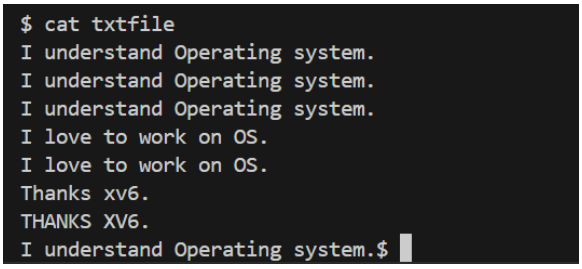
Project 4 Readme

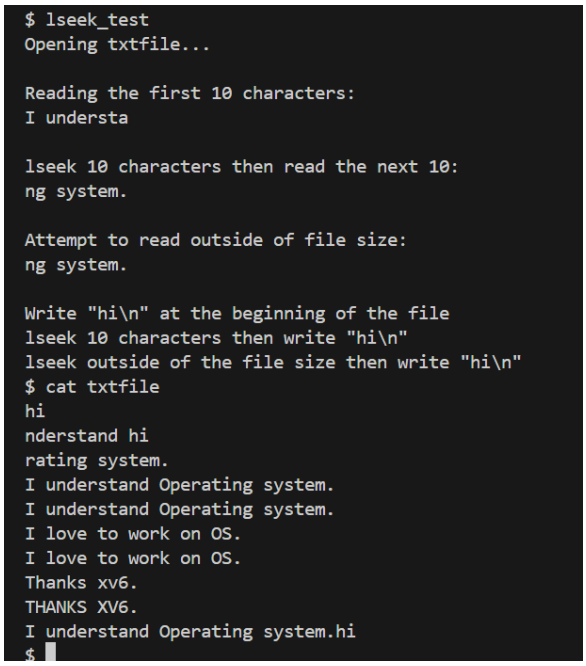
Group members (same group as project 1, 2 and 3)

- Molly Fawcett
- Shehzeen Syed
- Ethan Kocses

We ran xv6 through both WSL on Windows and directly on Ubuntu. xv6 was installed as detailed in project 0, except with a compiler downgrade.

Part 1: Adding lseek support





For this part of the project, we added support for the lseek system call in xv6. Inside lseek, we update the file structure's f->off field by adding the provided offset value to it. In our implementation, we handle the case where lseek moves the file offset (f->off) beyond the current file size (f->ip->size) by filling the holes with zero bytes. We calculated the size of the hole as the difference between the new offset and the file's current size. If the hole is greater than zero, we allocate a buffer of that size, initialize it with zeroes using memset, and then write each byte one by one to the file using writei(). This ensures that all the bytes between the old end of file and the new offset are explicitly written as zero.

To test the full behavior of lseek and our hole-filling logic, we wrote a user program called lseek_test. This program reads the first 10 characters, seeks forward 10 bytes and reads

again, writes "hi" at the beginning of the file, writes again at offset 10, and finally seeks to a large offset and writes once more. When attempting to read outside the file, it doesn't read thus the buffer variable doesn't get updated. So, when we print the buffer again, it prints the same thing as previously without anything additional. That last write triggers our hole-filling logic, so the file grows, and the gap is properly filled with zeros.

After running the program, we used cat txtfile to inspect the final contents of the file. The output confirms that lseek correctly updated the offset, that writing at a new position works as expected, and that data written far beyond the file size is preceded by a properly filled zero region. This demonstrates that both lseek and our hole-filling mechanism are functioning as intended. Even though we can't see the zeroes directly with cat, the fact that 'hi' appeared in the right place and the file didn't contain random garbage data proves the hole was filled properly.

Part 2: Adding support for symbolic links

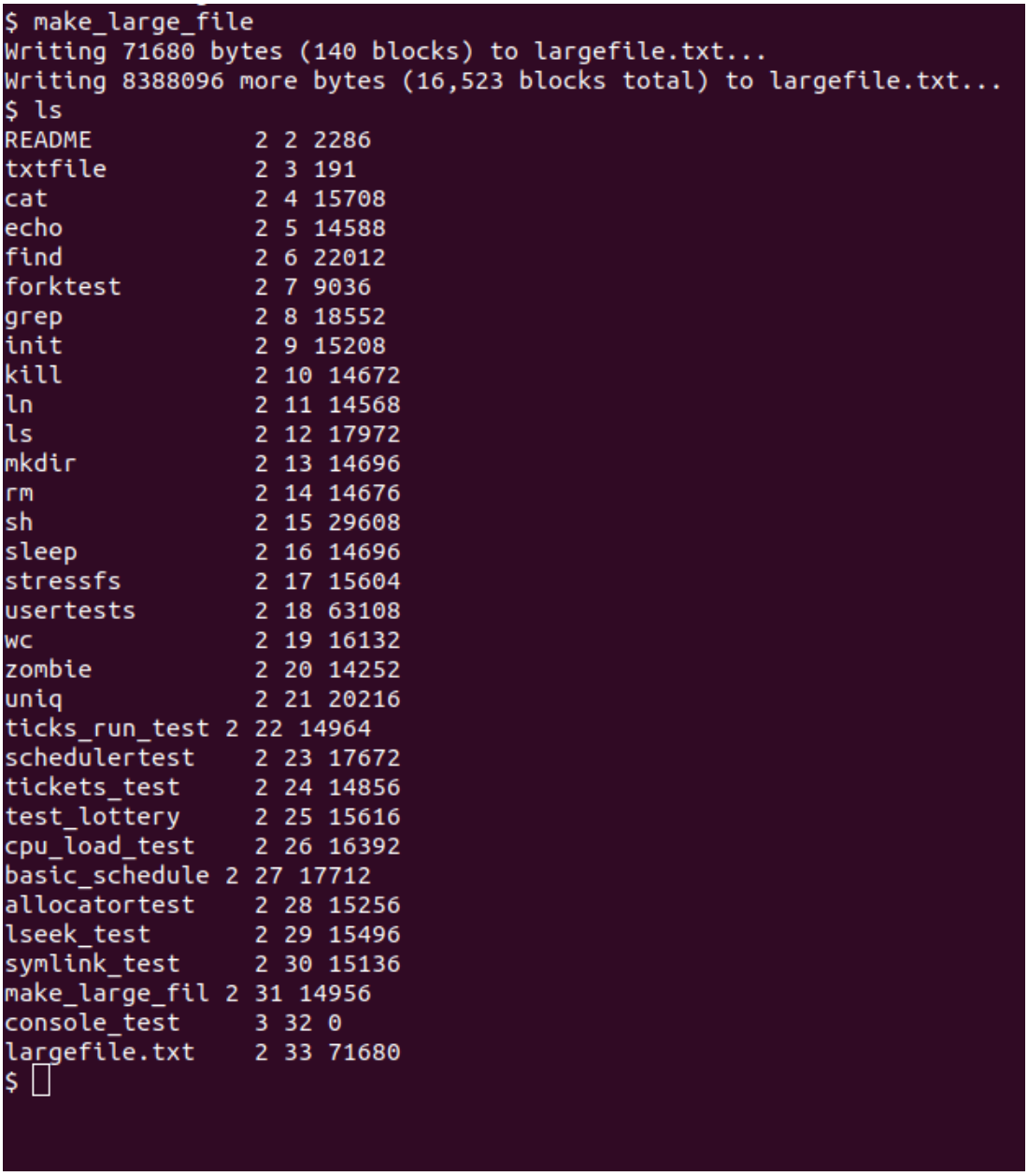
We modified the open() system call to handle symlinks by recursively resolving them. If a file is of type T_SYMLINK and the O_NOFOLLOW flag is not set, open() reads the stored path from the inode using readi(), null-terminates it, and recursively calls open() on that target path. This allows chained symlinks to be resolved. To avoid infinite loops caused by circular links, we included a depth counter that returns an error if the recursion exceeds 10 levels. If the user specifies the O_NOFOLLOW flag when calling open(), the kernel opens the symlink inode itself without resolving the target.

```
$ symlink_test
creating symlink: symlink1
Read from symlink: I understa
creating symlink: symlinkLoop2
creating symlink: symlinkLoop1
Failed to open symlink loop as expected
Read from symlink using O_NOFOLLOW: txtfile
$ symlink_test1
creating symlink: symlink3
Created link to symlink3 named 'link'
Reading from 'link':
Read: txtfile
link successfully unlinked
Confirming original txtfile still exist :  I understand Operati
$
```

To verify our symbolic link implementation, we created two user-level test programs that thoroughly test the functionality of symlink(), symlink resolution during open(), the O_NOFOLLOW flag behavior, and how link() and unlink() interact with symlinks. The first test program creates a symlink (symlink1) pointing to txtfile and confirms that reading from it returns the content of the target file, demonstrating that the kernel correctly follows the symlink. It also creates a symlink loop using symlinkLoop1 and symlinkLoop2, where each points to the other. Attempting to open this loop triggers our recursion depth limit and fails as expected, verifying that the kernel safely detects and blocks infinite symlink loops. Additionally, the program tests open() with O_NOFOLLOW on symlink1, and successfully reads the symlink's path (i.e txtfile), showing that the symlink was opened as a regular file, not followed. The second test program confirms that link() and unlink() do not follow symbolic links. It creates a link to the symlink symlink3, opens it with O_NOFOLLOW, and reads the symlink path content. Finally, it unlinks link and verifies that the symlink's target (txtfile) is unaffected.

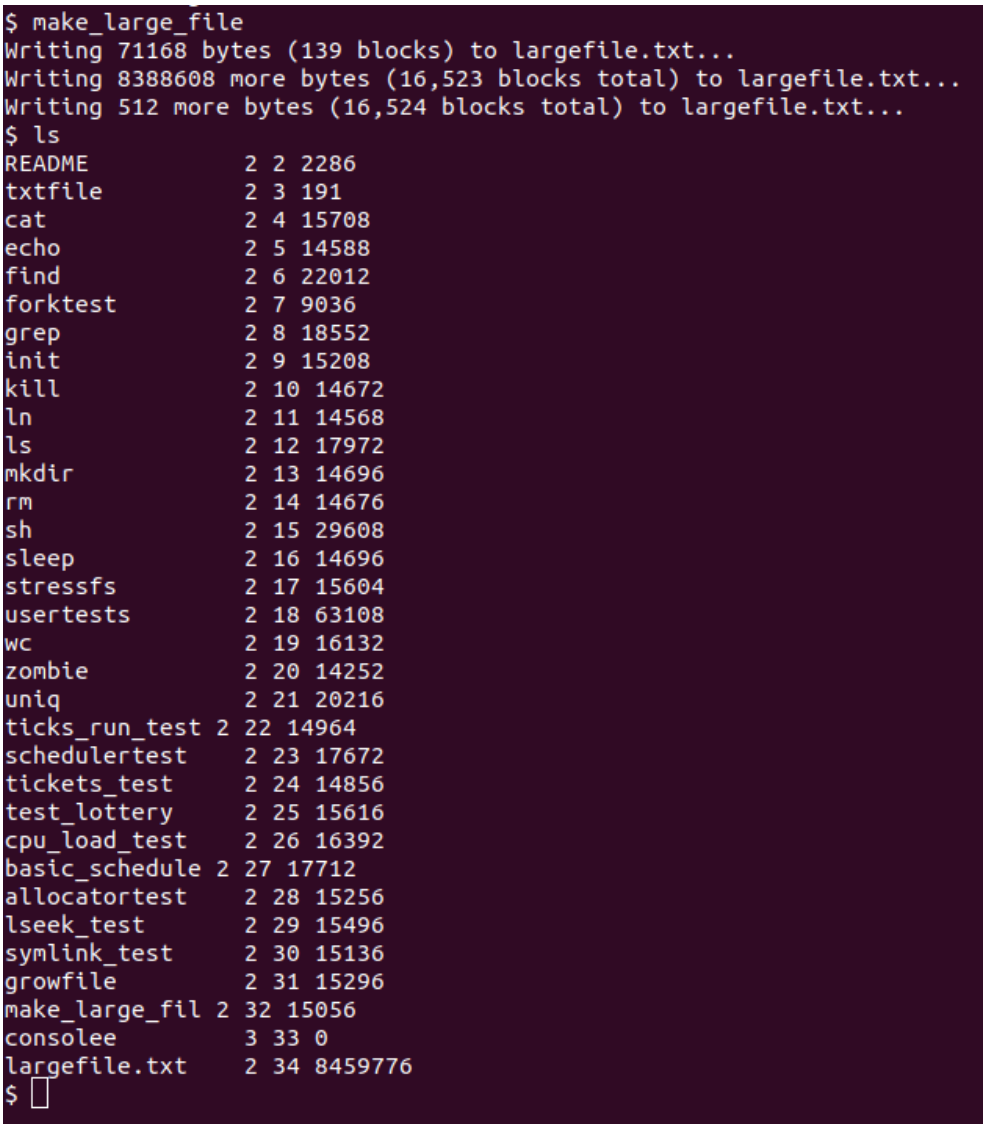
Part 3: Adding support for large files

First, we started with a user program called make_large_file.c that wrote to a file called largefile.txt. It first wrote 71680 bytes (140 blocks) of data to this file, then wrote the rest of the bytes separately to total to 16,523 blocks. We're not sure why, but we didn't get an error as described in the assignment. However, when using ls we found that the size of the file would never go over 140 blocks, even with some different tests (not shown here). So, we think that this is still a good indicator of the max file size. Below is a screenshot of what we described above. You can see largefile.txt at the bottom with only 140 blocks, even when trying to write 16,523 blocks.



Then, we implemented the doubly indirect block by removing 1 from the direct block count and changing the address arrays in the inode structs accordingly. We gave the doubly indirect block a size of 128 * 128 because it holds 128 indirect blocks which each hold 128 direct blocks. Then, we had to modify bmap() in fs.c to handle doubly indirect blocks. Bmap() now allocates the doubly indirect block only if it needs the space, then finds (or allocates) the proper indirect block within that depending on the input logical block number (bn), and finally finds (or allocates) the proper direct block within that indirect block, again based on bn. Finally, we had to modify itrunc() to have a nested for loop that frees all the blocks within the doubly indirect block.

We then modified the make_large_file.c user program to test the new max file size. As you can see below, it successfully writes a full 16,523 blocks (8,459,776 bytes) but does not write the extra block afterwords.



Part 4: Adding extent based files

For adding extent-based file support, we decided to use the suggested implementation of using the first 3 bytes for a direct data block pointer, and the last byte for length. This meant the maximum length for our extents would be 256.

To implement this structure, the bulk of our changes were made in the bmap() and itrunc) functions, similarly to in part 3. We had to rework how to 1) find an allocated block, 2) allocate a block that needs to be added to the file, and 3) free all blocks for the file.

This meant to find an allocated block: we had to loop through our list of direct pointers, split up the bits to read the information, count up how many blocks had been allocated so far by keeping track of the previous extents' lengths. When we find the extent that contains the logical block number, we can return it!

To allocate a block, we have to be a little clever. If we find ourselves looking at a direct pointer to nothing – we stop, similarly to the old implementation. We know we'll have to allocate a new block. But, there is a chance that we may actually just be able to add a new block to the previous extent, as long as its not 255 length yet AND the next free data block on disk is contiguous to the extent. If this is true, we just make the extent 1 block longer. If not, we start a new extent in the next direct pointer.

itrunc() for extents ended up being much simpler than for indirect blocks. Simply free as many contiguous blocks for each extent as the length value tells us to!

One of our test programs, extent_test, helps us observe what happens when one extent fills up, and we have to move to the next direct pointer. These print statements come from making a file as long as the max file size in our system: 256 length * 13 direct pointers = 3328 blocks.

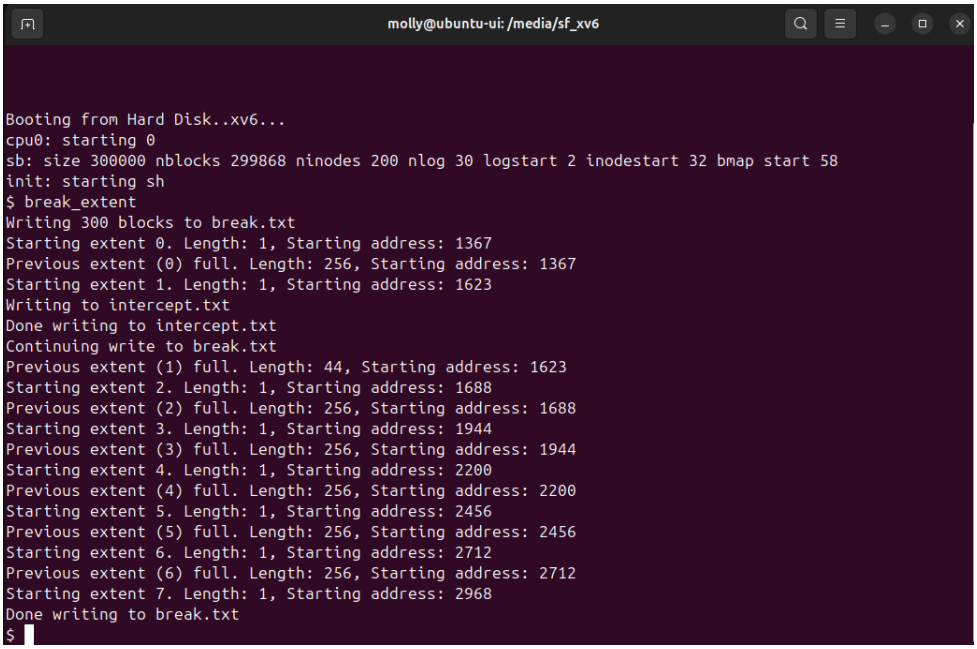
```
Writing 3328 (theoretical max) blocks to testlarge.txt
Starting extent 0. Length: 1, Starting address: 1368
Previous extent (0) full. Length: 256, Starting address: 1368
Starting extent 1. Length: 1, Starting address: 1624
Previous extent (1) full. Length: 256, Starting address: 1624
Starting extent 2. Length: 1, Starting address: 1880
Previous extent (2) full. Length: 256, Starting address: 1880
Starting extent 3. Length: 1, Starting address: 2136
Previous extent (3) full. Length: 256, Starting address: 2136
Starting extent 4. Length: 1, Starting address: 2392
Previous extent (4) full. Length: 256, Starting address: 2392
Starting extent 5. Length: 1, Starting address: 2648
Previous extent (5) full. Length: 256, Starting address: 2648
Starting extent 6. Length: 1, Starting address: 2904
Previous extent (6) full. Length: 256, Starting address: 2904
Starting extent 7. Length: 1, Starting address: 3160
Previous extent (7) full. Length: 256, Starting address: 3160
Starting extent 8. Length: 1, Starting address: 3416
Previous extent (8) full. Length: 256, Starting address: 3416
Starting extent 9. Length: 1, Starting address: 3672
Previous extent (9) full. Length: 256, Starting address: 3672
Starting extent 10. Length: 1, Starting address: 3928
Previous extent (10) full. Length: 256, Starting address: 3928
Starting extent 11. Length: 1, Starting address: 4184
Previous extent (11) full. Length: 256, Starting address: 4184
Starting extent 12. Length: 1, Starting address: 4440
Done writing
$ $
```

This is similarly demonstrated in our stat function after the txt file is made:

```
$ stat testlarge.txt
File type: T_EXTENT
Size: 1703936
Extent 0 starts at block 1368 and holds 256 blocks
Extent 1 starts at block 1624 and holds 256 blocks
Extent 2 starts at block 1880 and holds 256 blocks
Extent 3 starts at block 2136 and holds 256 blocks
Extent 4 starts at block 2392 and holds 256 blocks
Extent 5 starts at block 2648 and holds 256 blocks
Extent 6 starts at block 2904 and holds 256 blocks
Extent 7 starts at block 3160 and holds 256 blocks
Extent 8 starts at block 3416 and holds 256 blocks
Extent 9 starts at block 3672 and holds 256 blocks
Extent 10 starts at block 3928 and holds 256 blocks
Extent 11 starts at block 4184 and holds 256 blocks
Extent 12 starts at block 4440 and holds 256 blocks
$
```

Because we wrote all the bytes at once, we weren't able to demonstrate through this program that sometimes an extent may not actually reach its full potential length of 256. This version of break_extent demonstrates what happens when another file is written to in the middle of writing an extent file, causing one of its extents to get cut off.

(not the version of break_extent included in submission – that version writes enough blocks to cause an error)



Here, extent 1 gets cut off at just 44 blocks because intercept.txt was written at the next contiguous block when it got written.

The final version of break.txt demonstrates our theoretical max file size. We see that if we try to write any more than 3328 blocks, we receive an error:

.