WHY FILE SYSTEMS <u>AREN'T</u> SLOW

Professor Ken Birman
CS4414 Lecture 21

IDEA MAP FOR TODAY

We have seen that file systems come in many shapes, sizes, and run in many places!

Yet what file systems are doing is inherently high-latency: Fetching bytes from some random place on a storage unit that may be a rotating physical platter accessed by moving read heads.

Caching and the "working set"

Prefetching

Secondary indices

PERFORMANCE OF A FILE SYSTEM

Application must open the file

- Linux will need to access the directory
- … scan it to find the name and inode number
- … load the inode into memory
- … check access permissions

So, opening a file could involve 2 or more disk reads (more if the directory is large).

THE FUNDAMENTAL ISSUE?

Data transfers are pretty fast

- They use a feature called direct memory access (DMA)
- DMA can match memory speeds, if the disk can send/receive data that rapidly (some devices can, many can't).

But the *delays* for accessing storage are hard to eliminate. They come from the hardware required to talk to the disk (or network)

THE FUNDAMENTAL ISSUE?

To read data we pay (delay to talk to the disk) + (transfer time)

Obviously, the code itself needs to be as efficient as possible, but we won't get rid of these hardware costs

Can we find ways to "hide" this delay from the application?

READING THE FILE

The application might modify the seek pointer. This is free.

Then does a read of some number of bytes

- File system must look up the block "at" this offset into the file
- Could involve scanning several levels of disk-block indices if file is big
- Once it has the block number, file system reads that block
- Once the block is in the buffer pool, kernel copies the data to user space, tells the process how many bytes were read (in case of EOF)

… could easily require 3 or 4 disk I/O operations to do all of this.

EXAMPLE OF WORST-CASE PERFORMANCE

Many people use print statements to debug programs, and redirect into a file if there will be a lot of lines of output.

Each line is normally “written” as it is produced.

Each write is just like a read: copy to the kernel plus (perhaps) many disk writes to update the inode, block list and the block, plus (perhaps) to remove a block from the free list.

WHAT HAPPENS?

This works, but will often cause “slow motion” behavior

If you check, you’ll find that all of these I/O requests are causing a huge bottleneck.

This is why we use “buffered” I/O solutions. They save up 4K or 8K of bytes in a buffer, they write it all at once.

COSTS OF A KERNEL READ OR WRITE

The system call itself requires a trap, must save user context and switch into kernel context.

The kernel may have been busy; if so, your request could easily block waiting for locks or for a thread to service it. We need to fetch the actual data, which might not be in the buffer pool.

Memcpy from kernel to user, or user to kernel.

COSTS OF A KERNEL READ OR WRITE

The syste r context and swit

All of this could easily require several milliseconds. Although a millisecond is a small amount of time, you may be limited to several hundred such requests per second.

The kern could easily block wa We need to fetch the fer pool.

Memcpy from kernel to user, or user to kernel.

HOW IOSTREAMS DECIDES WHEN TO ISSUE A WRITE REQUEST

The std::endl object has two roles

- It has a "value", which is '\n' (the ASCII newline character)
- It also has a "side-effect", which is to cause the line to be written

Effect is that every line will trigger a write if you use std::endl.

In contrast, with '\n' you still get line by line printouts, but the data will be buffered until the iostream buffer is filled.

BUFFERED I/O IS *MUCH* FASTER, BUT…

Suppose your program happens to crash.

What would happen to the last 1.5K of print messages?

… they could have been in the I/O stream buffer, in memory, and would not be printed! The file of debug output will be missing hundreds of lines of output!

REMINDER: ZOOKEEPER

We mentioned that Zookeeper itself is fault-tolerant, but has “issues”.

It uses *checkpoints:* Periodically, it saves its state to disk.

Zookeeper can have amnesia when it recovers from a shutdown.

- The most recent updates can be lost.
- Fundamental issue: If Zookeeper checkpoints every update, it runs too slowly, so they only do it every 5 seconds!

SO… WHY IS NORMAL FILE I/O SO FAST? WE SAW THIS IN LECTURE 2 (A QUICK REVIEW)

Several factors come into play all at once

1. Linux retains blocks from the disk in the file system buffer pool and can respond to reads immediately if it gets a cache hit.
2. Linux uses a "write-through" policy: Writes update the block in the buffer pool. The program continues… the actual disk I/O might be delayed for a while.
3. Linux anticipates likely future reads and prefetches data
4. Many modern disks have caches of their own. For these, a disk read can be satisfied instantly if the block is in the disk cache

CACHING: THE CORE CHALLENGE IS TO HAVE THE WORKING SET IN THE CACHE

We use this term in several situations.

Linux sometimes does paging to reduce the pressure on memory. A process has the working set in memory if all the instructions and data it actually touches when running are resident.

Similarly, the disk buffer pool holds the working set if it already has a copy of the files the application is likely to access.

WHY WOULD THIS EVER HAPPEN?

Modern workloads often involve running some program again and again with many inputs unchanged.

For example, when training a vision system (a type of neural network called a CNN), we might reread the same input photos again and again while adjusting the CNN model parameters.

COLD START (FIRST READ) VERSUS WARM

The first time the files are accessed, Linux needs to read them.

But then they linger in cache, so the second and subsequent reads get cache hits on the buffer pool.

This is called a “warm cache” situation.

CACHE EVICTION ALGORITHMS

When a new block is loaded into a full cache, decides which to evict.

One option is to use Least Recently Used (LRU) caching. Evict the block that has not been touched in the longest amount of time.

Implementation: Keep a queue. As each block is touched, move it to the head of the queue. The LRU block is at the tail of the queue.

CACHE EVICTION ALGORITHMS

Issue with LRU: If we run a training system, as in the example, it may delay a long time before revisiting files.

Those blocks will often be evicted just before we finally access them again.

Causes a form of "thrashing": wasteful pattern of evicting blocks, then reloading them.

LEAST FREQUENTLY USED (LFU)

With this algorithm, we track how often each block is accessed.

Retain a block if it is accessed more frequently… evict a block that has not been accessed as often.

Issue: If the cache is full of heavily accessed files, but now we stop accessing them, they might never be evicted!

LFU WITH “AGING”

This is like LFU, but as time passes, older references count less.

Implemented by periodically multiplying the count by, e.g., 9/10

Effect is a form of LFU focused on “recent” accesses.

MULTILEVEL APPROACH

Similar to one of the thread scheduling policies we saw early in the course.

Partition the cache. Block migrates from partition to partition based on an access time or access frequency rule.

Now we can use a different eviction policy in each partition.

SECOND CHANCE CACHING

With multilevel approaches, one issue is that the partition sizes might not be ideal.

Suppose that a process would get 100% hits if 2/3rds of the cache is devoted to frequently accessed blocks. But we limit the process to 1/3 of the cache. We get a high miss rate.

A second-chance cache addresses this.

WHEN A BLOCK IS EVICTED, IT MOVES TO THE SECOND-CHANCE CACHE

We also write it to disk at this point, if the block is dirty.

The idea is that if we weren't really using the full size of one of the partitions, a cache-miss on the heavy-hitter partition might be followed by a cache-hit on the second-chance cache.

MULTIPROCESS CONSIDERATIONS

LRU and LFU are usually expressed in terms of a fixed-size cache.

But we might prefer to allocate different amounts of cache space to different processes!

How would we estimate how much each requires?

WORKING SET TRACKING

We use these methods when the amount of memory for each process might be varied – some will get more, some less.

Goal: Estimate the “working set” each process is accessing.

Definition: The working set is the set of pages or files or blocks being accessed during some window of time.

INSIGHT

Most applications have locality, meaning they loop and repeat the same things in time (temporal locality) and also access the same regions of memory for a while (spatial locality).

If the resident memory includes all the pages of code that are running, all the data this code accesses, and all the file blocks being processed right now, the program runs without pausing. Having the working set in memory is necessary and sufficient.

WORKING SET TRACKING

We start by introducing a clock, and need to track reads/writes.

The clock defines *epochs*, usually 100ms each. If a block is accessed, mark it as active.

If a file (or a block of a file) hasn't been accessed in *t* epochs, evict it (but in fact, use a second-chance cache). *t* is tunable.

WHAT IF THE KERNEL STILL WON'T HAVE ENOUGH SPACE?

If we are still short on space, we might evict a block some process is going to need.

But at this point, we know that this process would not run if we schedule it. We could actually evict its entire working set. Later, when resuming it, we could pull its whole working set back in!

This is a strategy called "swapping".

WHICH POLICIES ARE FOUND IN LINUX?

The answer turns out to vary depending on which Linux.

Moreover, some allow "power users" to tune these policies.

Even so, this set of methods is part of an engineering design pattern. Even without a std::xxx class supporting this pattern, we often use these ideas when designing big systems!

PREFETCHING IS ALSO A POWERFUL TOOL

When Linux sees that you have read two or more blocks in a row, it prefetches the next blocks.

Goal is to have a steady overlap of file access with reading.

This is hugely valuable on networks, which often have very high bandwidth but “relatively” high delays.

SECONDARY INDICES

Systems often use some form of sort to access data. But it may not be the “primary” sort, which is based on the primary keys.

If the same sort is used often, we precompute a “secondary index”.

We can use this to initiate prefetching. Linux has an “asynchronous I/O” option that can start a read in advance of when data will be needed.

LINUX ASYNCHRONOUS FILE READS

POSIX API for the file system: AIO

When you compile, must include the aio.h header and also provide C++ with a flag, -rt.

This flag appears *at the end* of the command line. It is actually being passed to the linker, not the compiler.

POSIX AIO OPERATIONS: KERNEL API

aio_read – like read, but returns an aio "id"
aio_write – like write, but returns an aio "id"
aio_fsync – asynchronously requests that data be flushed to disk
aio_error – error number returned by a failed aio request
aio_return – obtain the outcome (returned result) for a request
aio_suspend – wait for specified request(s) to complete
aio_cancel – cancel specified requests
lio_listio – enqueue a list of operations rather than just one

MICROSOFT ASYNCHRONOUS I/O CLASS

Called System.IO

Has a native implementation for the Windows kernel, which "supports" Linux, but can also be accessed directly from C++

Portable (open source) but not widely used in C++ programs.

A PUZZLE ABOUT FAST-WC

Think back to our fast word-count program from Lecture 1 and 2

It had a thread to open files, which is good… fast-wc has many files to open!

But then it just used Linux file I/O (POSIX read) into a character array: read, then scan. Read, then scan. Etc.

A PUZZLE ABOUT FAST-WC

Suppose that we instead mapped entire files with mmap?

Or used AIO: We could start the read on block k+1 as we scan block k.

Which would be faster?

… IT ISN'T OBVIOUS!

With mapped files, we eliminate the memcpy from the kernel to user space that occurs with read.

- With mmap, the file is directly in user space (blocks from the buffer pool are mapped into the user memory)
- But memcpy runs at 18GB/second on compute30, and the whole Linux source files, in total, were only 836MB. So copying takes a total of just 0.64s. Saving this amount of time won't help much.
- Suggests that mmap won't be a big win for fast-wc

WHAT ABOUT ASYNCHRONOUS I/O

Ken tried it! It has no measurable impact

What does this tell us?

WHAT ABOUT ASYNCHRONOUS I/O

Ken tri

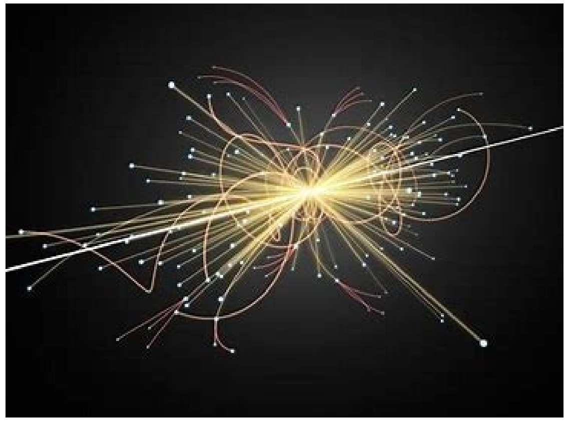
What

If asynchronous file reads don't help, this must mean that the application isn't pausing waiting for file reads to complete.

We know that memcpy, in total, is just 0.64s – and this cost is spread over all 24 cores, so any single core waits roughly 0.026s.

Conclusion? Linux prefetching must be working well enough to fetch the next block before we request it, so that read() doesn't really wait.

WHEN DOES ASYNCHRONOUS I/O HELP?



Imagine a program with a very random "looking" data access pattern, like a particle physics program doing analysis from detector data.

- Detectors produce gigabytes of data per event – huge files
- The application focuses on the data showing actual tracks

Linux won't anticipate this pattern of access, so asynchronous I/O could really help a lot. It offers a way to "tell Linux what to prefetch".

```
SELECT Orders.OrderID, Customers.CustomerName, …
FROM Orders where
           Orders.CustomerID=Customers.CustomerID;
```

A query such as this is fastest if both Orders and Customers are sorted by CustomerID.

DATABASE EXAMPLE

Many systems use big databases.

These manage data in *relations*, which are sorted tables. Each row has a primary key, and this is used for sorting.

But perhaps some query accesses the data using a <u>different</u> key.

```
SELECT Orders.OrderID, Customers.CustomerName, …
FROM Orders where
            Orders.CustomerID=Customers.CustomerID;
```

If Orders is sorted by OrderID and Customers by LastName, a secondary index for each will help.

DATABASE EXAMPLE

Here, the database will construct a fast search index: a data structure that tells it which row to access "next" (or even, which block in the file holding the data).

Using asynchronous I/O, the database can execute the query on the current block while having Linux prefetch the next blocks it will need to scan.

WHOLE-FILE PREFETCHING

With a network file system, it often makes sense to fetch the entire file the first time it is used.

In some systems, a prediction is even made *before the file is opened* and it is prefetched in anticipation.

This does use network resources, but hides the delays if the guess was valid!

WORKING SETS AT THE FILE LEVEL

These forms of file prefetching lead to the idea that groups of files are often accessed together.

The file system can potentially learn the group and fetch them all if any one is accessed.

In fact, you could imagine a helper file for each file: “if anyone accesses me, they will probably access xxx, yyy, and zzz too”

PRELOADING DLLS

An important case is when a process will use a collection of DLLs.

If Linux can anticipate which will be needed, it can load them all at once when the process is launched.

Many DLLs tend to all be used together, not in isolation. Linux has a “preload daemon” for this specific case!

FILE COMPRESSION IDEAS

File systems store files in blocks of fixed size

But do applications really access files block by block?

Some file systems have explored a mix of compression (which simply squeezes the file down) with *variable sized blocks*, aimed at transferring the entire "useful" portion of the file

ANTICIPATORY FILE CACHING IN THE NETWORK

Because more and more users are mobile, ISPs are thinking about how to improve performance and reduce load on the ISP network by caching videos and photos.

In these systems, as you move from place to place, they transfer "your" cache and prefetch data to the access point your device is associated with. This masks big last-moment delays!

IF ONE COPY IS GOOD… WHY NOT MORE?

Suppose that a storage system has huge capacity and is only partly in use.

Why not use this space in some way that could improve performance?

Systems based on this idea make extra copies just in case they might be used later.

PREDICTING POPULARITY

Facebook is an example of a company that uses machine learning to decide

- What to cache (and where to cache it)
- When to prefetch files (photos or videos)
- Which files may become viral
- Which files have gone cold and can be evicted

EXAMPLE: WHICH MIGHT GO VIRAL?

Photo of Megan and Harry buying ice cream for Archie.

Surveillance web cam photo that shows someone walking a dog.

Photo of your sister’s covid-safe wedding last week.

FACTORS THEY CONSIDER

Who is in the photo? Who follows these people?

How old is the photo? How good is the photo quality?

Are there “early indicators” that people find it interesting?

BOTTOM LINE SUMMARY?

A computer has a lot of capacity to do things concurrently.

Prefetching or preloading files is a huge win:

- The costs of data access aren't eliminated, but are mostly hidden
- The work of prefetching/preloading is often mostly in hardware
- We own the hardware… why not keep it busy?
- Tremendous variety of examples where the same basic ideas are employed for many different purposes.