On faster application startup times: Cache stuffing, seek profiling, adaptive preloading

bert hubert
Netherlabs Computer Consulting BV
bert.hubert@netherlabs.nl

Abstract

This paper presents data on current application start-up pessimizations (on-demand loading), relevant numbers on real-life harddisk seek times in a running system (measured from within the kernel), and shows and demonstrates possible improvements, both from userspace and in the kernel. On a side note, changes to the GNU linker are discussed which might help. Very preliminary experiments have already shown a four-fold speedup in starting Firefox from a cold cache.

1 1980s and 1990s mindset

The cycle of implementations means that things that were slow in the past are fast now, but that things that haven't gotten any faster are perceived as slow, and relatively speaking, are.

CPUs used to be slow and RAM was generally fast. These days, a lot of CPU engineering goes into making sure we do not mostly wait on memory.

Something like this has happened with hard disks. In the late 1980s, early 90s, there was a lot of attention for seek times, which was understandable as these were in the order of 70ms. These have been reduced, but not as much as disk throughput has increased. Typical measured seek times on laptop hard disks are still in the 15-20ms region, while 20 megabytes/second disk speeds would allow the disks of old to be read in their entirety in under 10 seconds.

2 Some theory

To retrieve data from disk, four things must happen:

1. The instruction must be passed to the drive
2. The drive positions its reading head to the proper position
3. We wait until the proper data passes under the disk
4. The drive passes the data back to the computer

It is natural to assume that seeking to locations close to the current location of the head is faster, which in fact is true. For example, current Fujitsu MASxxxx technology drives specify the 'full stroke' seek as 8ms and track-to-track latency as 0.3ms.

However, for many systems the actual seeking is dwarfed by the rotational latency. On average, the head will have to wait half a rotation for the desired data to pass by. A quick calculation shows that for a 5400RPM disk, as commonly found in laptops, this wait will on average be 5.6ms.

This means that even seeking a small amount will at least take 5.6ms.

The news gets worse - the laptop this article is authored on has a Toshiba MK8025GAS disk, which at 4200RPM claims to have an average seek time of 12ms. Real life measurements show this to be in excess of 20ms.

3 What this means, what Linux does

That one should avoid seeking by all means. Given a 20ms latency penalty, it is cheaper to read up to 5 megabytes speculatively to get to the desired location

In Linux, on application startup, the relevant parts of binaries and libraries get mmapped into memory, and the CPU starts executing the program. As the instructions are not loaded into memory as such, the kernel encounters page faults when data is missing, leading to disk reads to fill the memory with the executable code.

While highly elegant, this leads to unpredictable seek behaviour, with occasional hits going *backward* on disk. The author has discovered that if there is one thing that disks don't do well, it is reading backwards.

Short of providing a 'reverse' setting to the disk's engine, the onus is on the computer to optimize this away.

4 How to measure, how to convert

As binary loading is "automatic", userspace has a hard time seeing page faults. However, the recently implemented 'laptop mode' not only saves batteries, it also allows for logging of actual disk accesses.

At the level of logging involved, only PID, device id and sector are known, which is understandable as the logging infrastructure of laptop mode is mostly geared towards figuring out which process is keeping the disk from spinning down.

Typical output is:

```
bash(261): READ block 11916
  on hda1
bash(261): READ block 11536
  on hda1
bash(261): dirtied inode 737
  (joe) on hda1
bash(261): dirtied inode 915
  (ld-linux.so.2) on hda1
```

Short of dragging around a lot more infrastructure than is desireable, the kernel is in no position to help us figure out which files correspond to these blocks.

Furthermore, there is no reverse map in any sane fs to tell us which block belongs to which file.

Luckily, another recent development comes to our rescue: syscall auditing. This can be thought of as a global strace, allowing designated system calls to be logged, whatever their origin. This generates a list of files which might have caused the accesses.

This combined with the forward logical mapping facility used by lilo to determine the sector

locations of files allows us to construct a partial reverse map that should include all block accesses logged by the kernel.

From this we gather information which parts of which positions in whic files will be accessed or system or application startup.

5 Naively using the gathered information

Using the process above, a program was written which gathers the data above for a typical Debian Sid startup, up to and including the launch of Firefox. On next startup, a huge shell script used ‘dd’ to read in all relevant blocks, sequentially. Even without merging nearby reads, or or utilizing knowledge of actual disk layout, this sped up system boot measureably. Most noticeable was the factor of four improvement in startup times of Firefox.

In this process a few things have become clear:

- There are a lot of reads which cannot be connected to a file
- The ‘dd’ read script is very inefficient
- The kernel has its own ideas on cache maintenance and throws out part of the data
- Reads are spread over a large number of files

The reads which cannot be explained are in all likelihood part of either directory information or filesystem internals. These are of such quantity that directory traversal appears to be a major part of startup disk accesses.

It is interesting to note that only in the order of 40 megabytes of disk is touched on booting, leading to the tentative conclusion that all disk access could conceivably be completed in 2 seconds or less.

However, it is also clear that reads are spread over a large number of files, making naive applications of readahead(2) less effective.

6 More sophisticated ways of benefiting from known disk access patterns

Compiler, assmbler and linker work together in laying out the code of a program. Andi Kleen has suggested storing in an ELF header which blocks are typically read during startup, allowing the dynamic linker to touch these blocks sequentially. Howver, this idea is not entirely relevant anymore as most time is spent touching libraries, which will have differing access patterns for each file program using them.

Linus Torvalds has suggested that the only way of being really sure is to stuff the page cache with a copy of the data we know that is needed, and that we store that data in a sequential slab on disk so as to absolutely prevent having to seek.

The really dangerous bit is that we need to be very sure our sequential slab is still up to date. It also does not address the dentry cache, which appears to be a dominant factor in bootup.

Another less intrusive solution is to use a syscall auditing daemon to discover which application is being started and touch the pages that were read last time this binary was being started. During bootup this daemon might get especially smart and actually touch pages that it knows will be read a few seconds from now. The hard time is keeping this in sync.

7 Conclusions

Currently a lot of time is wasted during application and system startup. Actual numbers appear to indicate that the true amount of data read during startup is minimal, but spread over a huge number of files.

The kernel provides some infrastructure which, through convoluted ways, can help determine seek patterns that userspace might employ to optimize itself.

A proper solution will address both directory entry reads as well as bulk data reads.