Clonefiles

Mario Wolczko, Sep 2021

Clonefiles are a relatively new addition to various flavors of Unix (and Windows). They were added to macOS with APFS in macOS Sierra (10.12.4). Linux has *reflinks*, supported by XFS, BtrFS and OCFS2. Windows has had something similar in ReFS since 2016. In macOS they are created with the clonefile(2) system call; in Linux, you use ioctl_ficlone(2) (or ioctl_fideduperange(2)). At user level, there is `cp -c` (macOS) and `cp --reflink` (Linux).

What is a clonefile?

A clonefile (I'll use the macOS term) is created by cloning an existing file, and shares the original's data, and so needs no additional space for data (but of course needs space for metadata). Creating a clonefile is near-instantaneous, no matter the size of the file being cloned. Unlike a hard link, blocks in a clonefile are unshared when written to; *copy on write*. In the extreme case, after extensive modification a clonefile has no data shared with the file it was cloned from. I've described this as if the clone was somehow distinguished, but in fact the situation is completely symmetric; you can think of each file as being a clone of the other. You can, of course, clone a clone.

I've been working with clonefiles recently, as part of my work on persistence. They provide a mechanism by which a file can be quickly snapshotted to record its current state, something very useful for heap checkpoints. However, if you're in the habit of working with large clonefiles (which I am) there's a problem: none of the other Unix tools are aware of them. This isn't a correctness issue: any tool which works on a plain file will work on a clonefile. However, there are no tools to *observe* the sharing relationships between files (useful when debugging), and none of the standard tools are *optimized* for clonefiles. Case in point: if you compare two clones using `cmp(1)`, the shared blocks could obviously be omitted from the comparison, as they are by definition identical in the two files. But `cmp` reads and compares the data anyhow. For clones of, say, 100GB, `cmp` can run for many minutes, needlessly comparing each data block with itself. Note that `cmp` *is* optimized to detect hard links and skip the comparison, but hard links have been in Unix since the early days. In the rest of this post I will describe a tool I've written that can elucidate sharing relationships, and is also the basis for a clonefile-optimized `cmp`.

Detecting shared blocks

The first problem we need to address is how to tell when two files are sharing a block. Neither macOS nor Linux provides an interface which can answer this directly, but there is an indirect way: we can get the underlying physical address of each of the two blocks, and if they live in the same filesystem and have the same physical address they are the same block. I learned this technique from this post and the associated repo, which is about how to do this in macOS. However, I will switch to discussing Linux next, as the Linux interface has a useful abstraction, and will return to macOS later.

Extents

An *extent* is a contiguous region of a file which occupies contiguous physical storage. Once we know the logical offset (*offset*) and physical address of the start of the extent (*physaddr(start)*) we can easily calculate the physical address of any logical offset within the extent.

$$physaddr(offset) - physaddr(start) = offset - start$$

The Linux FIEMAP interface returns the set of physical extents which underlie a logical region (*start*, *length*) of a file. This is a relatively new interface; it has no manual page, but there is a text file with most of the details; the rest are available in the associated header file.

To read the set of extents for a region of a file we can do something like this:

```
    struct fiemap fm= { (__u64)start, (__u64)len, 0L, 0L, 0 };
    if (ioctl(fd, FS_IOC_FIEMAP, &fm) < 0)
        fail("Can't get # of extents : %s\n", strerror(errno));
    unsigned n= fm.fm_mapped_extents;
    struct fiemap *pfm= malloc_s(sizeof(struct fiemap)
                                 + n * sizeof(struct fiemap_extent));
    pfm->fm_start=      start;
    pfm->fm_length=       len;
    pfm->fm_flags=          0;
    pfm->fm_extent_count=   n;
    if (ioctl(fd, FS_IOC_FIEMAP, pfm) < 0)
        fail("Can't get list of extents : %s\n", strerror(errno));
    if (pfm->fm_mapped_extents != n)
        fail("file is changing: number of extents changed\n");
```

The code calls `ioctl(FS_IOC_FIEMAP)` twice: once to get the number of extents, and again to fill out the appropriately sized array of structs. Obviously, the file should not be undergoing modification while this is done, as the number of extents could change between the two calls.

Having done this, we can then make a printing utility to list the extents, e.g.:

```
$ extents -P -f -p /mnt/big/a
(1) /mnt/big/a
#              Logical        Physical          Length   Flags
                Offset          Offset
1                    0     25797865472          524288
2               524288     25798914048          524288  LAST
```

This shows a file of two extents, each half a megabyte in size.

Flags

In Linux, each extent comes with a set of flags, one of which tells us if the extent is shared (i.e., part of a clonefile):

```
$ extents -P -f -p /mnt/big/b1
(1) /mnt/big/b1
#              Logical        Physical          Length   Flags
                Offset          Offset
1                    0     25798389760          524288  SHARED
2               524288     25800028160          524288  LAST
```

However, it does not tell us with *which* other file(s) it is shared. For that, we need to compare the physical address (returned as another field of the struct) with the physical address of a candidate extent. For example, by comparing the last example with the following we can see that these files share the first extent:

```
$ extents -P -f -p /mnt/big/b
(1) /mnt/big/b
#              Logical        Physical          Length   Flags
                Offset          Offset
1                    0     25798389760          524288  LAST SHARED
```

We should be cautious when interpreting the information return by FIEMAP: it is internal to the implementation of the filesystem, and therefore dependent on a myriad of otherwise invisible details. For example, if we get the extent list for a newly created file, we might observe that the physical addresses have not yet been allocated because the file's data are still in flight:

```
$ dd if=/dev/random count=100 of=/mnt/big/c; extents -P -f -p /mnt/big/c
100+0 records in
100+0 records out
51200 bytes (51 kB, 50 KiB) copied, 0.00255701 s, 20.0 MB/s
(1) /mnt/big/c
#              Logical        Physical          Length   Flags
                Offset          Offset
1                    0               0           51200  LAST UNKNOWN DELALLOC
$ sync; extents -P -f -p /mnt/big/c
(1) /mnt/big/c
#              Logical        Physical          Length   Flags
                Offset          Offset
1                    0        10899456           51200  LAST
```

Holes

If we look at the sample output below we will see something strange: there is a gap between the two extents (i.e., the logical offset of the second is not the same as the logical offset of the end of the first).

```
$ extents -P -p -f /mnt/big/hole
(1) /mnt/big/hole
#              Logical        Physical          Length   Flags
                Offset          Offset
1                    0        11161600           53248
2                524288     17192820736           51200  LAST
```

This is because Unix files can contain *holes*, which are regions which were never written, and so do not have any physical storage allocated. When reading from the file (via read(2)), holes read as zeroes.

Detecting shared extents

Suppose we want to understand the current sharing relationship among a set of files. We can get the list of extents for each file and examine their physical storage locations to see which map onto the same location. All the cautions from the previous section apply; e.g., we should ensure that the data are stable (using `sync(1)` or `fsync(2)` can help).

Note, however, that the Linux `ioctl_ficlonerange(2)` interface allows a region of a file to be cloned to another file (or even the same file!) at *any* logical offset (although in practice sharing is restricted to block granularity). Hence the comparison has to compare every extent's physical locations with every other extent. One solution is to sort a single list of all extents by physical address and then look for overlaps. The output will be a list of sets of extents, in which every element of a set shares storage; any extent which does not share will be in a singleton. Using the extents' lengths as a secondary sort key simplifies the processing:

$$ext_A < ext_B \vDash physaddr(ext_A) < physaddr(ext_B)$$
$$\vee\ (physaddr(ext_A) = physaddr(ext_B) \wedge length(ext_A) < length(ext_B))$$

Let's say we are mid-way through the list; the current extent under consideration is *cur* and the next extent is *nxt*. The current extent *cur* is any element from a set of extents *sh* which are

known to share storage (i.e., have the same physical address and length). The physical address, logical address and length of an extent are available via functions *p*, *l* and *len*, resp.

```
if (p(cur) < p(nxt)) {
    if (p(cur) + len(cur) > p(nxt)) {
        len(cur)= p(nxt) - p(cur);
        tail_len= p(cur) + len(cur) - p(nxt);
        chop_insert_tails(sh, tail_len);
    }
    report(sh);
} else { // p(cur)==p(nxt)
    add(sh, nxt);
    if (len(cur) < len(nxt)) {
        len(nxt) -= len(cur);
        l(nxt) += len(cur);
        p(nxt) += len(cur);
        re_sort(&nxt);
    } else // len(cur)==len(nxt)
        get_next(&nxt);
}
```

If *cur* does not overlap *nxt* then we are finished with *cur*, since it must precede *nxt* (because of the sort order); we can report *sh* as a set of shared extents. If *cur* begins before *nxt* but overlaps *nxt* then we split each extent in *sh* at the overlap and reinsert the tails into the to-do list at the appropriate place to maintain sort order; we are then done with *sh*.

Otherwise, *cur* and *nxt* must begin at the same physical address. Either they are the same length (in which case *nxt* is added to *sh*) or *cur* is shorter than *nxt*, in which case we split *nxt* at the end of *cur*, adding the head to *sh* and putting the tail back in the to-do list.

Here's an example of this algorithm in action, listing the shared and unshared extents from the earlier example:

```
$ extents  /mnt/big/b /mnt/big/b1
(1) /mnt/big/b
(2) /mnt/big/b1

Shared:
File#:                                1                2
#               Length           Logical          Logical
                                  Offset           Offset
1               524288                 0                0

Not Shared:
(2) /mnt/big/b1
#              Logical          Length
                Offset
1               524288          524288
```

Clone-aware `cmp`

We now have enough to build a version of `cmp` which is clone-aware. I call it `ccmp`. It uses knowledge about shared extents to skip over them.

Done properly, this would probably be in C, derived from `cmp`. But, I'm going to be lazy here and instead write it in `bash`, using `cmp` itself to find differences within unshared regions — using the `-i` and `-n` flags to `cmp` we can direct to specific regions within the files to be

compared. I enhanced `extents` to report, for a pair of files, the boundaries of regions which are not shared and hence must be fed to `cmp`.

```
$ extents -c /mnt/big/b /mnt/big/b1

524288 524288 524288
```

The heart of `ccmp` looks something like this:

```
extents -c $file1 $file2 | while read start1 start2 len
do
  cmp -i $start1:$start2 -n $len $file1 $file2
done
```

`cmp` prints differences along with the offset relative to where it was asked to start comparing, so some post-processing is needed (not shown) to sanitize the output.

With some more shell hackery we can support most of `cmp`'s options, and even massage the output to be largely the same. See the repo for extra detail. Note that we cannot make `ccmp` into a POSIX-compliant `cmp`, because the spec. requires the output to include the line number — which can only be obtained by reading *all* the data.

Making it work on macOS

MacOS does not provide an extent interface as rich as Linux's. On macOS, we can get the physical address of a block using fcntl(2) with the command `F_LOG2PHYS_EXT`; this operation takes an open file descriptor and an offset in the file and returns the offset of the corresponding block on the device on which it resides. In an additional field of the struct passed to `fcntl` we can ask how many contiguous bytes remain at that offset, which is returned via the same field:

```
static off_t l2p(unsigned fd, off_t off, off_t max, off_t *pcontig) {
    struct log2phys ph= {0, max, off};
    if (fcntl((int) fd, F_LOG2PHYS_EXT, &ph) >= 0) {
        if (pcontig != NULL) *pcontig= ph.l2p_contigbytes;
        return ph.l2p_devoffset;
    }
    if (errno == ERANGE)
        return -1; // hole
    fail("fcntl failed! %s\n", strerror(errno));
    return -2; // never reached
}
```

In the event that we run into a hole, we can use the command `SEEK_DATA` with `lseek(2)` to skip over the hole to the next allocation. Here's the code to build extents from the `fcntl` returns:

```
    off_t off= 0L;
    while (off < file_size) {
        off_t contig;
        off_t ph= l2p(fd, off, file_size - off, &contig);
        if (ph >= 0) {
            if (contig <= 0) fail("contig not positive: %d\n", contig);
            new_extent(off, ph, contig);
            off += contig;
        } else { // skip over hole
            off= lseek(fd, off, SEEK_DATA);
            if (off < 0) fail("lseek failed: %d\n", strtoerror(errno));
        }
```

```
    }
```

Performance

Let's give it a spin. If we try `ccmp` on a pair of small files which are not clones, the performance is (as expected) worse than `cmp`, since we're doing extra work, and some of it in `bash`.

```
$ time cmp cmp.o opts.o
cmp.o opts.o differ: char 17, line 1
        0.00 real         0.00 user         0.00 sys
$ time ccmp cmp.o opts.o
cmp.o opts.o differ: char 17
        0.02 real         0.01 user         0.01 sys
```

However, if we apply it to clones, things improve. On my machine, a 2018 MacBook Pro, with the files on an external drive (you'll see why in a moment), comparing clones of 10 MB takes the same time in `cmp` and `ccmp`.

```
$ ls -lh foo bar
-rw-r--r--  1 mario  admin    10M Sep 12 13:40 bar
-rw-r--r--  1 mario  admin    10M Sep 12 13:40 foo
$ time cmp foo bar
        0.02 real         0.01 user         0.00 sys
$ time ccmp foo bar
        0.02 real         0.01 user         0.00 sys
```

Caveat: take all these numbers with a pinch of salt. If you repeat the comparisons, the times get shorter, presumably due to file caching, but I think the first timing is more representative of a real scenario.

Once we get bigger than 10MB, `ccmp` pulls ahead (if there is substantial sharing):

```
$ ls -lh foo bar
-rw-r--r--  1 mario  admin   1.0G Sep 12 13:42 bar
-rw-r--r--  1 mario  admin   1.0G Sep 12 13:42 foo
$ time cmp foo bar
        5.97 real         0.65 user         1.03 sys
$ time ccmp foo bar
        0.02 real         0.00 user         0.01 sys
```

And by the time we get to clones of a terabyte, `ccmp` is *much* faster:

```
$ ls -lh big*
-r--r--r--  1 mario  admin   1.0T Aug  4 14:12 big.jah
-r--r--r--  1 mario  admin   1.0T Aug  4 14:14 big2.jah
$ time ccmp -bl big.jah big2.jah | wc -l
     135
        18.2 real         3.94 user         5.01 sys
$ time cmp -bl big.jah big2.jah | wc -l
     135
       303m5.2 real      6m25.0s user      21m51.9 sys
```

You read that right: 18 seconds vs. 5 hours, a 1000-fold speedup! This is a comparison of two clonefiles with >100 randomly selected bytes altered in one.

On Linux I'll compare using XFS, since ext4 does not support reflink. (You can easily create an XFS filesystem in an ext4 file and mount it, as explained here.) On my aging (SSD-based) Linux laptop, the break even point for `cmp` vs `ccmp` is also at around 10 MB.

As an extreme test, let's create a file that contains only two blocks of random data, A and B, with each block reflinked repeatedly in the same file in the sequence ABABA…, repeated 1000 times:

```
$ ls -lh self.dat
-rw-r--r-- 1 mario mario 7.9M Sep 15 14:49 self.dat
```

If we compare the file with itself but at starting offsets differing by an even multiple of block size, all comparisons are elided, whereas at odd multiples nothing is elided:

```
$ time ccmp -i 0:8192 -bl self.dat self.dat | wc -l
cmp: EOF on self.dat
0
      0.05 real         0.02 user         0.04 sys
$ time ccmp -i 0:4096 -bl self.dat self.dat | wc -l
cmp: EOF on self.dat after byte 8196096
8164080
      7.30 real         7.01 user         0.52 sys
```

Summary

Clonefiles allow for block-level sharing in modern Unix filesystems, but because they're relatively new few tools know about them. I've described a new tool I've written to explore clonefiles, which can enumerate the sharing relationships among a set of files. It's also possible to make some tasks which involve enormous clone files much faster (when the clones share a lot) and my tool enables a version of `cmp` which can skip over shared regions.

Appendix: Source code

Source code is available at https://github.com/mwolczko/extents