Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

FS 2016 Prof. R. Wattenhofer

Computer Engineering II

Solution to Exercise Sheet 9

Basic

1 HDDs

a) For a random workload, the tracks in the middle of a platter are favored since on average, the distance to them 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	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
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 log-structured 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 log-structured SSDs, the FTL stores which logical address is mapped to which physical address. We will denote "logical address `a` is located at physical address `b`" by "`a → b`".

Direct mapped SSD:

write(0) content A

Content	A	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
State	V	E	E	E	i	i	i	i

write(6) content B

Content	A	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	B	$\perp$
State	V	E	E	E	E	E	V	E

write(4) content C

Content	A	$\perp$	$\perp$	$\perp$	C	$\perp$	B	$\perp$
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

Log-structured SSD:

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	$0 \to 0$, $6 \to 1$, $4 \to 2$, $1 \to 3$							
Content	A	B	C	D	$\perp$	$\perp$	$\perp$	$\perp$
State	V	V	V	V	i	i	i	i

write(0) content E

Table	$0 \to 4$, $6 \to 1$, $4 \to 2$, $1 \to 3$							
Content	A	B	C	D	E	$\perp$	$\perp$	$\perp$
State	V	V	V	V	V	E	E	E

garbage collection
First, we have to copy the live pages from block 0 to end of log and update our mapping information:

Table	$0 \to 4$, $6 \to 5$, $4 \to 6$, $1 \to 7$							
Content	A	B	C	D	E	B	C	D
State	V	V	V	V	V	V	V	V

Then we can erase block 0:

Table	$0 \to 4$, $6 \to 5$, $4 \to 6$, $1 \to 7$							
Content	$\perp$	$\perp$	$\perp$	$\perp$	E	B	C	D
State	E	E	E	E	V	V	V	V

write(4) content F

Table	$0 \to 4$, $6 \to 5$, $4 \to 0$, $1 \to 7$							
Content	F	$\perp$	$\perp$	$\perp$	E	B	C	D
State	V	E	E	E	V	V	V	V

garbage collection
Copy the live pages from block 1 to end of log, update FTL:

Table	$0 \to 1$, $6 \to 2$, $4 \to 0$, $1 \to 3$							
Content	F	E	B	D	E	B	C	D
State	V	V	V	V	V	V	V	V

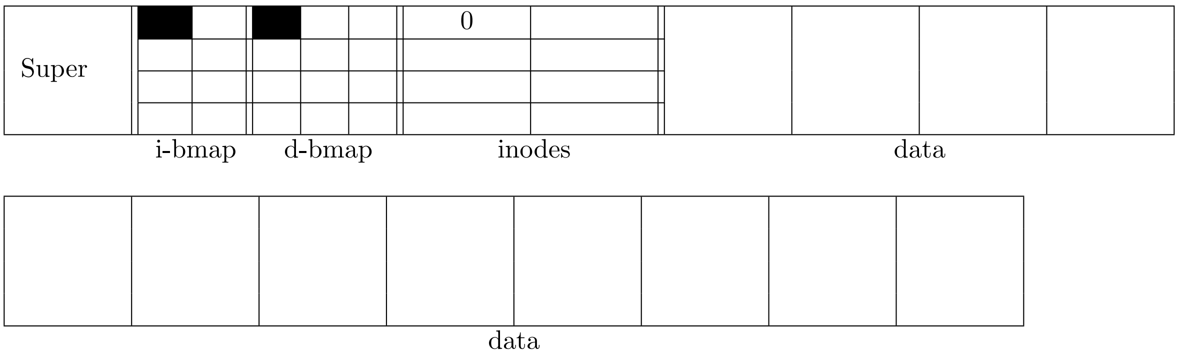
Erase block 1:

Table	$0 \to 1$, $6 \to 2$, $4 \to 0$, $1 \to 3$							
Content	F	E	B	D	$\perp$	$\perp$	$\perp$	$\perp$
State	V	V	V	V	E	E	E	E

3 File System

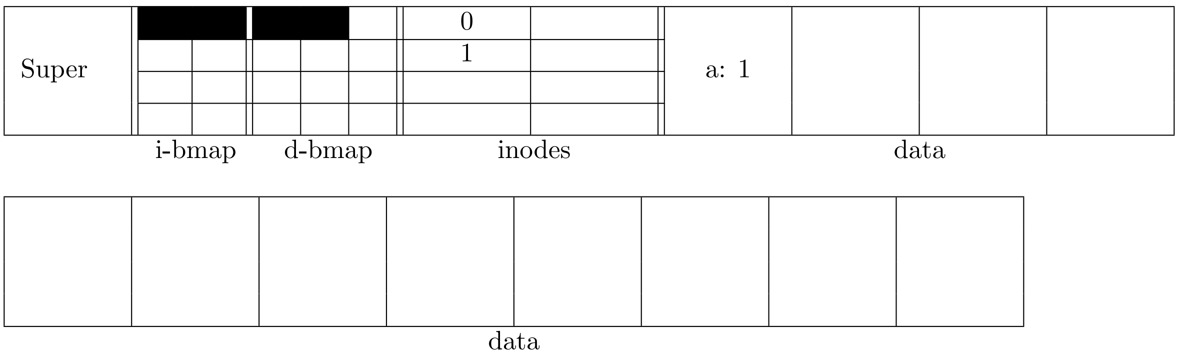
a) In an inode-based file system, every file (also directory) is represented by exactly one inode. The content of the inode representing a file is a set of pointers to the data blocks that contain the data of the file (plus a bunch of meta data that we don't care about here). For files, the data is just their content. For directories, the data is a set of mappings from the name of a file/directory within the directory to an inode representing that file/directory. 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.



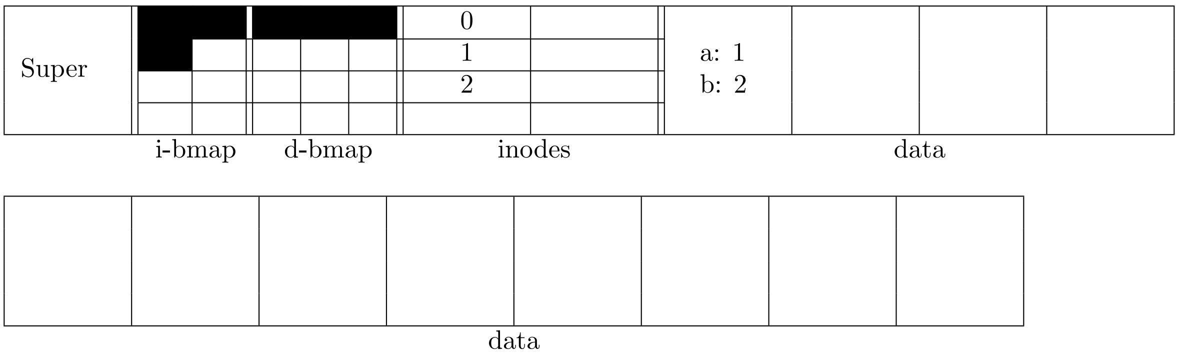
Initially, only inode 0 exists and contains a single pointer to data block 0. This inode represents the root directory. Since the root directory is initally empty, there is no data stored in the data block.

Command: mkdir /a

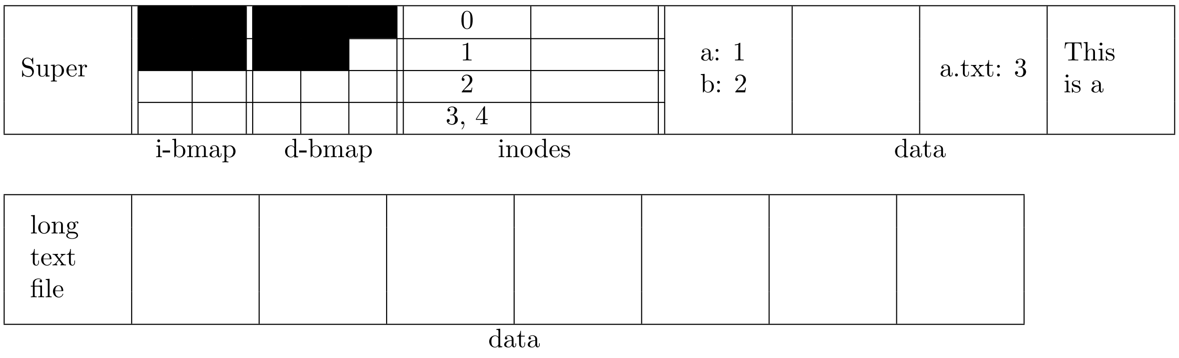


To create directory /a, we need a new inode — the inode bitmap tells us that inode 1 can be written — and reserve space for the directory — the data bitmap tells us that data block 1 is empty. We need to update the content of the root directory to include subdirectory /a. The mapping “a:1” means that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

Command: mkdir /b

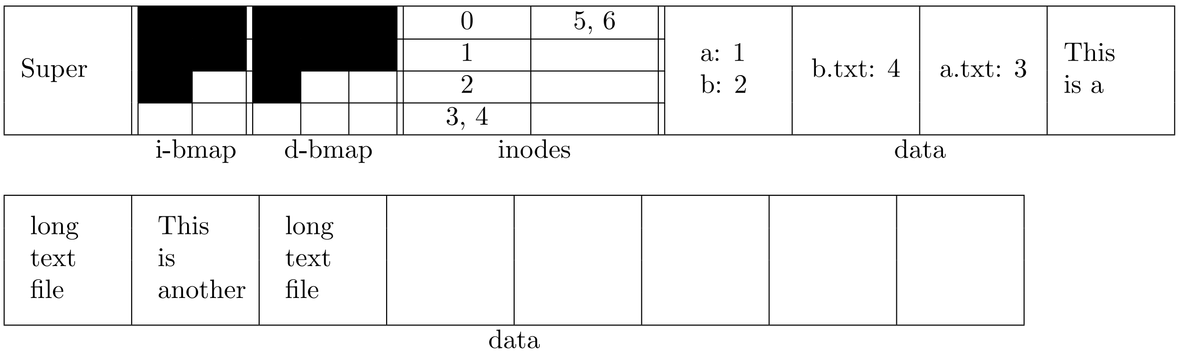


Command: echo “This is a long text file” > /b/a.txt

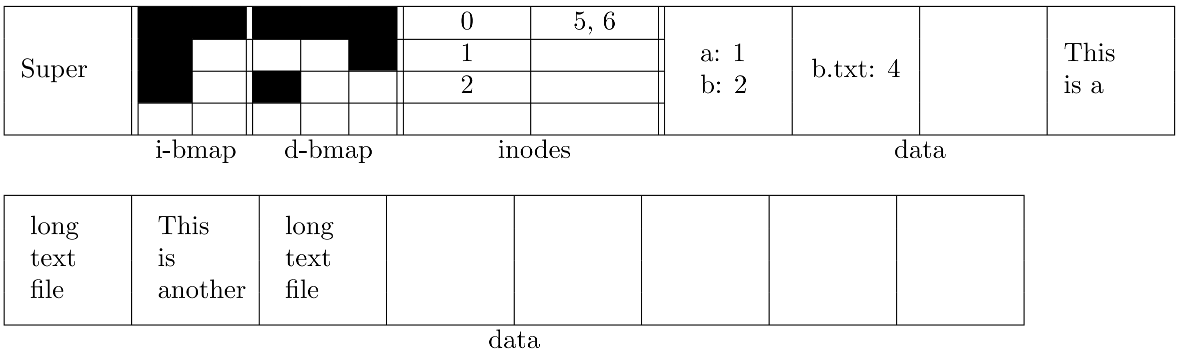


To create file /b/a.txt, we need a new inode — the inode bitmap tells us that inode 3 can be written — and reserve space for the file — the data bitmap tells us that data blocks 3 and 4 are empty. We need to update the content of the directory /b to include the information that file /b/a.txt is represented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

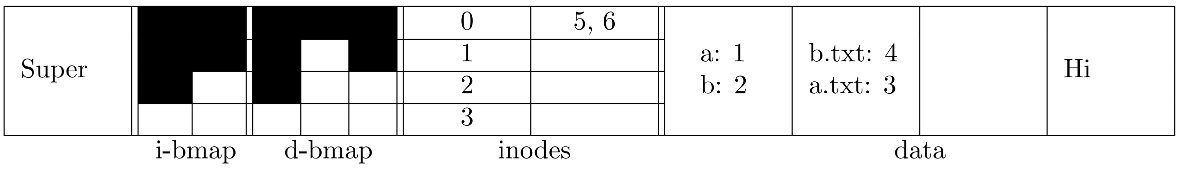


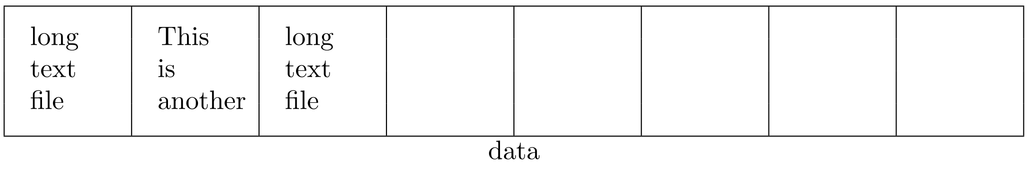
Command: rm /b/a.txt



Note what happens 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 address of inode table
- 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 revolved 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 blocks and point to the linked file's inode. In this sense, a hard link is a "normal" link. Note that each file has at least one hard link.
- Soft links are blocks (i.e. basically files) which store a path which has to be resolved to get to the linked file. Thus, two paths have to be resolved when accessing a symlink.