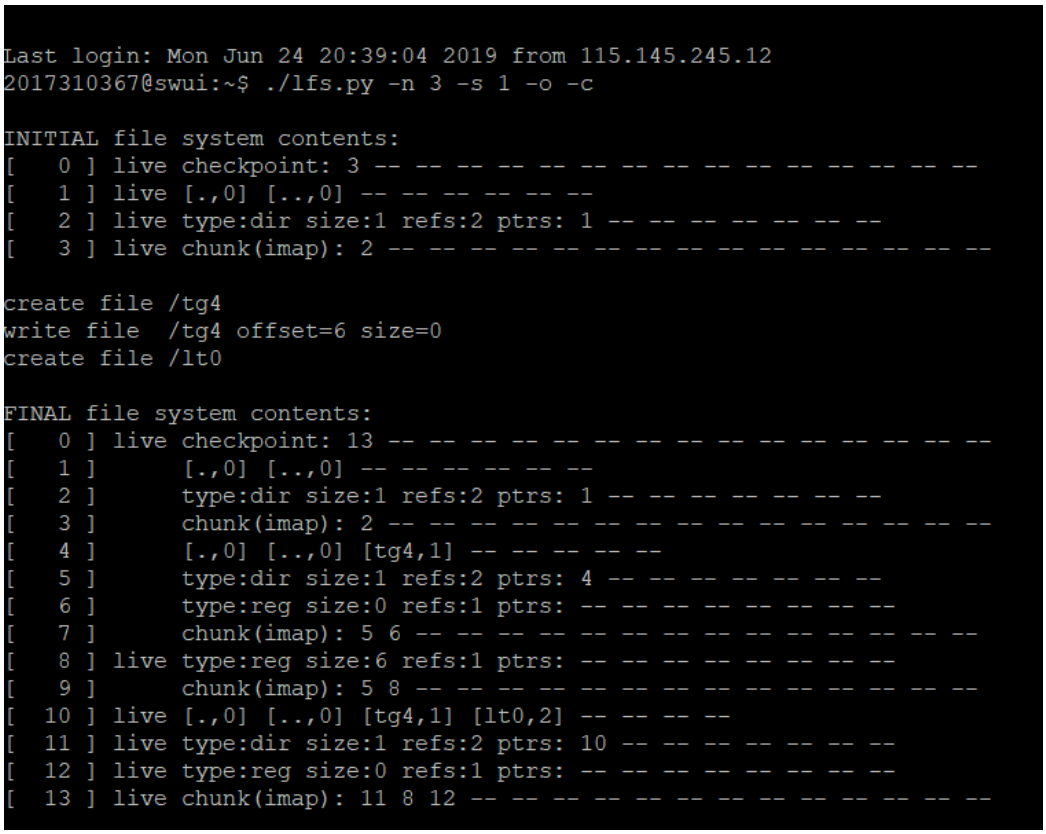
1. Run ./lfs.py -n 3, perhaps varying the seed (-s). Can you figure out which commands were run to generate the final file system contents? Can you tell which order those commands were issued? Finally, can you determine the liveness of each block in the final file system state? Use -o to show which commands were run, and -c to show the liveness of the final file system state. How much harder does the task become for you as you increase the number of commands issued (i.e., change -n 3 to -n 5)?



가장 먼저 intial file을 보면, 블록 0에는 체크 포인트가 적혀있다. 현재의 체크 포인트는 블록 3이다. 블록 3으로 가면 imap이 적혀있다. Inode가 가리키는 블록 2로 가면 현재 파일의 타입, 사이즈, 블록 정보를 알 수 있다. ptr에서 알려주는 데로 블록 1로 가면, 해당 파일의 데이터 정보를 볼 수 있다. initial에서는 자기자신을 의미하는 .과 부모를 의미하는 ..의 파일을 가지고 있음을 알 수 있다.

그러나 final file을 보면 체크포인트가 13으로 바뀌어있다. 즉, 블록 4~13까지는 새로운 명령에 의해 파일정보가 수정된 내용이 담겨있다. 블록 4를 보면 tg4라는 파일이 추가된 것을 알 수 있다. 따라서 첫번째 명령어는 create file /tg4임을 예상할 수 있다. 그 후 파일 메타데이터 정보를 변경하고 imap chunk 정보도 변경한다. 그 후 블록 8에서의 메타데이터를 보면 tg4 파일의 크기가 0

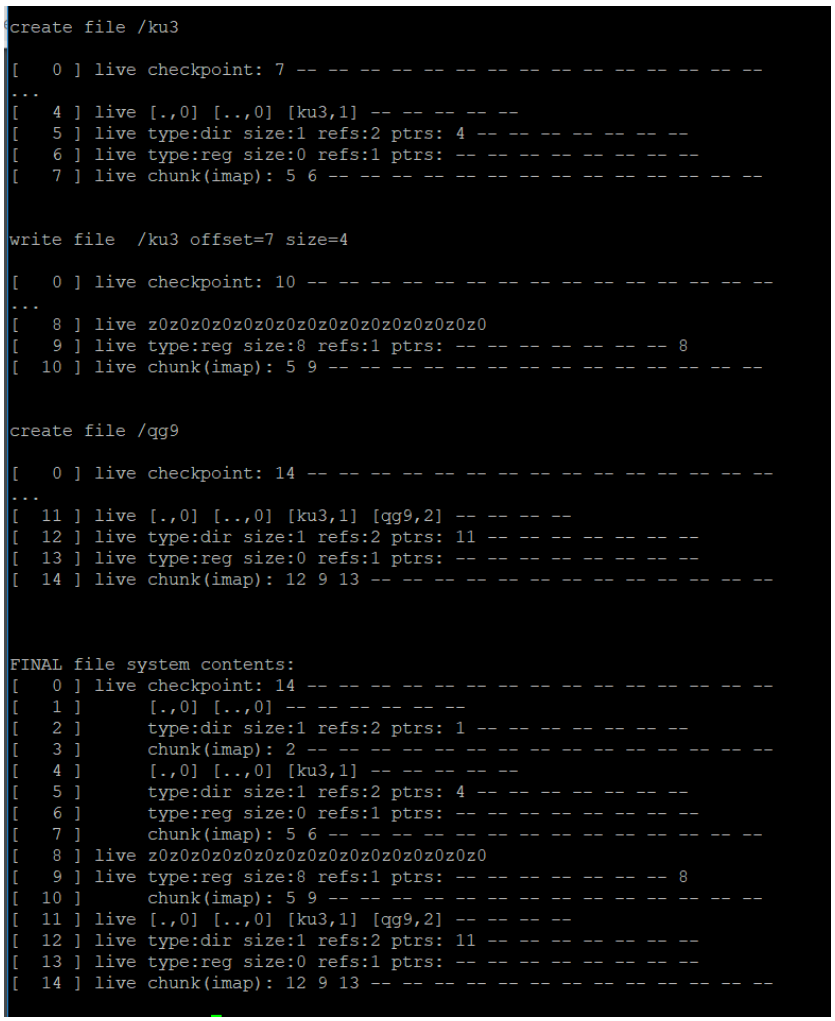
에서 6으로 바뀌어있다. 즉, tg4 write가 두번째 명령어라는 것을 예상할 수 있다. 그 후 imap chunk 정보를 변경했다. 블록 10을 보면 lt0이라는 파일이 추가된 것을 볼 수 있다. 따라서 세번째 명령어는 lt0 create 일 것이다. 그 이후 다시 파일 정보를 변경하고 imap chunk 정보를 변경한다. 이 과정까지 커밋이 완료되었으므로 변경이 완료된 블록 13이 새로운 체크포인트가 된 것이다.

```
2017310367@swui:~$ ./lfs.py -n 5 -s 1 -o -c

INITIAL file system contents:
[   0 ] live checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] live [.,0] [..,0] -- -- -- -- -- --
[   2 ] live type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] live chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

create file /tg4
write file  /tg4 offset=6 size=0
create file /lt0
write file  /lt0 offset=1 size=7
link file   /tg4 /oy3

FINAL file system contents:
[   0 ] live checkpoint: 26 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ]      [.,0] [..,0] -- -- -- -- -- --
[   2 ]      type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ]      chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ]      [.,0] [..,0] [tg4,1] -- -- -- -- --
[   5 ]      type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ]      type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ]      chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ]      type:reg size:6 refs:1 ptrs: -- -- -- -- -- -- -- --
[   9 ]      chunk(imap): 5 8 -- -- -- -- -- -- -- -- -- -- -- -- --
[  10 ]      [.,0] [..,0] [tg4,1] [lt0,2] -- -- -- --
[  11 ]      type:dir size:1 refs:2 ptrs: 10 -- -- -- -- -- -- --
[  12 ]      type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  13 ]      chunk(imap): 11 8 12 -- -- -- -- -- -- -- -- -- -- -- -- --
[  14 ] live f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0
[  15 ] live k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1k1
[  16 ] live a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2
[  17 ] live f3f3f3f3f3f3f3f3f3f3f3f3f3f3f3f3
[  18 ] live l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4l4
[  19 ] live m5m5m5m5m5m5m5m5m5m5m5m5m5m5m5m5
[  20 ] live g6g6g6g6g6g6g6g6g6g6g6g6g6g6g6g6
[  21 ] live type:reg size:8 refs:1 ptrs: -- 14 15 16 17 18 19 20
[  22 ]      chunk(imap): 11 8 21 -- -- -- -- -- -- -- -- -- -- -- -- --
[  23 ] live [.,0] [..,0] [tg4,1] [lt0,2] [oy3,1] -- -- --
[  24 ] live type:dir size:1 refs:2 ptrs: 23 -- -- -- -- -- -- --
[  25 ] live type:reg size:6 refs:2 ptrs: -- -- -- -- -- -- -- --
[  26 ] live chunk(imap): 24 25 21 -- -- -- -- -- -- -- -- -- -- -- -- --

2017310367@swui:~$
```

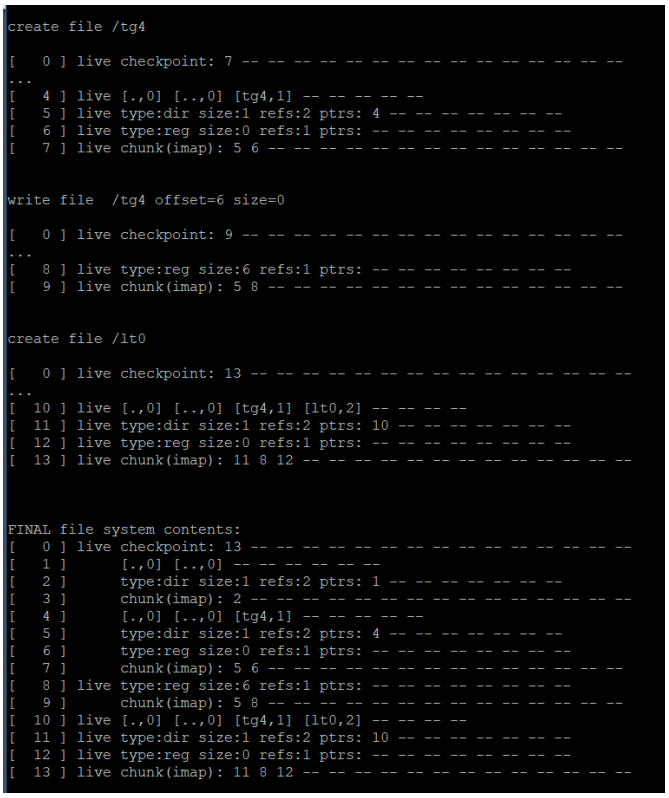
다음은 -n 플래그를 5로 바꾸었을 때의 결과이다. 즉, 명령어의 개수가 3개가 아니라 5개가 된 것이다. 역시 initial file에서는 블록 0~3에 걸쳐 초기 정보를 저장해두었다. Final file을 보면 체크포인트가 26인 것이 보인다. 즉, 4~26은 새로운 명령어들에 의해 파일이 변한 과정인 것이다. 블록 4에 의해 첫번째 명령어는 create tg4임을 알 수 있다. 블록 8에 의해 두번째 명령어는 write tg4임을 알 수 있다. 블록 10을 통해 세번째 명령어는 create lt0임을 알 수 있다. 블록 14~21을 통해 네번째 명령어는 lt0 파일에 write 임을 알 수 있다. 또한 블록 23을 통해 oy라는 파일이 새로 생겼으나 해당 메타데이터는 생기지 않은 것으로 보아 마지막 명령어은 link oy3임을 알 수 있다. 명령어의 개수가 많아질수록 파일 로그를 읽는 것이 점점 까다로워졌다.

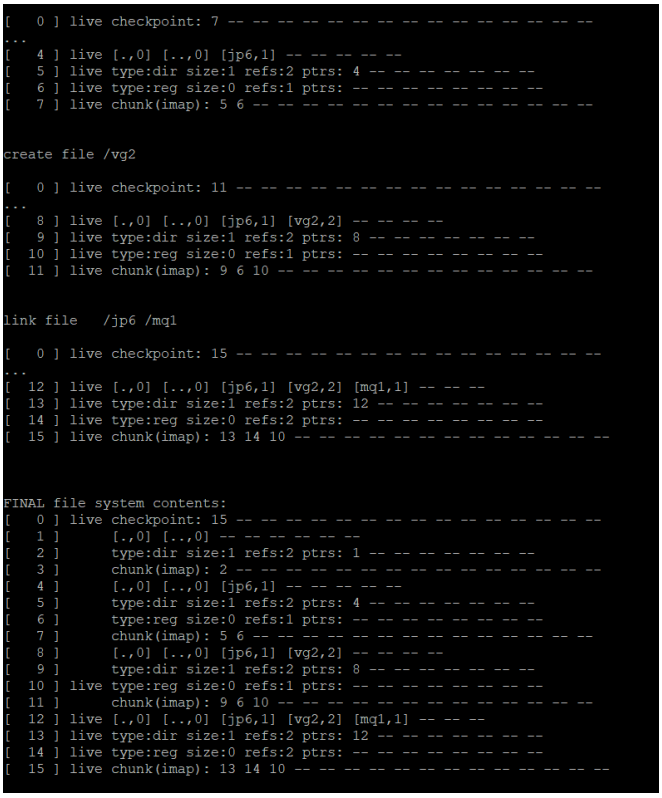
2. If you find the above painful, you can help yourself a little bit by showing the set of updates caused by each specific command. To do so, run ./lfs.py -n 3 -i. Now see if it is easier to understand what each command must have been. Change the random seed to get different commands to interpret (e.g., -s 1, -s 2, -s 3, etc.).

파일 로그를 조금 더 쉽게 읽기 위해 -I 플래그를 사용할 수 있다.



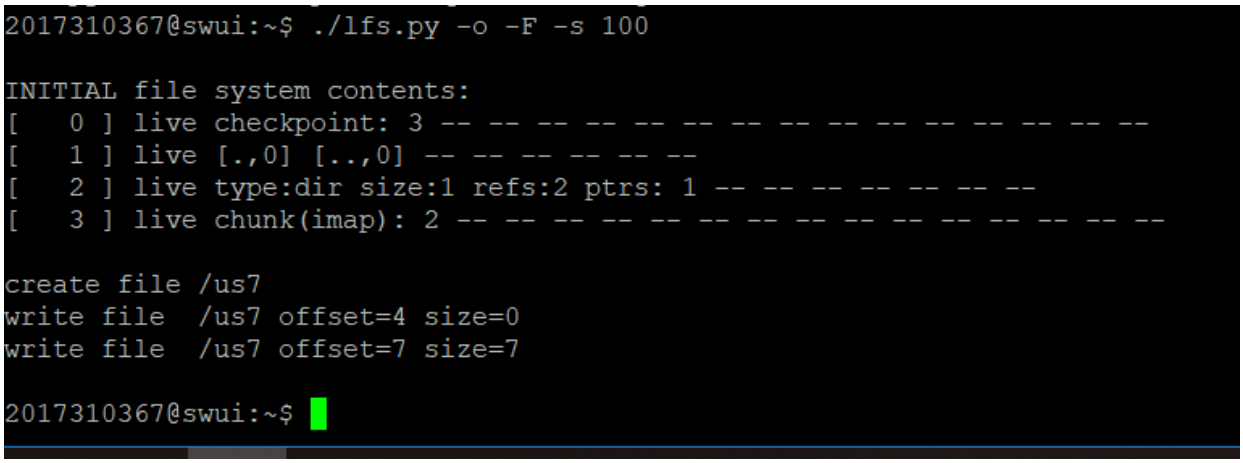
i플래그를 사용하자 파일 로그를 명령어에 따라 나누어 보여준다. 첫번째 명령어 create ku3에 의해 블록 4~7에 걸쳐 데이터를 블록에 쓰고, 메타데이터를 저장하고, imap chunk를 저장하는 것을 확인할 수 있다. 두번째 명령어 write ku3에 의해 블록 8~10에 걸쳐 데이터를 쓰고, 메타데이터를 저장하고, imap chunk를 저장하는 것을 확인할 수 있다. 역시 세번째 명령어 create qq9에 의해 블록 11~14에 걸쳐 데이터를 블록에 쓰고, 메타데이터를 저장하고, imap chunk를 저장하는 것을 확인할 수 있다. 그렇게 블록 14는 final file의 체크 포인트에 저장된다.





-s로 seed를 바꿔가며 파일의 변화를 살펴보았다. 역시 i 플러그를 넣으니 명령어 별로 블록을 나누어져 로그를 읽는 것이 훨씬 수월했다.

3. To further test your ability to figure out what updates are made to disk by each command, run the following: ./lfs.py -o -F -s 100 (and perhaps a few other random seeds). This just shows a set of commands and does NOT show you the final state of the file system. Can you reason about what the final state of the file system must be?



-F 플래그를 주자 final file을 보여주지 않는다. 그러나 명령어들을 통해 해당 결과를 예상할 수 있다. 가장 먼저 첫번째 명령어를 보면 us7이라는 파일을 만들었음을 알 수 있다. 따라서 블록 4에서는 파일데이터를 [.,0] [..,0] [us7,1] 로 변경할 것이다. 또한 블록 5에서는 type:dir size:1 refs:2 ptrs: 4 로 부모 디렉토리의 메타데이터를 바꾸고, ptr을 블록 4로 변경한다. 그 후 블록 6에서는 us7의 메타데이터를 만든다. 블록 7에서는 imap chunk를 업데이트한다. 두번째 명령어를 보면 us7 파일을 write했음을 알 수 있다. 따라서 블록 8에서는 us7의 메타데이터를 변경하고 블록 9에서는 imap chunk을 블록 8로 업데이트한다. 그 후 마지막 명령어를 보면 us7를 write한 것은 두번째 명령어와 같지만 size 변환까지 이루어졌다. 따라서 블록 10에서 해당 데이터를 작성하고 블록 11에서 us7의 메타데이터를 변경, 블록 12에서는 imap chunk을 블록 11로 업데이트한다. 따라서 이제 final file의 체크포인트는 11일 것이다.

4. Now see if you can determine which files and directories are live after a number of file and directory operations. Run tt ./lfs.py -n 20 -s 1 and then examine the final file system state. Can you figure out which pathnames are valid? Run tt ./lfs.py -n 20 -s 1 -c -v to see the results. Run with -o to see if your answers match up given the series of random commands. Use different random seeds to get more problems.

5. Now let's issue some specific commands. First, let's create a file and write to it repeatedly. To do so, use the -L flag, which lets you specify specific commands to execute. Let's create the file "/foo" and write to it four times: -L c,/foo:w,/foo,0,1:w,/foo,1,1:w,/foo,2,1:w,/foo,3,1 -o. See if you can determine the liveness of the final file system state; use -c to check your answers.

```
2017310367@swui:~$ ./lfs.py -n 1 -L c,/foo:w,/foo,0,1:w,/foo,1,1:w,/foo,2,1:
oo,3,1 -o -c

INITIAL file system contents:
[   0 ] live checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] live [.,0] [..,0] -- -- -- -- -- --
[   2 ] live type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- -- --
[   3 ] live chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

create file /foo
write file  /foo offset=0 size=1
write file  /foo offset=1 size=1
write file  /foo offset=2 size=1
write file  /foo offset=3 size=1

FINAL file system contents:
[   0 ] live checkpoint: 19 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ]      [.,0] [..,0] -- -- -- -- -- --
[   2 ]      type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- -- --
[   3 ]      chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ] live [.,0] [..,0] [foo,1] -- -- -- -- -- --
[   5 ] live type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- -- --
[   6 ]      type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ]      chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ] live v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0
[   9 ]      type:reg size:1 refs:1 ptrs: 8 -- -- -- -- -- -- -- --
[  10 ]      chunk(imap): 5 9 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  11 ] live t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0
[  12 ]      type:reg size:2 refs:1 ptrs: 8 11 -- -- -- -- -- --
[  13 ]      chunk(imap): 5 12 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  14 ] live k0k0k0k0k0k0k0k0k0k0k0k0k0k0k0k0
[  15 ]      type:reg size:3 refs:1 ptrs: 8 11 14 -- -- -- -- --
[  16 ]      chunk(imap): 5 15 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  17 ] live g0g0g0g0g0g0g0g0g0g0g0g0g0g0g0g0
[  18 ] live type:reg size:4 refs:1 ptrs: 8 11 14 17 -- -- -- --
[  19 ] live chunk(imap): 5 18 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

-L 플래그를 사용하여 명령어를 만들 수 있다. c,/foo 로 foo 파일을 만들고 w,/foo,0,1 로 foo 파일에 write(offset0, size 1)한다. w,/foo,1,1, w,/foo,2,1, w,/foo,3,1로 각각 foo 파일에 write(offset1, size 1), write(offset2, size 1), write(offset3, size 1)을 한다.

그 후 final file을 보면 블록 4~7에서 foo 파일을 만들고, 블록 8~18에 걸쳐 foo 파일에 write하고 있는 것을 볼 수 있다. 블록 8에서 데이터를 쓴 후 블록 9에서 메타데이터에 저장하고 블록 10에서 imap chnuk에 저장한다. 역시 세번째 명령어에 의해 블록 11에서 데이터를 쓴 후 블록 12에서 메타데이터에 저장하고 블록 13에서 imap chnuk에 저장한다. 네번째 명령어에 의해 블록 14에서 데이터를 쓴 후 블록 15에서 메타데이터에 저장하고 블록 16에서 imap chnuk에 저장한다. 마지막 명령어에 의해 블록 17에서 데이터를 쓴 후 블록 18에서 메타데이터에 저장하고 블록 19에서 imap chnuk에 저장한다. 체크포인트는 블록 19가 된다.

6. Now, let's do the same thing, but with a single write operation instead of four. Run ./lfs.py -o -L c,/foo:w,/foo,0,4 to create file "/foo" and write 4 blocks with a single write operation. Compute the liveness again, and check if you are right with -c. What is the main difference between writing a file all at once (as we do here) versus doing it one block at a time (as above)? What does this tell you about the importance of buffering updates in main memory as the real LFS does?

```
2017310367@swui:~$ ./lfs.py -n 1 -L c,/foo:w,/foo,0,4 -o -c

INITIAL file system contents:
[   0 ] live checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] live [.,0] [..,0] -- -- -- -- -- --
[   2 ] live type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] live chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

create file /foo
write file  /foo offset=0 size=4

FINAL file system contents:
[   0 ] live checkpoint: 13 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ]      [.,0] [..,0] -- -- -- -- -- --
[   2 ]      type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ]      chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ] live [.,0] [..,0] [foo,1] -- -- -- -- --
[   5 ] live type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ]      type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ]      chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ] live v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0
[   9 ] live t1t1t1t1t1t1t1t1t1t1t1t1t1t1t1t1
[  10 ] live k2k2k2k2k2k2k2k2k2k2k2k2k2k2k2k2
[  11 ] live g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3
[  12 ] live type:reg size:4 refs:1 ptrs: 8 9 10 11 -- -- -- --
[  13 ] live chunk(imap): 5 12 -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

이번에는 -L 플래그를 이용하여 c,/foo로 foo 파일을 만들고 w,/foo,0,4로 write(offset0, size 4) 하였다. 즉, 한꺼번에 데이터를 쓰는 것이다. 블록 4에서 foo 파일을 만들고 6, 7에서 메타데이터와

imap chunk를 업데이트한다. 그 후 블록 8~11에 걸쳐 데이터를 쓰고 블록 12에 메타데이터를 저장, 블록 13에 imap chunk를 저장한다. 문제 5와 결과는 비슷하지만 그 과정은 훨씬 짧은 것을 알 수 있다.

7. Let's do another specific example. First, run the following: ./lfs.py -L c,/foo:w,/foo,0,1. What does this set of commands do? Now, run ./lfs.py -L c,/foo:w,/foo,7,1. What does this set of commands do? How are the two different? What can you tell about the size field in the inode from these two sets of commands?

```
2017310367@swui:~$ ./lfs.py -L c,/foo:w,/foo,0,1 -o -c

INITIAL file system contents:
[   0 ] live checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] live [.,0] [..,0] -- -- -- -- -- --
[   2 ] live type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] live chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

create file /foo
write file  /foo offset=0 size=1

FINAL file system contents:
[   0 ] live checkpoint: 10 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ]      [.,0] [..,0] -- -- -- -- -- --
[   2 ]      type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ]      chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ] live [.,0] [..,0] [foo,1] -- -- -- -- --
[   5 ] live type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ]      type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ]      chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ] live v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0
[   9 ] live type:reg size:1 refs:1 ptrs: 8 -- -- -- -- -- -- --
[  10 ] live chunk(imap): 5 9 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

-L c,/foo:w,/foo,0,1를 통해 블록 4~7에 걸쳐 foo 파일을 만들고 블록 8~10에 걸쳐 foo에 write 작업을 한 것을 확인할 수 있다. Offset이 0이기 때문에 ptr을 보면 0번째 offset에 블록 8이 적혀 있음을 볼 수 있다.

```
2017310367@swui:~$  ./lfs.py -L c,/foo:w,/foo,7,1 -o -c

INITIAL file system contents:
[   0 ] live checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] live [.,0] [..,0] -- -- -- -- -- --
[   2 ] live type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] live chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

create file /foo
write file  /foo offset=7 size=1

FINAL file system contents:
[   0 ] live checkpoint: 10 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ]      [.,0] [..,0] -- -- -- -- -- --
[   2 ]      type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ]      chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ] live [.,0] [..,0] [foo,1] -- -- -- -- --
[   5 ] live type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ]      type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ]      chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ] live v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0
[   9 ] live type:reg size:8 refs:1 ptrs: -- -- -- -- -- -- -- 8
[  10 ] live chunk(imap): 5 9 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

역시 -L c,/foo:w,/foo,7,1를 통해 블록 4~7에 걸쳐 foo 파일을 만들고 블록 8~10에 걸쳐 foo에 write 작업을 한 것을 확인할 수 있다. 그러나 이 경우에는 offset이 7이기 때문에 ptr을 보면 7번째 offset에 블록 8이 적혀있음을 볼 수 있다.

8. Now let's look explicitly at file creation versus directory creation. Run simulations ./lfs.py -L c,/foo and ./lfs.py -L d,/foo to create a file and then a directory. What is similar about these runs, and what is different?

```
2017310367@swui:~$ ./lfs.py -L c,/foo -c -o

INITIAL file system contents:
[   0 ] live checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] live [.,0] [..,0] -- -- -- -- -- --
[   2 ] live type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] live chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

create file /foo

FINAL file system contents:
[   0 ] live checkpoint: 7 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ]      [.,0] [..,0] -- -- -- -- -- --
[   2 ]      type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ]      chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ] live [.,0] [..,0] [foo,1] -- -- -- -- --
[   5 ] live type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ] live chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

./lfs.py -L c,/foo를 실행했을 때의 결과이다.

```
2017310367@swui:~$ ./lfs.py -L d,/foo -o -c

INITIAL file system contents:
[   0 ] live checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] live [.,0] [..,0] -- -- -- -- -- --
[   2 ] live type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] live chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

create dir  /foo

FINAL file system contents:
[   0 ] live checkpoint: 8 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ]      [.,0] [..,0] -- -- -- -- -- --
[   2 ]      type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ]      chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ] live [.,0] [..,0] [foo,1] -- -- -- -- --
[   5 ] live [.,1] [..,0] -- -- -- -- -- --
[   6 ] live type:dir size:1 refs:3 ptrs: 4 -- -- -- -- -- -- --
[   7 ] live type:dir size:1 refs:2 ptrs: 5 -- -- -- -- -- -- --
[   8 ] live chunk(imap): 6 7 -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

./lfs.py -L d,/foo를 실행했을 때의 결과이다.

첫번째 경우, foo는 그저 파일이기 때문에 블록 4~7까지 파일을 만들고, 메타데이터를 만들고, imap chnuk에 저장한다. 반면, 두번째 경우 foo는 파일이 아니라 디렉토리이기 때문에 블록 4에서는 foo의 부모 디렉토리를 수정하고, 블록5에서는 foo 디렉토리를 만든다. 그 후 블록 6에서는 foo의 부모 디렉토리의 메타데이터를 수정하고 블록 7에서는 foo의 메타데이터를 수정한다. 블록 8에서는 imap chunk를 업데이트한다.

9. The LFS simulator supports hard links as well. Run the following to study how they work: ./lfs.py -L c,/foo:l,/foo,/bar:l,/foo,/goo -o -i. What blocks are written out when a hard link is created? How is this similar to just creating a new file, and how is it different? How does the reference count field change as links are created?

```
create file /foo

[   0 ] live checkpoint: 7 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
...
[   4 ] live [.,0] [..,0] [foo,1] -- -- -- -- --
[   5 ] live type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ] live chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- --


link file   /foo /bar

[   0 ] live checkpoint: 11 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
...
[   8 ] live [.,0] [..,0] [foo,1] [bar,1] -- -- -- --
[   9 ] live type:dir size:1 refs:2 ptrs: 8 -- -- -- -- -- -- --
[  10 ] live type:reg size:0 refs:2 ptrs: -- -- -- -- -- -- -- --
[  11 ] live chunk(imap): 9 10 -- -- -- -- -- -- -- -- -- -- -- -- --


link file   /foo /goo

[   0 ] live checkpoint: 15 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
...
[  12 ] live [.,0] [..,0] [foo,1] [bar,1] [goo,1] -- -- --
[  13 ] live type:dir size:1 refs:2 ptrs: 12 -- -- -- -- -- -- --
[  14 ] live type:reg size:0 refs:3 ptrs: -- -- -- -- -- -- -- --
[  15 ] live chunk(imap): 13 14 -- -- -- -- -- -- -- -- -- -- -- -- --



FINAL file system contents:
[   0 ] live checkpoint: 15 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ]      [.,0] [..,0] -- -- -- -- -- --
[   2 ]      type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ]      chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ]      [.,0] [..,0] [foo,1] -- -- -- -- --
[   5 ]      type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ]      type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ]      chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ]      [.,0] [..,0] [foo,1] [bar,1] -- -- -- --
[   9 ]      type:dir size:1 refs:2 ptrs: 8 -- -- -- -- -- -- --
[  10 ]      type:reg size:0 refs:2 ptrs: -- -- -- -- -- -- -- --
[  11 ]      chunk(imap): 9 10 -- -- -- -- -- -- -- -- -- -- -- -- --
[  12 ] live [.,0] [..,0] [foo,1] [bar,1] [goo,1] -- -- --
[  13 ] live type:dir size:1 refs:2 ptrs: 12 -- -- -- -- -- -- --
[  14 ] live type:reg size:0 refs:3 ptrs: -- -- -- -- -- -- -- --
[  15 ] live chunk(imap): 13 14 -- -- -- -- -- -- -- -- -- -- -- -- --
```

-L c,/foo:l,/foo,/bar:l,/foo,/goo를 -i로 자세히 살펴본 결과이다. 가장 먼저 foo 파일을 만들고 foo 파일에 bar 파일을 link한다. 그 후 역시 foo 파일에 goo 파일을 link한다. 해당 link들은 hard link로 파일을 데이터에 저장하고, 메타데이터를 만들고, chunk imap을 업데이트 한다는 점에서 비슷하다. 그러나 reference count를 세는 ref의 값이 하드링크가 만들어질수록 하나씩 증가한다는 점에서 다르다. 하드링크가 하나씩 생길 때마다 ref가 1씩 증가하고 있음을 볼 수 있다.

10. LFS makes many different policy decisions. We do not explore many of them here – perhaps something left for the future – but here is a simple one we do explore: the choice of inode number. First, run ./lfs.py -p c100 -n 10 -o -a s to show the usual behavior with the "sequential" allocation policy, which tries to use free inode numbers nearest to zero. Then, change to a "random" policy by running ./lfs.py -p c100 -n 10 -o -a r (the -p c100 flag ensures 100 percent of the random operations are file creations). What on-disk differences does a random policy versus a sequential policy result in? What does this say about the importance of choosing inode numbers in a real LFS?

```
2017310367@swui:~$ ./lfs.py -p c100 -n 10 -o -a s -c

INITIAL file system contents:
[   0 ] live checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] live [.,0] [..,0] -- -- -- -- -- --
[   2 ] live type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] live chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

create file /kg5
create file /hm5
create file /ht6
create file /zv9
create file /xr4
create file /px9
create file /gu5
create file /kv6
create file /wg3
create file /og9

FINAL file system contents:
[   0 ] live checkpoint: 43 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ]      [.,0] [..,0] -- -- -- -- -- --
[   2 ]      type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ]      chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ]      [.,0] [..,0] [kg5,1] -- -- -- -- --
[   5 ]      type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ]      chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ]      [.,0] [..,0] [kg5,1] [hm5,2] -- -- -- --
[   9 ]      type:dir size:1 refs:2 ptrs: 8 -- -- -- -- -- -- --
[  10 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  11 ]      chunk(imap): 9 6 10 -- -- -- -- -- -- -- -- -- -- -- -- --
[  12 ]      [.,0] [..,0] [kg5,1] [hm5,2] [ht6,3] -- -- --
[  13 ]      type:dir size:1 refs:2 ptrs: 12 -- -- -- -- -- -- --
[  14 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  15 ]      chunk(imap): 13 6 10 14 -- -- -- -- -- -- -- -- -- -- -- --
[  16 ]      [.,0] [..,0] [kg5,1] [hm5,2] [ht6,3] [zv9,4] -- --
[  17 ]      type:dir size:1 refs:2 ptrs: 16 -- -- -- -- -- -- --
[  18 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  19 ]      chunk(imap): 17 6 10 14 18 -- -- -- -- -- -- -- -- -- -- --
[  20 ]      [.,0] [..,0] [kg5,1] [hm5,2] [ht6,3] [zv9,4] [xr4,5] --
[  21 ]      type:dir size:1 refs:2 ptrs: 20 -- -- -- -- -- -- --
[  22 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  23 ]      chunk(imap): 21 6 10 14 18 22 -- -- -- -- -- -- -- -- -- --
[  24 ] live [.,0] [..,0] [kg5,1] [hm5,2] [ht6,3] [zv9,4] [xr4,5] [px9,6]
[  25 ]      type:dir size:1 refs:2 ptrs: 24 -- -- -- -- -- -- --
[  26 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  27 ]      chunk(imap): 25 6 10 14 18 22 26 -- -- -- -- -- -- -- -- --
[  28 ]      [gu5,7] -- -- -- -- -- -- --
```

./lfs.py -p c100 -n 10 -o -a s, 즉 순차적 실행을 했을 때의 결과이다.

```
2017310367@swui:~$ ./lfs.py -p c100 -n 10 -o -a r -c

INITIAL file system contents:
[   0 ] live checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] live [.,0] [..,0] -- -- -- -- -- --
[   2 ] live type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] live chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

create file /kg5
create file /hm5
create file /ht6
create file /zv9
create file /xr4
create file /px9
create file /gu5
create file /kv6
create file /wg3
create file /og9

FINAL file system contents:
[   0 ] live checkpoint: 52 38 -- -- -- 23 -- 13 53 -- -- 48 8 -- 33 --
[   1 ]      [.,0] [..,0] -- -- -- -- -- --
[   2 ]      type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ]      chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ]      [.,0] [..,0] [kg5,205] -- -- -- -- --
[   5 ]      type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ]      chunk(imap): 5 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ] live chunk(imap): -- -- -- -- -- -- -- -- -- -- -- -- -- -- 6 -- --
[   9 ]      [.,0] [..,0] [kg5,205] [hm5,114] -- -- -- --
[  10 ]      type:dir size:1 refs:2 ptrs: 9 -- -- -- -- -- -- --
[  11 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  12 ]      chunk(imap): 10 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  13 ] live chunk(imap): -- -- 11 -- -- -- -- -- -- -- -- -- -- -- -- --
[  14 ]      [.,0] [..,0] [kg5,205] [hm5,114] [ht6,20] -- -- --
[  15 ]      type:dir size:1 refs:2 ptrs: 14 -- -- -- -- -- -- --
[  16 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  17 ]      chunk(imap): 15 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  18 ]      chunk(imap): -- -- -- -- 16 -- -- -- -- -- -- -- -- -- -- --
[  19 ]      [.,0] [..,0] [kg5,205] [hm5,114] [ht6,20] [zv9,81] -- --
[  20 ]      type:dir size:1 refs:2 ptrs: 19 -- -- -- -- -- -- --
[  21 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  22 ]      chunk(imap): 20 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  23 ] live chunk(imap): -- 21 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  24 ]      [.,0] [..,0] [kg5,205] [hm5,114] [ht6,20] [zv9,81] [xr4,130] --
[  25 ]      type:dir size:1 refs:2 ptrs: 24 -- -- -- -- -- -- --
[  26 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  27 ]      chunk(imap): 25 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

./lfs.py -p c100 -n 10 -o -a r, 즉 랜덤실행을 했을 때의 결과이다. Inode순 대로 순차적 실행했을 때 훨씬 좋은 결과를 보이는 것을 알 수 있다. 순차실행을 했을 때에는 43개의 블록밖에 사용하지 않았지만 랜덤실행을 했을 때는 53개의 블록을 사용하였다. 따라서 아이노트를 어떻게 결정하느냐가 lfs 성능에 영향을 주며 실제 lfs에서 주요한 역할을 함을 알 수 있다.

11. One last thing we've been assuming is that the LFS simulator always updates the checkpoint region after each update. In the real LFS, that isn't the case: it is updated periodically to avoid long seeks. Run ./lfs.py -N -i -o -s 1000 to see some operations and the intermediate and final states of the file system when the checkpoint region isn't forced to disk. What would happen if the checkpoint region is never updated? What if it is updated periodically? Could you figure out how to recover the file system to the latest state by rolling forward in the log?

```
INITIAL file system contents:
[   0 ] live checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] live [.,0] [..,0] -- -- -- -- -- --
[   2 ] live type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] live chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

create file /om1

[   4 ] live [.,0] [..,0] [om1,1] -- -- -- -- --
[   5 ] live type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ] live chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- -- --


delete file /om1

[   8 ] live [.,0] [..,0] -- -- -- -- -- --
[   9 ] live type:dir size:1 refs:2 ptrs: 8 -- -- -- -- -- -- --
[  10 ] live chunk(imap): 9 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --


create file /eg5

[  11 ] live [.,0] [..,0] [eg5,1] -- -- -- -- --
[  12 ] live type:dir size:1 refs:2 ptrs: 11 -- -- -- -- -- -- --
[  13 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  14 ] live chunk(imap): 12 13 -- -- -- -- -- -- -- -- -- -- -- -- -- --



FINAL file system contents:
[   0 ] live checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ]      [.,0] [..,0] -- -- -- -- -- --
[   2 ]      type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ]      chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ]      [.,0] [..,0] [om1,1] -- -- -- -- --
[   5 ]      type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ]      type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ]      chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ]      [.,0] [..,0] -- -- -- -- -- --
[   9 ]      type:dir size:1 refs:2 ptrs: 8 -- -- -- -- -- -- --
[  10 ]      chunk(imap): 9 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  11 ] live [.,0] [..,0] [eg5,1] -- -- -- -- --
[  12 ] live type:dir size:1 refs:2 ptrs: 11 -- -- -- -- -- -- --
[  13 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  14 ] live chunk(imap): 12 13 -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

./lfs.py -N -i -o -s 1000을 했을 때 인덱스 오류가 일어나 ./lfs.py -N -i -o -s 999로 대체하였다. 보통 체크포인트는 가장 마지막에 업데이트된 imap chunk 를 가리키지만 해당 결과에서는 블록 3을 가리키고 있는 것을 확인할 수 있다. 체크포인트는 파일을 복구할 때 사용된다. 만약 체크포인트를 업데이트 하지 않으면 체크포인트 이후 해당하는 로그들은 복구할 수 없다. 그러나 만약 체크포인트를 너무 자주 업데이트 한다면 성능에 문제를 준다. 만약 파일을 disk에 저장하는 데에 문제가 생기면 체크포인트에 해당하는 블록을 찾아가서 로그를 차례대로 읽어 파일을 undo, redo 할 수 있다. 이러한 undo, redo를 토대로 파일을 복구할 수 있게 된다.