Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing

FS 2018 Prof. R. Wattenhofer

Computer Engineering II

Solution to Exercise Sheet Chapter 6

Basic

1 HDDs

a) For a random workload, we can assume that the head is equally likely to be on each page. As the outer tracks have more pages, the head is more likely to be on the outer tracks than in the middle tracks. Therefore, a new request that is a bit towards the outer track than in the middle track is served faster. (To calculate this exactly, one can assume that each track has data proportional to its distance from the center. Then, solve the equation involving the unknown track to which the expected seek time from any page is minimized.)

b) SSTF: starvation occurs if requests for a few closely grouped tracks come in frequently. For example: if requests for the outermost tracks are arriving so frequently that there always exists one unprocessed request at any time, then any request for the innermost track will not be processed.
SPTF: any form of continuous sequential access over few tracks (going back and forth) will result in requests for far away tracks never being processed.
SCAN, C-SCAN: if requests for the current track are being issued all the time, then neither of these modes will ever leave the track.
F-SCAN solves the problem for SCAN and C-SCAN by not considering requests that come in during a pass over the platter. The whole set of requests that exist at the beginning of a pass will be processed by the end of it, and requests coming in during the pass will only be procssed in the next pass.

c) It does not allow for any simple optimizations, such as grouping requests to sectors that lie close to each other or minimizing positioning times.

d) We need three values to find out how long it takes to read one sector: the time to seek the correct track T_{seek}, the time to rotate the platter to the correct position T_{rot}, and the time to read the sector T_{transfer}.

HDD 1 with 9000 rpm:
$T_{\text{seek}} = 5ms$
For random access, T_{rot} is half the time it takes for the disk to rotate since in expectation, a random destination is exactly half a rotation away.

$T_{\text{rot}} = \frac{1}{2}\frac{1}{9000}min \approx 3.33ms$
$T_{\text{transfer}} = \frac{4KB}{120MB/s} \approx 32.6\mu s$

This gives a rate of I/O of roughly $R_{\text{I/O}}\frac{4KB}{5ms+3.33ms+0.0326ms} \approx 0.478\frac{MB}{s}$. Since all sectors' positions are independent of each other, we can just divide the amount of data we want to read by the rate of I/O to get the time it will take in expectation to perform the read. Therefore, it takes HDD 1 roughly $\frac{200MB}{0.478MB/s} \approx 418s$ to process a 200MB random access workload.

HDD 2 with 5400 rpm:
$T_{\text{seek}} = 3ms$
$T_{\text{rot}} = \frac{1}{2}\frac{1}{5400}min \approx 5.56ms$
$T_{\text{transfer}} = \frac{4KB}{120MB/s} \approx 32.6\mu s$
This gives a rate of I/O of roughly $R_{\text{I/O}}\frac{4KB}{3ms+5.56ms+0.0326ms} \approx 0.466\frac{MB}{s}$. It takes HDD 2 roughly $\frac{200MB}{0.466MB/s} \approx 429s$ to process a 200MB random access workload.

2 SSDs

Block	0				1			
Page	0	1	2	3	4	5	6	7
Content	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
State	i	i	i	i	i	i	i	i

We start with an empty SSD with all cells being invalid (i). Programmed cells will be marked as valid (v), programmable ones as erased (e). We have to erase any invalid pages before programming them.

To save some space, we will only list the content and state of pages after this, plus the mapping table for the page-mapped SSD.

The command `write(x) content Y` comes from the OS and means "write content `Y` to logical address `x`". A logical address is an abstraction presented to the OS by the storage device; the OS only sees an array of disk positions it can operate on. This way, the OS can deal with completely different devices without knowing anything but this abstraction about them. The device itself translates every logical address to a physical address, i.e. in the case of SSDs: every logical address is mapped to some page of the SSD.

For direct mapped SSDs, logical addresses are directly used as physical addresses. For page-mapped SSDs, the FTL stores which logical page is mapped to which physical page. We will denote "logical address `a` is located at physical address `b`" by "`a → b`".

Garbage collection only happens in SSDs that use an FTL; its purpose is to make pages that are invalid useable again.

Direct mapped SSD:

write(0) content A

Content	A	⊥	⊥	⊥	⊥	⊥	⊥	⊥
State	v	e	e	e	i	i	i	i

write(6) content B

Content	A	⊥	⊥	⊥	⊥	⊥	B	⊥
State	v	e	e	e	e	e	v	e

write(4) content C

Content	A	⊥	⊥	⊥	C	⊥	B	⊥
State	v	e	e	e	v	e	v	e

write(1) content D

Content	A	D	$\perp$	$\perp$	C	$\perp$	B	$\perp$
State	v	v	e	e	v	e	v	e

write(0) content E

First erase block 0, remember what was in page 1; the SSD has some RAM built in that can be used for buffering, or to remember the contents of a block that is about to be erased as in our scenario.

Content	$\perp$	$\perp$	$\perp$	$\perp$	C	$\perp$	B	$\perp$
State	e	e	e	e	v	e	v	e

Now program the new content of page 0 and the old content of page 1.

Content	E	D	$\perp$	$\perp$	C	$\perp$	B	$\perp$
State	v	v	e	e	v	e	v	e

write(4) content F

erase first

Content	E	D	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
State	v	v	e	e	e	e	e	e

program

Content	E	D	$\perp$	$\perp$	F	$\perp$	B	$\perp$
State	v	v	e	e	v	e	v	e

Page-mapped SSD:

The row "Table" contains the pairs "logical page → physical page".

write(0) content A

Table	$0 \to 0$							
Content	A	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
State	v	e	e	e	i	i	i	i

write(6) content B

Table	$0 \to 0$, $6 \to 1$							
Content	A	B	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
State	v	v	e	e	i	i	i	i

write(4) content C

Table	$0 \to 0$, $6 \to 1$, $4 \to 2$							
Content	A	B	C	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
State	v	v	v	e	i	i	i	i

write(1) content D

Table	0 → 0, 6 → 1, 4 → 2, 1 → 3							
Content	*A*	*B*	*C*	*D*	⊥	⊥	⊥	⊥
State	v	v	v	v	i	i	i	i

write(0) content E

Table	0 → 4, 6 → 1, 4 → 2, 1 → 3							
Content	*A*	*B*	*C*	*D*	*E*	⊥	⊥	⊥
State	i	v	v	v	v	e	e	e

garbage collection

First, we copy the live pages from block 0 to end of log and update our mapping information; this is usually done first so the data isn't lost in case of a power failure.

Table	0 → 4, 6 → 5, 4 → 6, 1 → 7							
Content	*A*	*B*	*C*	*D*	*E*	*B*	*C*	*D*
State	i	i	i	i	v	v	v	v

Then we can erase block 0:

Table	0 → 4, 6 → 5, 4 → 6, 1 → 7							
Content	⊥	⊥	⊥	⊥	*E*	*B*	*C*	*D*
State	e	e	e	e	v	v	v	v

write(4) content F

Table	0 → 4, 6 → 5, 4 → 0, 1 → 7							
Content	*F*	⊥	⊥	⊥	*E*	*B*	*C*	*D*
State	v	e	e	e	v	v	i	v

garbage collection

Copy the live pages from block 1 to the next erased page, update FTL:

Table	0 → 1, 6 → 2, 4 → 0, 1 → 3							
Content	*F*	*E*	*B*	*D*	*E*	*B*	*C*	*D*
State	v	v	v	v	i	i	i	i

Erase block 1:

Table	0 → 1, 6 → 2, 4 → 0, 1 → 3							
Content	*F*	*E*	*B*	*D*	⊥	⊥	⊥	⊥
State	v	v	v	v	e	e	e	e

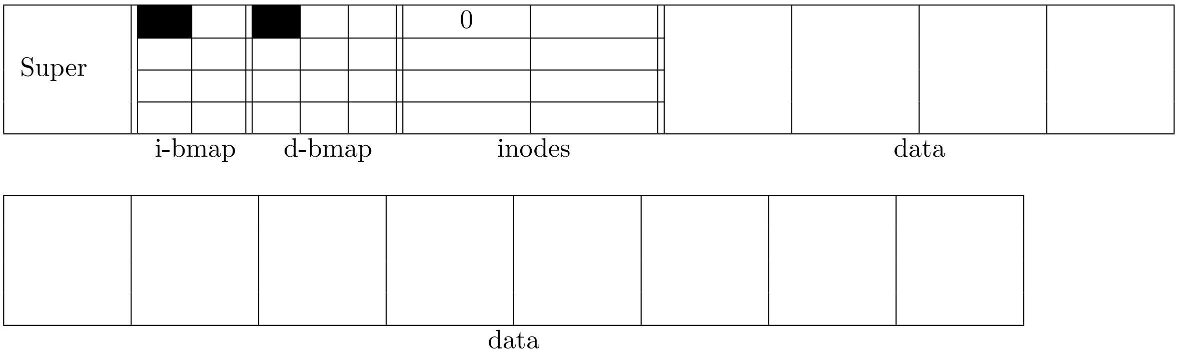
Advanced

3 File System

a) In an inode-based file system, every file (also directory) is represented by exactly one inode. The content of the inode representing a file is a set of pointers to the data blocks that contain the data of the file (plus a bunch of meta data that we don't care about here). For files, the data is just their content. For directories, the data is a set of mappings `(name: inode number)`. The bitmaps are only used to decide which cells in the inode region/-data region can be written to.

Note that Unix-like operating systems have file systems where each directory includes two subdirectories by default: `.` ("dot") and `..` ("dot-dot"). dot refers to the directory itself, dot-dot to its parent directory — which only in case of the root directory is the directory itself. We will not show entries for either.

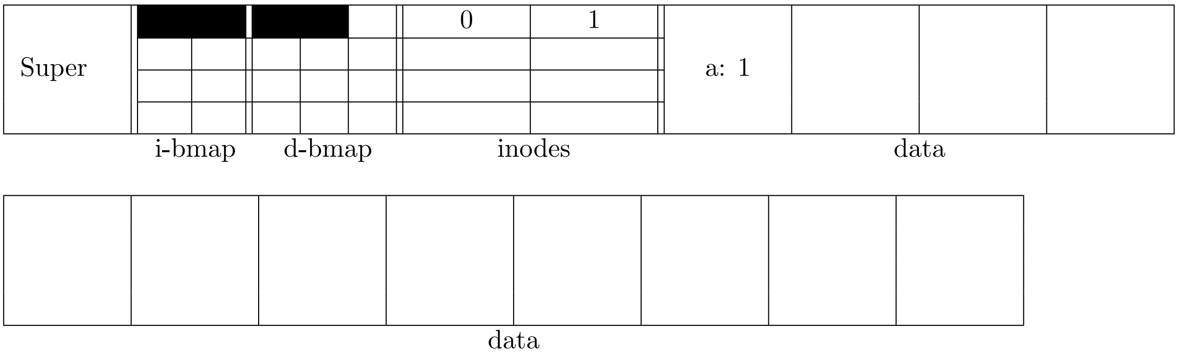
The file system starts out with the inode 0 representing the empty root directory; there is nothing in its data block yet since it is empty. Black cells in the bitmaps indicate that the respective bits are set. The numbers in inodes are the pointers to the data blocks of the represented file.



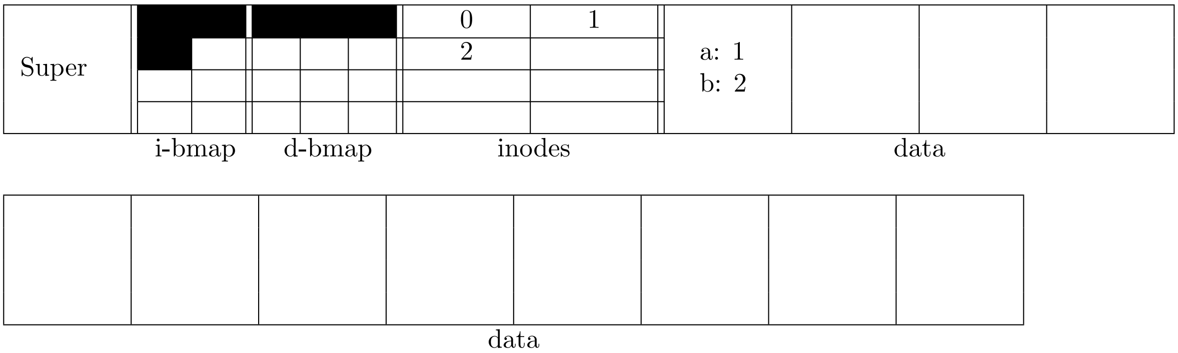
Initially, only inode 0 exists and contains a single pointer to data block 0. This inode represents the root directory. Since the root directory is initally empty, there is no data stored in the data block.

Command: mkdir /a

To create directory /a, we need a new inode — the inode bitmap tells us that inode 1 can be written — and reserve space for the directory — the data bitmap tells us that data block 1 is empty. We need to update the content of the root directory to include subdirectory /a. The mapping "a:1" means that the subdirectory a is represented by inode 1.

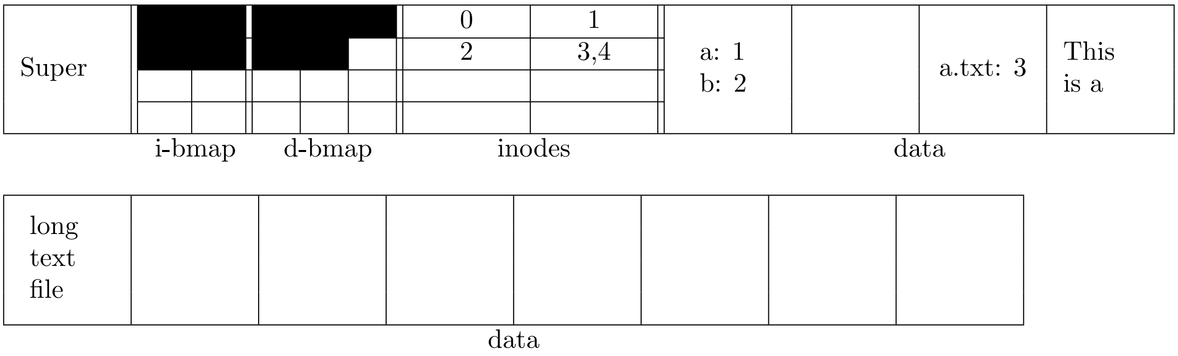


Command: mkdir /b

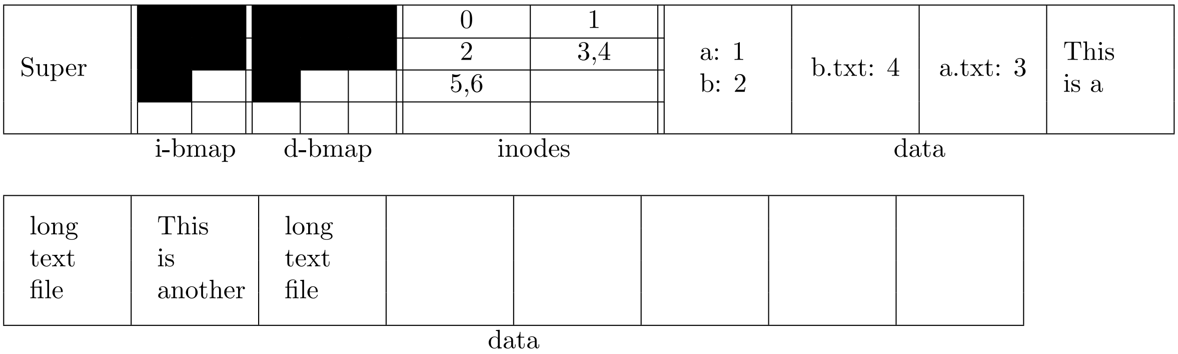


Command: echo “This is a long text file” > /b/a.txt

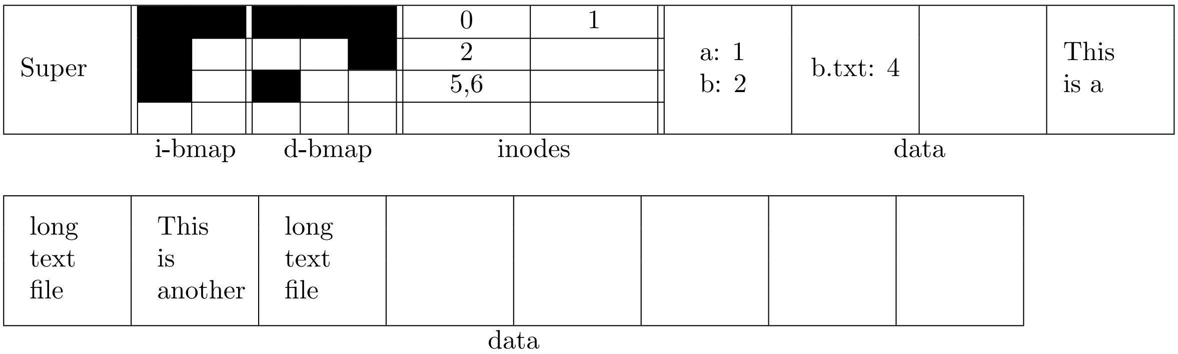
To create file /b/a.txt, we need a new inode — the inode bitmap tells us that inode 3 can be written — and reserve space for the file — the data bitmap tells us that data blocks 3 and 4 are empty. We need to update the content of the directory /b to include the information that file /b/a.txt is represented by inode 3. Note that inode 3 contains two pointers: one to data block 3, and one to data block 4.



Command: echo “This is another long text file” > /a/b.txt

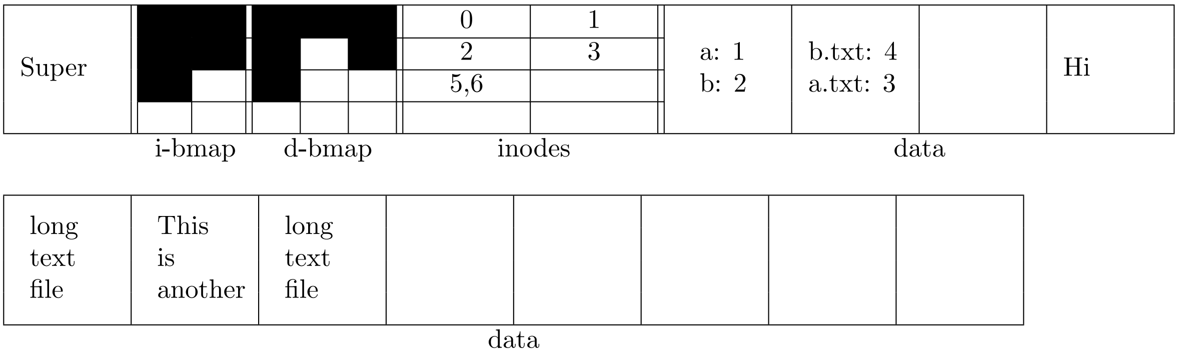


Command: rm /b/a.txt



Note what happens according to our solution when we delete a file: the inode bitmap marks the inode that represented the file as legal to be written to, the data bitmap does the same for the data blocks that contained the file, and the directory containing the file has the mapping from filename to inode removed. The content of the file itself is still on the disk! In fact, depending on the specific file system, even the pointer information stored in the inode may or may not still be on the disk, with just the inode bitmap indicating that that specific place in the inode can be written to. The ext2 file system did not delete the contents of an inode when a file was deleted, making file recovery as easy as flipping a bit in the inode bitmap along with the corresponding bits in the data bitmap (assuming none of the data blocks or the inode had been overwritten since the file was deleted). ext3 started setting all pointer fields stored in an inode that represents a deleted file to 0, which means the content of the inode was actually erased. The solution presented here assumes that the content of an inode that represents a deleted file is erased, while the content of the data blocks containing a deleted file are not.

Command: echo Hi > /a/a.txt



b) - Super: get starting address of inode region

- Inode table at index 0: get address(es) of '/' block(s)
- '/' block: get address of '/a' inode
- '/a' inode: get address(es) of '/a' block(s)
- '/a' block: get address of '/a/b.txt' inode
- '/a/b.txt' inode: get address(es) of '/a/b.txt/' block(s)

One important side note: the inode bitmap and the data bitmap are not involved when reading files (and therefore also not when resolving a path); they are only involved when we need to know where we can store now file data, or when we delete existing files.

c)
- Hard links are stored in the directory data blocks and point to the linked file's inode, i.e. a hard link is not a separate file, it is simply the tupel (name : inode number) as an entry in a directory's data black. In this sense, a hard link is a "normal" link. Each file has at least one hard link.
- A soft link is a file that has its own inode. Its data block stores a path which has to be resolved to get to the linked file. Thus, two paths have to be resolved when accessing a soft link.

4 Permissions

a) `weakuser` would need user execute permissions since he is the file owner. Even if the execute bit for *other* was set, `weakuser` would not be able to execute the file. Similarly, because `user` is not the owner of the file but is in its owner group, the group execute bit would have to be set for `user` to be able to execute it.

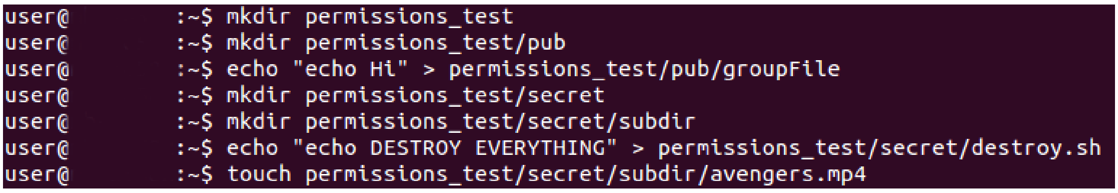
b) He can do all three. When the OS decides whether `weakuser` can execute the file, it checks whether he is the owner (he isn't), then if he is in the owner group. Since `weakuser` is in the owner group, and the group has execute permissions, he can execute the file. Only once the file is being executed will the `suid` bit change the user permissions of the new process in which the file is being executed to those of `user`. In short: to execute a file, the `suid/sgid` have no influence, only once the file is being executed does the newly started process get influenced by them.

Notice that the user permissions have `S`, i.e. the `suid` bit is set, but the owner does not have permissions to execute the file.

c) `weakuser` is *other* for that directory, and *other* does not have read privileges.

d) Set *other* read bit on `/secret/subdir/avengers.mp4`; `user` can do this since he owns the file — he does not need write permissions for it! To resolve the path, *other* needs execute permissions for all directories on the path, which he has. Notice that *other* does not have read permissions for `/secret/subdir/`, but he doesn't need those to resolve the path; he would only need them if he wanted to list the content of `/secret/subdir/`.

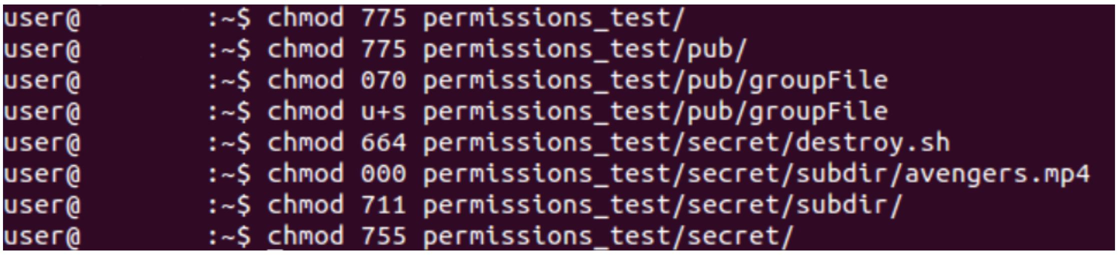
e) First we create the directory tree; the solution below shows a directory tree rooted at directory `permissions_test` in my home directory on Ubuntu.



```
user@          :~$ mkdir permissions_test
user@          :~$ mkdir permissions_test/pub
user@          :~$ echo "echo Hi" > permissions_test/pub/groupFile
user@          :~$ mkdir permissions_test/secret
user@          :~$ mkdir permissions_test/secret/subdir
user@          :~$ echo "echo DESTROY EVERYTHING" > permissions_test/secret/destroy.sh
user@          :~$ touch permissions_test/secret/subdir/avengers.mp4
```

Figure 1: Creating the directory tree.

Next we set the permissions.

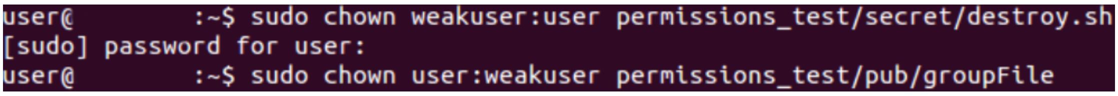


```
user@         :~$ chmod 775 permissions_test/
user@         :~$ chmod 775 permissions_test/pub/
user@         :~$ chmod 070 permissions_test/pub/groupFile
user@         :~$ chmod u+s permissions_test/pub/groupFile
user@         :~$ chmod 664 permissions_test/secret/destroy.sh
user@         :~$ chmod 000 permissions_test/secret/subdir/avengers.mp4
user@         :~$ chmod 711 permissions_test/secret/subdir/
user@         :~$ chmod 755 permissions_test/secret/
```

Figure 2: Setting permissions.

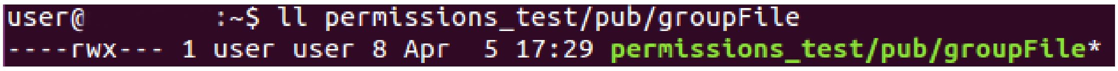
A short explanation for the command chmod: if we interpret a triad of the permission string as a binary number, we get a value between 0 and 7: read permissions are the 4-bit, write permissions are the 2-bit, execute permissions are the 1-bit. For example, 3 means "write and execute privileges" since $3 = 2 + 1$. Thus chmod 437 file will make the permission string of a regular file file to -r---wxrwx. Alternatively, you can set or remove (+ or −) read/write/execute permissions (r/w/x) for owner/group owner/other (u/g/o). For example, chmod o+rw file gives (+) read (r) and write (w) permissions to *other* (o) for file.

If you check the permissions now (e.g. the command ll permissions_test/secret/ will tell you the permission string for permissions_test/secret/), you will see the permissions are set correctly for every file except permissions_test/secret/destroy.fh since we have not yet set the sgid bit for that file yet. The reason we didn't show that yet is to illustrate that once we change the owner of a file using the command chown as in the following figure, both suid and sgid get cleared. You need superuser privileges to use chown — you cannot "gift" someone with a file unless you have superuser privileges.



```
user@        :~$ sudo chown weakuser:user permissions_test/secret/destroy.sh
[sudo] password for user:
user@        :~$ sudo chown user:weakuser permissions_test/pub/groupFile
```

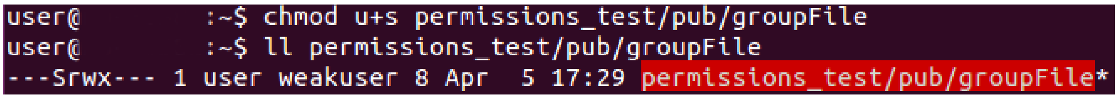
Figure 3: chown newOwner:newGroupOwner file changes the owner and group owner of file to newOwner and newGroupOwner respectively.



```
user@         :~$ ll permissions_test/pub/groupFile
----rwx--- 1 user user 8 Apr  5 17:29 permissions_test/pub/groupFile*
```

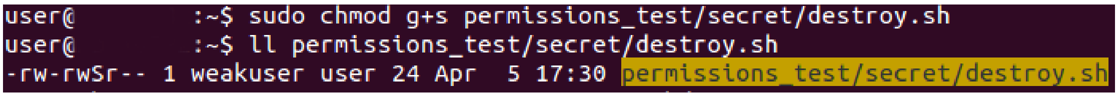
Figure 4: Checking permissions after changing owners and group owners using chown. The terminal on Ubuntu offers different color codings for some permission strings.

We now set the suid/sgid bits that we want again:



```
user@         :~$ chmod u+s permissions_test/pub/groupFile
user@         :~$ ll permissions_test/pub/groupFile
---Srwx--- 1 user weakuser 8 Apr  5 17:29 permissions_test/pub/groupFile*
```

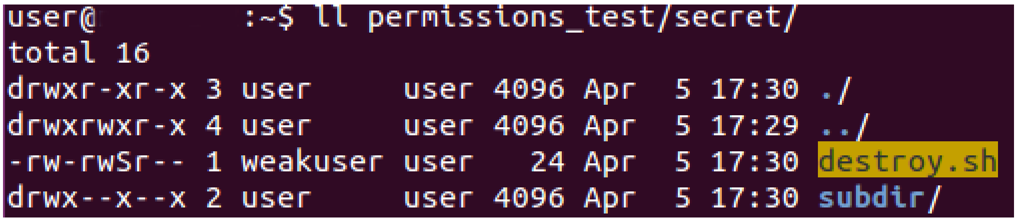
Figure 5: Setting the suid bit for groupFile again...



```
user@          :~$ sudo chmod g+s permissions_test/secret/destroy.sh
user@          :~$ ll permissions_test/secret/destroy.sh
-rw-rwSr-- 1 weakuser user 24 Apr  5 17:30 permissions_test/secret/destroy.sh
```

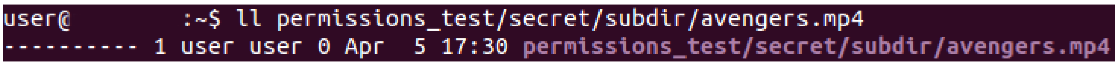
Figure 6: ...and the `sgid` for `permissions_test/secret/destroy.sh`. We need `sudo` here if we are not logged in as `weakuser` since only the owner and the superuser can set file permissions.

As an example, we check the permissions for two files:



```
user@          :~$ ll permissions_test/secret/
total 16
drwxr-xr-x 3 user     user 4096 Apr  5 17:30 ./
drwxrwxr-x 4 user     user 4096 Apr  5 17:29 ../
-rw-rwSr-- 1 weakuser user   24 Apr  5 17:30 destroy.sh
drwx--x--x 2 user     user 4096 Apr  5 17:30 subdir/
```

Figure 7: Checking permissions for `permissions_test/secret/`...



```
user@          :~$ ll permissions_test/secret/subdir/avengers.mp4
---------- 1 user user 0 Apr  5 17:30 permissions_test/secret/subdir/avengers.mp4
```

Figure 8: ...and for `permissions_test/secret/subdir/avengers.mp4`.

You can verify that the permissions and owners for the rest of the directory tree are set correctly as well.