DAW Design & Implementation #10

Disks, Filesystems and Disk I/O

Storing Data

- A heirarchy of storage types (by size & speed):
  - Registers
  - Cache memory
  - Main memory
  - Disk storage
- Recorded audio is generally too large for anything except disk storage

Disk Storage

- Magnetic medium, spread over a circular surface (contrast to the linear surface of a tape)
- Possible to change the magnetization
- Spin the surface
- Put an electromagnetic “head” near the surface
- “write” == change the magnetically encoded value on the disk
- “read” == pick up the variations
- Improvements: coat both surfaces, make it spin faster, add more surfaces (platters) in parallel (each with their own “head”)

Critical Properties of Disk Storage

- *Rotational/Seek Latency*: how long does it take before the head can read the required data?
- *Streaming bandwidth*: assuming no other limitations, how many bytes/second can the disk read/write from/to the bus?

Data Layout

- The disk surface is divided into concentric circles ("tracks" or "cylinders")
- Each track is divided into a set of sectors
- A sector is the smallest unit of storage that can be read/written; typically 512 bytes
- Note: most modern disks turn at constant angular velocity. Thus data is distributed more sparsely on the outer tracks than the inner tracks

Rotational/Seek Latency

- Time to access a particular byte of information
- Disk controller must:
- (a) identify the relevant head to read the data
- (b) position the head over the correct track ("seek") (5-10ms)
- (c) wait for correct rotational position of platter
- (d) read the data
- Typical consumer disks rotate at 5400-7200 RPM
- Fast disks rotate at 10k-15k RPM
- Maximal rotational latency = 1 rotation
- 5400RPM = 11ms, 10k RPM = 7ms

Streaming Data Rate

- Absolute limit is determined by the type of disk/bus connection
- IDE, EIDE, SCSI (1,2,3), SATA etc.
- No real disk approaches this limit
- Current electronics: reads @ 25-40Mb/sec
- Always an optimistic value; assumes everything else about the data access is “perfect”

Making Disks Faster

- Improve head speed
- Increase angular velocity
- Make platters smaller (seek time is reduced)
- Increase bits-per-inch (sector density) (seek time & number of seeks is reduced)
- One head per track (no seeks; very expensive)
- Add a data cache (duh!)
- And now ... solid state disks

What we need for audio

- Typical “heavy” load: simultaneous playback and recording of 24 (mono) tracks
- 4 bytes per sample
- 48000 samples per second
- About 4.3MB/sec streaming writes, and another 4.3MB/sec streaming reads
- 10 years ago: hard to do without expensive SCSI disks (without RAID)
- Now: almost any disk can do this (in theory)
- So now, double the sample rate ...
- Halve the bytes/sample ...
- etc.

What's a Filesystem

- The disk provides a medium for storing data
- It doesn't provide a system for storing data
- A filesystem is a system for allocating the sectors on a disk for use in storing data.
- The filesystem tells us where to store newly written data, and where to read existing data
- Essentially a function that maps from one tuple (*filename,offset-in-file*) to another (*sector-on-disk,offset-in-sector*)

Filesystem Basics

- Filesystems allocate storage space in "blocks" (aka "clusters"), not sectors
- 1 block == a series of contiguous sectors
- Typical block sizes: 1kB – 16kB
- Filesystems need an allocation table to know which blocks have been used
- Filesystems need a way to refer to a particular file (a "filename")
- Filenames may be organized in a heirarchy

`/Users/paul/source/ardour/3.0/libs/ardour/audioengine.cc`

- Some filesystems may use a flat namespace
- Most filesystems have file *metadata* stored somewhere (e.g. last modified, owner)

Should we use a custom filesystem?

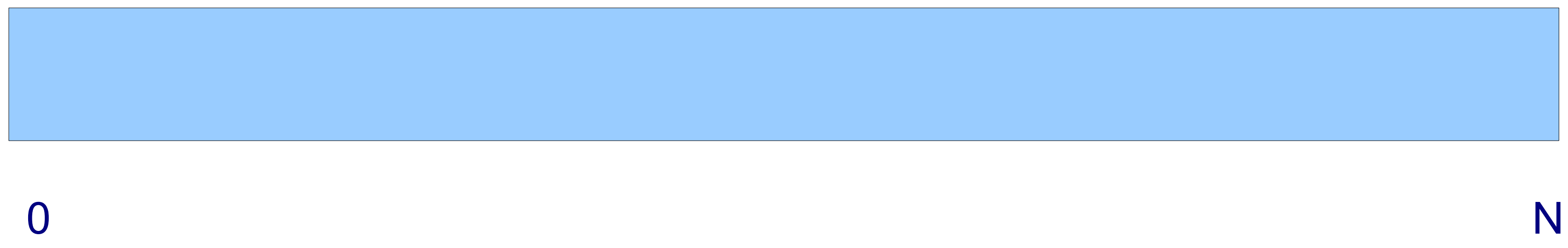
- 15-20 years ago, disks were not fast enough to support streaming data rates required for multitrack audio with normal filesystems
- Several efforts were made to develop custom filesystems for audio/video work
- It is still true that these custom designs would be faster than standard filesystems
- Improvements in disk technology have made them unnecessary
- RAID is always an option

Audio & the Filesystem

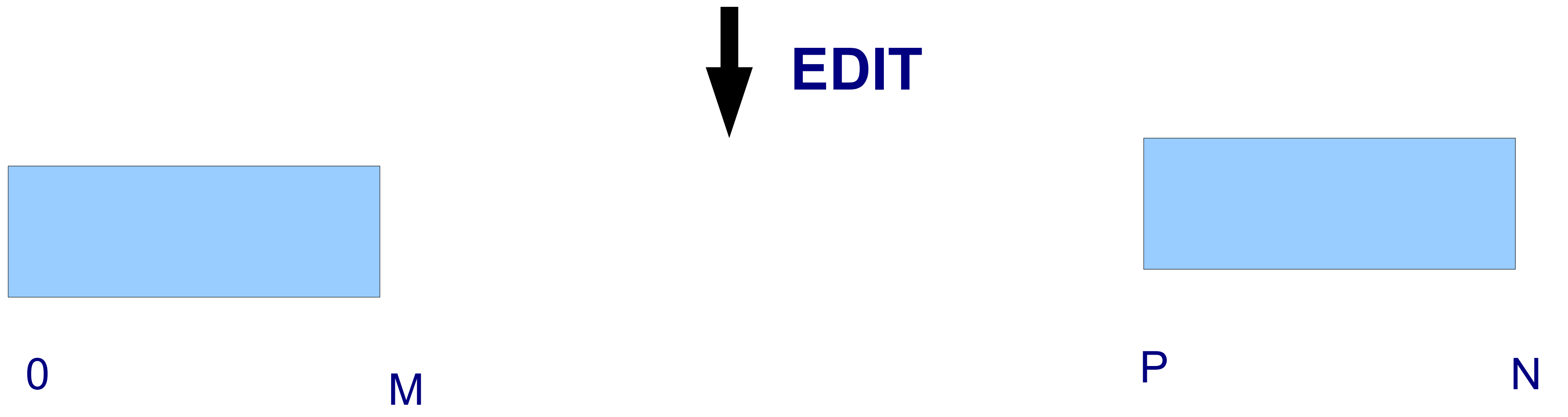
- We can't read on demand: too slow
- Too much seeking on disk will always be a problem
- Most files are “big” (by filesystem standards)
- Most access is *sequential*: we don't read a byte here and another byte somewhere else
- Most files don't change much after being written

Editing & Audio File Structure

- Special considerations for a non-linear, non-destructive editor (like a DAW; not like a soundfile editor)
- Consider a single file representing a single sound
- First, mono
- Lets edit it



Playlist = “file Foo, start at 0, length N”



Playlist = “file Foo, start at 0, length M,
file Foo, start P, length N-P”

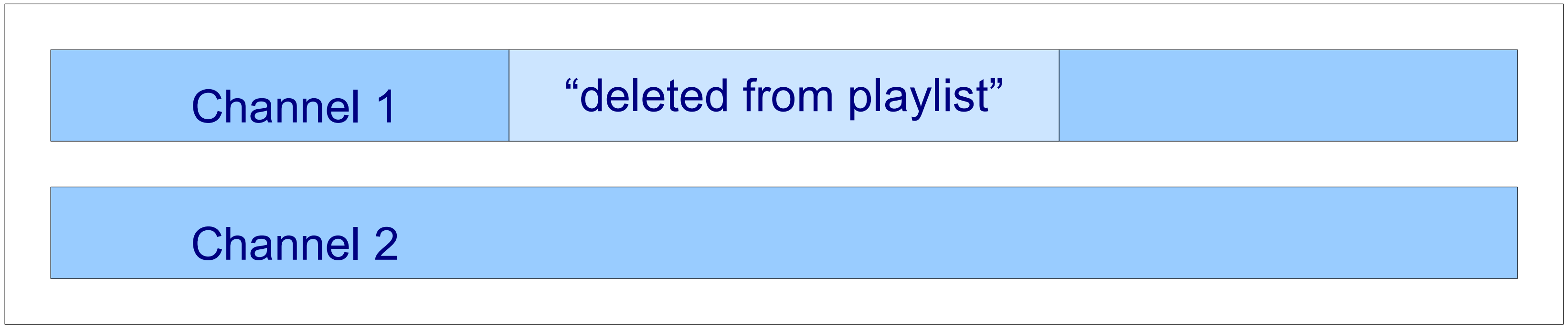
```
Playlist = “file Foo, start at A, length L1,
            file Foo, start at B, length L2,
            ....
            file Foo, start at Z, length Ln”
```

- Every region might represent another seek
- What happens if the file is not mono?

Editing & Non-Mono Files

- For some audio sources, non-mono is fine because we will always edit the channels simultaneously
- E.g. recording an electronic keyboard, or a stereo mic pair
- But: suppose we record a drumkit with 16 microphones, do we store the data in a 16 channel file, or 16 mono files?

1 stereo file



- Reading data for 1 channel implies reading the other
- Every time we read channel 2 in the “deleted” range, we also read channel 1
- Now imagine this with 16 channels ...
- Lots of pointless disk I/O

Sound file editor issues

- The situation can be even worse for a destructive editor
- If the user edits just 1 channel of the full set, saving the modified file requires rewriting the all the data
- Contrast with the situation with mono files: only the edited channels need to be written back to disk

Moral

- Use mono soundfiles where possible
- But also: more edits generally means more seeks, and seeks are slow.

Allocation Issues

- Filesystems have different strategies for allocating blocks to files
- Simplest is to just allocate “next free block” to new data writing requests
- Leads to massive “fragmentation”

Fragmentation

- Non-fragmented file allocation: file is stored in a series of contiguous sectors
- Create 2 files. Delete the first. Now create a larger file.
- The 3rd file ends up being split across the space left behind by the deleted file and the rest of “free space” - it is “*fragmented*”
- Reading this file will involve seeking - slow

More Complex Filesystems

- “first free block” is a very naive allocation strategy
- If we knew a file would need 10 blocks, we could reserve it ahead of time ... then never get any fragmentation
- In general the filesystem can't know this
- But it can guess!

A strategy for writing audio files

- Assumptions: the file(s) are big, several may be written at once
- There is an optimum write-chunk size for every filesystem
- Interleave writing of each file
- Read with the same chunk size
- Not guaranteed to provide optimal allocation, but likely to

Other files too!

- Peakfiles are "summaries" of max/min sample values at 1 or more resolutions
- Used for drawing waveforms
- Typical resolutions: 1 pair of values per 256/1024/4096/16k samples
- We write these in small chunks (as the values are computed)
- Helps to pre-allocate space for the file to minimize fragmentation
- `ftruncate(1)` is a POSIX system call that can be used for this
- Ardour preallocates 128kB blocks for peak files

Disk I/O Thread Design

- Cannot ever, ever, ever do disk I/O from the audio (“realtime”) thread – too slow
- SSD? Still not a good idea
- Another thread or multiple threads required
- Simplest design: 1 thread to read & write data
- Wake it up from the audio thread when necessary
- Playback buffers getting empty
- Recording buffers getting full

Disk Butler Design in Ardour

- Assumptions: disk I/O is s-l-o-w
- Goals: no starvation of tracks, sensible disk allocation results
- Use a round-robin approach
- Write `min(chunksize,requested)` bytes for each track
- Loop until no tracks need any more writes
- Repeat for reads
- Go back to sleep

```
while (butler_should_run) {
      wait_for_wakeup ();

      while (tracks_need_writing()) {
         foreach track {
            track->write (min (chunksize,
                          track->to_be_written()));
      }

      while (tracks_need_reading ()) {
         foreach track {
            track->read (min (chunksize,
                         track->to_be_read())));
       }
}
```

Solid State Disks

- Instead of spinning magnetic media, non-volatile memory circuits
- Eliminate seeks and rotational latency
- Access latency drops to microseconds
- But ...
- You must erase a block before it can be (re)written, which can be slow
- Too many rewrites in the same place will “wear out” the memory, so “wear levelling” algorithms are required
- Most current filesystems are optimized around the idea that seeks are slow, which is no longer true

SSD 2

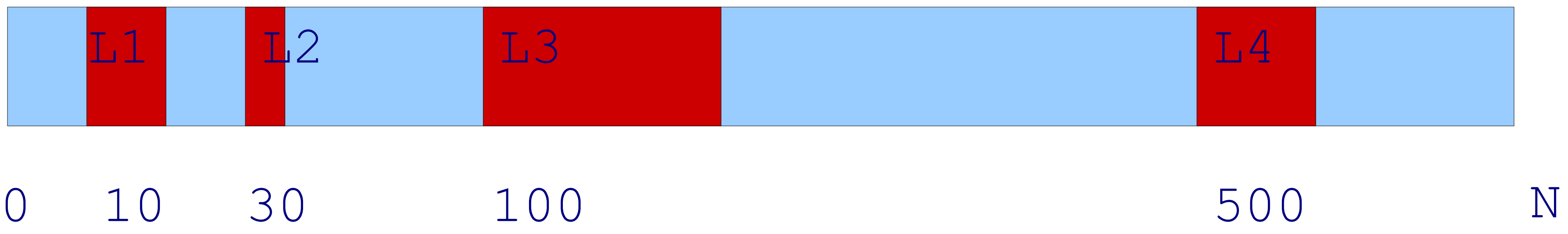
- Thus, SSD's really require special filesystems
- Another reason why not using a custom filesystem for audio/video is a good idea
- Basically, SSD & audio is an unknown right now

A Filesystem Trick

- Destructive recording (“tape mode”) is a variation on normal DAW operation
- Normal: each record pass writes data to a new file (or least, a new part of an existing file)
- Destructive: each record pass overwrites any existing data corresponding to the same location on the timeline

Destructive Recording 2

- What happens when you read data from a part of the file that hasn't been recorded to?
- Depends on the filesystem!
- Most (all?) Unix filesystems have the following behaviour:
- After setting the length of a file with `ftruncate(1)`, any reads to areas of the file not yet written will return bytes set to zero
- 4 bytes set to zero == a 4 byte sample set to a value of zero
- Result: no magic games to make destructive recording work, and no setup of the files to contain “silence”



```
ftruncate (f, N);
...
pwrite (f,buf, L1, 10);
...
pwrite (f, buf, L2, 30);
...
pwrite (f, buf, L3, 100);
...
pwrite (f, buf, L4, 500);
...
pread (f, buf, L5, 200); <<== ?????
```