Section 11: File Systems, Reliability, and Queueing Theory

CS162

April 12, 2019

Contents

1 Warmup **2**

2 Vocabulary **4**

3 Problems **6**

3.1 Extending an inode 6

3.2 Queuing Theory 11

3.3 Tying it all together 11

1 Warmup

What are the ACID properties? Explain each one and discuss the implications of a system without that property.

Atomicity - data left in intermediate state
Consistency - data left in invalid state
Isolation - concurrent transactions may interfere with each other
Durability - crashes may destroy committed transactions

Name 2 different RAID levels that offer redundancy. For each level, explain how a recovery program could recover data from a degraded array.

RAID 1 - The recovery program copies all the data from the good disk to a replacement
RAID 5 - The recovery program uses the data and parity bits from the N - 1 good disks to rebuild the data and parity that should belong on the Nth disk.

Explain the difference between a hard link and a soft link (symbolic link).

A hard link is just a directory entry. Multiple hard links can exist for a single inode. Modifying the name of a hard link will not modify the inode itself or any other hard links to that inode. Multiple hard links to the same file must all be on the same filesystem, since hard links point to an inode, and an inode can only be on a single filesystem. Hard links contribute to an inode's reference count.

A soft link (symbolic link) is a special file system object that contains a path to another file or directory. When programs access soft links, the standard library will resolve the soft link's target file. Soft links do not guarantee the existence of the target, like hard links do. Soft links can point to files and directories in different filesystems. Soft links do not contribute to the reference count.

How could you implement hard links for the FAT file system? What problem would you encounter?

In FAT, each directory entry contains the first block number for that file. This is because the blocks of a file is determined by its first block, i.e. the entry point into the FAT. Thus, a hard link might be implemented in a FAT system by having a directory entry contain the same first block number as another directory entry.

The problem is that a FAT entry, unlike an inode, doesnt keep track of how many directory entries refer to that entry. Thus, if a user deletes a file the block of that file will be put back on the list of free blocks, even if another hard link to that file exists.

What is a journaled file system? Explain the purpose of the file system's "journal".

A journaled file system maintains a journal, which contains the most recent updates to file system metadata. The journal is used to recover the file system's state, in case of a power failure or a system crash.

Discuss the advantages and drawbacks of memory mapped file accesses compared to traditional disk accesses for small random file reads and writes to many files of varying size.

Memory mapped files allow for in-place updates without the need to seek into the file to the appropriate location.

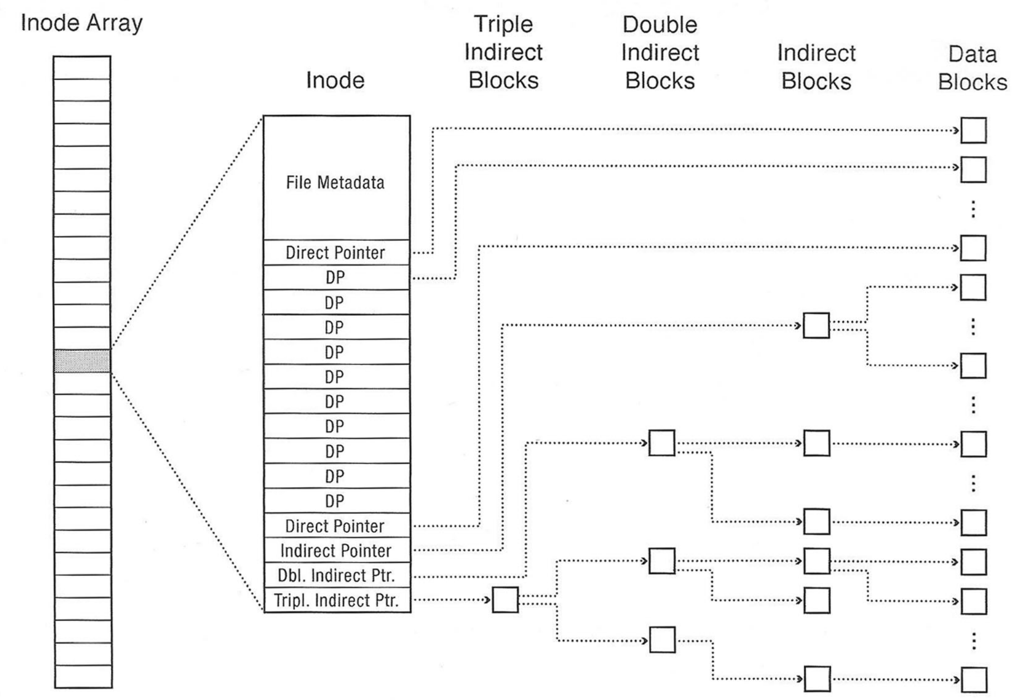
However, since this access pattern accesses data in many different files, it will incur overhead for paging the files in and out. This may or may not be worse than the overhead of traditional I/O, which involves overhead in the syscall framework and copying memory.

2 Vocabulary

- **Unix File System (Fast File System)** - The Unix File System is a file system used by many Unix and Unix-like operating systems. Many modern operating systems use file systems that are based off of the Unix File System.

- **inode** - An inode is the data structure that describes the metadata of a file or directory. Each inode contains several metadata fields, including the owner, file size, modification time, file mode, and reference count. Each inode also contains several data block pointers, which help the file system locate the file's data blocks.

  Each inode typically has 12 direct block pointers, 1 singly indirect block pointer, 1 doubly indirect block pointer, and 1 triply indirect block pointer. Every direct block pointer directly points to a data block. The singly indirect block pointer points to a block of pointers, each of which points to a data block. The doubly indirect block pointer contains another level of indirection, and the triply indirect block pointer contains yet another level of indirection.



- **Transaction** - A transaction is a unit of work within a database management system. Each transaction is treated as an indivisible unit which executes independently from other transactions. The ACID properties are usually used to describe reliable transactions.

- **ACID** - An acronym standing for the four key properties of a reliable transaction.

  Atomicity - the transaction must either occur in its entirety, or not at all.

  Consistency - transactions must take data from one consistent state to another, and cannot compromise data integrity or leave data in an intermediate state.

Isolation - concurrent transactions should not interfere with each other; it should appear as if all transactions are serialized.

Durability - the effect of a committed transaction should persist despite crashes.

- **Idempotent** - An idempotent operation is an operation that can be repeated without effect after the first iteration.
- **Logging file system** - A logging file system (or journaling file system) is a file system in which all updates are performed via a transaction log ("journal") to ensure consistency, in case the system crashes or loses power. Each file system transaction is first written to an append-only redo log. Then, the transaction can be committed to disk. In the event of a crash, a file system recovery program can scan the journal and re-apply any transactions that may not have completed successfully. Each transaction must be idempotent, so the recovery program can safely re-apply them.
- **Queuing Theory** Here are some useful symbols: (both the symbols used in lecture and in the book are listed)
  - μ is the average service rate (jobs per second)
  - T_{ser} or S is the average service time, so $T_{ser} = \frac{1}{\mu}$
  - λ is the average arrival rate (jobs per second)
  - U or u or ρ is the utilization (fraction from 0 to 1), so $U = \frac{\lambda}{\mu} = \lambda S$
  - T_q or W is the average queuing time (aka waiting time) which is how much time a task needs to wait before getting serviced (it does not include the time needed to actually perform the task)
  - T_{sys} or R is the response time, and it's equal to $T_q + T_{ser}$ or $W + S$
  - L_q or Q is the average length of the queue, and it's equal to λT_q (this is Little's law)

3 Problems

3.1 Extending an inode

Consider the following `inode_disk` struct, which is used on a disk with a 512 byte block size.

```
/* Definition of block_sector_t */
typedef uint32_t block_sector_t;

/* Contents of on-disk inode. Must be exactly 512 bytes long. */
struct inode_disk
  {
    off_t length;                       /* File size in bytes. */
    block_sector_t direct[12];          /* 12 direct pointers */
    block_sector_t indirect;            /* a singly indirect pointer */
    uint32_t unused[114];               /* Not used. */
  };
```

Why isn't the file name stored inside the `inode_disk` struct?

The file name belongs in the directory entry.

What is the maximum file size supported by this inode design?

It is $2^{16} + 12 \times 2^{9}$ bytes

How would you design the in-memory representation of the indirect block? (e.g. the disk sector that corresponds to an inode's `indirect` member)

You could use a `block_sector_t[128]`, which is exactly 512 bytes.

Implement the following function, which changes the size of an inode. If the resize operation fails, the inode should be unchanged and the function should return `false`. Use the value `0` for unallocated block pointers. You do not need to write the inode itself back to disk. You can use these functions:

- "`block_sector_t block_allocate()`" – Allocates a disk block and returns the sector number. If the disk is full, then returns 0.
- "`void block_free(block_sector_t n)`" – Free a disk block.
- "`void block_read(block_sector_t n, uint8_t buffer[512])`" – Reads the contents of a disk sector into a buffer.
- "`void block_write(block_sector_t n, uint8_t buffer[512])`" – Writes the contents of a buffer into a disk sector.

```
bool inode_resize(struct inode_disk *id, off_t size) {
  block_sector_t sector;
  for (int i = 0; i < 12; i++) {
    if (size <= 512 * i && id->direct[i] != 0) {
      free_block(id->direct[i]);
      id->direct[i] = 0;
    }
    if (size > 512 * i && id->direct[i] == 0) {
      sector = allocate_block();
      if (sector == 0) {
        inode_resize(id, id->length);
        return false;
      }
      id->direct[i] = sector;
    }
  }
  if (id->indirect == 0 && size <= 12 * 512) {
    id->length = size;
    return true;
  }
  block_sector_t buffer[128];
  if (id->indirect == 0) {
    memset(buffer, 0, 512);
    sector = allocate_block();
    if (sector == 0) {
      inode_resize(id, id->length);
      return false;
    }
    id->indirect = sector;
  } else {
    block_read(id->indirect, buffer);
  }
  for (int i = 0; i < 128; i++) {
    if (size <= (12 + i) * 512 && buffer[i] != 0) {
      free_block(buffer[i]);
      buffer[i] = 0;
    }
    if (size > (12 + i) * 512) && buffer[i] == 0) {
      sector = allocate_block();
      if (sector == 0) {
        inode_resize(id, id->length);
        return false;
      }
      buffer[i] = sector;
    }
  }
  block_write(id->indirect, buffer);
  id->length = size;
  return true;
}
```

Another solution that you may find useful

```
#include <stdio.h>
#include <unistd.h>

typedef uint32_t block_sector_t;

struct inode_disk
{
    off_t length; /* File size in bytes. */
    block_sector_t pointers[13]; /* 12 direct pointers and 1 indirect pointer*/
    uint32_t unused[114]; /* Not used. */
};

struct indirect_disk
{
    block_sector_t pointers[128];
}

bool calculate_indices(int blocknumber, int *offsets, int *offset_cnt)
{
    if (sector_idx < 12)
    {
        offsets[0] = sector_idx;
        *offset_cnt = 1;
        return true;
    }
    sector_idx -= 12;
    if (sector_idx < PTRS_PER_SECTOR)
    {
        offsets[0] = 12;
        offsets[1] = sector_idx % PTRS_PER_SECTOR;
        *offset_cnt = 2;
        return true;
    }
    return false;
}

bool inode_change_block(struct inode_disk *id, block_sector_t block, bool add)
{
    int offsets[2];
    int offset_cnt;
    int i = 0;
    uint8_t zeros[512];
    struct indirect_disk cur;
    block_sector_t cur_indirect;

    memset(zeros, 0, sizeof(zeros));
    calculate_indices(block, offsets, &offset_cnt);
    for (i = 0; i < offset_cnt; i++)
    {
        if (i == 0)
        {
            if (add && id->pointers[offsets[0]] == 0)
            {
                block_sector_t next_indirect;
                if ((next_indirect = block_allocate()) == 0)
                    return false;
                id->pointers[offsets[0]] = next_indirect;
                block_write(next_indirect, zeros);
                cur_indirect = next_indirect;
            }
            if (!add && ((offset_cnt == 1) || (offset_cnt == 2 && offsets[1] == 0)))
            {
                block_free(id->pointers[offsets[0]]);
                id->pointers[offsets[0]] = 0;
            }
        }
        else
        {
            block_read(cur_indirect, &cur);
            if (add && cur.pointers[offsets[1]] == 0)
            {
                block_sector_t next_indirect;
                if ((next_indirect = block_allocate()) == 0)
                    return false;
                cur.pointers[offsets[1]] = next_indirect;
                block_write(next_indirect, zeros);
                block_write(cur_indirect, &cur);
                cur_indirect = next_indirect;
            }
            if (!add)
            {
                block_free(cur.pointers[offsets[1]]);
                cur.pointers[offsets[1]] = 0;
                block_write(cur_indirect, &cur);
            }
        }
    }

}

bool inode_resize(struct inode_disk *id, size_t size)
{
    size_t cur_blocks;
    size_t new_blocks;
    off_t cur;
    off_t d_cur;
    cur_blocks = id->length/BLOCK_SIZE;
    new_blocks = size/BLOCK_SIZE;
    if (new_blocks > cur_blocks)
    {
        //allocate blocks from [cur_blocks+1, new_blocks];
        for (cur = cur_blocks + 1; cur <= new_blocks; cur++)
        {
            if (!inode_change_block(id, cur, true))
            {
                // must deallocate if failed to allocate
                for (d_cur = cur_blocks + 1; d_cur < cur; d_cur++)
                {
                    inode_change_block(id, d_cur, false);
                }
                return false;
            }
        }
        id->length = size;
        return true;
    }
    else if (cur_blocks > new_blocks)
    {
    //deallocate blocks from [new_blocks+1, cur_blocks];
        for (cur = new_blocks + 1; cur <= cur_blocks; cur++)
            inode_change_block(id, d_cur, false);
        id->length = size;
        return true
    }
    else
    {
        id->length = size;
    }
}
```

3.2 Queuing Theory

Explain intuitively why response time is nonlinear with utilization. Draw a plot of utilization (x axis) vs response time (y axis) and label the endpoints on the x axis.

Even with high utilization (99%), some of the time (1%), the server is idle, which is a waste. All this wasted time adds up, and in the steady state, the queue becomes very long.
Graph should be linear-ish close to $u = 0$ and grow asymptotically toward ∞ at $u = 1$.

If 50 jobs arrive at a system every second and the average response time for any particular job is 100ms, how many jobs are in the system (either queued or being serviced) on average at a particular moment? Which law describes this relationship?

$50 \times 0.1 = 5$ (5 jobs at any time). This is Little's law.

Is it better to have N queues, each of which is serviced at the rate of 1 job per second, or 1 queue that is serviced at the rate of N jobs per second? Give reasons to justify your answer.

One server that can process N jobs per millisecond is faster. Better response time ($\frac{1}{N}$ sec vs 1 sec) and better utilization (no load-balancing problems), which gives you lower queuing delays on average.

What is the average queueing time for a work queue with 1 server, average arrival rate of λ, average service time S, and squared coefficient of variation of service time $\mathbf{C}$?

$T_q = T_{ser}(\frac{u}{1-u})(\frac{C+1}{2})$ where $u = \lambda S$

What does it mean if $\mathbf{C} = 0$? What does it mean if $\mathbf{C} = 1$?

If $\mathbf{C} = 0$, then your arrival rate is regular and deterministic, which means that tasks arrive at a constant rate.
If $\mathbf{C} = 1$, then your arrival rate can be modeled as a Poisson distribution, and the interval between arrivals can be modeled as a exponential distribution.

3.3 Tying it all together

Assume that you have a disk with the following parameters:

- 1TB in size
- 6000RPM
- Data transfer rate of 4MB/s (4×10^6 bytes/sec)
- Average seek time of 3ms
- I/O controller with 1ms of controller delay
- Block size of 4000 bytes

What is the average rotational delay?

$\frac{1}{2} \times \frac{60\text{sec/minute}}{6000\text{RPM}} = 5\text{ms}$

What is the average time it takes to read 1 random block? Assume no queuing delay.

$\frac{4{,}000\text{bytes}}{4{,}000{,}000\text{bytes/sec}} = 1\text{ms}$, and $1 + 3 + 5 + 1 = 10\text{ms}$

Will the actual measured average time to read a block from disk (excluding queuing delay) tend to be lower, equal, or higher than this? Why?

It will be lower, because the operating system will use a disk scheduling algorithm to improve locality. This model assumes the disk is always seeking to a random location.

Assume that the average I/O operations per second demanded is 50 IOPS. Assume a squared coefficient of variation of $\mathbf{C} = 1.5$. What is the average queuing time and the average queue length?

$$
\begin{aligned}
T_q &= T_{ser}\left(\frac{u}{1-u}\right)\left(\frac{\mathbf{C}+1}{2}\right) \\
u &= \lambda T_{ser} \\
u &= 50\text{IOPS} \times 0.01\text{sec} \\
u &= 0.5 \\
T_q &= 10\text{ms}\left(\frac{0.5}{1-0.5}\right)\left(\frac{1.5+1}{2}\right) \\
T_q &= 12.5\text{ms} \\
L_q &= T_q\lambda \\
L_q &= 0.0125 \times 50 \\
L_q &= 0.625 \text{ operations}
\end{aligned}
$$