 -c 옵션이 동시에 들어온 경우에도 help() 를 출력해주고 프로그램 실행을 종료한다. ssu\_remove() 와 ssu\_remove\_all() 함수의 실행모습은 아래에 자세하게 서술하였다.     [ssu\_remove.c 의 순서도] |

## **ssu\_shared.h : 공유되는 함수/변수 선언/구현**

### <ssu\_shared 함수 명세개요>

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| #ssu\_shared.h 헤더 설명   * 기본적으로 공유되는 ACTUAL\_PATH, BACKUP\_PATH 등을 공유하고 , userad, /etc/passwd 를 보고 판단하는과정은 ssu\_backup에서만 동작하거나 ssu\_exec가 따로 동작한다면 큰 문제가 발생할 것이라는 판단하에 공유헤더를 만들어 동일한 메모리 접근 내지 동일한 동작을 하도록 설계되어있다. * 그리고 ssu\_exec.h 에서 사용되는 매크로를 담아뒀다. * [ 1. 매크로 명세]   // MAXPATHLEN, MAXFILELEN, MAXPROMPTLEN 들은 파일최대길이, 실제 절대경로 파일 최대길이, 프롬포트 길이 등을 의미한다.  #define MAXPATHLEN          4097  #define MAXFILELEN          256  #define MAXPROMPTLEN        1024  // 백업경로에 있는 파일을 다룰 때 YYMMDDHHMMSS 를 설정할 때 문자열에 사용될 매크로이다.  #define TIME\_TYPE           20  // 파일을 복사하거나 해싱할 떄 버퍼크기를 다음과 같이 설정하였다  (16KB 크기의 버퍼를 기준으로 가져오도록 설정했다)  #define BUFSIZE             1024\*16  // filenodes의 해시길이를 저장할 때 쓰는 매크로다. MD5보다 긴 SHA-1 의 길이를 기준으로 설정되었다.  #define HASH\_LEN            41  // flist에서 사용되는 매크로이다. 파일/디렉토리 테이블의 크기를 40개로 선할당해주었다.  #define START\_FLIST\_IDX     40  // filenodes에서 해당 파일 길이 이상의 파일은 백업하지 않도록 설정하였다.  (현재 100MB 이상의 파일은 해싱을 하지 않는다.)  #define MAX\_FILE\_SIZE       100000000  // 딕셔너리 생성 (mkdir) 생성 시 mode 값을 다음과 같이 설정하였다.  // S\_IRWXU | S\_IRWXO | S\_IRWXG 로 설정한 값이다.  #define DIRECTRY\_ACCESS     0777  // 아래 매크로는 파일 크기를 확인할 때 사용되는 매크로이다.  // \_STR 부분에는 해당 '문자열' 을 의미하고 , \_RETURN\_TYPE는 함수를 종료시킬 때  // 해당 이름을 우선으로 user 를 만들어서 백업경로를 제작.  #define BACKUP\_USERNAME     "junhyeongBKS"  return Type을 설정해주는 부분이다.  #define LEGTH\_ERR(\_STR, \_RETURN\_TYPE)     if (strlen(\_STR) >= MAXPATHLEN)\  {\      printf("%s string length is %ld, It is over max limit length(%d)\n", \_STR,strlen(\_STR) ,MAXPATHLEN);\      return \_RETURN\_TYPE;\  }   * [ 2. 전역변수 명세] * ssu\_exec.h, backup.c의 함수들은 연산이 용이하도록 actual\_path를 구해서 사용하도록 되어있다. * ACTUAL\_PATH 는 사용자 user id에 대한 홈 경로로 /home/<username> 으로 구성되어있다. 이는 get\_actuaalpath() (현재 작업공간을 기준으로 홈 경로를 결정) 하거나 get\_actualpath2(경로) (새로 입력받은 파일을 기준으로 /home/<username>를 결정) 한다. * 예시1> /home/junhyeong/go2/test1.c 라는 파일이 있을 때 ACTUAL\_PATH 는 user 폴더 바로 상위폴더부터 끝까지 경로를 의미한다. 위의 예시에서는 /home/junhyeong 가 ACTUAL\_PATH 가 된다. * Filenode 멤버변수에도 actual\_path가 존재하는데 이때 actual\_path는 ACTUAL\_PATH 는 다른 경로를 의미한다. * 예시2> /home/junhyeong/go2/test2.c 가 존재한다면 ACTUAL\_PATH /home/junhyeong 이고 Filenode 의 actual\_path는 go2/test2.c 가 된다.   // 현재 위치 getcwd() 사용. || 입력받은 파일의 username 이용  char ACTUAL\_PATH [MAXPATHLEN];  // /home/junhyeongBKS/backup 폴더를 생성해줌.  char BACKUP\_PATH [MAXPATHLEN];  // 만약, 에러가 발생하여 BACKUP\_USERNAME을 사용하지 못하거나, user를 추가하지 못하거나 /etc/passwd에서 파일을 찾을 수 없는 경우 사용  char changed\_username[256];   * [3. 함수 명세]   // 파일 사이즈 체크함수.  int file\_size\_check (char file\_names[]);  //★ 시작은 무조건 ACTUAL\_PATH, BACKUP\_PATH부터 구할 것.  void get\_actualpath();  // 입력된 파일 기준 actual\_path  void get\_actualpath2(char\* orignal\_path);  //$HOME 경로를 기준으로 $HOME/backup 경로 생성  void get\_backuppath();  //동일 유저명이 존재하는지 확인  int exist\_username();  // /home 디렉토리를 가진 아무 User나 가져오는 함수.  char\* find\_anyuser();   * [4. 함수 동작원리, 프로토타입 및 상세설명]   int file\_size\_check (char file\_names[]);   * file\_names 가 상대경로인지 절대경로인지 판별하여 변환해주는 함수 * 일때, 상대경로는 (. 이나 아예파일이름으로 들어온 경우는) 원래 linux 취지에 맞게 프로그램이 실행된 위치 즉, 프로그램이 실행될 때 자연스럽게 getcwd 했을 때 나오는 경로로 설정하였다. * ~ 와같은 $HOME 경로의 경우에는 새로 생성한 /HOME/junhyeongBKS 를 기준으로 홈 경로로 설정했다. * ex) /home/oslab/P1/ssu\_backup 을 실행한다면 상대경로는 /home/oslab/P1 이 될 것이고, ~ 홈 경로는 /home/junhyeongBKS 가 된다.  |  |  | | --- | --- | | 파라미터 설명 | char file\_names[]   * + file\_names 로 들어온 파일은 함수 내부에서 절대경로로 바꿔서 준다. 그렇기에 file\_names는 할당된 포인터가 들어가야한다. | | 작동원리 | 1. 함수 내부에서 상대경로를 판별한다..  2A. 상대경로 형태가 ~ 라면 BACKUP(생성된 사용자 $HOME) 를 기준으로 ~뒷 경로와 $HOME을 결합한다.  2B. 상대경로 형태가 . 혹은 바로 파일명 (/가 안 나옴) 인 경우에는 realpath 를 호출하여 경로를 변환해준다.  3. 그리고 ACTUAL\_PATH (들어온 데이터의 /home/<USERNAME> 을 설정해준다.  4. 새롭게 만들어진 경로가 **최대 길이(4096)**를 초과하는지 확인한다.  5. 경로가 최대길이를 초과하지 않는다면 함수 파라미터인 file\_names에 해당경로를 strcpy 를 통해 붙여넣는다. | | 리턴값 | 성공하면 1 , 실패하면 0 |   void get\_actualpath();   * 현재 작업 경로를 바탕으로 ACTUAL\_PATH를 설정한다.  |  |  | | --- | --- | | 작동원리 | 1. 함수 내부에서 getcwd를 호출한 경로를 받아온다.  2. /home/ 경로 이후에 있는 '/' 토큰을 기준으로 /home/<username> 의  username을 찾는다.  3. USERNAME을 찾은 경우에는 ACTUAL\_PATH에 복사한다.  -> 파일 길이가 무조건 줄어들기 때문에 파일크기 체크하지 않음. |   void get\_actualpath2(char\* orignal\_path);   * original\_path 경로를 기준으로 /home/<user\_name> 을 추출한다.  |  |  | | --- | --- | | 파라미터 | char\* orignal\_path   * + 추출받을 orignal\_path 경로 | | 동작원리 | 1. 상대경로로 들어온 경우 해당 경로를 realpath() 를 호출하여 절대경로로 변경한다.  2. 해당 경로에 /home/ 이 있는지 확인하여 없다면 즉시 return 한다.  3. original\_path 에서 /home/ 다음에 / 토큰까지 기준으로 username을 찾는다.  -> 만약, 토큰이 없다면 /home/ 뒤의 모든 내용이 uesrname으로 처리한다.  4. <FILENAME>, <NEWNAME> 옵션에 들어올 ACTUAL\_PATH를 /home/<username> 형태로 복사한다. |   void get\_backuppath();   * getenv(“HOME”) 을 호출하여 해당 사용자 아이디 $HOME 을 기준으로 백업경로 생성. * 만약 $HOME 경로가 /root 인 경우 pwd 를 통해 현재 자신 경로를 기준으로 backup 경로 생성.   int exist\_username();   * 백업하려는 username "**junhyeongBKS"** 가 /etc/passwd 에 있는지 확인한 후 있다면 1 ,없다면 0, open 에러를 발생한다면 -1을 리턴한다.  |  |  | | --- | --- | | 동작 원리 | /etc/passwd 값을 : 토큰으로 무한정 분리시켜보며 **junhyeongBKS (BACKUP\_USERNAME)** 과 일치하는지 확인하고 있다면 1, 없다면 0 return | | 리턴값 | 성공하면 0, 실패하면 1 |   char\* find\_anyuser(); //실제 사용 X   * 백업 경로 생성에 실패하였으므로 /etc/passwd 의 가장 첫번째 /home 경로를 백업경로로 설정한다.  |  |  | | --- | --- | | 동작원리 | 1. /etc/passwd 를 FILE 구조체로 받아온다.  2. fgets 함수의 주요특징 중 한 라인씩 받아오는 기능을 활용하여 한 줄씩 받아온다.  3. 아래 예시와같이 /etc/passwd 의 경로는 **사용자아이디(username)** : 비밀번호 : uid : gid : 설명 **: 홈 경로** **이므로 :** 토큰 기준으로 1번째 (username) 과 **6번쨰 home 경로** 이므로 토큰을 6번 분할하여 1,6번째 토큰을 각각 username, home 경로로 설정한 다음, home 경로에 /home 경로가 발생되는 가장 첫 경로를 전역변수 changed\_username 로 설정한다. | | 리턴값 | 성공하면 /home/<username> 경로, 실패시 NULL | |

## *ssu\_exec.h : add, remove, recover에 사용될 모든 함수들의 명세서***.**

### ssu\_exec 함수 명세 개요

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | | #ssu\_exec.h 함수 설명   * ssu\_exec 파일에는 add, remove, recover 연산에 필요한 함수들이 수록되어있다. * 소스코드 내부를 살펴보면 대략 10개의 매크로와 50개의 함수들로 구성되어 있다   [ 2. 구조체 명세]  ssu\_exec.h 에 선언된 구조체는 크게 4가지가 있다. 노드의 역할을 하는 filenode와 이를 연결하는 구조체인 flist, rlist, frlist 등이 존재한다.  <구조체 소개>  1. Filenode  - 연결리스트에 연결된 node를 의미한다.  - 해당 노드에는 pathname(파일원래경로), filename (파일이름), actualpath  (실질적경로), inversepath (상대경로), hash(해당 파일의 해시값), file\_stat  해당 파일의 stat 구조체), back\_up\_time (백업될 시간과(ex> 230312201830)),  마지막으로 자신과 연결될 filenode 포인터 next가 존재한다. (자기참조구조체).  **2. Mnode**  - Filenode 에서 same\_next 기능을 추가하여 구현한 기능으로 단방향 연결리스트에서 체이닝 연결리스틀 지원하는 구조체이다. Filenode 기능을 추가하여 만들었기 때문에 구조체 내부변수에 Filenode가 들어간다.  3. **rlist**  - 일반적인 연결리스트 형태로 구현되어있다.  - 데이터가 새로 갱신될 때마다 헤더부분이 늘어나는 구조형태를 띄고 있다.  4. Mlist  - 명세에 적혀있는대로 각 명령어에서 파일노드들을 연결리스트 형태로 관리할 수 **있도록 제작된 Mlist 이다. 체이닝연결리스트를 지원하며 즉, 같은 이름이지만 다른** 시간 백업노드를 가진 노드에 대하여 same\_next 라는 포인터를 제공하여 연결할 수 있도록 해준다.  6. mrlist  - 후에 add, recover, remove 연산 시 필요한 mlist 와 rlist를 한꺼번에 받아오는 구조체이다. 실 사용에시) 일반적으로 ssu\_recover\_default 함수를 통해 백업할 경로를 받아온 rlist 리스트를 ssu\_recover\_default 함수 내부에서 상대적 위치인 BACKUP 경로로 변환한 백업경로의 파일리스트를 mlist로 뽑아오는데 이때 두 리스트를 모두 가져온다.    typedef struct filenode {      //초기화할 때 설정해줘야하는 값      char path\_name [MAXPATHLEN];        // 파일 경로 저장      char file\_name [MAXFILELEN];        // 파일 이름 저장  // 백업파일/원래경로를 제외한 실질적 경로.      char actual\_path [MAXPATHLEN];      char inverse\_path [MAXPATHLEN];     // 상대경로 저장      char hash[HASH\_LEN];                // 해시값 저장 (해시길이 : 40)      struct stat file\_stat; // stat 저장 (바이트 크기 등에 사용.) -> st\_size      char back\_up\_time [TIME\_TYPE]; // 백업 타이밍 설정        //나중에 설정할 값들.      struct filenode\* next;   // Reg 파일의 경우 연결리스트 사용  }Filenode;      typedef struct managenode {      Filenode\* filenode;      //나중에 설정할 값들.      struct managenode\* next;  // 같은 파일 , 다른 백업시간의 파일들을 연결하는 포인터.      struct managenode\* same\_next;      struct managenode\* same\_next\_tail;   // same\_next에 연결된 노드 중 가장 마지막 노드, 삽입시간을 O(1) 로 하기위함. 없으면 NULL      int same\_cnt; // 자기 자신 노드를 포함하고 same\_next에 연결된 파일 개수  }Mnode;    /\*\*   \* : rlist : 단순 연결리스트 헤더   \*  :꼬리부분과 헤더부분 존재.   \*  (꼬리부분 :rear (헤더)) + (헤더부분 : header(새로 추가될 때마다 갱신되는 부분))   \*  \*/  typedef struct rlist{  // rlist의 꼬리부분 (가장 먼저 들어온 데이터의 filenode 포인터가 들어감)      Filenode\* header;  // rlist의 헤드부분 (가장 최근에 들어온 데이터의 filenode 포인터가 들어감)      Filenode\* rear;      int file\_cnt; // 현재 rlist에 들어있는 파일의 개수.  }Rlist;    typedef struct mlist{      Mnode\* dir\_head;      Mnode\* file\_head;      Mnode\* dir\_tail;          // 연결리스트를 바로 연결하기 위한 구조. O(1)      Mnode\* file\_tail;      int file\_cnt;                       // 파일 총 개수      int dir\_cnt;                        // 디렉토리 총 개수      //★file\_rear\_table, file\_cnt\_table 는 max\_file\_cnt를 따라감.  }Mlist;   typedef struct mrlist{      Rlist\* rlist;      Mlist\* mlist;  }MRlist;  [4. 함수 명세] | |  | | /\*\* 추가도구\*/  char\* replace (char\* original, char\* rep\_before, char\* rep\_after, int cnt);  // 문자열 교체함수 (중요한 점은 char\* a = replace() 형태로 쓸것.)  int kmp (char\* origin, char\* target);  // 문자열 교체함수에서 KMP 알고리즘 이용.  //해싱함수 : opt:0->md5, opt:1->SHA1  char\* do\_hashing(FILE \*f, int opt);   //option 0 :md5, 1: sha1  char\* hash\_to\_string(unsigned char \*md);  //hash 16비트 md 를 문자로 변환하는 함수.  // 해시 비교함수  int hash\_compare (Filenode\* a\_node, Filenode\* b\_node); //a와 b filenode 비교함수  int hash\_compare\_one (Filenode\* a\_node, char\* path\_name, int opt, int f\_opt);  // a filenode와 path\_name 을 비교하는 함수.  // 구조체 초기화 함수들.  Filenode\* new\_filenode (); // 기본 초기화  Filenode\* new\_filenodes (char\* filename, int opt, int f\_opt); // 파일 초기화, option 0: original, 1: backup  Rlist\* new\_Rlist(); //  MRlist\* new\_MRlist(Mlist\* mlist, Rlist\* rlist);  void print\_node (Filenode\* node);  // Filenode Unit 상태 출력  void print\_rlist (Rlist\* rlist); // rlist 모든 요소 출력  void rappend (Rlist\* rlist, char\* file\_name, int opt, int f\_opt); // Rlist 에 file\_name 경로 데이터 단순 연결  Filenode\* rpopleft (Rlist\* rlist);// Rlist 큐 popleft  void free\_rlist(Rlist\* rlist); // rlist 모든 요소 동적할당 해제  void free\_mrlist(MRlist\* mrlist);  // mrlist 모든 요소 동적할당 해제  // 파일 복사 함수  int file\_cpy (char\* a\_file, char\* b\_file); (a\_file-->b\_file로 파일복사)  //파일 복사 함수 (a노드 path\_name --> invere\_path 로 복사)  int node\_file\_cpy (Filenode\* a\_node);  int make\_directory (char\* dest);    // 현재시간 \_230227172231 (현재시간 생성 개체)  char\* curr\_time();  // 파일 탐색 함수.  Rlist\* original\_search(char\* file\_name, int f\_opt, int all); // 그냥 연결리스트 구현 (동작확인완료 . 3.04)  Mlist\* backup\_search(char\* file\_name, int f\_opt, int all);  // 해시 체이닝(연결리스트) 구현. (동작확인완료 . 3.04)  int scandir(const char \*dirp, struct dirent \*\*\* namelist,              int(\*filter)(const struct dirent \*),              int(\*compar)(const struct dirent\*\*, const struct dirent \*\*)); // scandir, alphasort 명시  int alphasort(const struct dirent \*\*d1, const struct dirent \*\*d2);  // 1. add 계열함수  int ssu\_add (char\* file\_name, int flag, int f\_opt);  // add 명령어를 수행하는 함수.  // 2. remove 계열 함수.  void ssu\_remove (char\* file\_name, int a\_flag); //ssu\_recover\_default() 재탕  void ssu\_remove\_all();   // BACKUP 경로에 있는 모든 경로를 mlist로 뽑아와서 popleft 하여 모두 지우는 함수.  void scandir\_makefile\_for\_remove(Mlist\* mlist, char \*parent\_folder);  // parent\_folder 를 바탕으로 mlist 안의 모든 하부디렉토리 파일을 mlist로 구성함.  Mlist\* check\_backup\_file\_for\_remove(char\* file\_name, int flag\_a);  // 백업할 file\_name을 상대적인 BACKUP\_PATH 경로로 전환  // 3. recover 계열 함수.  int ssu\_recover (char\* file\_name, int flag\_d, int flag\_n, char\* new\_name, int f\_opt);  // replace이용. (캡슐함수)  // file\_name => new\_name으로 inverse\_path 변경.  int modify\_inversepath (Filenode\* file\_name, char\* new\_name, int flag\_d);  // d\_flag 설정 여부 확인하여 상대경로 file\_name MRlist 반환  MRlist\* ssu\_recover\_default (char\* file\_name, int d\_flag, int f\_opt);  // 같은 디렉토리에 대한 230227172413 15bytes 등 출력.  void print\_time\_and\_byte (Filenode\* node);  void append\_samefile (Mlist\* mlist, char\* original\_file\_name, int f\_opt);  // origin path 에 있는 같은 이름의 파일 백업 파일 긁어오기.  int check\_backup\_file(char\* file\_name, int flag\_d);  //상대적인 경로에 파일이 있는 경우 만들어줌  void make\_file\_name(char\* file\_name);  void scandir\_makefile(char \*parent\_folder, char\* original\_path);  void main\_help(); // 전체 help 출력  /\*\* scandir 함수\*/  // 전역변수 scandir\_filename 과 file\_name 이 동일한 백업경로 파일리스트 mlist 추출 scandir filter  int scandir\_filter (const struct dirent\* entry);  // . , .. 를 제외한 dirent 구조체를 가져오는 filter (알 수 없는 오류로 사용불가)  int basic\_filter (const struct dirent\* entry);  // mlist 에 file\_path (백업할 경로) 를 상대 백업경로로 변경하여 해당하는 모든 백업경로의 데이터를 mlist 를 가져옴  int append\_samefile2 (Mlist\* mlist, char\* file\_path, int opt, int f\_opt);  char\* scandir\_filename; // 전역변수 scandir\_filename을 설정함으로 scandir\_filter 로 scandir fileter로 사용  /\*\* 모두 03.27 추가 : 파일 관리자 구조체\*/  Mnode \*new\_mnodes(char\* file\_name, int opt, int f\_opt); // Mnode 생성자 블럭 (filenode->구조체이용)  Mlist\* new\_mlist(); // mlist 생성자 블럭  void mappend (Mlist\* mlist, char\* file\_name, int opt, int f\_opt); // mlist 에 딕셔너리/파일 등 연결리스트에 연결해주는 구조체 (dir,file 구분연결)  void print\_mlist (Mlist\* mlist); // 파일 관리된 mlist 출력  void free\_mlist(Mlist\* mlist); // add,remove,recover 함수 호출 후 삭제.  void pop\_mlist (Mlist\* mlist, char\* delete\_string); // 백업 경로에 있는 파일 삭제 시 관리  Mlist\* manage\_backup\_path\_file(); // 백업 경로에 있는 모든 파일 mlist화  /\*\*   \* option에 따라 파일/디렉토리를 하나씩 지우는 함수.   \* option : file=0, direcory=1,   \*  return : 딕셔너리인 경우에는 스택을 활용하기 위해 char\* 동적할당해서 return해줌,   \*  일반파일의 경우에는 상관없음.   \*/  char\* popleft\_mlist(Mlist\* mlist, int option); // mlist 에서 딕셔너리 혹은 파일을 하나씩 삭제 : option(0: 파일 삭제, 1: 딕셔너리삭제)  void\* pop\_dict\_mlist(Mlist\* mlist); // mlist 딕셔너리 전용 삭제 (stack형태로 지워야하기 때문)  /\*\* help 함수 추가 03.15\*/  void main\_help\_add(); // add help 함수 출력  void main\_help\_remove(); // remove help 함수 출력  void main\_help\_recover(); // recover help 함수 출력 | |  | |

### ssu\_exec 함수 내부 구현 로직

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| <0. 추가 도구들>  // 문자열 교체함수 (중요한 점은 char\* a = replace() 형태로 쓸것.)  char\* replace (char\* original, char\* rep\_before, char\* rep\_after, int cnt);  int kmp (char\* origin, char\* target); // 문자열 교체함수에서 KMP 알고리즘 이용.   * kmp 알고리즘을 이용하여 original 문자열의 rep\_before 부분을 rep\_after 부분으로 교체해주는 함수 * cnt 가 0 이면 일치하는 모든 rep\_before을 rep\_after로 바꿔주고, cnt가 0이아닌 양의 정수가 할당될 시, cnt 만큼 문자열을 교체해줌 ex> what a good day -> what a sad day 를 바꾸려고 할 때, replace("what a good day", "good", "sad", 0);  |  |  | | --- | --- | | 작동 원리 | C++ 과 다르게 C언어는 문자열을 교체하는 함수가 지원되지 않았다.  그래서 Mainstream에서 substring을 찾아서 해당 위치에 pattern을 교체해주는 함수를 제작하여 사용하였다.    예를들어, ABCABCABC 가 있다고 하였을 때 BC 를 찾는 알고리즘은 KMP 알고리즘을 사용하였다. KMP 알고리즘의 간략한 설명을 하자면, 접미어와 접미어가 일치하는 부분을 고려하여 문자열이 일치하지 않는다면 pattern 크기 만큼 이동하였다가  pattern 의 해당부분이 접두어랑 일치하는 길이만큼 되돌려주는 알고리즘으로서  시간복잡도가 O(N) 이되는 알고리즘을 이용하여 pattern 문자열과 일치하는 인덱스를 int\* 테이블로 전달받아 사용된다.    마지막의 cnt는 일치하는 문자열을 몇개까지 바꿀 것인지 묻는 옵션이다. 만약 0을 넘겨주게되면 rep\_before과 일치하는 모든 문자열을 rep\_after로 바꿔준다. 해당함수는 pattern문자를 추가한 문자열을 동적으로 할당하여 주기때문에 return 받은 문자열은 꼭 free를 해서 동적할당을 해주어야한다.  요약하자면 아래와 같이 작동한다.    1. kmp로 일치 문자열 찾기  2. ptr = 일치문자열 + 찾을 문자열길이 해서 뒷부분 복사  3. 원본 -> 바꿀 문자열 바꾸기  4. 붙여넣기     주의할 점은 : 할당된 문자열이 오기 때문에 사용 후 free를 해줄 것을 명심하라. |           <1. ssu\_exec.h 실행초기에 실행될 함수들>    void main\_help\_add();  void main\_help\_remove();  void main\_help\_recover();  void main\_help();   * 잘못된 인수, 잘못된 경로 등 , help를 입력헀을 때 출력되는 함수 * main\_help : help 전체 출력 help 문장은 아래와 같음 * main\_help\_add : main\_help() 중 add 부분만 출력 * main\_help\_recover : main\_help() 중 recover 부분만 출력 * main\_help\_remove : main\_help() 중 remove 부분만 출력  |  |  | | --- | --- | | help 시 출력되는 문장 | "Usage:\n"  "  > add <FILENAME> [OPTION]\n"  "    -d : add directory recursive\n"  "  > remove <FILENAME> [OPTION]\n"  "    -c : remove all file(reculsive)\n"  "    -a <NEWNAME> : clear backup directory\n"  "  > recover <FILENAME> [OPTION]\n"  "    -d : recover directory recursive\n"  "    -n <NEWNAME> : recover file with new name\n"  "  > ls\n"  "  > vi\n"  "  > vim\n"  "  > help\n"  "  > exit\n" |     //해싱함수 : opt:0->md5, opt:1->SHA1  char\* do\_hashing(FILE \*f, int opt);   // 파일의 내용물을 읽어 해시를 읽어주는 함수  char\* hash\_to\_string(unsigned char \*md); // EVP\_DigestFinal\_ex 를통해 전달받은 16진수 문자열을 일반 문자열로 변환시켜주는 함수.   * <https://www.openssl.org/docs/man1.1.1/man3/EVP_MD_CTX_new.html> 참고하여 제작 * opt의 값이 0이면 MD5, 1이면 SHA-1 로 해시를 구해준다.  |  |  | | --- | --- | | 작동원리 | 1. EVP\_get\_digestbyname 안에 3항연산자를 이용하여 MD5, SHA-1 에 맞는 MVP\_MD 구조체를 받아온다.  2. mdctx라는 EVP\_MD\_CTX 구조체를 새로할당받아 EVP\_DigestInit\_ex 에 MVP\_MD 에 맞는 EVP\_MD\_CTX로 변환시켜준다.  3. 파일의 내용을 계속 받아와서 mdctx 를 계속 업데이트 해준다. (해시업데이트)  4. 해싱이 끝나면 해싱된 값을 EVP\_DigestFinal\_ex 을 호출하여 문자열(md)을 받아온다.  5. 해당 문자열은 16진수로 표현된 문자열이므로 hash\_to\_string 함수를 내부적으로 호출하여 일반문자열 형태로 변환시켜준다. |  * 성공 시 해시값 (할당되어 있으므로 다 쓰고 해제할 것), 실패 시 NULL 반환     char\* curr\_time(); // 현재시간 \_230227172231   * (현재시간 생성 개체) 과 같은 현재 시각에 대한 정보를 YYMMDDHHMMSS 형태로 받아오는 함수.  |  |  | | --- | --- | | 작동원리 | 1. time\_t 구조체를 (time.h 에 정의되어있는) localtime 을 호출하여 현재 시각에 대한 구조체( tm )를 받아온다.  2. tm 구조체의 tm\_year, tm\_mon, tm\_mday, tm\_hour, tm\_min, tm\_sec 멤버변수들을 조합하여 YYMMDDHHMMSS 형태로 변환하여 반환해준다. |  * 현재시각에 대한 정보를 문자열화 하여 반환해줌  (해당 함수는 동적할당하여 전달하기 때문에 끝나고 할당해제를 해줘야한다.)     <2. 구조체 초기화 함수들.>  Filenode\* new\_filenode ();  // 기본 초기화   * Filenode 를 생성할 때 멤버변수들을 (NULL 내지 0) 으로 초기화 시켜주는 함수 * OOP 를 위한 일종의 생성자처럼 다룸. * 일반적으로 new\_filenodes 내부에서 실행되도록 설정.   // 파일 초기화, option 0: original, 1: backup  Filenode\* new\_filenodes (char\* filename, int opt, int f\_opt);   * Filenode 구조체를 filename의 경로로 들어왔을 때 해당 경로를 절대경로화 시켜서  Filenode 내부 멤버변수에 맞는 값들을 할당  |  |  | | --- | --- | | parameter 설명 | opt  0: 원래경로 (ex> /home/junhyeong/go2/file.cpp) 1: 백업경로 (ex> /home/junhyeong/backup/go2/file.cpp)  f\_opt  0: MD5  1: SHA-1 | | 멤버변수 목록 | char path\_name [MAXPATHLEN];  // 파일 경로 저장      char file\_name [MAXFILELEN];    // 파일 이름 저장      char actual\_path [MAXPATHLEN];   // 실질적 경로      char inverse\_path [MAXPATHLEN];   // 상대경로      char hash[HASH\_LEN];                 // 해시값 저장 (해시길이40:)      struct stat file\_stat;               // stat 저장      char back\_up\_time [TIME\_TYPE];     // 백업 타이밍 설정      struct filenode\* next;               // Reg 파일의 경우 연결리스트 사용 | | 동작원리 | if filename == test/test1.txt, opt==0, f\_opt=0 일 때.    0. filenode 를 새로 할당한다.  1. filename가 상대경로로 들어온 경우 opt에 맞게 절대경로화 시켜준다.  -> 이미 초벌로 filename은 절대경로로 들어오게 세팅되어 있지만, 안정성을 위해 한 번 더 처리해주도록 설정  (filename -> /home/junhyeong/test/test1.txt)  2. filename이 존재하는 경로인지 체크하고 존재하지 않으면 ' $(PATH) is not existed!' 메시지를 띄워주고 NULL 을 return하고 종료한다.  3. 파일의 stat 구조체를 구하여 file\_stat 구조체에 복사해준다.  4. 해당 파일이 딕셔너리파일 이나 Regular 파일이 아닐 시 종료한다. (Fifo 파일 같은 경우면 파일처리를 해주지 않는다.)  ※ FIFO 파일의 경우 해싱과정에서 무한루프를 돌아버리는 문제가 발생하여  해시를 구하지 않음.  5. 해당 path\_name을 바탕으로 ACTUAL\_PATH 변수를 토큰으로 ACTUAL\_PATH 뒷 부분 경로를 가져와서 actual\_path에 복사해준다.  (actual\_path = "/test/test1.txt")  6. path\_name의 가장 마지막 '/' 문자를 기준으로 file\_name을 구한다.  (file\_name = 'test1.txt')  7. 해당 opt에 따라 상대경로를 구해준다. : 해당 예시에서는 opt=0 이므로 상대경로는 BACKUP\_PATH + ACTUAL\_PATH가 된다.  (opt : 0 일 때 BACKUP\_PATH + actual\_path 멤버변수)  (opt: 1일 떄 ACTUAL\_PATH + actual\_path 멤버변수)  8. 해당 파일의 해시값을 do\_hashing을 통해 구해준다.   * + 만약 해당 파일이 디렉토리 파일이거나 파일의 용량(사이즈)이 MAX\_FILE\_SIZE (현재 10MB) 를 넘는 경우  "$(path\_name) file size is $(file\_stat.st\_size), pass\n" 에러 메시지 출력 후 백업시켜주지 않고 NULL 리턴   + 만약 해시값이 MD5 의 경우 32바이트만 사용하고 나머지 8바이트는 사용하지 않으므로 뒷 8바이트는 '\0' 으로 초기화   9. filenode 를 return해준다. |     Flist\* new\_mlist ();   * mlist 하나를 새로 할당하고 mlist 의 멤버변수들을 (NULL 내지 0) 으로 모두 초기화해줌 * flist의 생성자 같은 역할을 해준다.   Rlist\* new\_Rlist();   * rlist 하나를 새로 할당하고 rlist 의 멤버변수들을 (NULL 내지 0)으로 모두 초기화해줌 * rlist의 생성자 같은 역할을 수행한다.   FRlist\* new\_MRlist(Mlist\* mlist, Rlist\* rlist);   * mrlist 하나를 새로 할당하고 rlist, mlist를 NULL로 초기화 시켜준다. * mrlist의 생성자 같은 역할을 수행한다.     void print\_node (Filenode\* node); // Filenode Unit 상태 출력   * Filenode의 path\_name, file\_name, actual\_path, inverse\_path, hash, 딕셔너리 여부 등을 출력해주는 함수   void print\_rlist (Rlist\* rlist); // rlist 모든 요소 출력   * rlist의 모든 filenode의 path\_name 을 출력해주는 함수.   void print\_mlist (Flist\* flist); // flist 모든 요소 출력   * fmist의 모든 filenode 에 대해 호출해줌 * 만약 file 테이블에 같은 이름(file\_name)의 노드가 존재할 시 >> same file 을 출력하고 해당 path\_name 을 출력해줌.   // Rlist 에 file\_name 경로 데이터 단순 연결  void rappend (Rlist\* rlist, char\* file\_name, int opt, int f\_opt);   * rlist에 단순 해당 file\_name 의 filenode를 할당하여 연결리스트 연결  |  |  | | --- | --- | | parameter 설명 | Rlist\* rlist   * + 해당 file\_name 을 기준으로 filenode를 생성하여 연결할 rlist 생성,     char\* file\_name   * + 해당 파일 경로     int opt  0: 원래경로 (ex> /home/junhyeong/go2/file.cpp) 1: 백업경로 (ex> /home/junhyeong/backup/go2/file.cpp)  int f\_opt  0: MD5  1: SHA-1 |       **/\*\*\*\*\*\*\*\*\*\* 파일관리 구조체 Mlist 구현 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**  Mnode \*new\_mnodes(char\* file\_name, int opt, int f\_opt);   * 파일 관리 구조체를 지원하기 위한 (Mlist)에 추가될 Mnode를 구현하기 위한 Mnode 생성자 함수 * filenode 구조체에서 same\_next 포인터를 지원함으로써 단순 연결리스트에서 나아가 **같은 파일명에 같은 수준에서의 연결리스트 구현**  |  |  | | --- | --- | | parameter 설명 | char\* file\_name   * + Mnode에 추가할 파일 이름   int opt (append 함수에 넘겨줄 인자때문에 사용 실상 1만 사용)  0 : Original\_path (백업할 경로)  1 : Backuped\_path (백업 경로)  int f\_opt (append 함수에 넘겨줄 인자때문에 사용)  0 : MD5  1 : SHA-1 |   Mlist\* new\_mlist();   * 파일관리 구조체 Mlist 를 할당받고 모든 요소를 초기화시켜주는 함수   void mappend (Mlist\* mlist, char\* file\_name, int opt, int f\_opt);   |  |  | | --- | --- | | parameter 설명 | Mlist\* mlist   * + 파일 관리 구조체 컨테이너 리스트 Mlist   char\* file\_name   * + Mnode에 추가할 파일 이름   int opt (append 함수에 넘겨줄 인자때문에 사용 실상 1만 사용)  0 : Original\_path (백업할 경로)  1 : Backuped\_path (백업 경로)  int f\_opt (append 함수에 넘겨줄 인자때문에 사용)  0 : MD5  1 : SHA-1 | | 동작 원리 | 1. file\_name을 바탕으로 새로운 mnode를 만든다. (newfile)  2. stat구조체를 이용하여 st\_mode를 통해 해당 구조체가 딕셔너리/일반파일 인지 판단한다.  3A-1 : 해당 파일이 딕셔너리라면 next포인터에 단순 연결한다.  3B-1 : 해당 파일이 일반파일인데 option 이 0 이라면 동일한 파일이 존재할 수 없기 때문에 단순연결한다.  3B-2 : 해당 파일이 일반파일이고 option 이 1이라면 백업시간 ('\_') 을 기준으로 토큰을 분리하여 같은 파일이 있는지 확인한다.  3B-3A : 분리된 토큰 기준으로 같은 파일이 존재한다면 same\_next 포인터에 newfile을 연결한다.  3B-3B : 분리된 토큰 기준으로 같은 파일이 존재하지 않는다면 next에 단순 연결한다. |   void print\_mlist (Mlist\* mlist);   * 파일 관리 구조체 Mlist의 모든 요소들을 출력해주는 함수.   void free\_mlist(Mlist\* mlist);   * mlist의 모든 요소들을 free 해주는 함수.   **void pop\_mlist (Mlist\* mlist, char\* delete\_string);**  // mlist 파일 array 대해 추가. option 0: orignal, 1: Backup  void mappend (Mlist\* mlist, char\* file\_name, int opt, int f\_opt);   * 파일 관리 구조체 Mlist 에서 딕셔너리 와 일반 파일들을 분리해서 append하고,  같은 이름을 가진 파일들은 next 포인터가아닌 same\_next포인터에 연결되도록 해주는 append함수  |  |  | | --- | --- | | parameter 설명 | Mlist\* mlist   * + 해당 file\_name 을 기준으로 filenode를 생성하여 연결할 mlist     char\* file\_name   * + 해당 파일 경로     int opt  0: 원래경로 (ex> /home/junhyeong/go2/file.cpp) 1: 백업경로 (ex> /home/junhyeong/backup/go2/file.cpp)  int f\_opt  0: MD5  1: SHA-1 | | 동작원리 | if filename == test/test1.txt, opt==0, f\_opt=0 일 때.  0. 해당 pathname을 이름을 기준으로 Mnode객체 하나를 할당받는다. (inverse\_path 등 여러 유틸리티를 활용하기 위함)  1. 해당 filename 데이터가 딕셔너리라면 그냥 dir\_array 에 Filenode를 추가한다.  2A. opt == 0 이라면 rappend와 같이 일반적인 연결리스트를 연결한다. (큐 형태처럼 다음노드가 dir/file\_head가 새로 생성된 Mnode 를 연결하는 형태)  2B. opt가 1이라면 dir\_array 는 opt=0 일떄와 동일하게 작동, file의 경우에는 file\_<time> 에서 time 만 다른 Mnode 의 경우 next 포인터에 연결되는 것이 아니라 same\_next 포인터에연결되어 현재 링크드 리스트와 독립적으로 연결하게 함.  이떄 same\_next 포인터를 활용하기 위해 filenode를 Mnode의 멤버변수로 사용하고 same\_cnt, same\_next\_tail 등을 지원하여 파일리스트 추가/삭제를 용이하게 해줬음. |   Mlist\* manage\_backup\_path\_file();   * 백업 경로의 모든 파일을 불러오는 함수 (remove 할 때 파일관리를 위해사용되는 함수)  |  |  | | --- | --- | | 동작 원리 | 1. backup\_search() 에 BACKUP 경로를 넣어 백업경로에 있는 모든 파일리스트를 Flist로 받아온다.  2. 받아온 Flist 를 바탕으로 Mlist를 재구성한다.  3. 임시로 받아온 flist는 free를 한다.  4. 받아온 Mlist를 Return한다. |   char\* popleft\_mlist(Mlist\* mlist, int option);   * mlist 요소들을 option(0:파일, 1:딕셔너리) 를 tail로부터 하나씩 삭제하는 함수. * remove 에서 사용될 리스트 삭제함수  |  |  | | --- | --- | | 인자 설명 | Mlist\* mlist   * + 파일리스트를 관리중인 mlist   int option   * + 0 : 파일 을 file\_tail 으로부터 하나 삭제   + 1 : 딕셔너리를 dir\_tail 으로부터 하나 삭제 | | 동작 설명 | <option : 0>  1. dir\_tail 노드를 삭제할 노드 delnode 로 설정  2. dir\_tail노드를 dir\_tail->next 로 설정함으로써 dir\_tail 을 하나 삭제  3. mlist->dir\_cnt 를 하나 줄임  4. 삭제된 del\_node 를 free 하고, 문자열 길이만큼 문자열 할당해서 반환    <option : 1>  1. file\_tail 노드를 삭제할 노드 delnode 로 설정  2A. 만약 삭제할 노드에 same\_next 노드가 연결되어있으면 해당 same\_next 노드를 file\_tail 노드로 설정  -> 삭제 시 삭제될 delnode의 same\_cnt, same\_next\_tail 정보등을 모두 복사  2A-2 삭제하면서 file\_tail->same\_cnt, 를 하나 줄임  2B. same\_next가 NULL 이라면 file\_tail = file\_tail->next; 를 하여 맨 앞 노드 삭제  2B-2. 삭제 후 mlist->file\_cnt 하나 줄임  3 . 2A,2B 과정 중 모두 mlist->total\_file\_cnt 를 줄임  4. return NULL; | | return | 딕셔너리 option:1 을 삭제 시 삭제한 딕셔너리 파일경로 반환 실패 시 NULL  : 해당 파일경로는 pop\_dict\_mlist 에서 스택형태로 삭제되기 위해 사용  파일은 NULL 반환 |       void pop\_dict\_mlist(Mlist\* mlist);   * mlist 의 모든 디렉토리를 stack 의 pop 형태로 삭제 (mlist 에서 pop 된 딕셔너리 경로를 역순으로 삭제) * 이것은 앞 부분에 상위 디렉토리가 먼저 들어가기 때문에 Queue 의 pop으로는 한계가 있어 제작 * 구조를 stack 형태로 변환 시 파일 탐색이 어려워져 위의 형태 체택  |  |  | | --- | --- | | 인자 설명 | Mlist\* mlist   * + 파일 구조가 관리되고 있는 mlist | | 동작 설명 | 1. mlist의 딕셔너리 개수만큼 char\*\* 동적 배열 할당  2. mlist 의 모든 딕셔너리 개수만큼 (mlist->dir\_cnt) 딕셔너리를 모두 pop 하면서 할당받은 동적배열의 역순으로 채움  3. 할당받은 동적배열의 0번 인덱스부터 remove 를 호출하여 경로 삭제 와 동시에 동적할당 해제  4. 할당받은 char\*\* 동적배열 free |   void free\_rlist(Rlist\* rlist);   * rlist의 모든 데이터요소 (filenode) 를 할당해제해주는 함수.   void free\_mlist(Mlist\* mlist);   * mlist의 모든 데이터요소 (Mnode, Filenode) 등을 할당해제 해줄 때 사용   void free\_mrlist(MRlist\* mrlist);   * frlist안의 mlist, frlist를 할당해제해주고mrlist도 할당해제 해주는 함수.     // 파일 복사 함수  int file\_cpy (char\* a\_file, char\* b\_file);   * a\_file 의 내용을 b\_file 의 경로로 복사해주는 함수. (파일 존재여부 확인하지 않고 덮어씌워버림)  |  |  | | --- | --- | | 동작 원리 | 저수준 파일 입출력 (open, read, write) 사용  1. fd1(a\_file)를 read 해서 fd2(b\_file) 에 write 해주는 방식으로 구현  2. 만약 a\_file이 접근불가능하거나 없는 파일일 경우 에러메시지 출력 후 return 0  3. b\_file의 경우 해당 파일의 경로의 부모디렉토리가 존재하는지 체크 후 없으면 만들어주는 make\_directory 함수(아래 설명) 이용 |  * 성공 시 1 실패시 0 return     int node\_file\_cpy (Filenode\* a\_node);   * file\_cpy의 a\_file 은 a\_node의 path\_name, b\_file은 a\_node의 inverse\_path 로 복사해주는 함수 * 원래경로 -> 상대경로로의 복사 유틸리티를 위하여 제작.   int make\_directory (char\* dest);   * dest 목적지 까지 부모 디렉토리가 모두 존재하는지 체크 후 존재하지 않는다면 해당 부모 명으로 디렉토리 생성  |  |  | | --- | --- | | 작동 원리 | 해당 dest를 /단위로 strtok 해주어 해당 단계에 해당 디렉토리가 있는지 확인 후 없으면 DIRECTORY\_ACCESS 권한으로 디렉토리 생성  ※ DIRECTORY\_ACCESS 는 기본적으로 0777 으로 설정되어있음 |       < 3. 구조체 탐색함수>  Rlist\* original\_search(char\* file\_name, int f\_opt, int all); // 그냥 연결리스트 구현   * original 경로에 존재하는 모든 파일에 대하여 단순 연결리스트 구현 (단순 rlist 연결)  |  |  | | --- | --- | | parameter 설명 | char\* file\_name  파일경로  int f\_opt  0: MD5  1: SHA-1  int all  0: 해당 파일경로 하나에 대해서만 수행  1: 해당 파일경로에 대해서 하위폴더를 모두 탐색 | | 작동원리 | 1. 해당 file\_name 에 대하여 filenode 를 생성 ( file\_path, path\_name 등 유틸리티 활용을 위해 사용)  2\_1. file\_name이 딕셔너리 파일일 경우  2\_1\_A. 딕셔너리 파일인데 all이 0일 경우 그 딕셔너리에 관해서만 scandir 을 활용하여 하위 파일들을 rappend 수행  2\_2\_B. 딕셔너리 파일인데 all이 1일 경우 rlist를 실시간으로 업데이트 하면서 해당 rlist가 끝에 도달할 때 까지 재귀적으로 파일을 rappend 해줌.    2-2. file-\_name이 딕셔너리 파일이 아닐 경우 해당 파일만 rappend 해준다. |   Mlist\* backup\_search(char\* file\_name, int f\_opt, int all);   * 해시 체이닝(연결리스트) 구현. * 백업 경로에 존재하는 모든 하위 파일에 대하여 mlist 구조체를 업데이트해주는 함수  |  |  | | --- | --- | | parameter 설명 | char\* file\_name  파일경로  int f\_opt  0: MD5  1: SHA-1  int all  0: 해당 파일경로 하나에 대해서만 수행  1: 해당 파일경로에 대해서 하위폴더를 모두 탐색 | | 작동원리 | 1. 해당 file\_name 에 대하여 filenode 를 생성 ( file\_path, path\_name 등 유틸리티 활용을 위해 사용)  2\_1. file\_name이 딕셔너리 파일일 경우  2\_1\_A. 딕셔너리 파일인데 all이 0일 경우 그 딕셔너리에 관해서만 scandir 을 활용하여 하위 파일들을 mappend 수행  2\_2\_B. 딕셔너리 파일인데 all이 1일 경우 flist를 실시간으로 업데이트 하면서 해당 mlist의 리스트가 끝에 도달할 때 까지 재귀적으로 파일을 mappend 해줌.    2-2. file\_name이 딕셔너리 파일이 아닐 경우 해당 파일만 mappend 해준다. | | 리턴값 | 성공 시 추출된 상대경로(BACKUP) 파일리스트 mlist 반환 실패시 NULL |     // 1. add 계열함수  int ssu\_add (char\* file\_name, int flag, int f\_opt);   * Original Path 경로를 입력받게 되면 해당 경로를 기준으로 백업하는 함수. * 백업 과정에는 해시값을 비교하는 과정이있음 * d 옵션을 사용하는 경우 해당경로를 기준으로 하위디렉토리 및 파일들을 전부 백업해줌 * d 옵션을 사용하는데 Regular 파일이 오는 경우 d 옵션이 없는 것처럼 동작하도록 구현 * d 옵션을 사용하지 않는데 디렉토리 파일이 온다면 "$(filename) is a directory file" 이라는 에러메시지와 함께 종료되도록 구현  |  |  | | --- | --- | | parameter 설명 | char\* file\_name  파일경로    int flag  0: 플래그 없음  1: d 플래그 존재    int f\_opt  0: MD5  1: SHA-1 | | 작동원리 | 1. 해당 file\_name 에 대하여 filenode 를 생성 ( file\_path, path\_name, inverse\_path등 유틸리티 활용을 위해 사용)  2. file\_name으로부터 해당 파일의 original\_path 와 inverse\_path 경로 추출  2-2. 해당 경로를 바탕으로 **파일관리를 위한 리스트 Mlist 생성 + mlist\_update()**  3A. parameter 의 flag가 1인 경우 (d옵션이 있는 경우)  3A-2A. 해당 file\_name이 디렉토리파일이 아닌 경우 flag설정을 0 으로 변경 후 3B 과정 실행  3A-2B. 해당 file\_name이 디렉토리인 경우  3A-2B-1. original\_path를 기준으로 original\_search 함수 호출 (all 옵션 사용) -> original\_path 하부파일들에대한 rlist 추출  (단순 연결리스트로 해당 디렉토리 하부디렉토리 및 파일 모두 append)  3A-2B-2. backup\_path를 기준으로 backup\_Search 함수 호출 (all 옵션 사용)-> inverse\_path 의 하부파일들에 대한 mlist 추출  (완전 연결리스트 체이닝방식으로 상대적인 backup 경로에 있는 파일들을 모두 백업)  3A-2B-3. mlist와 rlist 의 file\_name이 동일한 경우를 체크하여 동일한 경우가 있다면 해시값을 일차적으로 비교  3A-2B-4A. 해시값이 동일한경우가 존재한다면 "$(filename의 path\_name) is already backuped" 을 출력하고 continue  3A-2B-4B. 해시값이 동일하지 않는다면 현재시각을 추가하여 backup 경로에 파일 백업. (node\_file\_cpy 함수 호출)  3B. parameter 의 flag가 0인 경우 (d옵션이 없는 경우)  3B-2A. 해당 file\_name이 디렉토리파일인 경우 "$(file\_name의 path\_name) is a directory file" 출력 후 종료.  3B-2B. 해당 file\_name이 디렉토리가 아닌 경우  3A-2B-1. original\_path를 기준으로 original\_search 함수 호출 (all 옵션 사용X)  -> original\_path 에 대해서만 rlist 추출  3A-2B-2. backup\_path를 기준으로 backup\_Search 함수 호출 (all 옵션 사용X)  -> inverse\_path 의 하부파일들에 대해서만 mlist 추출  3A-2B-3. mlist와 rlist 의 file\_name이 동일한 경우를 체크하여 동일한 경우가 있다면 해시값을 일차적으로 비교  3A-2B-4A. 해시값이 동일한경우가 존재한다면 "$(filename의 path\_name) is already backuped" 을 출력하고 함수종료  3A-2B-4B. 해시값이 동일하지 않는다면 현재시각을 추가하여 backup 경로에 파일 백업. (node\_file\_cpy 함수 호출) |     [ssu\_add 함수 내부 로직 순서도]  // 2. remove 계열 함수.  void ssu\_remove (char\* file\_name, int a\_flag);   * filename 기준으로 상대경로에 있는 파일에 대해서 삭제 진행. * 첫 번째 인자가 a\_flag가 들어오지 않았는데 디렉토리 파일이 들어온 경우 에러처리  |  |  | | --- | --- | | parameter 설명 | char\* file\_name  파일경로    int a\_flag  0: 플래그 없음  1: d 플래그 존재 | | 작동원리 | 1. filename에 해당하는 filenode 생성 (유틸리티 활용 목적)  1-A. filename 이 NULL 이라면, 백업 경로에만 존재한 파일이 있는지 확인을 위해 check\_backup\_file\_for\_remove 를 호출하여 mlist를 추출  1-B. 받아온 mlist 역시 NULL 이라면 함수 종료    2A. a\_flag가 존재한다면 ssu\_recover\_default 함수 호출 (all 옵션 사용)  -> 해당 filename을 기준으로 하위디렉토리/파일을 모두 mlist에 append  2A-1. 받아온 mlist의 file 리스트에 존재하는 파일들에 대하여 파일 총 크기 total\_file\_cnt 크기만큼 popleft\_mlist() 를 호출하여 mlist가 빌 때까지 삭제  2A-2. 삭제되면서 backup file removed 출력  2A-3. mlist에서 pop\_dict\_mlist를 호출하여 mlist의 딕셔너리 리스트를 삭제해가며 딕셔너리 삭제  2B. a\_flag가 존재하지 않는다면  2B-1A 해당 filename이 디렉토리 파일이라면 "$(파일경로) is a directory file" 출력 후 종료  2B-1B 해당 filename이 디렉토리파일이 아니라면  2B-1B-1. ssu\_recover\_default 함수 호출 (all 옵션 사용 X) 하여 mlist 받아옴  2B-1B-2A. 받아온 mlist의 file\_tail 의 same\_cnt가 아무것도 없다면 (요소가 1개)라면 해당 파일 삭제 후 "backup file removed" 출력 후 종료  2B-1B-2B. 받아온 mlist의 file\_tail 의 요소가 2개 이상이라면  2A-1B-2B-1. "Choose file to recover" 로 프롬포트를 띄운 후 사용자에게 입력받게 한다.  (정상값 (1~file 개수) 이 입력이 올 때 까지 무한 반복  2A-1B-2B-2. 입력받은 값이 정상값이면 해당 파일 삭제 후 "backup file removed" 출력 후 종료 |       [ssu\_remove함수 내부 로직 순서도]  void ssu\_remove\_all(); // scandir 사용.   * backup 디렉토리에 있는 모든 파일/디렉토리 에 대해서 삭제 진행 * backup 디렉토리에 파일이 하나도 존재하지 않는 경우 "no file(s) in the backup" 출력 후 종료 * 삭제 후 "backup directory cleared($(파일 삭제 개수) regular files and $(디렉토리 개수) subdirectories totally" 출력 * 디렉토리 삭제과정이 문제가 생길 수 있어 재귀적으로 모두 삭제될 때까지 삭제 진행  |  |  | | --- | --- | | 동작 원리 | 1. BACK\_PATH에 있는 파일들을 backup\_path 함수를 호출하여 모두 mlist에 저장  2. 해당 backup 디렉토리에 파일이 하나도 존재하지 않는다면  "no file(s) in the backup" 출력 후 종료  3. 받아온 mlist 의 mlist->total\_file\_cnt 가 0이될 때까지 popleft\_mlist 를 호출하여 mlist 요소 삭제  4. 이후 pop\_dict\_mlist 를 호출하여 딕셔너리 전체 삭제 (출력없음)  5. 사용이 끝난 파일관리리스트 free\_mlist 호출 후 반환 |     [ssu\_remove\_all 함수 구조도]   Mlist\* check\_backup\_file\_for\_remove(char\* file\_name, int flag\_a);  // 백업할 file\_name을 상대적인 BACKUP\_PATH 경로로 전환   * remove 할 file\_name이 백업파일에만 존재하고 실제경로에는 존재하지 않는지 검사하는 함수  |  |  | | --- | --- | | 인자 설명 | char\* file\_name,   * + Backup 경로와 상대적인 위치에 있는 경로   int flag\_a   * + 0 : -a 플래그 없음   + 1 : -a 플래그 있음 | | 동작 원리 | 1. mlist를 하나 할당받는다.  2. 해당 mlist의 acual\_path 를 backup\_path로 변환한다.  ex) /home/sp2023/go2 --> /home/junhyeongBKS/go2 로 변환  3. appnd\_samefile2 함수를 호출하여 해당 위치와 상대적인 위치에 있는 mlsit를 받아온다.  4. 만약, 해당 mlist가 NULL 이라면  5. 만약 해당 mlist가 딕셔너리 파일인데 flag\_a 가 없다면 "$(file\_name) is a dictionary file" 출력 후 NULL 리턴  5B. scandir\_makefile\_for\_remove 를 호출하여 해당 딕셔너리 하부 파일 들을 모두 받아와서 mlist로 파일 리스트 관리 | | 리턴값 | 해당 상대적인 위치에 파일이 없을 경우 NULL  해당 상대적인 위치에 파일이 존재하는 경우 파일리스트를 관리하고 있는 mlist 반환 |   // 3. recover 계열 함수.  // replace이용. (캡슐함수)  int ssu\_recover (char\* file\_name, int flag\_d, int flag\_n, char\* new\_name, int f\_opt);   * recover 과정을 수행하는 캡슐함수이다. -d 옵션과 -n 옵션을 동시에 처리할 수 있으며 옵션이 없는 경우도 처리할 수 있다. * n 옵션이 들어온다면 백업경로의 filenodes 대상으로 inverse\_path modify\_inversepath(); 함수를 내부적으로 처리하여 백업 경로 이름 변경 가능   (해당 함수 설명 아래에 존재)   * d 옵션이 들어온다면 file\_name의 하위 디렉토리/파일을 재귀적으로 모두 복구 * d 옵션이 들어왔는데 file\_name이 디렉토리 파일이 아니면 d옵션이 아닌 것처럼 동작 * d 옵션이 들어오지 않고 file\_name이 디렉토리 파일이면 "$(file\_name의 path\_name) is a directory file" 에러 메시지 출력 후 종료 * 백업 파일에만 존재하고 원래 경로에 존재하지 않는 파일에 대해서 복구  |  |  | | --- | --- | | parameter 설명 | char\* file\_name   * + inverse\_path 를 바꿀 노드     iint flag\_d  0: d 옵션 없음  1: d 옵션 활성화    int flag\_n  0: n 옵션 없음  1: n 옵션 활성화    char\* new\_name   * + file\_name의 inverse\_path를 변경할 새 경로 이름     int f\_opt  0: MD5  1: SHA-1 | | 동작원리 | 0. filename 에 대한 filenode 할당 (filenode 유틸리티 사용)  1. 만약 filename이 NULL 이라면 backup 경로에만 있는 파일인지 체크를 위해 check\_backup\_file 함수 호출  1B. filenode 재호출 후 NULL 이면 종료    2A. d 옵션이 있다면  2A-1A. d 옵션이 있는데 filename이 regular 파일이면 2B 과정으로 이동  2A-1B. d 옵션 && filename이 directory 파일이면 해당 path\_name에 대해 ssu\_recover\_default 함수 호출  2A-1B-1. n옵션이 존재한다면 modify\_inversepath 함수 호출 후 inverse\_path 경로 변경 (없으면 동작X)  2A-1B-2. 본격적인 recover 진행 처리 (rlist의 모든 요소에 대해 아래 과정 실행)  2A-1B-2A. rlist와 file\_name이 일치하는 mlist 를 순차적으로 순회하는데 그때, 특정 파일리스트 Mnode 의 same\_next가 NULL인 경우 (Mnode->same\_cnt개수가 1개인 경우)  2A-1B-2A-2 : 해당 리스트를 복구하며 출력문을 띄워줌  2A-1B-2B. rlist와 file\_name이 일치하는 mlist의 same\_cnt가 2개 이상인 경우  2A-1B-2B-1. "Choose file to recover" 로 프롬포트를 띄운 후 사용자에게 입력받게 한다.  (정상값 (1~file 개수) 이 입력이 올 때 까지 무한 반복  2A-1B-2B-2. 입력받은 값이 정상값이면 해당 파일과 백업파일의 해시값 비교  2A-1B-2B-2A-1A 해시값이 같으면 "is same file, so Don't backup\n" 출력 후 continue  2A-1B-2B-2A-1B 해시값이 다르면 "$(백업경로) backup file recover to $(백업될 경로)" 출력 후 continue  2A-1B-2B-2A-2. 선택된 시간 ex> 230313170313 은 저장되었다가 차후 같은 시간을 가지는 백업 파일에 대해 일괄 백업 진행    2B. d 옵션이 없다면  2B-1A. 해당 파일이 directory 파일이면 실행종료  2B-1B. 해당 path\_name에 대해 ssu\_recover\_default 함수 호출 (d\_flag=0 으로 실행되어 해당파일에 대해서만 실행)  2B-1B-1. n옵션이 존재한다면 modify\_inversepath 함수 호출 후 inverse\_path 경로 변경 (없으면 동작X)  2B-1B-2. 본격적인 recover 진행 처리 (한 파일에 대해서만 실행)  2B-1B-2A. rlist와 file\_name이 일치하는 mlist의 file\_tail 의 same\_next가 NULL 인 경우 (same\_cnt가 1개인경우)  2B-1B-2A-2 : 해당 리스트에 대하여 복구하고 출력문을 띄워줌  2B-1B-2B. rlist와 file\_name이 일치하는 mlist의 file\_tail 의 same\_next가 NULL 이 아닌 경우 (same\_cnt 가 2개 이상인 경우)  2A-1B-2B-1. "Choose file to recover" 로 프롬포트를 띄운 후 사용자에게 입력받게 한다.  (정상값 (1~file 개수) 이 입력이 올 때 까지 무한 반복  2A-1B-2B-2. 입력받은 값이 정상값이면 해당 파일과 백업파일의 해시값 비교  2A-1B-2B-2A-1A 해시값이 같으면 "is same file, so Don't backup\n" 출력 후 continue  2A-1B-2B-2A-1B 해시값이 다르면 "$(백업경로) backup file recover to $(백업될 경로)" 출력 후 continue  2A-1B-2B-2A-2. 선택된 시간 ex> 230313170313 은 저장되었다가 차후 같은 시간을 가지는 백업 파일에 대해 일괄 백업 진행 |     [ssu\_recover함수 내부 로직 순서도]    < 5. add, recover, remove 내부에서 사용되는 함수들>  int hash\_compare (Filenode\* a\_node, Filenode\* b\_node);   * a\_node 와 b\_node 의 해시를 비교해주는 함수. * return 값 같으면 1 다르면 0   int hash\_compare\_one (Filenode\* a\_node, char\* path\_name, int opt, int f\_opt);   * a\_node 와 path\_name 에 있는 해당 경로의 파일의 해시가 같은지 비교해주는 함수 * path\_node 는 내부적으로 filenode 를 할당하여 해시를 구한 후 사용   (filenode의 내부적으로 파일체크, 상대경로의 절대경로화, 해싱 등 여러 유틸리티를 제공해주기 때문)   * return 값 같으면 1 다르면 0   // file\_name => new\_name으로 inverse\_path 변경.  int modify\_inversepath (Filenode\* file\_name, char\* new\_name, int flag\_d);   file\_name의 invrese\_path를 new\_name의 경로로 변경시켜주는 함수 (recover -n 옵션을 지원하기 위해 만들어진 함수)   * return값 성공 시 1 실패시 0  |  |  | | --- | --- | | parameter 설명 | Filenode\* file\_name   * + inverse\_path 를 바꿀 노드     char\* new\_name   * + file\_name의 inverse\_path를 변경할 새 경로 이름     int flag\_d  0: d옵션 없음  1: d옵션 활성화   * + d 플래그가 설정되어있는지 확인 : d 플래그가 설정되어 있다면 **new\_name은 디렉토리명으로 인식**하여 변경함. | | 동작원리 | 0. 디렉토리거나 ACTUAL\_PATH 가 없는 filenode를 인자로 받았을 경우 에러메시지 출력 후 return  1. newname을 절대경로화 시켜준다. (현재 위치가 ~/go2 라고 하였을 때)  ex> hihi -> /home/junhyeong/go2/hihi  1A. flag\_d 가 1일 경우   * + newname 부분 뒤에 file\_name의 actual\_path 부분에서 시간부분을 떼고 추가시킨다.   ex> hihi -> /home/junhyeong/go2/**hihi/go2/a.c**  1B. flag\_d 가 0일 경우   * + filename의 inverse\_path를 new\_name으로 변경시켜버린다.   2. filename의 path\_name과 inverse\_path 의 타입 (.txt, .c 등등) 이 일치하는지 체크 후 일치하지 않으면 warning 발생시킴 |   MRlist\* ssu\_recover\_default (char\* file\_name, int d\_flag, int f\_opt); // d\_flag 설정 여부 확인   * 원래 file\_name 기준으로 백업폴더로 이동하여 recover할mlist, rlist들을 자동으로 구해오는 함수.  |  |  | | --- | --- | | parameter 설명 | char\* file\_name  : orignal\_path 경로 (상대/절대경로) : 함수 내부적으로 backup 경로로 변환하여 활용  int d\_flag  0 : d플래그 없음 (단일 파일백업)  1 : d플래그 활성화 (모든 파일 백업)    int f\_opt  0: MD5  1: SHA-1 | | 동작원리 | 0. filename 에 대한 filenode 할당 (filenode 유틸리티 사용)  1A. d\_flag 설정 시  1A-1A. 해당 file\_name이 디렉토리 파일이 아닌 경우 1B 과정으로 진행  1A-1B. 해당 file\_name에 해당하는 original\_search 함수 호출 (all 옵션으로 실행) (단순 연결리스트로 하부 디렉토리 탐색)  1A-1B-1. original\_search 함수로부터 받은 rlist 의 모든 요소에 대하여 append\_samefile() 호출  (상대적인 경로에 있는 같은 파일들을 전부 mlist에 mappend)  1B. d\_flag 설정 X 시  1B-1A. 해당 file\_name이 디렉토리 파일인 경우 "$(file\_name의 path\_name) is a directory file" 출력 후 종료.  1B-1B. 해당 file\_name에 해당하는 original\_search 함수 호출 (all 옵션으로 실행X) (해당 파일만 탐색)  1A-1B-1. original\_search 함수로부터 받은 rlist 의 모든 요소에 대하여 append\_samefile() 호출  (상대적인 경로에 있는 같은 파일들을 전부 mlist에 mappend) | | 리턴 값 | 성공 시 file\_name의 상대경로(백업경로) 에 있는 파일들 리스트(mlist) 반환 실패시 NULL |   // 같은 디렉토리에 대한 230227172413 15bytes 등 출력.  void print\_time\_and\_byte (Filenode\* node);   * 파일 경로명과 바이트수를 출력해주는 함수 (같은 파일 존재시 선택하게 할 때 사용) * 3자리마다 (‘, ‘) 를 넣어 주는 기능 구현, 구현방식은 배열에 담은 문자열 리스트를 3개씩 인덱스를 이동하며 ,를 붙여 sprintf를 재갱신하는 방식 사용   // origin path 에 있는 같은 이름의 파일 백업 파일 긁어오기.  void append\_samefile (Mlist\* mlist, char\* original\_file\_name, int f\_opt);   * Orignal Path에 존재하는 파일명과 같은 backup path의 파일들을 mlist에 mappend해주는 함수. * ex> ~/home/a.txt 가 존재한다면 ~/backup/a.txt\_230227172413, ~/backup/a.txt\_230227172453, ~/backup/a.txt\_230227172459 등등을 mappend 해준다.  |  |  | | --- | --- | | parameter 설명 | Mlist\* mlist  해당 Filenode 들을 appned 시킬 mlist 구조체    char\* original\_file\_name  Original 경로에 존재하는 파일    int f\_opt  0: MD5  1: SHA-1 | | 동작원리 | 0. 디렉토리거나 ACTUAL\_PATH 가 없는 filenode를 인자로 받았을 경우 에러메시지 출력 후 return  1. original\_file\_name를 경로로하는 filenode 할당 (file\_stat, inverse\_path 등 유틸리티 활용목적)  2. inverse\_path 라는 문자열에 original 경로의 상대적인 백업경로를 담는다.  3. original\_file\_name 의 inverse\_path에서 부모디렉토리를 scandir을 통해 탐색하여  (탐색 중간에는 filename->inverse\_path는 해당 경로에 있는 파일명으로 포인터를 활용하여 계속 변경된다.)  inverse\_path와 filename->inverse\_path와 파일명이 동일한 파일 있으면 mlist에 mappend해준다. |     int check\_backup\_file(char\* file\_name, int flag\_d); //상대적인 경로에 파일이 있는 경우 만들어줌   * filenode 는 기본적으로 Origianl 경로에 있는 파일만 구조체를 구해준다. 그러므로 백업경로에만 있고 원래경로에는 없는 파일의 경우 기본적으로 해당 original 경로에 파일을 생성시켜준 후 위의 유틸리티 함수들을 사용해야한다. 이를 실행시켜줄 보조함수다.  |  |  | | --- | --- | | parameter 설명 | char\* file\_name   * + original\_backup\_path 이름 (내부적으로 상대적인 backup경로 파일과 비교함)     int flag\_d  0: d 옵션 없음  1: d 옵션 활성화   * + d플래그 존재하는지 체크 | | 동작 원리 | 1. 문자열 처리를 통해 file\_name의 백업 경로에 해당 파일이 존재하는지 체크한다.  1-B. 만약 flag\_d 가 0 이고 && file\_name이 디렉토리 파일이라면 오류 출력 후 종료  2. Orignal Path에 파일이 존재한다면 Original Path에 파일명과 동일한 임시파일을 만들어준다.  3. 만약 file\_name이 디렉토리 파일이라면 디렉토리 밑에 있는 하위파일들을 전부 살려낸다.  재귀적으로 파일을 살리는 것은 아래 scandir\_makefile 를 재귀적으로 호출하여 구현하였다. |     void make\_file\_name(char\* file\_name);   * 해당 file\_name 의 경로로 임시 파일을 생성한다.  (위의 check\_backup\_file 함수와 , scandir\_makefile 와 같이 recover 을 할 때 백업경로에만 존재하는 파일을 처리할 때 사용)  |  |  | | --- | --- | | 인자설명 | char\* file\_name   * file 이름 | | 동작 원리 | 1. file\_name 의 부모 디렉토리까지 디렉토리가 존재하는지 체크 후 없다면 생성해준다.  make\_directory 함수 이용  2. 해당 file\_name의 이름으로 임시적으로 file\_name 파일을 생성한다. |   void scandir\_makefile(char \*parent\_folder, char\* original\_path);   * parent\_folder (백업경로) 에 존재하는 파일을 original\_path에 임시적으로 파일들을 생성해주는 함수. * 백업 경로에만 존재하고 실제경로에 존재하지 않는 경우 임시적으로 해당경로에 파일을 생성하는 함수.  |  |  | | --- | --- | | 인자설명 | char \*parent\_folder   * 백업경로의 부모 경로   char\* original\_path   * 임시 파일을 생성할 백업당할 경로 | | 동작 원리 | 1. scandir 을 이용하여 parent\_folder 안의 파일들을 모두 가져옴  2. 해당 파일이 Regular 파일인 경우   * + make\_directory 생성 후 make\_file\_name 함수를 호출하여 임시파일생성   3. 해당 파일이 .(현재디렉토리), ..(하위디렉토리) 를 제외한 디렉토리 파일인 경우 재귀호출 |     void scandir\_makefile\_for\_remove(Mlist\* mlist, char \*parent\_folder);   * parent\_folder 경로 (Backup 경로) 로 들어와서 해당 Regular 파일/ 디렉토리 파일(하부디렉토리 모두) 포함해 다 읽어온 후 mlist에 mappend 해주는 재귀함수.  |  |  | | --- | --- | | parameter 설명 | Mlist\* mlist  parent\_folder 경로의 하부디렉토리를 mappend할 mlist 리스트    char \*parent\_folder  백업경로에서 탐색할 경로 | | 동작 원리 | 1. 백업 경로 parent\_folder 를 scnadir 을 호출하여 dirent 구조체를 얻는다.  2. 받아온 dirent 구조체의 끝까지 탐색한다.  2A. dirent 구조체가 디렉토리인 경우  2A-1scandir\_makefile\_for\_remove 를 재귀호출하여 1번부터 다시 실행한다. (DFS)  2B. dirent 구조체가 Regular 파일인 경우mlist에 mappend 한다.  3. 다 쓰고 필요없어진 dirent는 free해준다. |     <6. 유지 및 추가 연구 함수>  int scandir\_filter (const struct dirent\* entry);   * scandir 함수 중 filter 인자로 넣어주는 함수 * 전역변수 scandir\_filename 가 entry->d\_name 의 문자열 중 일부라도 일치하면 True * 만약 /home/junhyeong/backup/a.c 를 찾고싶으면  /home/junhyeong/backup/a.c\_230314222304 같은 파일을 찾기 위함.   int basic\_filter (const struct dirent\* entry);   * scandir 함수 중 "." (현재파일) , ".." (이전파일) 을 거르고 나머지 파일들을 구해주는함수 * 실제 사용하기엔 맨 아래페이지에서 설명할 특정오류 때문에 사용 할 수 없었음.   int append\_samefile2 (Mlist\* mlist, char\* file\_path, int opt, int f\_opt);   * file\_path (절대경로만 지원) 의 마지막 파일명의 부모디렉토리를 탐사하여  내부적으로 scandir\_filter을 이용하여 file\_path 와 동일한 file\_name 을 갖는 파일 탐색함수  |  |  | | --- | --- | | parameter 설명 | Mlist\* mlist   * + file\_path와 <time>을 제외한 파일명이 같은 백업파일을 가져오기 위한 파일관리리스트   char\* file\_path   * + 백업 파일 경로   int opt (append 함수에 넘겨줄 인자때문에 사용 실상 1만 사용)  0 : Original\_path (백업할 경로) (사실상 사용하지 않음)  1 : Backuped\_path (백업 경로)  int f\_opt (append 함수에 넘겨줄 인자때문에 사용)  0 : MD5  1 : SHA-1 | | 동작 원리 | 1. file\_path 를 기준으로 문자열 처리를 통해 순수파일명과 부모디렉토리를 분할  ex> /home/junhyeong/backup/go2/file.cpp 라고 하였을 때  parent\_path = " /home/junhyeong/backup/go2"  only\_file\_path = "file.cpp"   * + strrchr 이용   2. 해당 부모디렉토리가 접근가능하면 scandir 함수에 scandir\_filter 함수 (위에 정의) 를 이용하여 같은 파일명을 추출  3. 만약 파일명이 같은 dirent 변수가 존재한다면 mappend | |

# 실행결과 (캡처)

## ssu\_backup

### 해시 인자 없이 입력된 경우

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### 정의 되지 않은 명령어 입력 시

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### help 입력 시

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### 프롬포트 상에서 엔터만 입력 시 프롬포트 재출력

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

## add

### add만 입력 시 / add 상대경로 / add 절대경로

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### -d 없이 디렉토리 / home 경로 벗어난 경우, backup 경로를 포함하고 있는 경우

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### add ./ -d (-d옵션 상대경로)

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### add /home/junhyeong/lect -d (-d옵션 상대경로)

* 텍스트, 명판이(가) 표시된 사진

  자동 생성된 설명

### 잘못된 옵션 들어왔을 경우

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

## remove

### remove 만 입력시

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### remove ssu\_exec.h (상대경로 + 동일한 파일, 백업본이 여러개 존재할 때)

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### remove /home/junhyeong/go2/ssu\_add.c (절대경로 + 동일한파일 삭제과정)

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### remove /home/junhyeong/onlybackup/a.cpp (백업파일에만 존재하는 파일 삭제)

* 텍스트이(가) 표시된 사진

  자동 생성된 설명텍스트이(가) 표시된 사진

  자동 생성된 설명

### emove 9.ProcessCommunicate -a (remove -a 옵션 사용시)

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### remove test\_a (디렉토리인데 -a 옵션을 안붙인 경우 예외처리)

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### remove -a -c (-a, -c 옵션을 동시에 받은경우)

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### remove -c (-c 옵션 예시)

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

## recover

### recover 만 입력시

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### recover ssu\_remove.c (recover <FILENAME: 상대경로>)

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### 복구 파일이 백업디렉토리에만 존재하는 경우

* 텍스트이(가) 표시된 사진

  자동 생성된 설명텍스트이(가) 표시된 사진

  자동 생성된 설명

### recover -d 옵션 ( + 해시값 동일한 경우 백업진행하지 않는 예시)

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### recover KMP.c -n (newname 설정 하지 않은경우 Usage 출력 후 종료)

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

### recover KMP.c -n new\_KMP (-n 옵션 중 뒤의 type이 일치하지 않는경우 waring 띄워줌) recover KMP.c -n new\_KMP.c (-n 옵션 예제)

* 텍스트이(가) 표시된 사진

  자동 생성된 설명텍스트이(가) 표시된 사진

  자동 생성된 설명

### recover ssu\_remove.c -n linux\_remove.c (-n 옵션 사용 중 백업파일이 여러 개 있는 경우)

* 텍스트이(가) 표시된 사진

  자동 생성된 설명텍스트이(가) 표시된 사진

  자동 생성된 설명

### ls

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

## vim

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

## help

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

## exit

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

## makefile

텍스트이(가) 표시된 사진

자동 생성된 설명

# 보완점

## ***scandir 파라미터 중 filter 사용의 실패***

* scandir 안의 필터를 사용하면 scandir 결과 중 필터링 한 dirent 노드만 추출할 수 있다고 알려져있다. 하지만, 실제로 필터링을 해본 결과 UUU, VV 와 같은 이상한 파일들을 dirent가 생성해내는 등의 오류로 인해서 신뢰성 (잔고장이 안나는 것) 을 높히기 위해서 scandir 필터링 방법의 알고리즘 구현을 하지 못했다는 아쉬움이 남는다.
* 텍스트이(가) 표시된 사진

  자동 생성된 설명
* 이러한 점들을 해결하는 방법으론 filter 내에서 filter 된 dirent 구조체의 d\_node 를 access 하여 확인하는 방법도 존재하나, 안정성과 dirent 구조체를 해제하는 과정에서 이중 free 문제등의 발생으로 실제 적용은 힘든 부분이 존재했다.

### scandir 필터 함수 예시

|  |
| --- |
| * scandir 내부에서 사용될 필터함수의 예시 : 실제로는 remove 함수 내부함수로밖에 사용되지 못했다. * 함수 설명 : 전역변수 scandir\_filename 에서 \_시간부분 을 제외한 부분이 같은 dirent 구조체만 추출하는 filter   int scandir\_filter (const struct dirent\* entry)  {      if (strlen(scandir\_filename) == 0 || entry == NULL)          return 0;      if (strstr(entry->d\_name, scandir\_filename) != NULL)      {          char tmp\_check [MAXFILELEN] = {0,};          strcpy(tmp\_check, entry->d\_name);          char \*seper = strrchr(tmp\_check, '\_');          if (seper != NULL)              \*seper = '\0';            if (strcmp(tmp\_check, scandir\_filename) == 0)              return 1;          else              return 0;      }      return 0;  } |

## ***함수 모듈화의 미숙함***

첫 번째 flist의 보완점과도 깊이 연관이 되어있는 문제이기도 한 함수 모듈화의 실패는 이번 설계과제에서 가장 아쉬운 점으로 꼽을 수 있다. remove에서 사용된 scandir\_makefile\_for\_remover, heck\_backup\_file\_for\_remove 함수의 경우 scandir\_makefile, check\_backup\_file 함수와 상당 부분 비슷한 기능을 큰 기능차이를 가지고 있어 함수를 재활용하기 쉽지 않은 부분이 있었다. 예를 들어, scandir\_makefile 함수의 경우 original path 를 파라미터로 받아 상대적인 backup 경로에 해당 original\_path가 존재하는데 정작 origianl\_path에 파일이 없다면 original path에 **파일을 만들어주는** 함수로 설계되었다.

재활용이 힘든 부분은 **‘파일을 만들어 주는**’ 부분을 분리시키지 않은 채 original\_path의 상대경로 파일이 존재하면 리스트로 뽑아내는 로직과 + **파일을 만들어 주는** 로직이 섞여있어 오히려 파일을 생성시키면 안되는 remove 과정에서 재활용을 할 수 없어 scandir\_makefile\_for\_remover 를 새로 정의해서 만들어야되는 문제점이 발생했다. 그리고 이미 코드가 꽤 많은부분 진행이된 상태여서 다른 함수끼리 독립성 (independence) 이 깨진 상태가 되어있었다. 그래서 어느 부분을 건들인다면 ssu\_backup 시스템 전체가 망가질 수 있어 쉽게 수정하지 못했다는 한계점이 남아있었다.

이러한 한계점을 바탕으로 얻은 교훈은 함수를 제작할 때 그 함수 내에서 기능3 = 기능1 + 기능2 +... 형태로 만들어지는 함수라면, 기능1, 기능2 에 대한 함수를 만들어 모듈화 시켜놓는 것이 프로그램 가독성 면에서나, 효율 면에서 유리할 수 있겠다는 교훈을 얻을 수 있었다.

# 참고 자료 및 각주

|  |
| --- |
| * [MD5] <https://ko.wikipedia.org/wiki/MD5> * [SHA-1] <https://ko.wikipedia.org/wiki/SHA-1> * [KMP 알고리즘] <https://bowbowbow.tistory.com/6> * [Rabin-karp 알고리즘] <https://yjg-lab.tistory.com/218> * [strrchr 사용법] <https://www.ibm.com/docs/ko/i/7.3?topic=functions-strrchr-locate-last-occurrence-character-in-string> * [scandir 사용법] <https://www.it-note.kr/16> * [openssl 설치방법] <https://sangchul.kr/462> * [libssl 설치] <https://i5i5.tistory.com/505> * [md5 깃허브] <https://github.com/Chronic-Dev/openssl/tree/master/crypto/md5> * [sha1 깃허브] <https://github.com/Chronic-Dev/openssl/tree/master/crypto/sha> * [md5\_init > EVP\_MD\_CTX 대체] <https://www.openssl.org/docs/man1.1.1/man3/EVP_MD_CTX_new.html> * [각 종 알고리즘 아이디어] : 이것이 취업을 위한 코딩테스트다 with 파이썬 , 나동빈 저자 * [자료구조 참고]: C++ 자료구조론, 이석호 저자 * [C언어 기초문법] 열혈 C 프로그래밍, 윤성우 저자 * [getopt 사용법1] <https://soooprmx.com/c-%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%A8-%ED%8C%8C%EB%9D%BC%EB%AF%B8%ED%84%B0%EB%A5%BC-%EC%B2%98%EB%A6%AC%ED%95%98%EB%8A%94-getopt-%EC%82%AC%EC%9A%A9%EB%B2%95/> * [getopt 사용법2] <https://man7.org/linux/man-pages/man3/getopt.3.html> * [getopt 사용법3] <https://badayak.com/entry/C%EC%96%B8%EC%96%B4-%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%A8-%EC%8B%A4%ED%96%89-%EC%9D%B8%EC%88%98-%EC%A0%95%EB%B3%B4-%EA%B5%AC%ED%95%98%EB%8A%94-%ED%95%A8%EC%88%98-getopt> * [getopt 사용법4] <https://reakwon.tistory.com/107> * [개발형 폰트 제공] <https://dlxdlx.tistory.com/3> * [makefile 제작방법] <https://velog.io/@hidaehyunlee/Makefile-%EB%A7%8C%EB%93%A4%EA%B8%B0> * [lcrypto makefile] <https://stackoverflow.com/questions/55579144/how-to-fix-makefile-to-properly-include-lcrypto-to-avoid-linker-error-undefine> * [openssl 빌드방법] <https://www.lesstif.com/system-admin/openssl-compile-build-6291508.html> |