Lecture 25 — More Concurrency in File Systems

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

November 14, 2022

Copy-Modify-Merge

Thus far when we talk about modification of shared data we follow a model that could be described as “Lock-Modify-Unlock”.

But you’ve also used `git` or some other version control system (`svn`) that uses a different model: Copy-Modify-Merge.

Example: you and your lab partner work on something together…

Transaction

A transaction is a grouping of operations that belong together and should be treated as an indivisible unit.

Bad things can happen when an intermediate state of a multiple-step operation becomes inadvertently visible.

Most of the examples looked at things like `x++;`

In the copy-modify-merge scenario, people can make their changes separately and then we try to put them all together.

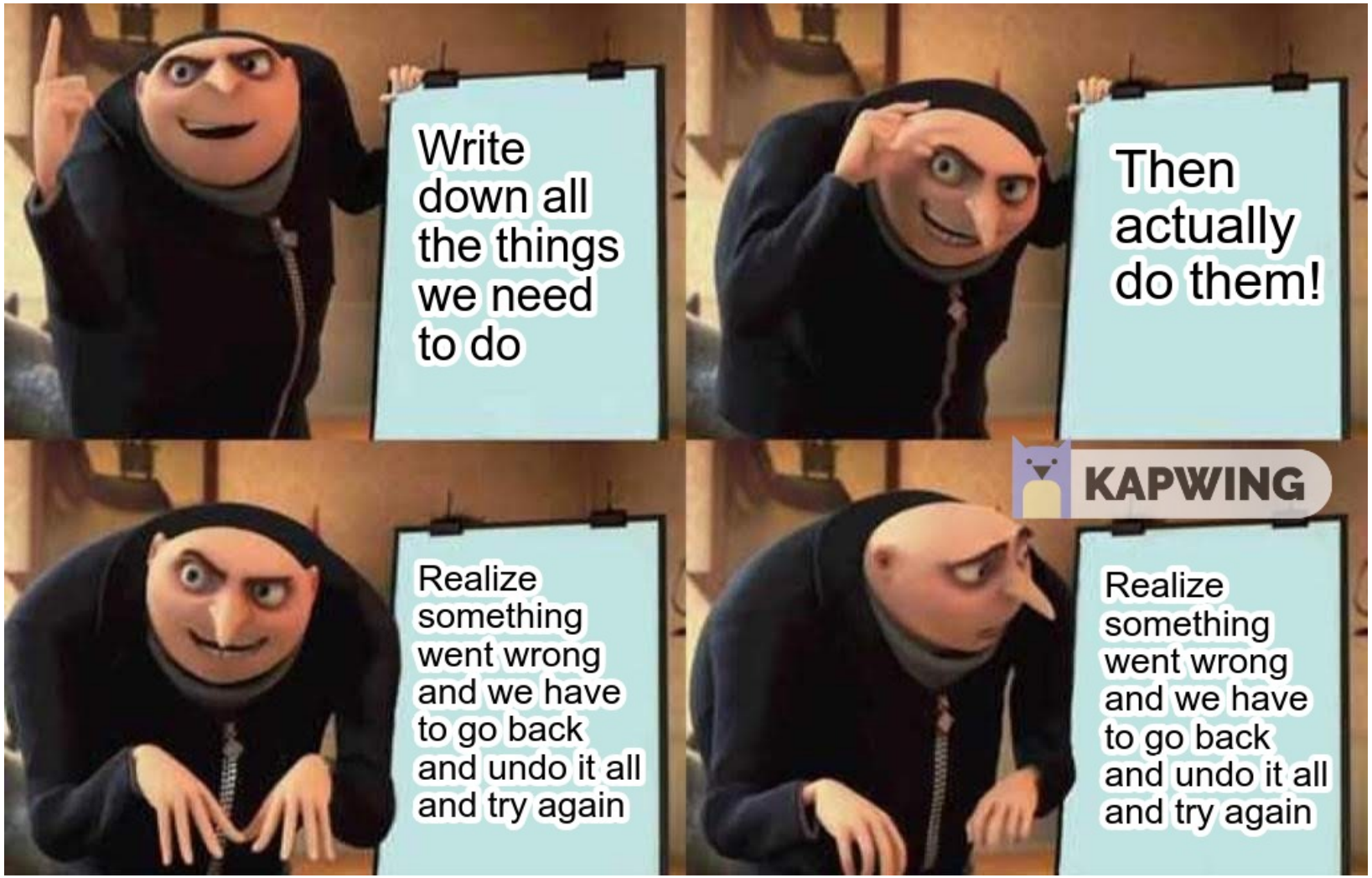
Plan your Moves

A transaction has a begin transaction statement, then the operations to take place in the transaction, and finally an end transaction statement.

Execution looks something like writing down the transaction into a log, doing the operations in the transaction.

When the last one is complete, if all went well, marking the transaction as successful.

I have a cunning plan...



Copy-Modify-Merge

In the case of version control, if there are merge conflicts then we are notified that the merge cannot take place until conflicts are resolved.

But in file systems, the last write wins.

You have *N* unread notifications…

I’m an inbox-zero kind of person so this picture hurts:

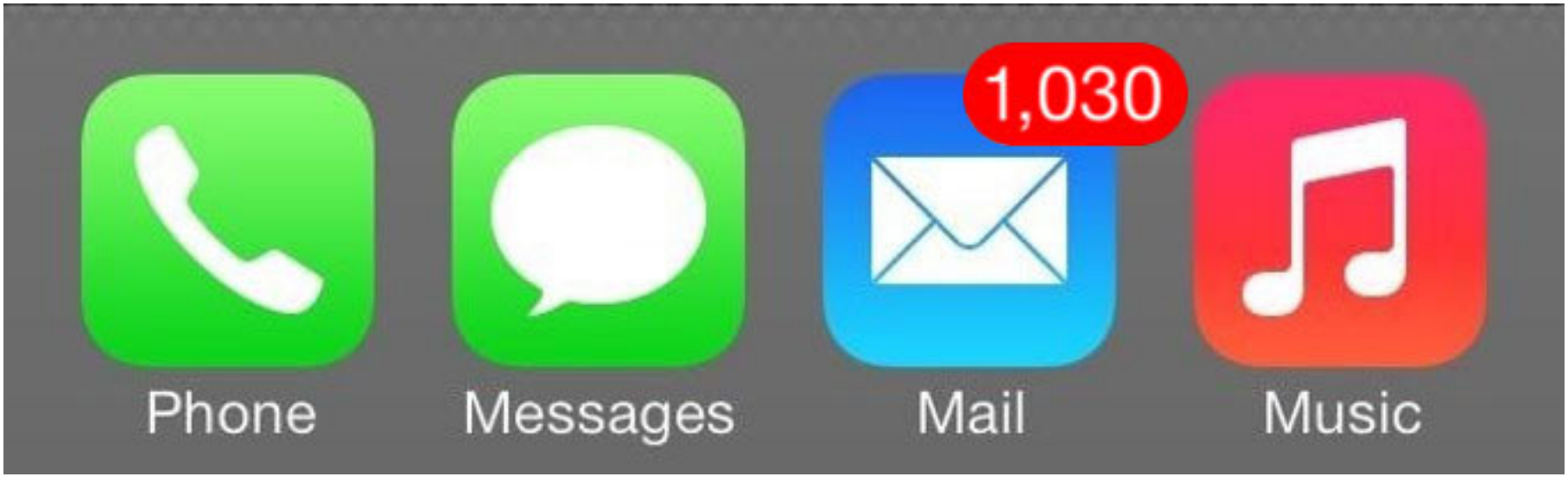


Image credit: Dhvanesh Adhiya

Notify Me

Another way that we can use the file system (in Linux only!) for synchronization or concurrency control is through the use of `inotify`.

Using this API, you can register your program as being interested in the events.

You say you want to watch a file or directory, and when an event occurs, then your program is informed

OVERWATCH!

The steps are:

1. Use an initialization function to create the management structure (and get a file descriptor back to refer to it).
2. Then you tell the kernel what files you are interested in by adding them to the structure (you can also remove them).
3. To collect an event, use `read` on the file descriptor. Each call returns one or more event structures.
4. If you're done, close the file descriptor representing the management structure, which conveniently cleans everything up for you.

The mechanism is not recursive.

Notification API: Initialization

The API calls:

```
int inotify_init( ); /* Returns file descriptor referring to the struct */
```

Initialization doesn't require any arguments, so that's quite convenient.

Notification API: Add

The API calls:

```
int inotify_add_watch( int fd, const char* pathname, uint32_t mask );
```

Adding an item to the watch takes as an argument the inotify structure to add it to, the name of the file to add, and a mask.

You must have at least read permission on the file to be able to watch it.

The mask is how we specify details about the events that we are interested in.

We have to save the return value of the add function if we want to use the remove function.

Notification API: Remove

The API calls:

```
int inotify_rm_watch( int fd, uint32_t wd );
```

Remove takes as an argument what was returned from add.

When we're completely done, just call close on the file descriptor representing the inotify.

TV has so many channels...

There are about 23 different events that you can watch for using the bit mask.

Bit Value	Description
`IN_ACCESS`	File accessed (read/execute)
`IN_ATTRIB`	Metadata changed, such as permissions
`IN_CLOSE_WRITE`	File opened for writing was closed
`IN_CLOSE_NOWRITE`	File not opened for writing was closed
`IN_CREATE`	File or directory created in watched directory
`IN_DELETE`	File or directory deleted from watched directory
`IN_DELETE_SELF`	Watched file or directory deleted
`IN_MODIFY`	File modified (write, for example)
`IN_OPEN`	File opened
`IN_ALL_EVENTS`	Watch for all of the above (and a few more)

Well, Now What?

Imagine we have set up some files that we would like to watch.

When ready for such an event, use `read` on the file descriptor for the inotify.

If an event occurred, you get back a structure `inotify_event`.

```
struct inotify_event {
  int      wd;       /* Watch descriptor */
  uint32_t mask;     /* Mask describing event */
  uint32_t cookie;   /* Unique cookie associating related events (for rename(2)) */
  uint32_t len;      /* Size of name field */
  char     name[];   /* Optional null-terminated name */
};
```

Weird: the size of an inotify event is thus the structure size plus the length of the array, i.e.: `sizeof( struct inotify_event) + len`.

Das ist Nicht Gut

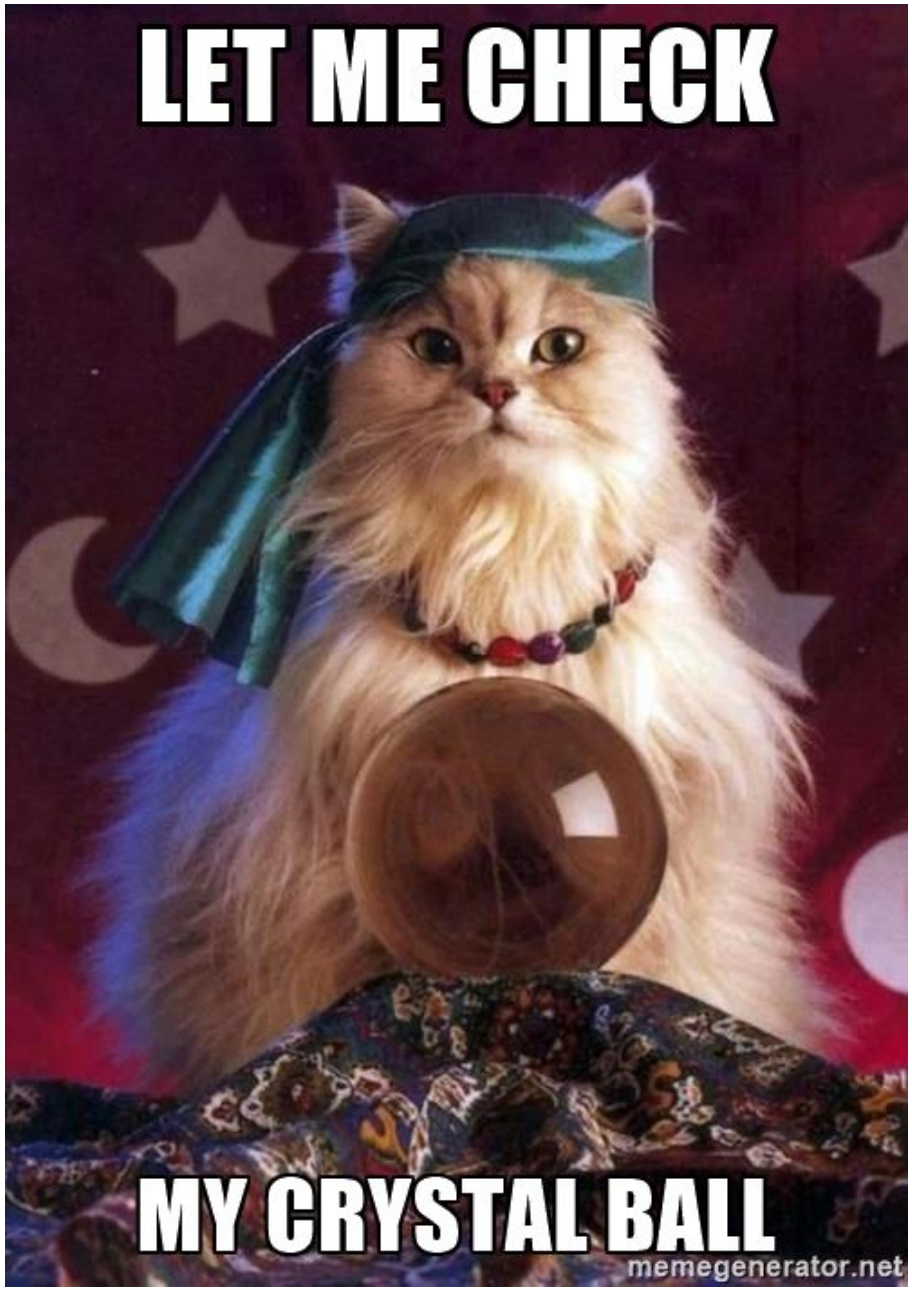
Usually when we do a read we need to know how many bytes we'd like to read.

If we're reading a struct, we know the size of the struct, but now it depends on the length of the data you get back.

Your standard clairvoyance problem.

One approach is to just make the buffer really big…

Consult an Expert?



If you don’t know, ask!

`ioctl` can tell you what you want to know!

`ioctl( fd, FIONREAD, &numbytes )` updates `numbytes` with the number of bytes currently available to read from the inotify instance.

If multiple events occurred you can get multiple structures back (if your buffer is big enough).

Example of using inotify

```
const char filename[] = "file.lock";

int main( int argc, char** argv ) {
  int lockFD;
  bool our_turn = false;

  while( !our_turn ) {
    lockFD = open( filename, O_CREAT | O_EXCL );
    if ( lockFD == -1 ) {
      printf( "The lock file exists and process %ld will wait its turn...\n",
       getpid() );
      int notifyFD = inotify_init( );
      uint32_t watched = inotify_add_watch( notifyFD, filename, IN_DELETE_SELF );

       /* Read the file descriptor for the notify -- we get blocked here
          until there's an event that we want */
      int buffer_size = sizeof( struct inotify_event ) + strlen( filename ) + 1;
      char* event_buffer = malloc( buffer_size );
      printf("Setup complete, waiting for event...\n");
      read( notifyFD, event_buffer, buffer_size );
```

Example of using inotify

```
      struct inotify_event* event = (struct inotify_event*) event_buffer;
      /* Here we can look and see what arrived and decide what to do.
         In this example, we're only watching one file and one type
         of event, so we don't need to make any decisions now */

      printf("Event occurred!\n");

      free( event_buffer );
      inotify_rm_watch( lockFD, watched );
      close( notifyFD );
    } else {
      char* pid = malloc( 32 );
      memset( pid, 0, 32 );
      int bytes_of_pid = sprintf( pid, "%ld", getpid() );

      write( lockFD, pid, bytes_of_pid );
      free ( pid );
      close( lockFD );
      our_turn = true;
    }
  }

  printf("Process %ld is in the area protected by file lock.\n", getpid());
  remove( filename );
  return 0;
}
```

Consistency Checking and Journalling

Unfortunately, an error, crash, or power failure or something similar may result in a loss of data or inconsistent data in the file system.

The directory structures, pointers, inodes, et cetera are all data structures and if they become corrupted it may lead to serious problems.

We may need to check for consistency:

Consistency Checking

We could check for inconsistent data periodically (e.g., on system boot up) and many operating systems do so.

This is, of course, an operation that will consume a very large amount of time while the whole disk is scanned.

UNIX: `fsck`. Windows: `chkdsk`/`scandisk`.

These tools will look for inconsistent states (e.g., a file that claims to be 12 blocks but the linked list contains only 5) and will attempt to repair it.

Its level of success depends on the nature of the problem and the implementation of the file system.

Preventing the Problem

Obviously we would like to prevent the problem, if we can.

All modern OS file systems use transactions to ensure consistency.

We'll talk about ZFS, APFS, and NTFS…

ZFS uses the idea of transactions, making sure that the state is always consistent on disk.

Much like the copy-modify-merge model, data is copied, then changed, then rewritten.

Blocks are never overwritten with new data.

Instead, a transaction writes all data and metadata to new blocks.

Only when the transaction is complete, any references to the old blocks are replaced with the location of the new blocks.

Then the old pointers and blocks can be cleaned up (reused or disposed of).

Interesting weakness: what if the disk is totally full?!

APFS

Like some version control systems, APFS brings the ability to take snapshots of the file system.

Freeze the state of the file system and from there any additional changes are “diffs” against that base state, meaning only new things take up space.

This is potentially quite helpful for taking backups!

… You do take backups, right?

Time Machine, Whyyyyy

The performance of your system can be degraded while traditional backups are being taken.

Time-consuming: computing a diff between the last backup copy and the current.

APFS approach is faster, but also a way to avoid corruption.

You can replay changes as needed to get the file back to *a* consistent state.

More Copies, More Good?

The APFS does potentially harm the most common “backup” system of non-technical users: take a copy of the file and put it in a different folder.

APFS will not actually duplicate the data on the same volume.

It sounds like they’re doing you a favour if you think of this as just reducing wasted space.

But from the perspective of redundancy: if that part of the disk is damaged then all copies are lost.

Marketing-Speak, Engage

Somewhat like ZFS, the APFS approach to avoiding inconsistent data amidst a crash is something like copy-on-write.

In typical Apple fashion they were pretty vague about what this means…

The APFS lead developer Dominic Giampaolo just says it's a "novel copy-on-write metadata scheme" but also somehow not exactly the same as ZFS's single-atomic-update approach.

Example: NTFS (Windows File System)

NTFS uses several different storage levels:

1. **Sector**
2. **Cluster**
3. **Volume**

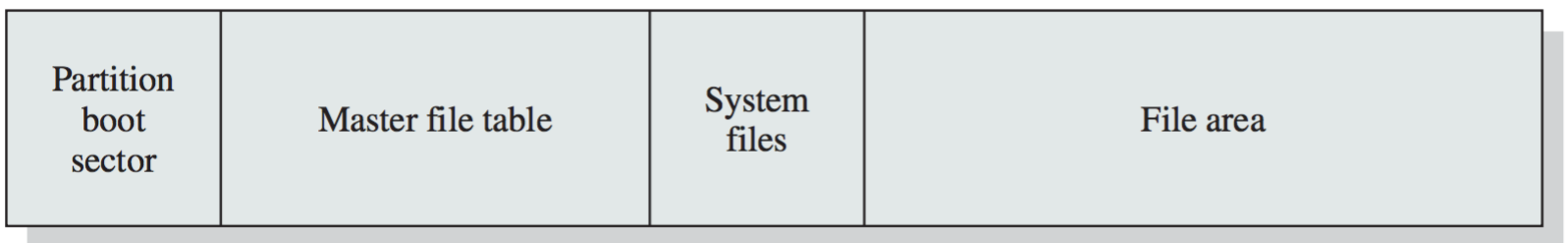
The cluster is the fundamental unit of allocation of NTFS.

This allows the file system to be independent of the size of physical sectors on the disk.

A volume contains file system information, a collection of files, and free space.

The logical volume may be some of a physical disk, all of one, or spread across multiple physical disks.

NTFS Volume Layout



NTFS System Files

The Master File Table (MFT) contains information about all the files and folders.

A block is allocated to system files that contain important system information:

1. **MFT2**
2. **Log File**
3. **Cluster Bitmap**
4. **Attribute Definition Table**

NTFS Journalling

NTFS uses journalling to ensure that the file system will be in a consistent state at all times, even after a crash or restart.

There is a service responsible for maintaining a log file that will be used to recover in the event that things go wrong.

Note that the goal of recovery is to make sure the system-maintained metadata is in a consistent state; user data can still get lost.

This was a Microsoft design decision.

Sorry, You Lose

A particular write may not have taken place because of a crash, resulting in some data loss for you, the user.

But at least the system will always remain in a consistent state.

As a side benefit, we can sometimes re-order the writes to get better performance.

NTFS Journalling Implementation

The actual implementation of journalling:

1. Record the change(s) in the log file in the cache.
2. Modify the volume in the cache.
3. The cache manager flushes the log file to disk.
4. Only after the log file is flushed to disk, the cache manager flushes the volume changes.

Just Deal with This

What's really interesting about this is that the changes are carried out in the background, that is to say, asynchronously.

A program can say that it wants to write some data, and not have to wait for the data to be written before going on to the next thing.

How interesting!

Can we get that behaviour in our (regular) program? Yes we can...