Optional assignment for refresher module quiz

- Implement IPC using shared memory in xv6
  - To be done individually

- Submit a design documentation by 1st April

- The design documentation should contain
  - New system call APIs to setup IPC, input parameters, return value
  - A sample program that uses those APIs

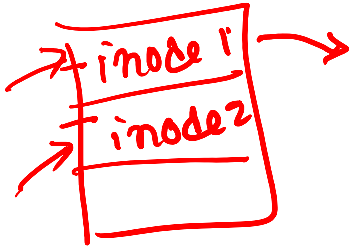
Inode

- Inode contains metadata corresponding to a file or directory
- in-memory inode structure also contains a lock
- Before, reading/writing to a file xv6 acquires the inode lock
- Multiple threads cannot read/write the file at the same time

Disk accesses are slow

- To eliminate the cost of slow disk reads an in-memory buffer cache is maintained by xv6

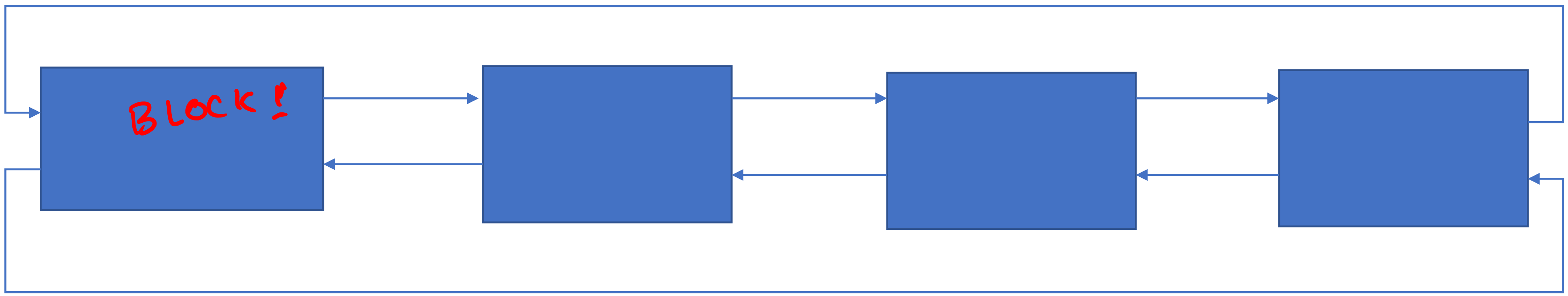
Buffer cache



buf = bread()

bwrite();

brelse



next

prev

Sector ID

char data[512];

= bread(BLOCK1),

brelse

Buffer cache

- Synchronize access to disk block
  - Only one thread can access a disk block at a given time

Buffer cache replacement

- LRU replacement policy is used in xv6
  - On every access xv6 moves the buffer to the front of the linked list
  - The last node in the linked list is the least recently used buffer

Buffer cache

```
struct buf {
  int flags;        // BUSY or DIRTY or CLEAN
  uint blockno;  // block number
  ... prev , next
  uchar data[512];
};
```

Buffer cache

```
buf = bread (dev, blknum);
…
bwrite (buf);
brelse (buf);
```

Can we avoid double copy?

Application:

read (fd, buf, size);

--

Kernel:

sector_id = offset_to_sector (fd);

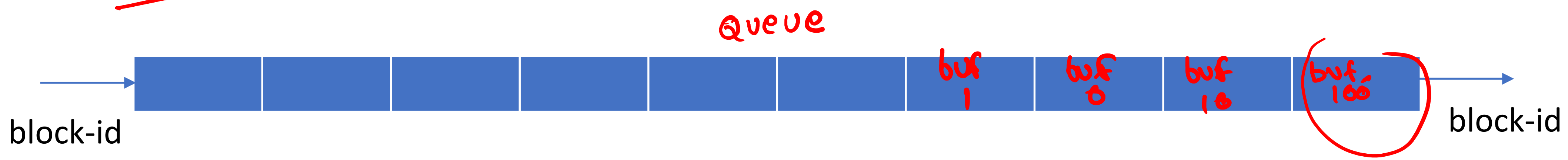
kbuf = bread (dev, sector_id);

memcpy (buf, kbuf, …);

bwrite(buf, 100)
bwrite(buf, 10)
bwrite(buf, 0)
bwrite(buf, 1)

read(sector_id, number of sectors)

idestart



Wait until disk is ready

Configure disk device to read/write contiguous sectors

Disk interface takes sector-id of the first sector and the number of sectors to read/write data

Data is written to the port 0x1F0

ideintr

Disk device generates an interrupt when the writing is complete or the data is ready for reading

Data is read from the port 0x1F0

In the interrupt handler, xv6 pops the buffer from the queue and initiate next request (if present) is the queue

What is the disk scheduling policy in xv6?

FIFO

FCFS scheduling

- First-come, first-served

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

53 – 98 – 183 – 37 – 122 – 14 – 124 – 65 – 67

Total head movement = 640

Problem: not the best algorithm for performance

SSTF scheduling

- Shortest seek time first

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

53 – 65 – 67 – 37 – 14 – 98 – 122 – 124 – 183

Total head movement = 236

Problem: starvation

a - 65
b - 65

buf = bread(65)
buf[100] = 20;

buf
65

buf = bread(65)
buf[100] = 40;

buf
65

SSTF

- SSTF is not the optimal algorithm

- Total head movement can be further reduced by selecting a different node which is not at the shortest distance

Elevator algorithm

- Starts from one end of the disk and moves forward to the other end of the disk, servicing in between requests

- After reaching the other end, the head moves backwards to the front of the disk, servicing intermediate requests

Elevator algorithm

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

Say head is moving towards zero

53 – 37 – 14 – 65 – 67 – 98 – 122 – 124 – 183

No starvation!

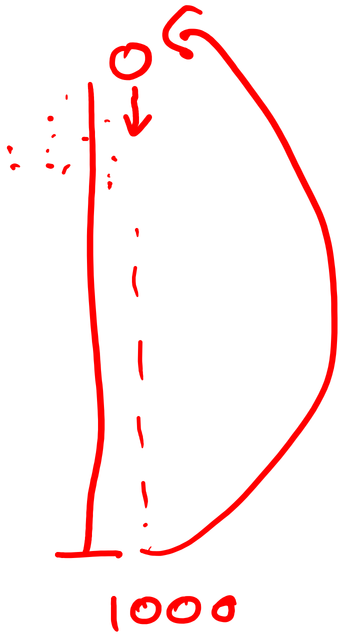
Circular SCAN algorithm

- After reaching the other end reset the head to beginning of the disk

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

53 – 65 – 67 – 98 – 122 – 124 – 183 – 14 – 37



Crash recovery

- A file system must be aware that a power failure may happen at any time
- A power failure must not put the file system in an inconsistent state due to partial write

Create

echo > a/b

1. allocate inode “x” for file b

2. create an entry which points to “x” in directory “a”

3. Initialize inode “x”

Create

a → dangling pointer

echo > a/b

1. allocate inode “x” for file b

2. create an entry which points to “x” in directory “a”

3. Initialize inode “x”

A power failure after step 2 will leave a dangling pointer in directory “a”

very bad!

Create

echo > a/b

1. allocate inode “x” for file b

2. Initialize inode “x”

3. create an entry which points to “x” in directory “a”

A power failure after step 2 will create space leak.

not so bad!

Assumption

- Writes to individual sector on disk are atomic

- Even if a power failure happens in between the sector write, the disk has enough energy to complete the write

Unlink

- rm a/b

1. Free data blocks of file “b”
2. Free inode block
3. Remove directory entry corresponding to “b” from “a”

Unlink

- rm a/b

1. Free data blocks of file “b”

2. Free inode block

3. Remove directory entry corresponding to “b” from “a”

If a crash happens after step 1, inode will point to dangling pointers

If a crash happens after step 2, directory “a” will point to dangling pointer

Filesystem inconsistency

- Dangling references
  - inode -> free blocks
  - directory entry -> free inode

- A space leak is not as bad as dangling references

Synchronous metadata update

- Always initialize the disk block before creating references
  - Initialize (update) disk block in buffer cache
  - Initiate write to disk
  - wait for write to finish
  - Create references
- Always erase the references before freeing the disk blocks

Synchronous metadata update

- Create
  - allocate file inode (update dirty bit on disk)
  - Initialize inode (wait until inode is updated on disk)
  - Create directory entry
- Unlink
  - remove directory entry
  - free inode
  - free data blocks

Space leak

- A separate program (fsck on Linux) walks the entire filesystem to free the unreferenced but allocated disk blocks
- fsck scans the whole filesystem to fix inconsistencies
- May take a long amount of time depending on the size of the disk

Can fsck detect all types of inconsistencies?

- No

mv a/b c/d

1. remove “b” (inode “x”) from “a”
2. add an entry “d” (inode “x”) to “c”

A crash after step 1 will delete the file “b”

Can fsck detect all types of inconsistencies?

- No

mv a/b c/d

1. add an entry “d” (inode “x”) to “c”
2. remove “b” (inode “x”) from “a”

A crash after step 1, leave the inode “x” with multiple references to it

fsck won’t fix it

Write back cache

- Write back cache improves performance
  - but makes recovery harder

- Unlink
  - remove directory entry
  - free inode
  - free data blocks

- Somehow remember the dependency between writes before persisting them to disk

Logging

- To eliminate the shortcomings of synchronous metadata update, xv6 implements logging
- In this design, data is first saved to a temporary disk location (also called a log) before the original write
- Logging ensures atomicity of operations

Operation

mv a/b c/d

begin_op()

1. add an entry “d” (inode “x”) to “c”
2. remove “b” (inode “x”) from “a”

end_op()

xv6 guarantees the atomicity of writes enclosed with begin_op and end_op. In this case, after the crash, the file system will see either both the changes or none of them.

Logging

syscall:

```
    begin_op();
    …
    bp = bread(…);
    bp->data[…] = …;
    log_write (bp);
    brelse (buf);
    …
    end_op();
```

xv6 ensures the atomicity of writes enclosed with begin_op and end_op.

Logging

- Multiple system calls can log in parallel

- The data is saved on the disk and log asynchronously
  - write back cache

- Faster than synchronous writes

Group commit

Ensure no other operation is executing in parallel

Write all outstanding writes to log space

Write log header (contains locations of logged block)

; after this point data is guaranteed to be recovered after crash

Copy data from the log to the original disk locations

Erase the log

Advantages of group commit

- Lazy writing to log (write back cache)
- Lazy writing to original locations
- Writing in batches (good for disk scheduling algorithm)

Recovery

- If a complete log header is present, the recovery code copy data from log space to their original locations

- Erase the log

Group commit

- Why can other operations not execute in parallel with group commit?

Concurrent group commit

disallow new operations

wait for existing operations to finish

make a copy of dirty blocks

allow new operations

write copied data to the log file

write log header

copy data from log file to the original disk locations

Concurrent group commit

- The previous algorithm can also use copy on write optimization to further reduce the overheads
- Notice, that the group commit is only copying from memory to memory when the concurrent operations are disallowed
- During the actual disk write the concurrent operations are allowed

Linux ext3 also implements logging

- Concurrent operations can execute along with group commit
- Group commit after a fixed interval (say 5 sec)
- Improve performance

What is the problem with periodic commit?

- synchronous writes succeeds even though the data is not written to the disk
  - e.g., write system call

- Application centric view

- User centric view

What is the problem with periodic commit?

- How does user know that data has been written to the disk?
  - If the program prints something on the console, or sending something over the network, etc.

  - What if the user is notified about the disk write but the data has not been updated on the disk
    - very bad

  - What if application is notified about the disk write but not the user
    - acceptable

Periodic commit

- The output is not externalized to the user until the commit occurs
  - OS buffers writes to console or network

- Improving throughput at the cost of latency