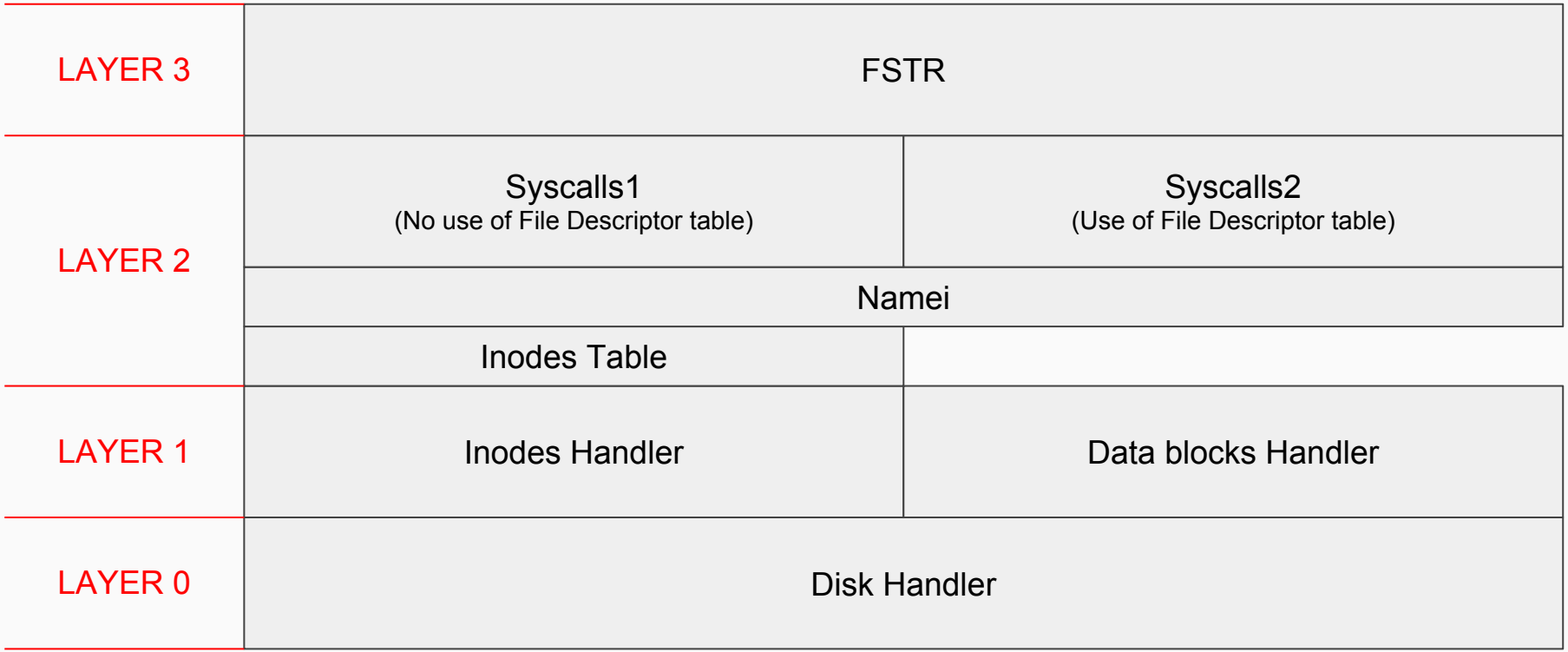
FSTR

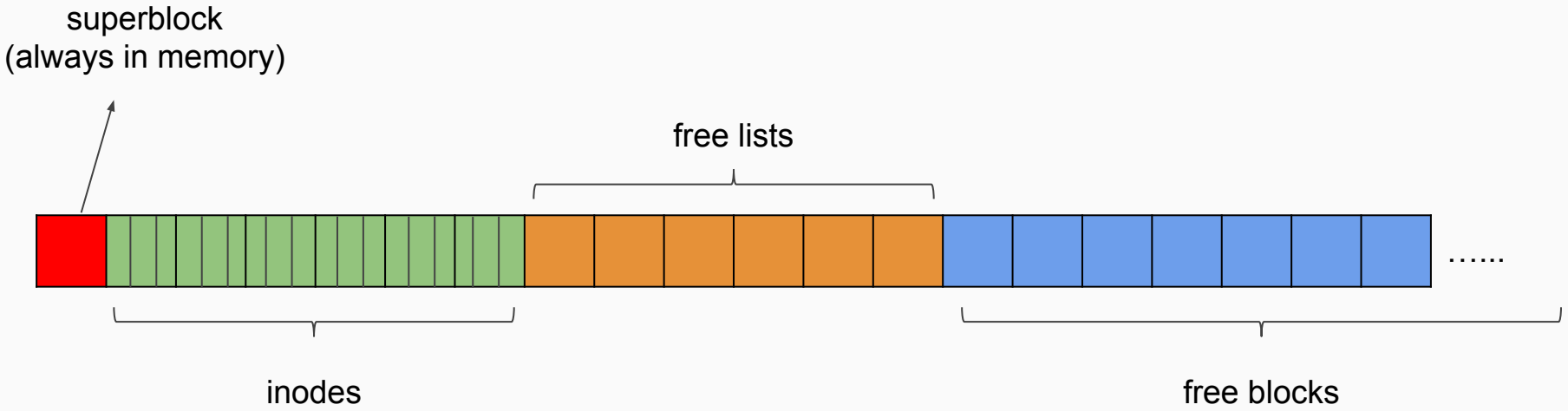
File System That Rocks

Tanuj Mittal
Aviral Takkar
Francois Malinowski

Presentation of layers



Layout of superblock, inodes, freelists



DONE BY mkfs.fstr UTILITY

Inodes Handler

Designed based on the algorithms presented in **The Design of UNIX OS** by **Maurice J Bach.**

Exposes three functions:

- **IGET -** fetches the given inode from disk
- **IPUT -** writes the given inode to disk; frees given inode if the number of links to it are 0 by calling IFREE
- **IALLOC -** searches for a free inode and allocates it

One internal function:

- **IFREE -** frees given inode and writes it to disk

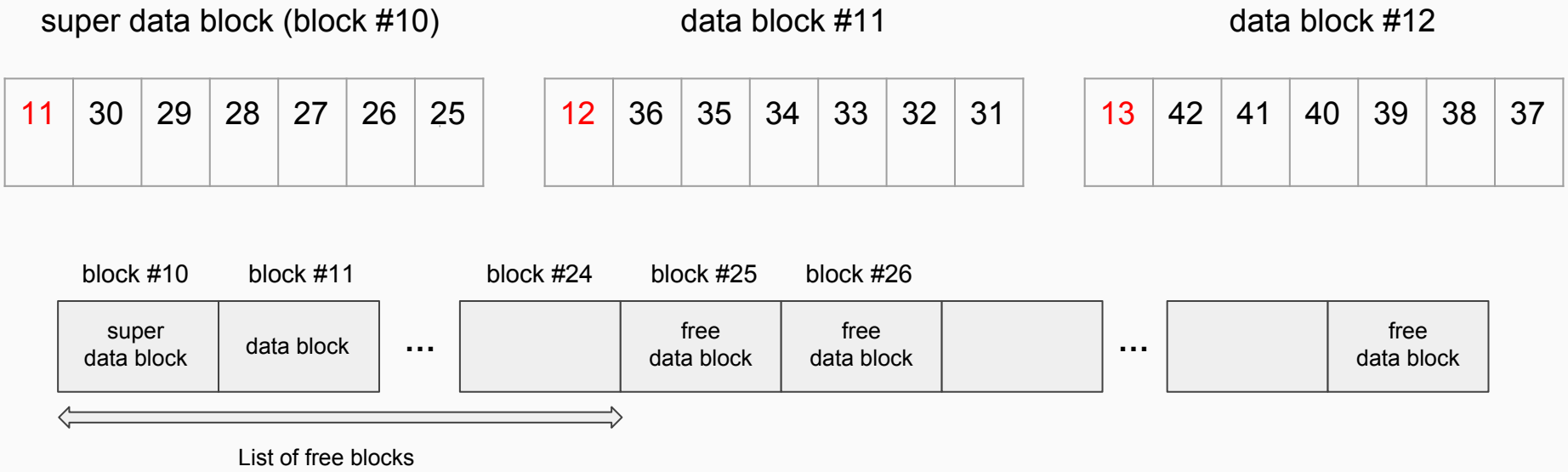
Inodes Handler

IGET (*inode_number, inode_buffer*)	**IPUT** (*inode*)	**IALLOC** (*inode_buffer*)	**IFREE** (*inode_buffer*)
• Read inode block (**BREAD**) • Copy inode from offset in block into *inode_buffer*	• If *inode->links = 0,* free all data blocks of file. **IFREE**(*inode*). Return • Read inode block (**BREAD**) • Copy *inode* into inode block at calculated offset • Write *inode* block (**BWRITE**)	• Find *next_free_inode* by linearly searching through all inodes. Return failure if not found. • **IGET**(*next_free_inode, inode_buffer*) • Set links_nb and default file type • **IPUT** (*inode_buffer*)	• Clear all fields of *inode_buffer* • *inode_buffer->TYPE = TYPE.FREE* • Read *inode_buffer* block **(BREAD**) • Copy *inode_buffer* to calculated offset • Write block (**BWRITE**)

How do we handle data blocks?

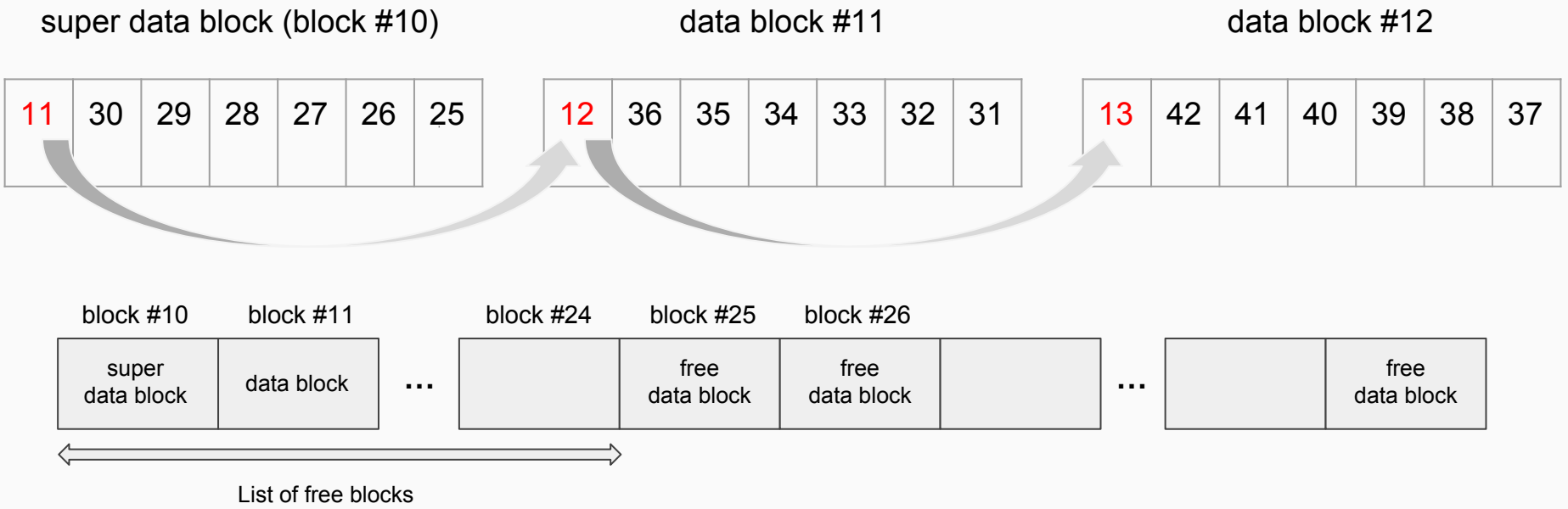
How do we handle data blocks?

How does the list of free blocks look like?



How do we handle data blocks?

How does the list of free blocks look like?



How do we allocate a data block?

How do we handle data blocks?

How do we alloc a data block?

super data block (block #10)

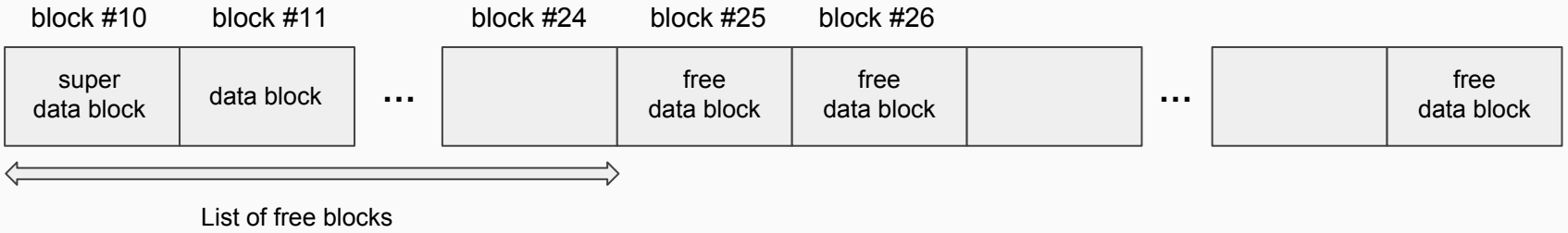
11	30	29	28	27	26	25

data block #11

12	36	35	34	33	32	31

data block #12

13	42	41	40	39	38	37



How do we handle data blocks?

How do we alloc a data block?

super data block (block #10)

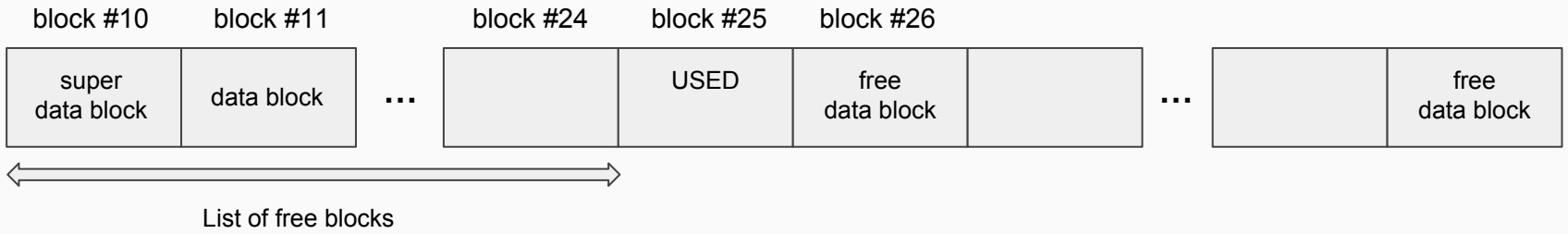
11	30	29	28	27	26	0

data block #11

12	36	35	34	33	32	31

data block #12

13	42	41	40	39	38	37



How do we handle data blocks?

How do we alloc a data block?

super data block (block #10)

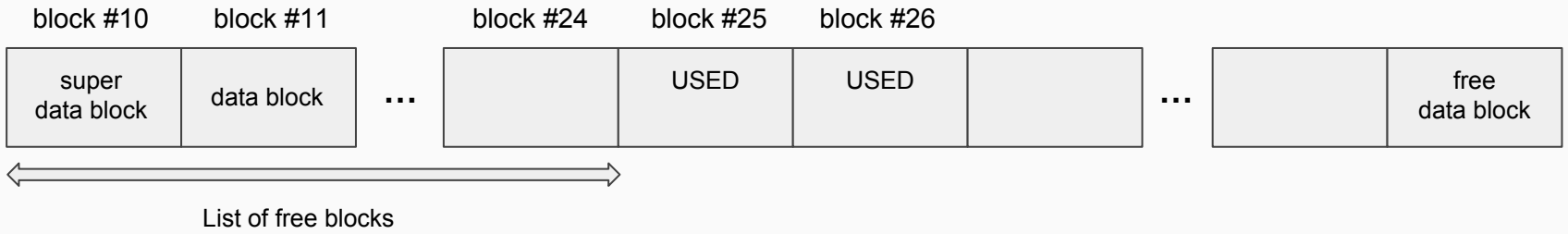
11	30	29	28	27	0	0

data block #11

12	36	35	34	33	32	31

data block #12

13	42	41	40	39	38	37



How do we handle data blocks?

How do we alloc a data block?
What happen for the last block number?

super data block (block #10)

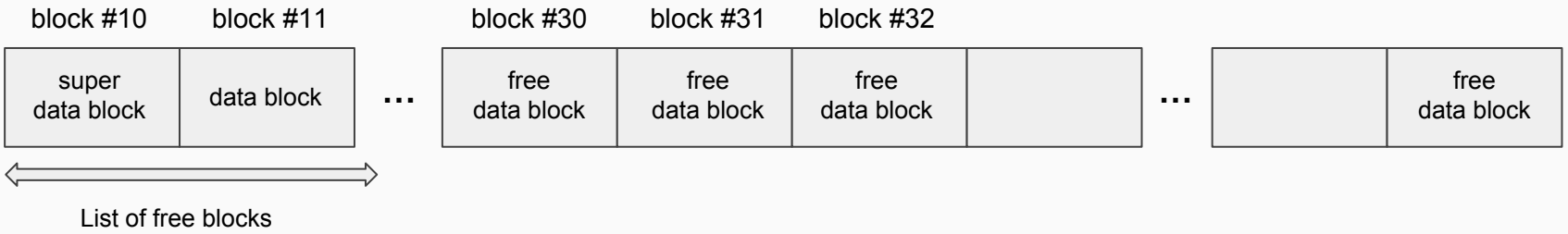
11	30	0	0	0	0	0

data block #11

12	36	35	34	33	32	31

data block #12

13	42	41	40	39	38	37



How do we handle data blocks?

How do we alloc a data block?
What happen for the last block number?

super data block (block #10)

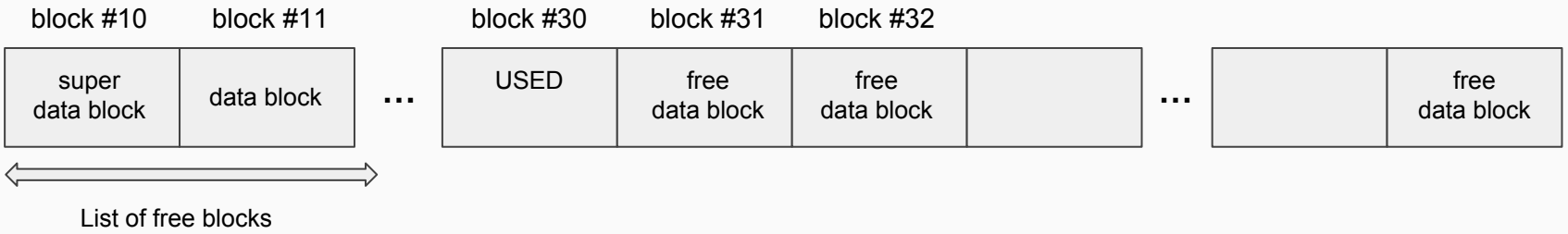
11	0	0	0	0	0	0

data block #11

12	36	35	34	33	32	31

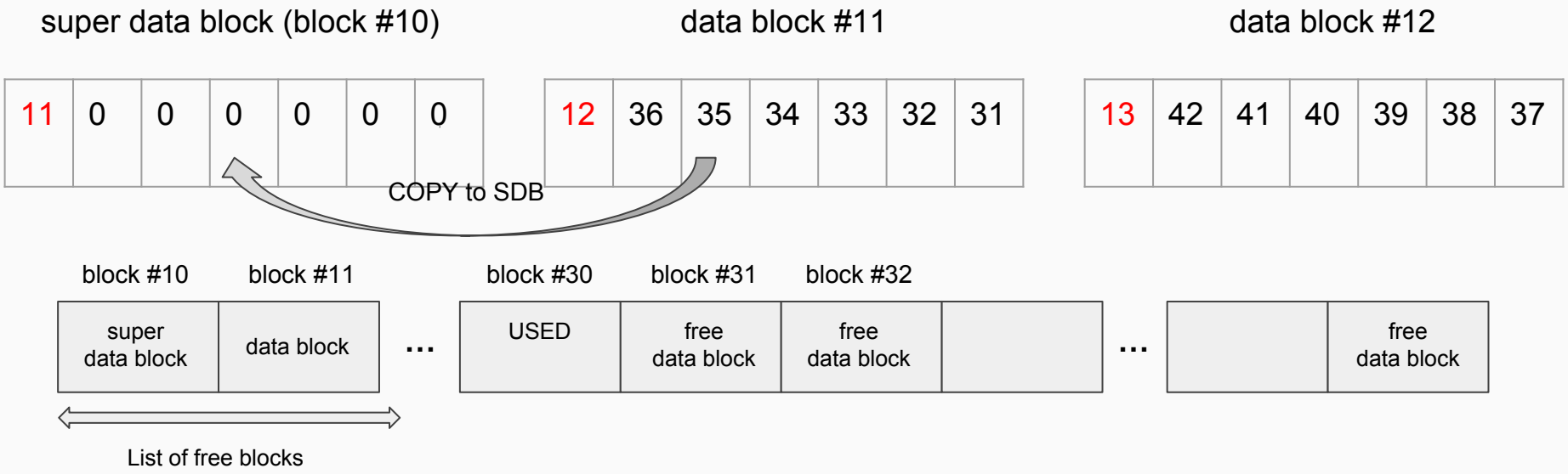
data block #12

13	42	41	40	39	38	37



How do we handle data blocks?

How do we alloc a data block?
What happen for the last block number?



How do we handle data blocks?

How do we alloc a data block?
What happen for the last block number?

data block #11

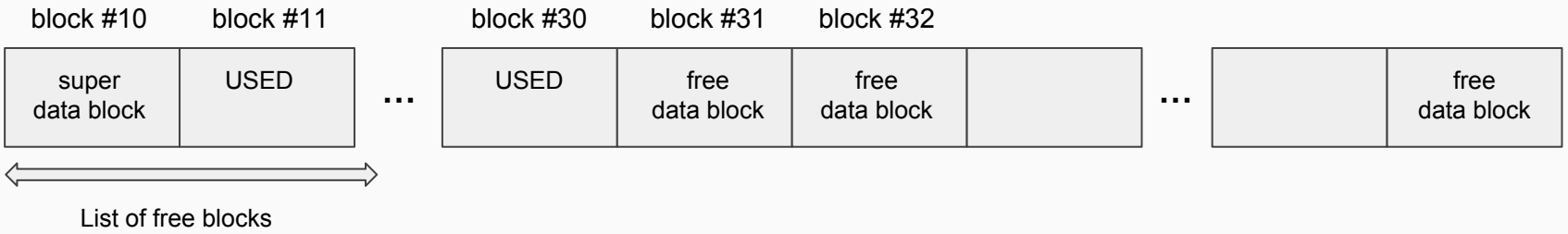
0	0	0	0	0	0	0

super data block (block #10)

12	36	35	34	33	32	31

data block #12

13	42	41	40	39	38	37



How do we handle data blocks?

How do we alloc a data block?
What happen for the last block number?

data block #11

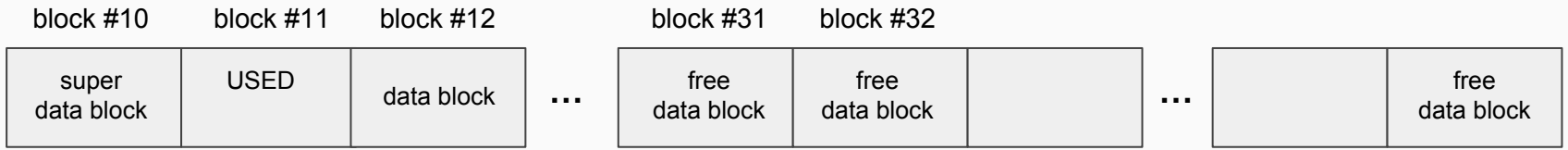
0	0	0	0	0	0	0

super data block (block #10)

12	36	35	34	33	32	31

data block #12

13	42	41	40	39	38	37



How do we free a data block?

How do we handle data blocks?

How do we free a data block?
e.g.: block #87

super data block (block #10)

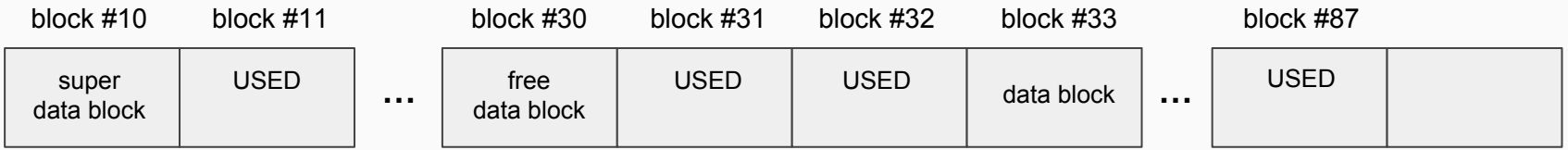
33	30	0	0	0	0	0

data block #33

73	38	45	51	65	64	63

data block #73

22	58	57	56	55	74	71



How do we handle data blocks?

How do we free a data block?
e.g.: block #87

super data block (block #10)

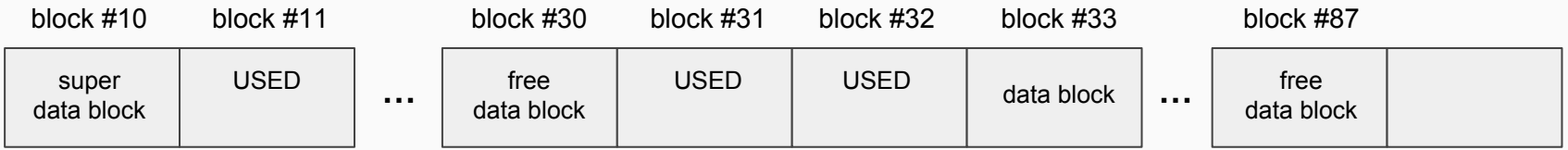
33	30	87	0	0	0	0

data block #33

73	38	45	51	65	64	63

data block #73

22	58	57	56	55	74	71



How do we handle data blocks?

How do we free a data block?
e.g.: block #15

super data block (block #10)

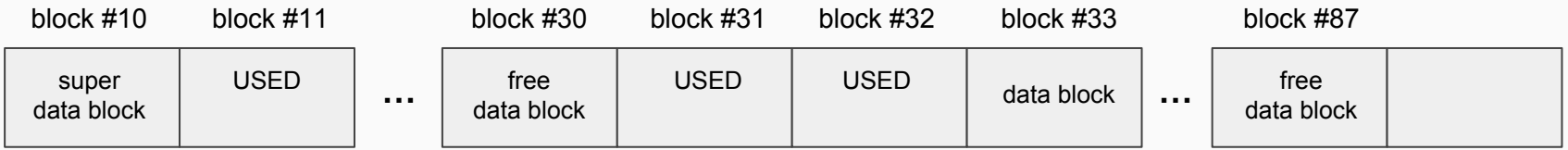
33	30	87	29	24	21	35

data block #33

73	38	45	51	65	64	63

data block #73

22	58	57	56	55	74	71



How do we handle data blocks?

How do we free a data block?
e.g.: block #15

data block #15

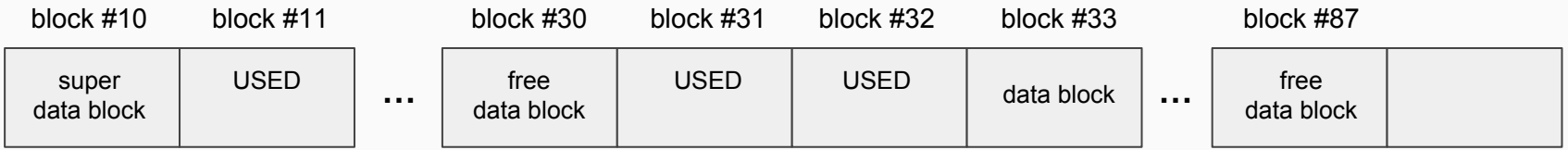
USED BLOCK

super data block (block #10)

33	30	87	29	24	21	35

data block #33

73	38	45	51	65	64	63



How do we handle data blocks?

How do we free a data block?
e.g.: block #15

data block #15

super data block (block #10)

33	30	87	29	24	21	35

data block #33

73	38	45	51	65	64	63

block #10	block #11		block #30	block #31	block #32	block #33		block #87	
super data block	USED	...	free data block	USED	USED	data block	...	free data block	

How do we handle data blocks?

How do we free a data block?
e.g.: block #15

data block #15

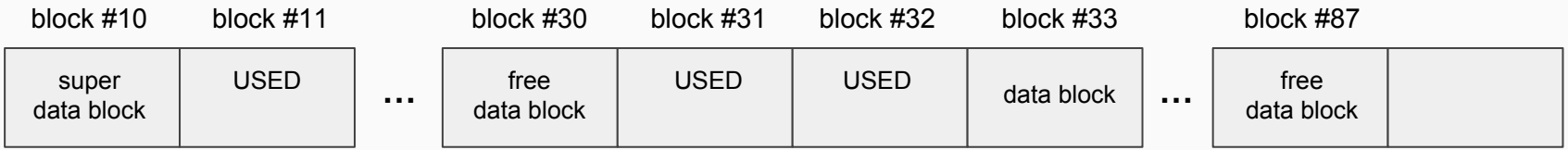
33	30	87	29	24	21	35

super data block (block #10)

15	0	0	0	0	0	0

data block #33

73	38	45	51	65	64	63



How do we handle data blocks?

How do we free a data block?
e.g.: block #15

super data block (block #10)

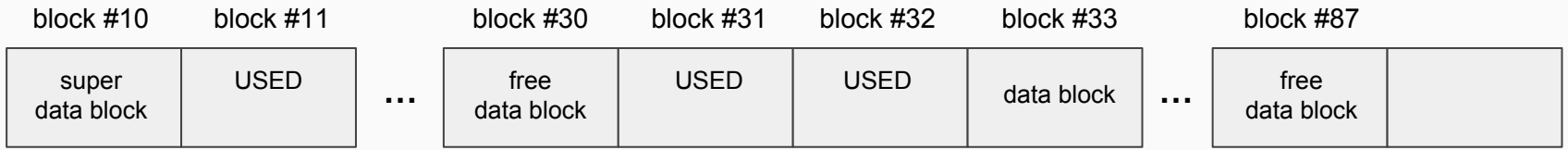
15	0	0	0	0	0	0

data block #15

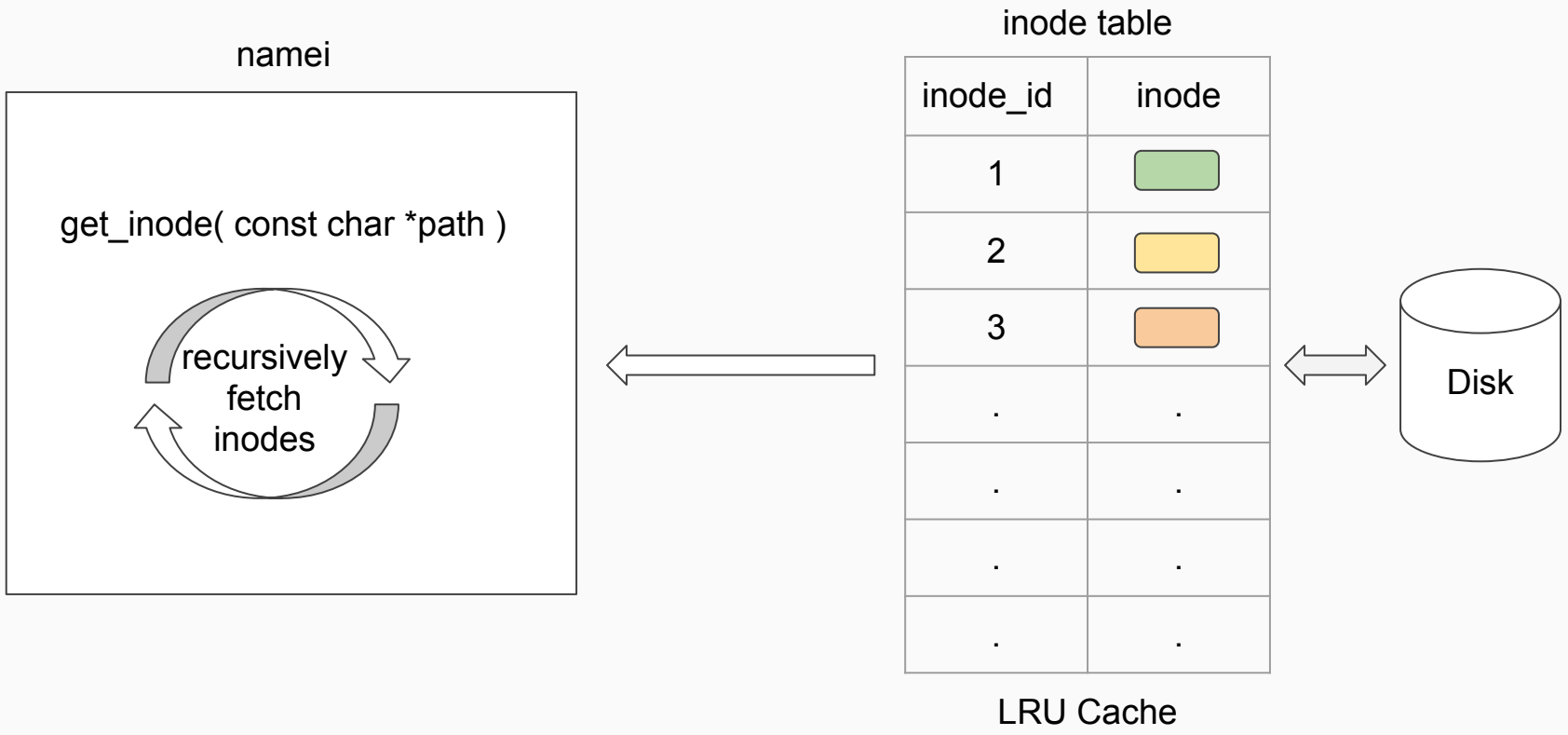
33	30	87	29	24	21	35

data block #33

73	38	45	51	65	64	63



namei and inode table



Implementation of:

- int **syscalls1__mkdir**(const char *path, mode_t mode)
- int **syscalls1__mknod**(const char *path, mode_t mode, dev_t dev)

- int **syscalls1__readdir**(const char *path, void *buffer, fuse_fill_dir_t filler, off_t offset)
- int **syscalls1__rename**(const char *oldpath, const char *newpath)
- int **syscalls1__unlink**(const char *path)
- int **syscalls1__rmdir**(const char *path)

- int **syscalls1__lstat**(const char *path, struct stat *buf)
- int **syscalls1__utimens**(const char *path, const struct timespec tv[2])

- int **syscalls1__chmod**(const char *path, mode_t mode)
- int **syscalls1__chown**(const char *path, uid_t uid, gid_t gid)

How do we handle system calls?

Implementation of:

- int **syscalls2__open**(const char *path, int oflag, ...)
- ssize_t **syscalls2__pread**(int fildes, void *buf, size_t nbyte, off_t offset)
- ssize_t **syscalls2__pwrite**(int fildes, const void *buf, size_t nbyte, off_t offset)
- int **syscalls2__close**(int fildes)

These functions need to handle file descriptors

Tests

4 types of tests were performed on FSTR:

- **Unit tests** (per module) with Unity C Testing Framework
  http://www.throwtheswitch.org/unity/
- **Bash scripts** to do integration testing (thousands of files in a folder, recursive folders)
- **Manual testing** (creation/copy of big files ~1Gb)
- **Bonnie** (used for benchmarks and for testing)
  http://www.textuality.com/bonnie/intro.html

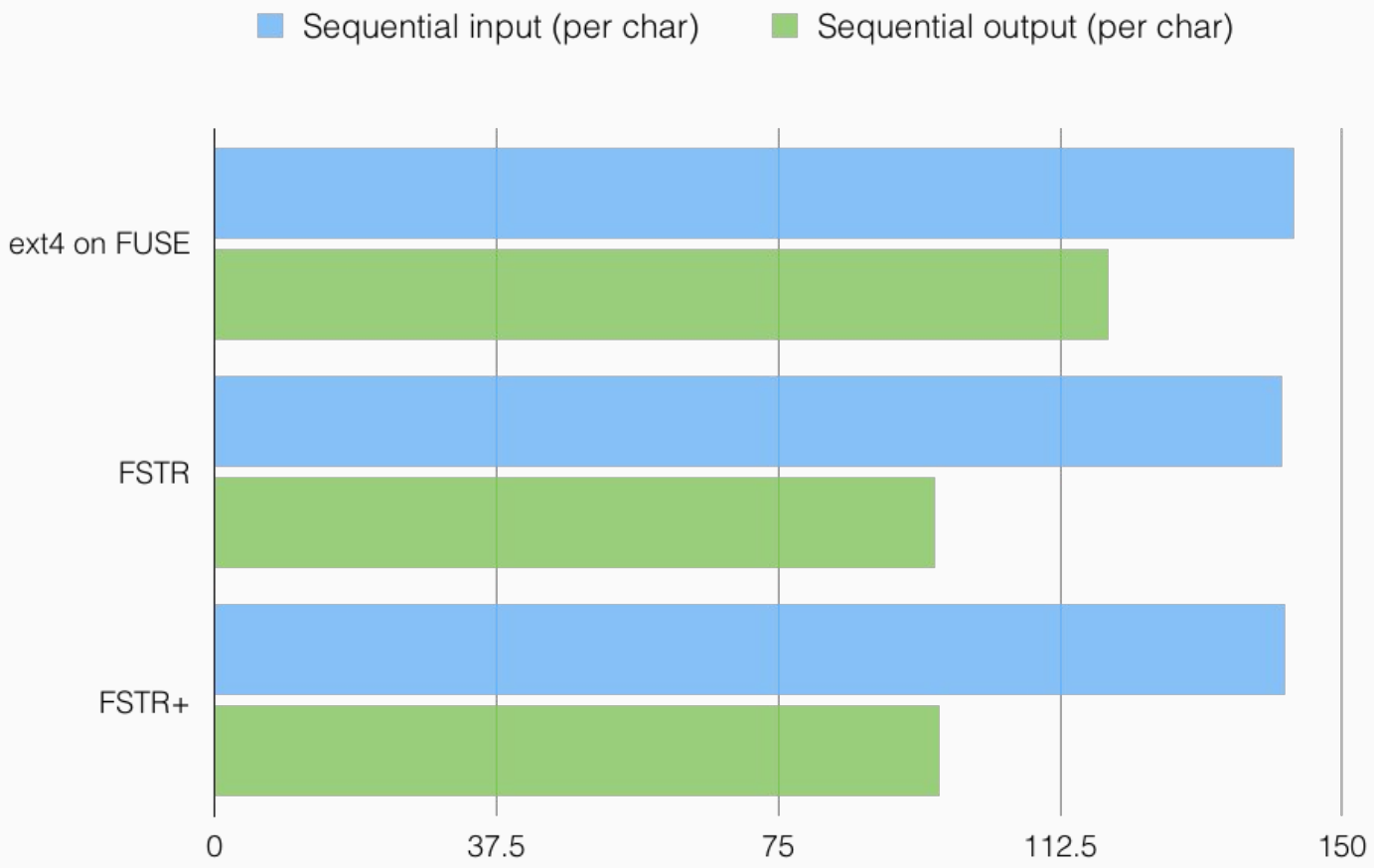
Travis CI to detect and avoid broken builds

It would have been impossible to debug without unit and layer testing

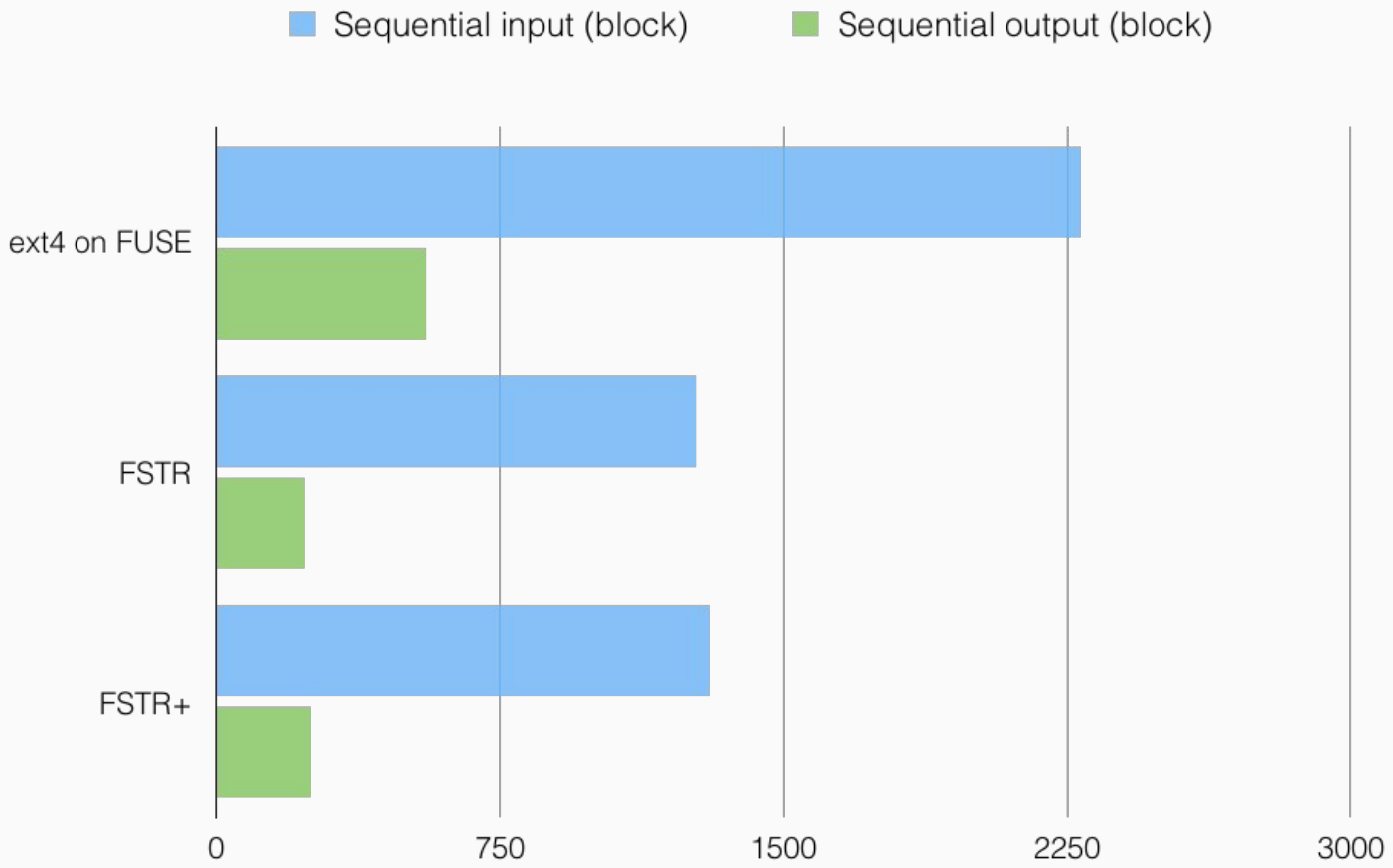
What Bonnie does

- a series of tests on a file of known size (100 MB default)
- **Sequential Output :** file written and rewritten using *putc()* and *write(2); tests the effectiveness of caching*
- **Sequential Input :** file read using *getc()* and *read(2)*
- **Random Seeks**

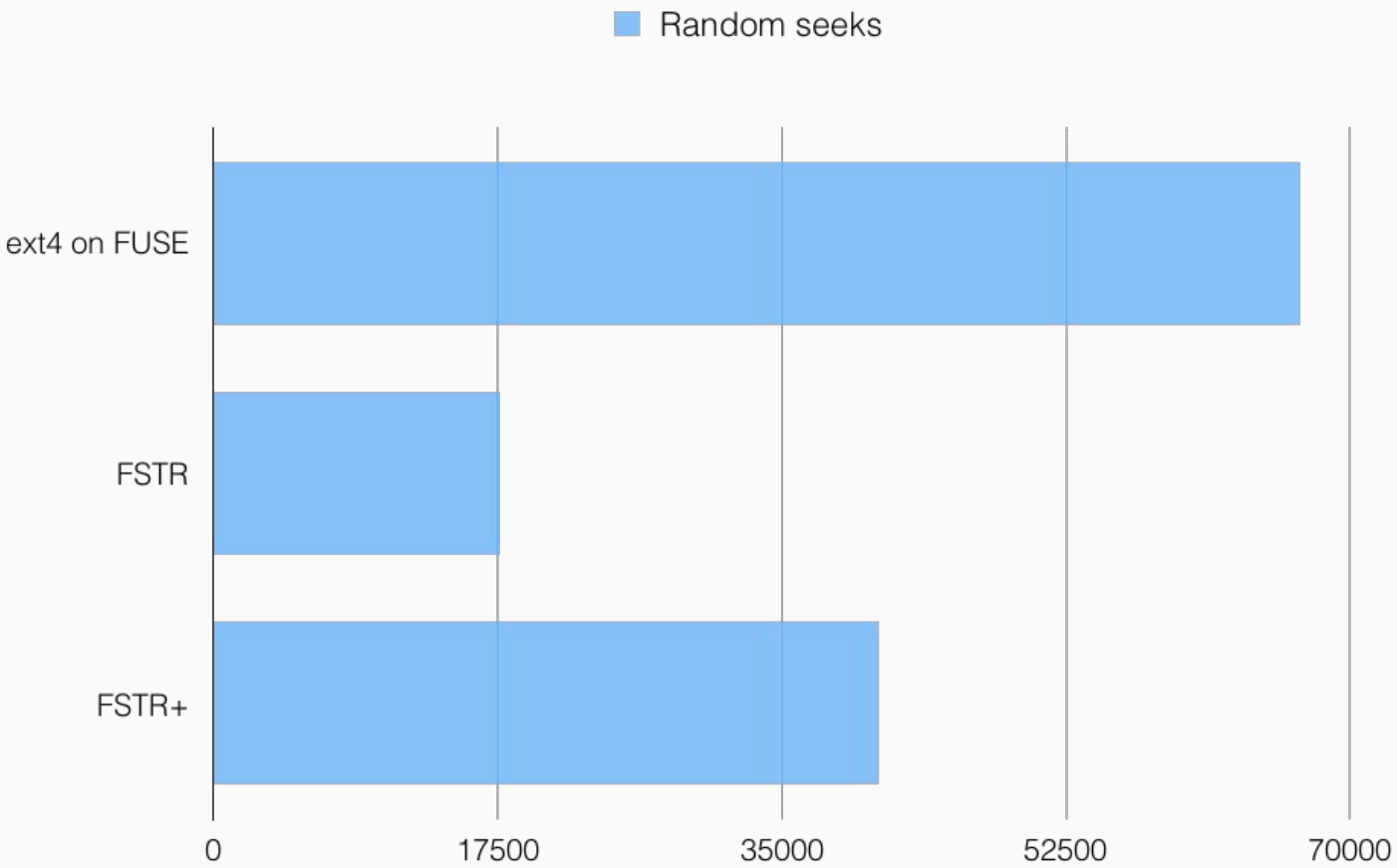
Benchmarks - *Sequential Character I/O*



Benchmarks - *Sequential Block I/O*



Benchmarks - *Random Seeks*



Key takeaways

- **layered development** makes unit testing easy
- **unit testing is effective** in catching bugs early on
- **stack variables > heap variables** for avoiding memory leaks
    - caller is responsible for allocation of memory
- **gdb is really helpful** for debugging

Questions?