Discussion 9: File Systems (continued), Distributed Systems

November 15, 2023

Contents

1 File Growth

In this question, we will explore how to grow a file in Pintos. This will be very similar to one of your tasks in Project File System.

In Pintos, `struct inode_disk` from `filesys/inode.c` represents an inode. Let's examine a modified `struct inode_disk` with 12 direct pointers and an indirect pointer.

```
#define BLOCK_SECTOR_SIZE   512

typedef uint32_t block_sector_t;

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

1. What is the purpose of the `unused` member in `struct inode_disk`?

   Each inode must be of size equivalent to the block size. The direct pointers, indirect pointer, and size only take up
   $$4 \times (12 + 1 + 1) = 56 \text{ bytes}$$
   As a result, we need to pad the rest of the struct with $512 - 56 = 456$ bytes, or equivalently an array of 114 32-bit integers.

2. What is the maximum file size supported by this file system?

   There are $512/4 = 128$ direct pointers in an indirect block, so the maximum file size is
   $$\begin{aligned} 512 \times (12 + 128) &= 2^9 \times (2^2 \times 3 + 2^7) \\ &= 2^{10} \times (2 \times 3 + 2^6) \\ &= 70 \text{ KiB} \end{aligned}$$

3. What data structure should you to represent an indirect block?

   Since an indirect block is an array of direct pointers, `block_sector_t[128]` would be adequate.

4. Implement `inode_resize` to grow or shrink the inode based on the given `size`. If the resize operation fails for any reason, the inode should be unchanged and the function should return `false`. Assume unallocated block pointers have value `0`.

```
/* Allocates a disk sector and returns its number. */
block_sector_t block_allocate(void);

/* Frees disk sector N. */
void block_free(block_sector_t n);

/* Reads contents of disk sector N into BUFFER. */
void block_read(block_sector_t n, uint8_t buffer[512]);

/* Write contents of BUFFER to disk sector N. */
void block_write(block_sector_t n, uint8_t buffer[512]);
```

```
bool inode_resize(struct inode_disk* id, off_t size) {
  block_sector_t sector;

  /* Handle direct pointers. */
  for (int i = 0; i < 12; i++) {
    if (size <= BLOCK_SECTOR_SIZE * i && id->direct[i] != 0) {
      /* Shrink. */
      ________________________________;
      ________________________________;
    } else if (size > BLOCK_SECTOR_SIZE * i && id->direct[i] == 0) {
      /* Grow. */
      ________________________________;
    }
  }

  /* Check if indirect pointers are needed. */
  if (id->indirect == 0 && size <= 12 * BLOCK_SECTOR_SIZE) {
    id->length = size;
    return true;
  }
  block_sector_t buffer[128];
  memset(buffer, 0, 512);
  if (id->indirect == 0) {
    /* Allocate indirect block. */
    ________________________________;
  } else {
    /* Read in indirect block. */
    ________________________________;
  }

  /* Handle indirect pointers. */
  for (int i = 0; i < 128; i++) {
    if (size <= (12 + i) * BLOCK_SECTOR_SIZE && buffer[i] != 0) {
      /* Shrink. */
      ________________________________;
      ________________________________;
    } else if (size > (12 + i) * BLOCK_SECTOR_SIZE && buffer[i] == 0) {
      /* Grow. */
      ________________________________;
    }
  }
  if (size <= 12 * BLOCK_SECTOR_SIZE) {
    ________________________________;
    ________________________________;
  } else {
    ________________________________;
  }
  id->length = size;

  return true;
}
```

```
bool inode_resize(struct inode_disk* id, off_t size) {
  block_sector_t sector;

  /* Handle direct pointers. */
  for (int i = 0; i < 12; i++) {
    if (size <= BLOCK_SECTOR_SIZE * i && id->direct[i] != 0) {
      /* Shrink. */
      block_free(id->direct[i]);
      id->direct[i] = 0;
    } else if (size > BLOCK_SECTOR_SIZE * i && id->direct[i] == 0) {
      /* Grow. */
      id->direct[i] = block_allocate();
    }
  }

  /* Check if indirect pointers are needed. */
  if (id->indirect == 0 && size <= 12 * BLOCK_SECTOR_SIZE) {
    id->length = size;
    return true;
  }
  block_sector_t buffer[128];
  memset(buffer, 0, 512);
  if (id->indirect == 0) {
    /* Allocate indirect block. */
    id->indirect = block_allocate();
  } else {
    /* Read in indirect block. */
    block_read(id->indirect, buffer);
  }

  /* Handle indirect pointers. */
  for (int i = 0; i < 128; i++) {
    if (size <= (12 + i) * BLOCK_SECTOR_SIZE && buffer[i] != 0) {
      /* Shrink. */
      block_free(buffer[i]);
      buffer[i] = 0;
    } else if (size > (12 + i) * BLOCK_SECTOR_SIZE && buffer[i] == 0) {
      /* Grow. */
      buffer[i] = block_allocate();
    }
  }
  if (size <= 12 * BLOCK_SECTOR_SIZE) {
    /* We shrank the inode such that indirect pointers are not required. */
    block_free(id->indirect);
    id->indirect = 0;
  } else {
    /* Write the updates to the indirect block back to disk. */
    block_write(id->indirect, buffer);
  }
  id->length = size;

  return true;
}
```

5. How would you modify your solution to the previous question to handle sector allocation failures (i.e. disk runs out of space)?

Any time a new sector is allocated using `block_allocate`, you should add a check that resembles the following.

```
sector = allocate_block();
if (sector == 0) {
  inode_resize(id, id->length);
  return false;
}
```

2 Reliability

Availability

Availability is the probability that the system can accept and process requests. This is measured in "nines" of probability (e.g. 99.9% is said to be "3-nines of availability").

Durability

Durability is the ability of a system to recover data despite faults (i.e. fault tolerance). It's important to note that durability does not necessarily imply availability.

When making a file system more durable, there are multiple levels which we need to concern ourselves with. For small defects in the hard drive, Reed-Solomon error correcting codes can be used in each disk block. When using a buffer cache or any other delayed write mechanism, it's important to make sure dirty data gets written back to the disk. To combat unexpected failures or power outages, the computer can be built with a special, battery-backed RAM called non-volatile RAM (NVRAM) for dirty blocks in the buffer cache.

To make sure the data survives in the long term, it needs to be replicated to maximize the independence of failures. **Redundant Array of Inexpensive Disks (RAID)** is a system that spreads data redundantly across multiple disks in order to tolerate individual disk failures. RAID 1 will **mirror** a disk onto another "shadow" disk. Evidently, this is a very expensive solution as each write to a disk actually incurs two physical writes. RAID 5 will stripe data across n multiple disks to allow for a single disk failure. Each stripe unit consists of $n-1$ blocks and one **parity block**, which is created by XOR-ing the $n-1$ blocks. To recover from a disk failure, the system simply needs to XOR the remaining blocks.

Reliability

Reliability the ability of a system or component to perform its required functions under stated conditions for a specified period of time. This means that the system is not only up (i.e. availability), but also performing its jobs correctly. Reliability includes the ideas of availability, durability, and security.

One approach taken by FAT and FFS (in combination with `fsck`) is **careful ordering and recovery**. For instance, creating a file in FFS may be broken down into the following steps

1. Allocate data block.
2. Write data block.
3. Allocate inode.
4. Write inode block.

5. Update free map.
6. Update directory entry.
7. Update modify time for directory entry.

To recover from a crash, `fsck` might take the following steps.

1. Scan inode table.
2. If any unlinked files (not in any directory), delete or put in lost and found directory.
3. Compare free block bitmap against inode trees.
4. Scan directories for missing update/access times.

It's important to note that this is not a foolproof method. While there are a few ways that failures can happen in spite of this method, the file system will be recoverable most of the times.

The other approach is to use **copy on write (COW)**. Instead of updating data in-place, new versions are written to a new location on disk, and the appropriate mappings and references to these data are subsequently updated. Since the mappings and references are updated last, this allows for easy recovery if the system crashes sometime in the middle of updating data since the old data and mapping will still be in tact. Furthermore, data is being only added, not modified, so batch updates and parallel writes can help improve performance.

2.1 Concept Check

1. What benefit with regards to read bandwidth might you see from using RAID 1?

   Read bandwidth can be doubled since there are two copies of the same data.

2. What is the minimum number of disks to use RAID 5?

   3. A disk is recovered by XOR-ing at least two other disks. It's important to note that the RAID level is not an indication of how many disks are needed.

3. RAID 4 had a dedicated disk with all the parity blocks. On the other hand, RAID 5 distributes the parity blocks across all disks in a round robin fashion. Why is this approach beneficial in terms of write bandwidth?

   When updating data, the parity block always needs to be written to. If all the parity blocks are in one disk like in RAID 4, this becomes a bottleneck as multiple writes to disk ultimately contend on one disk. As a result, it's better to distribute the parity blocks evenly such that more writes can happen without contending for the same disk.

4. How can COW help with write speeds?

   COW can transform random I/O into sequential I/O. For instance, take the example of appending a block to a file. In a traditional in-place system like FFS, the free space bitmap, file's inode, file's indirect block, and file's data block all need to be updated. On the other hand, COW could just find sequential unused blocks in disk and write the new bitmap, inode, indirect block, and data block.

3 Distributed Systems

3.1 Concept Check

1. The vanilla implementation of 2PC logs all decisions. How could 2PC be optimized to reduce logging?

   Abort decisions can be ignored and not logged with the idea of presumed abort. On recovery, if there is nothing logged, then we can simply assume that the decision made was to abort.

2. 2PC exhibits blocking behavior where a worker can be stalled until the coordinator recovers. Why is this undesirable?

   If a worker is blocked on this coordinator, then it may be holding resources that other transactions may need.

3. An interpretation of the End to End Principle argues that functionality should only be placed in the network if certain conditions are met.

   Only If Sufficient
   Don't implement a function in the network unless it can be completely implemented at this level.

   Only If Necessary
   Don't implement anything in the network that can be implemented correctly by the hosts.

   Only If Useful
   If hosts can implement functionality correctly, implement it in the network only as a performance enhancement.

   Consider the example of the reliable packet transfer: making all efforts to ensure that a packet sent is not lost or corrupted and is indeed received by the other end. Using each of the three criteria, argue if reliability should be implemented in the network.

   Only If Sufficient
   No. It is not sufficient to implement reliability in the network. The argument here is that a network element can misbehave (i.e. forwards a packet and then forget about it, thus not making sure if the packet was received on the other side). Thus the end hosts still need to implement reliability, so it is not sufficient to just have it in the network.

   Only If Necessary
   No. Reliability can be implemented fully in the end hosts, so it is not necessary to have to implement it in the network.

   Only If Useful
   Sometimes. Under circumstances like extremely lossy links, it may be beneficial to implement it in the network.

   Lets say a packet crosses 5 links and each link has a 50% chance of losing the packet. Each link takes 1 ms to cross and there is an magic oracle tells the sender the packet was lost. The probability that a packet will successfully cross all 5 links in one go is $(1/2)^5 = 3.125\%$. This means the end hosts need to try 32 times before it expects to see the packet make it through, taking up to # of tries $\times$ max # of links per try $= 32 \times 5 = 160$ ms.

   Likewise at each hop, if the router itself is responsible for making sure the packet made it to the next router, each router would know if the packet was dropped on the link to the next router. Thus each router only has to send the packet until it reaches the next router, which will be twice on average. So to send this packet, it will take on average # of tries per link # number of links = $2 \times 5 = 10$ ms. This is a huge boost in performance, which makes it useful to implement reliability in the network under some cases.

4. Why would you ever want to use UDP over TCP?

UDP is used in applications that prioritize speed and low overhead, and either don't care about being "lossy" or implements their own protocol for reliability. For example, in streaming audio or video, it doesn't matter if some packets are lost or corrupted, as long as most of the packets are sent, the user will still be able to hear/see the data well enough. Another example would be any application where real time data is very important (e.g. real time news, weather, stock price tracking, etc), and we don't have time for anything beyond best effort delivery.

3.2 Two Phase Commit

Consider a system with one coordinator (C) and three workers (W_1, W_2, W_3). The following latencies are given for each worker.

Worker	Send/Receive (each direction)	Log
W_1	400 ms	10 ms
W_2	300 ms	20 ms
W_3	200 ms	30 ms

You may assume all other latencies not given are negligible. C has a timeout of 3 s, log latency of 5 ms, and can communicate with all workers in parallel.

1. What is the minimum amount of time needed for 2PC to complete successfully?

For each phase, the worker needs to receive the message, log a result, and send back a message. Since each worker can operate in parallel, only the longest latency matters. Using the latencies given, W_1 has the longest round trip time latency with $400 + 10 + 400 = 810$ ms. Therefore, the two phases will require 1620 ms. However, we also need to add in the time that it takes for the coordinator to log the global decision, so the minimum amount of time is 1625 ms. The reason this is the minimum amount of time is because this assumes no failures.

2. Consider that all three workers vote to commit during the preparation phase. The coordinator broadcasts a commit decision to all the workers. However, W_2 crashes and does not recover until immediately after the coordinator's timeout phase. Does this transaction commit or abort? What is the latency of this transaction, assuming no further failures?

The transaction still commits since a commit decision was made by the coordinator. The preparation phase will take 810 ms as calculated before, and the commit decision needs to be logged which takes 5 ms. The first try of the commit phase will take 3000 ms (i.e. the timeout). The second try will only take $300 + 20 + 300 = 620$ ms because W_2 is the only worker that needs to commit; all other workers succeeded on the first try. In total, the latency of this transaction is $810 + 5 + 3000 + 620 = 4435$ ms.