1 Inode leak and its kernel remedy (50 points)

Task-1

To create a real inode leak, we need to create a situation where an inode is allocated but becomes unreachable.

So the strategy is:

1. Create a file, then unlink it while keeping it open. This creates an inode that has no directory entry but is still allocated
2. If we crash before closing the file descriptor, the inode becomes leaked

Please see the `ileak.c` in attachment

Task-2

I implement the second solution mentioned in the xv6 book, which maintains a list of inodes that need cleanup. This is more efficient than scanning the entire file system.

This task I use claude to help me solve the problem, generally I just ask it for help, and do the code review for it(with some small change).

Key Components:

1. Extended Superblock: Added fields to track orphaned inodes:
    - `norphans`: Number of orphaned inodes
    - `orphans[MAXORPHANS]`: Array of orphaned inode numbers
2. Orphan Management Functions:
    - `add_orphan(inum)`: Adds an inode to the orphan list when its link count drops to 0 but reference count > 0
    - `remove_orphan(inum)`: Removes an inode from the orphan list when properly freed
    - `cleanup_orphans()`: Processes all orphaned inodes during recovery
3. Modified System Calls:
    - `sys_unlink()`: Detects when an inode becomes orphaned and adds it to the list
    - `iput()`: Removes inodes from orphan list when they're properly freed
4. Recovery Integration:
    - `fsinit()`: Calls `cleanup_orphans()` during file system initialization after reboot

When `unlink()` is called on a file that still has open file descriptors, the system detects that `nlink == 0` but `ref > 1`, indicating an orphaned inode. Then we persistently stored the orphaned inode number in the superblock's orphan list.

After a crash and reboot, `fsinit()` automatically calls `cleanup_orphans()`, which:

- Reads the orphan list from the superblock
- For each orphaned inode, frees its data blocks and marks it as unused
- Clears the orphan list

If the process exits normally, `iput()` is called, which removes the inode from the orphan list and frees it properly.

2 Information leak, a user-level remedy, and a little hack (50 points)

Task-1

When we run:

```
echo Passwd > test
```

The file system writes the string "Passwd" to two different locations on disk:

1. Data block: The actual file content is written to a data block
2. Log block: Due to xv6's logging system, the same data is also written to the log as part of the transcation

The path is like:

```
1. begin_op() - Start file system transaction
2. File content "Passwd" written to log block
3. File content "Passwd" written to data block
4. end_op() - Commit transaction
```

So `grep -oba Passwd fs.img` finds the pattern in:

1. Location 1: The log area of the file system
2. Location 2: The actual data block where the file content resides

This is caused by the write-ahead logging design, where data exists temporarily in both the log and the final location.

Task-2

When we delete the file with `rm test`:

1. Directory entry removed: The file name "test" is removed from the directory
2. Inode freed: The inode is marked as free
3. But the data block wont immediately cleared: The actual content "Passwd" would remain in the data block

The file system only removes metadata (directory entry, inode, .etc) but doesnt zero out the data blocks for performance reasons. The data block still contains "Passed" until it gets reused by another file.

This is why grep still finds one match, the data block content persists even after deletion.

Task-3

When we create new files:

```
echo NTNU > file2
echo CSIE > file3
```

The key difference from Task 1 is the logging behavior:

For Task 1 (single file creation):

```
1. begin_op()
2. Write "Passwd" to log block
3. Write "Passwd" to data block
4. end_op() - but log may not be immediately cleared
```

Result: 2 copies exist temporarily (log + data)

But for Task 3 (multiple file operations):

```
1. begin_op()
2. Create file2, write "NTNU" to log
3. Write "NTNU" to data block
4. Create file3, write "CSIE" to log
5. Write "CSIE" to data block
6. end_op() - commit and clear log
```

The crucial difference is log management. When we perform multiple operations in sequence:

- Log gets reused/overwritten: The "NTNU" entry in the log gets overwritten when "CSIE" is written
- Log clearing: After the transaction completes, the log area is more likely to be cleared
- Only data block remains: Only the actual file content in the data block persists

xv6's logging system has limited log space. When we create multiple files. The log entries for earlier operations (like "NTNU") get overwritten by later operations (like "CSIE")

I try to grep the "CSIE" and it get the same output with the Task 1:

```
$ echo MES_NTNU_TEST > file
$ echo MES_CSIE_TEST > file2
$ ls
```

```
.              1 1 1024
..             1 1 1024
README         2 2 2292
cat            2 3 34264
echo           2 4 33184
forktest       2 5 16176
grep           2 6 37520
init           2 7 33648
kill           2 8 33104
ln             2 9 32920
ls             2 10 36288
mkdir          2 11 33160
rm             2 12 33152
sh             2 13 54728
stressfs       2 14 34048
usertests      2 15 179352
grind          2 16 49400
wc             2 17 35216
zombie         2 18 32528
ileak          2 19 38408
console        3 20 0
file           2 21 14
file2          2 22 14
$ QEMU: Terminated
mes@MesDesktop:~/NTNU-OS/xv6-riscv$ grep -oba MES_NTNU_TEST fs.img
825344:MES_NTNU_TEST
mes@MesDesktop:~/NTNU-OS/xv6-riscv$ grep -oba MES_CSIE_TEST fs.img
3072:MES_CSIE_TEST
826368:MES_CSIE_TEST
```

Task-4

I think the key insight is to overwrite the data blocks before deleting the file, ensuring the original content is completely destroyed.

1. Create a file with sensitive data:

   ```
   echo "SENSITIVE_PASSWORD_123" > secret.txt
   ```

   and we can check the data does exist:

   ```
   $ cat secret.txt
   "SENSITIVE_PASSWORD_123"
   ```

2. Overwrite with large amounts of dummy data:

   ```

   ```

```
echo "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" >
secret.txt
echo "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB" >
secret.txt
echo "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC" >
secret.txt
echo "DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD" >
secret.txt
```

3. delete the file

```
rm secret.txt
```

4. check data not exist:

```
grep -oba "SENSITIVE_PASSWORD_123" fs.img
```

Task-5

Actually I didnt get the point but okay, cant we just grep it like what we did above?

But anyway, let's add the file first:

```
echo MyPass=061208 > test.txt
```

then remove it:

```
rm test.txt
```

and grep it:

```
grep -oba "MyPass=[0-9]\{6\}" fs.img
```

and we get the passwd...?

```
mes@MesDesktop:~/NTNU-OS/xv6-riscv$ grep -oba "MyPass=[0-9]\{6\}" fs.img
825344:MyPass=061208
```