Bucket Brigades:
Data Management for Apache 2.0

Cliff Woolley
Apache Portable Runtime Project

Overview

- What are buckets and what good are they?
- Data structures
- Debugging
- Operations
- Case study
- Stupid bucket tricks

What are buckets?

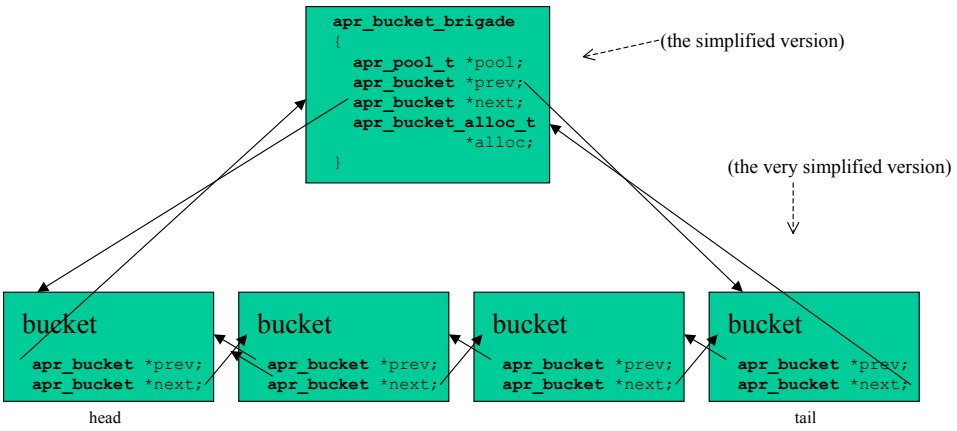
- Simply: a data source abstraction
- A convenient, efficient way of handing this data around without unnecessary copying

Apache filters don't have to know how to read from each different data source – just how to use the abstract interface. So whether the data is coming from a file or from a socket or whether it resides on the heap or on the stack or is a constant string literal… the filters don't have to treat them specially. This is basically the embodiment of the OO programming mindset – though it happens to be implemented in C in this case.

The other reason for buckets was that we needed a clean, efficient way to process data streams and pass the streams from one filter to another in as close to a zero-copy way as possible. We'll take a look at the features of buckets and brigades that make that happen.

The brigade data structure

- At its heart, it's just a ring:

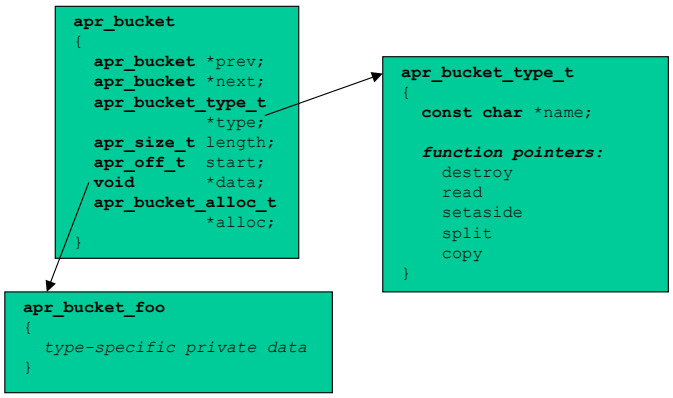


Specifically, it's a doubly-linked list where the tail points back to the head. There is a magic pointer value called a "sentinel" that is before the head and after the tail. The sentinel looks like a bucket in that it has a prev and next pointer, but it's not a bucket (in that you can't do any of the bucket operations on it or you'll segfault).

The buckets shown here are just kind of place holders – a real bucket has lots of other stuff like size and data pointers and so on – but we'll get to that next. For now, all you need to see is that each bucket has a prev and next just like the sentinel, and when you connect them all up, they form a ring.

In the apr_bucket_brigade data structure, the other things we see are a pool and a "bucket allocator." The apr_bucket_brigade structure itself is allocated from the pool, but other than that, all the pool is used for is for cleanup convenience. When the pool that a brigade is in goes away, the pool cleanup routine for the brigade automatically deletes all the buckets still in the brigade. There are some pitfalls (especially in regard to memory leaks) here that we'll talk about in more detail later – the short version is that you have to be careful to make sure that all of your buckets get deleted one way or another since the buckets, unlike the brigade, are *not* allocated from a pool, they're allocated from an apr_bucket_alloc_t (hence the other pointer in the brigade structure). Bucket allocators are similar in some ways to pools, but they give the allocated memory an unlimited lifetime and have no notion of a "cleanup."

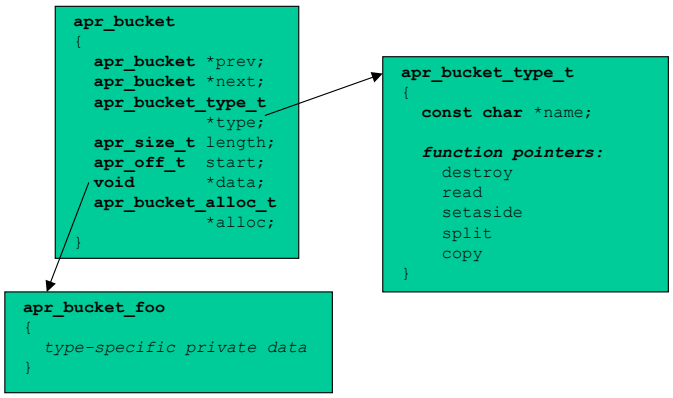
The bucket data structures



Okay, lots to explain here. I've left out a few fields, but these are kind of the highlights.

Let's look first at the apr_bucket structure itself. We've already seen the prev and next fields. Each bucket has a type pointer that points to an apr_bucket_type_t. The type essentially tells you what kind of structure to expect to find in the void *data pointer. There's also a pointer to the bucket allocator used to create this bucket (which can then be used to create private structures, buffers, and copies of the bucket, for example). Finally we have the ever-important start and length. Note that each bucket is limited in size (length) to the number of bytes that can be represented by an apr_size_t. If your data is longer than that, no problem; a brigade can hold up to an apr_off_t's worth – you just have to split it into multiple buckets. This size_t vs off_t thing has been debated ad nauseum… what you see is what we've ended up with. It fixes come problems and causes others, but overall it's the best compromise we've seen so far. The "start" offset is used so that when you want to chop some data off the beginning of the bucket, all you do is tweak the start parameter rather than having to copy the rest of the data to a new buffer. This can also be used to split buckets without copying the data – you end up with two (reference counted) buckets sharing a single copy of the data; the two buckets are identical except that their start and length variables are adjusted accordingly.

The bucket data structures



Then, as I mentioned, we have the apr_bucket_type_t structure. This is one that you should never have to mess with if all you do is *use* buckets. You'll only have to mess with it if you make new bucket types. Each bucket type (FILE, HEAP, etc) has a static const instance of this type structure. It defines the function pointers that operate on buckets of that type and in general allows you to know what kind of bucket it is – this is very similar in concept to a C++ virtual function table. I should take a step back here, though, and say that in most cases filters should not know or care what the type of a given bucket is – manipulating a bucket's internals defeats the whole purpose of the abstraction. If you *do* have to find out the type of a bucket, there are some macros you can use that will make it easier. Similarly, there are macros that extract the function pointers for you so it looks to you like a simple function call (eg, apr_bucket_read(e) rather than e->type->read(e)). I'll get to those in a bit.

Finally, there is the "apr_bucket_foo" structure as I've called it here. Each bucket type has its own private data structure – apr_bucket_heap for HEAP buckets, apr_bucket_file for FILE buckets, and so on. It should virtually never be the case that you need to deal with these directly from the outside; all access to them is through the five function pointers.

Standard disclaimer: There are a few fields (two or three) that I've omitted here; they're used internally. Some of them have accessor macros that I'll get to in a bit, but the rest you don't need to worry about at all.

Debugging brigades

- Examining a brigade with gdb:

```
(gdb) dump_brigade b
dump of brigade 0x8299d90
   | type     (address)    | length | data addr  | contents               | rc
-------------------------------------------------------------------------------
 0 | HEAP     (0x082973a0) | 287    | 0x082c5a58 | [HTTP/1.1 200 OK~~...] | 1
 1 | FILE     (0x08295820) | 2048   | 0x08296d60 | [**unprintable**]      | 1
 2 | EOS      (0x08296fe0) | 0      | 0x00000000 |                        | n/a
end of brigade
```

Perhaps the best way to understand what's going on with these structures is to actually look at some examples, so I'll take a detour here and point out some of the debugging tools we have made available. Apache 2.0's top-level directory contains a file called ".gdbinit" that has some gdb macros for looking at brigades and buckets. They do the work of figuring out the bucket types and interpreting the bucket content appropriately, etc.

Here is a (reasonably simple) sample brigade. In this example, we have three buckets that make up our brigade. These buckets represent an entire response: one bucket of data on the heap that contains the HTTP headers, one bucket that contains a handle to the file being served, and a final one called an EOS bucket that signifies that no further buckets will be used in this response. Three things are of particular note here. First, both the HEAP and FILE bucket types are reference counted… in this case there is only one reference to each of the "data addr" structures (and therefore only one reference to the "contents"). That brings up the second point, which is that the macro is unable to tell us what's in the FILE bucket, because the contents of the file are not in memory. No problem – the file will be sent directly to the client using sendfile() if possible. You could always call apr_bucket_read() on that FILE bucket to find out what's in the file, but that would cause it to no longer be a FILE bucket. (Buckets can "morph" themselves to other types; I'll talk about at toward the end of the talk.)

Debugging brigades

- Examining a brigade with gdb:

```
(gdb) dump_brigade b
dump of brigade 0x8299d90
   | type     (address)    | length | data addr  | contents               | rc
-------------------------------------------------------------------------------
 0 | HEAP     (0x082973a0) | 287    | 0x082c5a58 | [HTTP/1.1 200 OK~~...] | 1
 1 | FILE     (0x08295820) | 2048   | 0x08296d60 | [**unprintable**]      | 1
 2 | EOS      (0x08296fe0) | 0      | 0x00000000 |                        | n/a
end of brigade
```

The final thing I want to emphasize is that even though the entire response is contained in one brigade in this example, that is absolutely NOT guaranteed to be the case, and in fact usually isn't. You as a filter author need to expect that the response will come in multiple brigades (ie, multiple calls to your filter)… you need to set aside only those buckets you *have to* hang onto, and pass along any you're done with to the next filter. Only when you see an EOS bucket can you assume that you're really at the end; but at that point you have to keep in mind that you have to finish what you're doing and pass along all remaining buckets because your function won't get called again for this request.

Debugging buckets

- Examining a bucket with gdb:

```
(gdb) dump_bucket 0x08296d60
 bucket=MMAP     (0x08296d60) length=21     data=0x08296fe0
     contents=[~1asdlkfqbasaser...] rc=1

(gdb) p *(apr_bucket*)0x08296d60
$1 = {link = {next = 0x8296c20, prev = 0x82a5744}, type = 0x401a9820,
  length = 21, start = 65, data = 0x8296fe0,
  free = 0x806d424 <apr_bucket_free>, list = 0x8295800}

(gdb) p *((apr_bucket*)0x08296d60)->type
$2 = {name = 0x401a7e9a "MMAP", num_func = 5, is_metadata = APR_BUCKET_DATA,
  destroy = 0x4019ce3c <mmap_bucket_destroy>,
  read = 0x4019cdd0 <mmap_bucket_read>,
  setaside = 0x4019cf68 <mmap_bucket_setaside>,
  split = 0x806f654 <apr_bucket_shared_split>,
  copy = 0x806ea84 <apr_bucket_shared_copy>}
```

Similar to the dump_brigade gdb macro, we have a dump_bucket macro. It's not *quite* as useful as dump_brigade, but it does have one handy feature – it dumps out the bucket contents for you so you don't have to do a bunch of pointer casting by hand. The …] means that this is just the beginning of the data, and there's more. The ~ is a placeholder for a non-printable character.

If you really want to dig into the bucket, you can always print it out and see *all* the fields. If you have a variable of type apr_bucket*, you can just dereference it. Or if you have an address, you can cast the address to an apr_bucket* as I've done here and dereference THAT. From there, you can get to the bucket type and see all of the function pointers.

Operations

- Brigade macros
- Brigade functions
- Bucket macros
- Bucket functions

For completeness, I'll go through all of the operations you can perform on buckets and brigades. I'll just it the highlights as far as describing them, as they're all well-documented in the apr_buckets.h header file, but I want you to at least have a rough familiarity with them before we go any further.

Brigade operations

- APR_BRIGADE_SENTINEL(b)
- APR_BRIGADE_EMPTY(b)
- APR_BRIGADE_FIRST(b)
- APR_BRIGADE_LAST(b)

APR_BRIGADE_SENTINEL(b) takes a bucket brigade as a parameter and returns the magic pointer value that is the sentinel for the brigade. If a bucket has that pointer value as its prev or next pointer, you know that that bucket is at the head or tail of the brigade, respectively.

APR_BRIGADE_EMPTY(b) tells you whether brigade b is empty or not.

APR_BRIGADE_FIRST(b) returns a pointer to the first bucket in brigade b.

APR_BRIGADE_LAST(b) returns a pointer to the last bucket in brigade b.

Brigade operations

- APR_BRIGADE_FOREACH(e, b)
  - be VERY careful on this one

```
This is the same as either:
      e = APR_BRIGADE_FIRST(b);
      while (e != APR_BRIGADE_SENTINEL(b)) {
          ...
          e = APR_BUCKET_NEXT(e);
      }
  OR
      for (e = APR_BRIGADE_FIRST(b);
           e != APR_BRIGADE_SENTINEL(b);
           e = APR_BUCKET_NEXT(e)) {
          ...
      }
```

This one is so incredibly tricky that I had to give it its own whole slide. Be *very* careful using it. If one of the two loops shown above would not work for you, APR_BRIGADE_FOREACH won't work for you.

Specifically, be aware that you cannot change the value of e within the foreach loop, nor can you destroy the bucket it points to. Modifying the prev and next pointers of the bucket is dangerous but can be done if you're careful. If you change e's value or destroy the bucket it points to, then APR_BRIGADE_FOREACH will have no way to find out what bucket to use for its next iteration. The reason for this can be seen by looking closely at the equivalent loops given in the tip above.

Because of how tricky this can be and how little work it actually saves you, we've decided that this macro is more trouble than it's worth and we're going to get rid of it at some point in the future. It's still there for now though (even though it's deprecated), so I wanted you to be aware of it and what havoc it can wreak since you might still come across it in older code.

Brigade operations

- APR_BRIGADE_INSERT_HEAD(bb, e)
- APR_BRIGADE_INSERT_TAIL(bb, e)
- APR_BRIGADE_CONCAT(a, b)
- APR_BRIGADE_PREPEND(a, b)

These are all reasonably obvious. Here a, b, and bb are brigades and e is a bucket. In the concat and prepend case, a will get the combined list of buckets that were originally in a and b. b is left empty afterward. This is done without copying any data and, even better, without traversing either brigade. These are constant time operations. So if you have to move a bunch of buckets from one brigade to another, which is quite common in filters, use these macros to do it!

Brigade operations

- apr_brigade_create(bb)
- apr_brigade_destroy(bb)
- apr_brigade_cleanup(bb)
- apr_brigade_split(bb, e)
- apr_brigade_partition(bb, offset, e)
- apr_brigade_length(bb, readall, length)

So that does it for macros. Let's not get too bogged down in details here because most of these are either obvious or sufficiently well documented that I don't need to spend a lot of time on them. Just for the sake of completeness, though, what follows is a list of all the functions you can run on a brigade.

Of note on this slide: apr_brigade_destroy() and apr_brigade_cleanup() are the same thing in that both destroy all of the buckets in the brigade; the difference is that apr_brigade_destroy() unregisters the pool cleanup afterward. So if you're done with the brigade for good, use apr_brigade_destroy(); if you're done with the buckets but might want to put other buckets in the brigade later (rather than having to create a new brigade for the new buckets), use apr_brigade_cleanup(). apr_brigade_split() splits a brigade at the given bucket and returns the second brigade. apr_brigade_partition() does the same basic thing as split, except that it splits at a given offset into the brigade rather than at a certain bucket.

Brigade operations

- apr_brigade_create(bb)
- apr_brigade_destroy(bb)
- apr_brigade_cleanup(bb)
- apr_brigade_split(bb, e)
- apr_brigade_partition(bb, offset, e)
- apr_brigade_length(bb, readall, length)

apr_brigade_length() does what it sounds like. The only trick there is the readall flag; if it's set, then any buckets of an unknown length (like pipe or socket buckets) will be read from in order to determine their length. In some cases you might not want to do that, so if you set readall to 0 then apr_brigade_length() will return a length of –1 (the "unknown" length) if it encounters any buckets of an unknown length.

One other important point: several of these functions return apr_status_t's – you should always check the return codes. For example, if apr_bucket_read() returns an error, you'll want to know about it, and several of the functions above call apr_bucket_read() and pass back any errors that might result from the read.

Brigade operations

- apr_brigade_flatten(bb, c, len)
- apr_brigade_pflatten(bb, c, len, pool)
- apr_brigade_split_line(bbout, bbin, block, maxbytes)
- apr_brigade_to_iovec(bb, vec, nvec)

These functions are basically for extracting the content of brigades for use by code that doesn't understand brigades, and moreover expects to handle *all* of the data as one contiguous chunk (for the flattening or for line extraction) or as an iovec.

The word to the wise is this: wherever you possibly can, use apr_bucket_read() to read the data from buckets a bit at a time. Avoid these functions unless you have absolutely no alternative. They defeat the zero-copy mentality of bucket brigades and/or can lead to unexpectedly large amounts of data in memory at a time. With apr_bucket_read() it's much easier to make sure you have finite storage requirements.

Brigade operations

- apr_brigade_write(bb, flush, ctx, str, nbyte)
- apr_brigade_writev(bb, flush, ctx, vec, nvec)
- apr_brigade_puts(bb, flush, ctx, str)
- apr_brigade_putc(bb, flush, ctx, c)
- apr_brigade_putstrs(bb, flush, ctx, …)
- apr_brigade_vputstrs(bb, flush, ctx, va)
- apr_brigade_printf(bb, flush, ctx, fmt, …)
- apr_brigade_vprintf(bb, flush, ctx, fmt, va)

On the flip side, these functions are essentially all about getting data *into* brigades from other sources. Here, the rule of thumb is to pick the function that best matches your existing data format and results in the least number of calls to apr_brigade_*(). As an example, if you have a bunch of strings you've generated, you could put them into the brigade with multiple calls to apr_brigade_printf() or apr_brigade_write(), but if they're preformatted and you have them all on hand at the same time, apr_brigade_writev() might be faster. The latter is actually a relatively recent addition to the API, added for this very reason.

Bucket operations

- APR_BUCKET_INSERT_BEFORE(a, b)
- APR_BUCKET_INSERT_AFTER(a, b)
- APR_BUCKET_NEXT(b)
- APR_BUCKET_PREV(b)
- APR_BUCKET_INIT(b)

APR_BUCKET_INSERT_BEFORE and _INSERT_AFTER do what they sound like. They insert bucket b before or after bucket a, respectively.

APR_BUCKET_NEXT and _PREV simply return a pointer to the bucket before or after bucket b.

APR_BUCKET_INIT is used internally by bucket creation functions to initialize the prev and next pointers. Unless you're writing your own custom bucket type, you shouldn't need this one.

Bucket operations

- APR_BUCKET_REMOVE
- apr_bucket_destroy
- apr_bucket_delete

The three of these are actually all macros, even though the latter two are named like functions. It's easy to confuse the three, so I wanted to give them extra attention. Keeping them straight is important in avoiding segfaults and memory leaks and other nastiness.

APR_BUCKET_REMOVE() is a macro that unlinks a bucket from its brigade. The previous and next buckets are adjusted to point to each other, and at that point you have a free-floating bucket. The bucket's memory is not freed, so you're free to put the bucket in some other brigade. But whatever you do, don't just drop it on the floor, or you'll have a memory leak.

apr_bucket_destroy() frees the memory of a bucket without making any attempt to cleanup the brigade that that bucket is in.

apr_bucket_delete() is defined as the combination of the two:

```
#define apr_bucket_delete(e) {                  \
    APR_BUCKET_REMOVE(e);                       \
    apr_bucket_destroy(e);                      \
}
```

You have to just choose the one that fits the situation.

Bucket operations

- APR_BUCKET_IS_FLUSH(b)
- APR_BUCKET_IS_EOS(b)
- APR_BUCKET_IS_FILE(b)
- APR_BUCKET_IS_PIPE(b)
- APR_BUCKET_IS_SOCKET(b)
- APR_BUCKET_IS_HEAP(b)
- APR_BUCKET_IS_TRANSIENT(b)
- APR_BUCKET_IS_IMMORTAL(b)
- APR_BUCKET_IS_MMAP(b)
- APR_BUCKET_IS_POOL(b)

- APR_BUCKET_IS_METADATA(b)

Finally, we have macros to tell us what kind of bucket we're looking at.

FLUSH and EOS buckets are special "metadata" buckets that have zero length but are still important; they tell us something about the data in the rest of the brigade. Buckets like these will have a special metadata flag set in their apr_bucket_type_t that can be checked with APR_BUCKET_IS_METADATA(b). An empty bucket is safe to remove from the brigade if and only if it does not contain metadata.

Then we have the "data" bucket types: FILE, PIPE, SOCKET, HEAP, TRANSIENT, IMMORTAL, MMAP, and POOL. Most of them should be self-explanatory except for maybe TRANSIENT and IMMORTAL. A "transient" bucket contains data that is only guaranteed to exist until the current function call returns, such as data in a buffer on the stack. A filter that sees a transient bucket must copy the data somewhere else if it can't pass it right along to the next filter.

These are all the types defined within apr-util, but that's not to say there can't be other types; the way the buckets code is designed, it's easy for 3rd-party code to define custom bucket types. Apache, for example, defines an ERROR metadata bucket type that passes along information about HTTP error codes.

Bucket operations

- apr_bucket_read(e, str, len, block)
- apr_bucket_setaside(e, p)
- apr_bucket_split(e, point)
- apr_bucket_copy(e, c)

These “functions” are really macros that call the type-specific function pointer for each operation. FILE buckets are read by a different function than HEAP buckets, for example, even though both have the same API and return the same result: a pointer to a string of bytes that are the contents of the bucket. Split and copy use reference counting wherever possible to avoid copying the actual data; setaside is used to tell the buckets code that you want the bucket contents to be guaranteed to have a lifetime at least as long as that of pool p.

Bucket operations

- apr_bucket_setaside_noop
- apr_bucket_setaside_notimpl
- apr_bucket_split_notimpl
- apr_bucket_copy_notimpl
- apr_bucket_destroy_noop

There are a set of “placeholder” functions that you can use if you’re writing your own bucket type and some of the functions are either not implemented (meaning it’s an error to call that function on that bucket) or a no-op (meaning the function just doesn’t do anything but it isn’t an error to call it). You’ll notice that not *all* of the five functions have placeholders; that’s because the read() function *must* be implemented by all bucket types. Destroy must also be implemented, though it might not do anything.

In any event, even if a given bucket type doesn’t implement some of the functions, it will have *something* in all five function pointers. That way we never have to check if the pointer is NULL before calling the function. If a bucket chooses not to implement one of the functions, it just uses one of the placeholder functions.

Bucket operations

- apr_bucket_simple_split
- apr_bucket_simple_copy
- apr_bucket_shared_make
- apr_bucket_shared_destroy
- apr_bucket_shared_split
- apr_bucket_shared_copy

In addition to the placeholders that signify “nothing to see here, move on,” there are “default” functions you can use if your bucket type fits a certain mold. “Simple” buckets are non-reference-counted buckets like the TRANSIENT and IMMORTAL buckets. “Shared” buckets are reference-counted buckets like HEAP or POOL buckets. These functions just do the work of manipulating the start and offset values for you and updating the reference count if applicable.

Bucket operations

- The create functions
  - eg: apr_bucket_heap_create(buf, nbyte, freefunc, list)
- The make functions
  - eg: apr_bucket_heap_make(b, buf, nbyte, freefunc)

Each of the bucket types has a “create” function and a “make” function. I won’t list them all individually because that would be a bit redundant. The key point is that the difference between the create functions and the make functions is that create gives you a whole new bucket, and make turns an existing bucket into the requested type. Under the covers, the create functions allocate a bucket and then call make, though the make functions also turn out to be useful in “bucket morphing.”

Case study

- Two sides to the issue:
  - code that *implements* buckets
  - code that *uses* buckets

Our case study at this point will focus on code that *implements* buckets rather than code that *uses* buckets, since the filters presentation just given by Greg Ames and Jeff Trawick contained a nice case study of an example filter that *uses* buckets. Clearly both are important, but in the interest of avoiding too much duplication, I'll refer you to their course notes if you missed the filters talk. As for our look at code that *implements* buckets, it turns out that HEAP buckets are pretty much as straightforward as they get, so let's pick those for our case study.

Case study: HEAP buckets

```
APU_DECLARE_DATA const apr_bucket_type_t apr_bucket_type_heap = {
    "HEAP", 5, APR_BUCKET_DATA,
    heap_bucket_destroy,
    heap_bucket_read,
    apr_bucket_setaside_noop,
    apr_bucket_shared_split,
    apr_bucket_shared_copy
};

APU_DECLARE(apr_bucket *) apr_bucket_heap_create(const char *buf,
                                                 apr_size_t length,
                                                 void (*free_func)(void *data),
                                                 apr_bucket_alloc_t *list)
{
    apr_bucket *b = apr_bucket_alloc(sizeof(*b), list);

    APR_BUCKET_INIT(b);
    b->free = apr_bucket_free;
    b->list = list;
    return apr_bucket_heap_make(b, buf, length, free_func);
}
```

First we see the apr_bucket_type_t that defines the functions that make up the HEAP bucket type. As you can see, heap buckets only need to implement two of the five functions on their own – the "default" functions take care of the rest. The 5 indicates that this bucket has five function pointers… though that's pretty useless information because it's not currently used anywhere at all and I can't even really imagine how it *would* be used. But it's there anyway. APR_BUCKET_DATA is value for the "is_metadata" flag, simply indicating that this is a data bucket rather than a metadata bucket, in which case the value would have been APR_BUCKET_METADATA.

Next we have apr_bucket_heap_create(). This isn't part of the apr_bucket_type_t because each type's create and make functions take different parameter lists. Only *after* the data has been put in a bucket can the code start to ignore the bucket's type and let the buckets code do the rest. The only things that will look unfamiliar about apr_bucket_heap_create() are the things about freelists – apr_bucket_alloc, apr_bucket_free, the apr_bucket_alloc_t *list variable, etc. Basically all of this stuff is just a fancy replacement for malloc() and free(). In fact, originally we did use malloc() and free(), but we found them to be too much of a performance penalty, so we added the bucket allocators. They just keep freelists of chunks of memory cleverly sized to match the allocation patterns of the bucket codes, thus allowing memory allocation to happen at almost no cost.

Case study: HEAP buckets

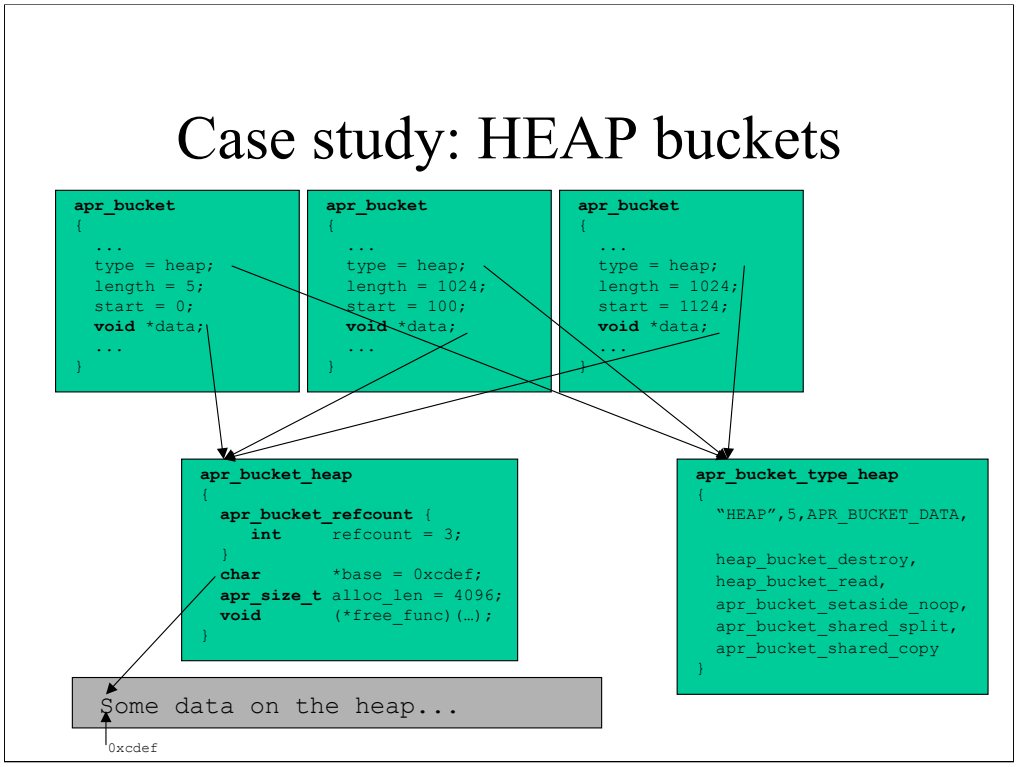
```
static apr_status_t heap_bucket_read(apr_bucket *b, const char **str,
                                     apr_size_t *len,
                                     apr_read_type_e block)
{
    apr_bucket_heap *h = b->data;

    *str = h->base + b->start;
    *len = b->length;
    return APR_SUCCESS;
}
```

A bucket read function has basically one task in life: to do whatever is needed to get the data into memory and return a pointer to it. In the HEAP bucket case, it's already in memory, so all we have to do is account for the starting offset and return our pointer. I suppose it would be useful at this point to know what's in the apr_bucket_heap structure, which is the private data structure for HEAP buckets (the one that hangs off the void *data pointer in the apr_bucket structure).

Case study: HEAP buckets

```
struct apr_bucket_heap {
    /** Number of buckets using this memory */
    apr_bucket_refcount  refcount;
    /** The start of the data actually allocated.  This should never be
     * modified, it is only used to free the bucket.
     */
    char    *base;
    /** how much memory was allocated */
    apr_size_t  alloc_len;
    /** function to use to delete the data */
    void (*free_func)(void *data);
};
```

So here's the apr_bucket_heap structure. Its first element is an apr_bucket_refcount structure which just contains a reference count for the heap data pointed to by this bucket. All "shared" bucket types will have an apr_bucket_refcount member at the beginning of their private data structure. The base pointer points to the block of memory on the heap; alloc_len tells you how big that block of memory is. free_func is a pointer to the function we'll use to free the memory when the last reference goes away. This could be free(), apr_bucket_free(), or even possibly some other allocator's free function.



Just to make sure we're on the same page, let's look at a set of HEAP buckets that all point to the same region of memory. That basically means that we originally had a single HEAP bucket but we've since copied and/or split it a number of times, so now the reference count is >1. As you can see, it's not just the data on the heap that's shared; it's the apr_bucket_heap structure as well. The apr_buckets all point to the same apr_bucket_heap, and basically only differ in that their start and length values are different.

Case study: HEAP buckets

```
APU_DECLARE(apr_bucket *) apr_bucket_heap_make(apr_bucket *b, const char *buf,
                                               apr_size_t length,
                                               void (*free_func)(void *data))
{
    apr_bucket_heap *h = apr_bucket_alloc(sizeof(*h), b->list);

    if (!free_func) {
        h->alloc_len = length;
        h->base = apr_bucket_alloc(h->alloc_len, b->list);
        if (h->base == NULL) {
            apr_bucket_free(h);
            return NULL;
        }
        h->free_func = apr_bucket_free;
        memcpy(h->base, buf, length);
    }
    ...
```

Okay, back to the actual code.

Here we have apr_bucket_heap_make(), which does the bulk of the work for apr_bucket_heap_create and is also used to morph other bucket types into HEAP buckets. On this page, all we're doing is allocating an apr_bucket_heap structure, and then if free_func was NULL, that indicates that the calling code wants us to copy the data they've given us to our own heap buffer. So we allocate a buffer with apr_bucket_alloc(), note that it should later be freed with apr_bucket_free(), and copy the data into it.

Case study: HEAP buckets

```
    ...
    else {
        h->base = (char *) buf;
        h->alloc_len = length;
        h->free_func = free_func;
    }

    b = apr_bucket_shared_make(b, h, 0, length);
    b->type = &apr_bucket_type_heap;

    return b;
}
```

If the free_func is *not* NULL, that means that we can just take whatever buffer was handed to us and use it directly. The caller is giving us the function to use to free the data when all references to it go away. That function might be apr_bucket_free(), it might be plain-old free(), or it might be something else.

After that, we call apr_bucket_shared_make, which essentially just sets the reference count in the apr_bucket_heap (h) to 1 and initializes the start and length parameters of the apr_bucket (b) to whatever we pass in (0 and length, respectively, in this case), and attaches h to b->data.

We set type type field of b, and we're done.

Case study: HEAP buckets

```
static void heap_bucket_destroy(void *data)
{
    apr_bucket_heap *h = data;

    if (apr_bucket_shared_destroy(h)) {
        (*h->free_func)(h->base);
        apr_bucket_free(h);
    }
}
```

Finally, we have the heap_bucket_destroy() function. Most of the reference-counted bucket types have destroy functions that look quite similar to this. apr_bucket_shared_destroy() checks the reference count and decrements it and returns 1 if we're destroying the last reference (0 otherwise). So if what we're destroying is not the last reference, heap_bucket_destroy just decrements the reference count and that's it. If we ARE destroying the last reference, we also free the base pointer with whatever free function was specified when we made the heap bucket and then free the apr_bucket_heap structure as well.

Case study: HEAP buckets

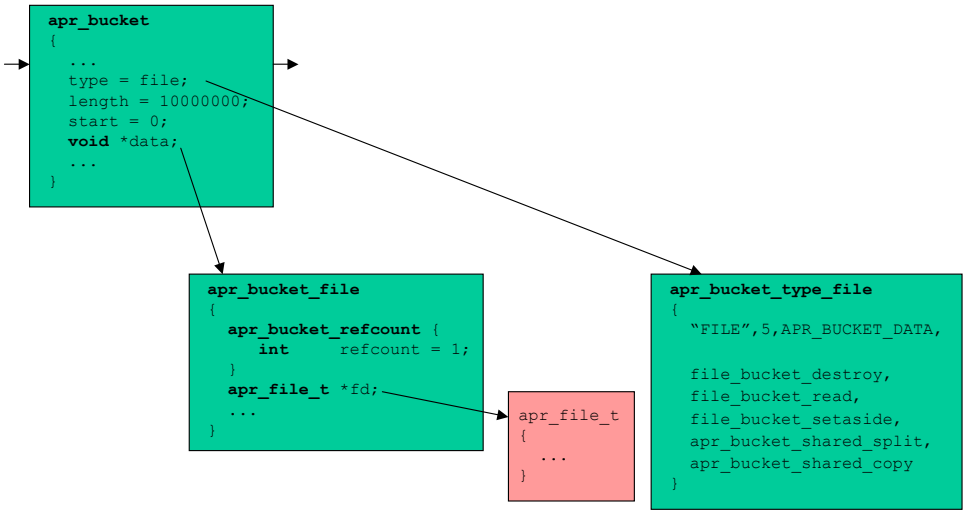
- That’s it!

Stupid bucket tricks: morphing

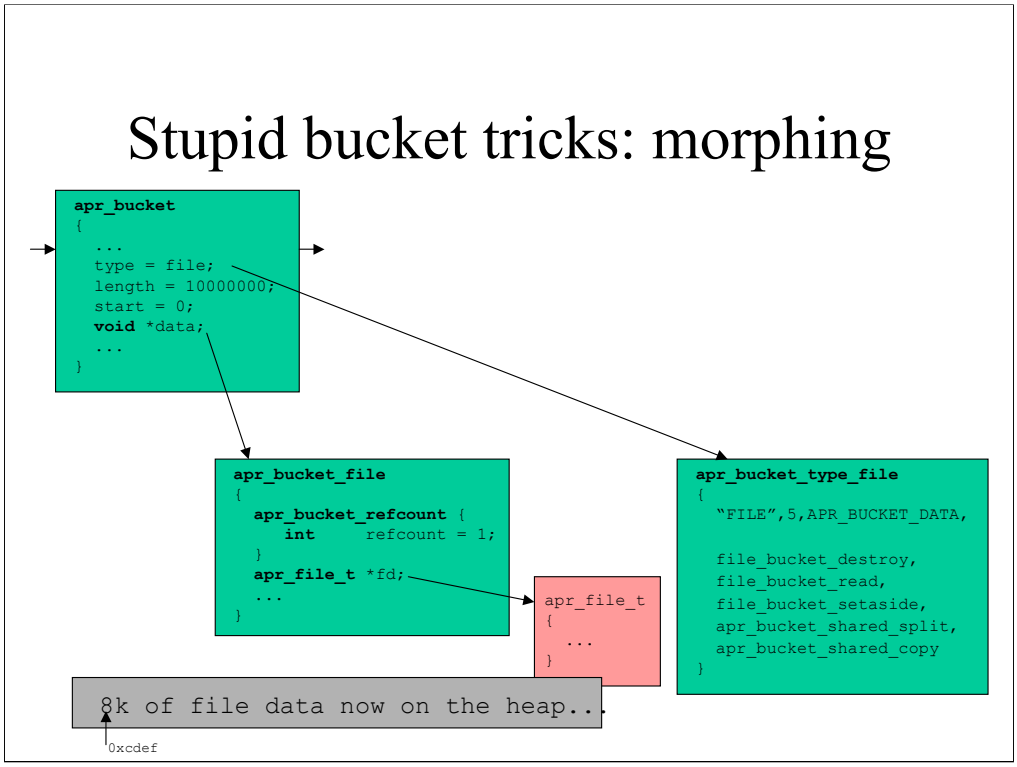
- What happens when you call apr_bucket_read() on a FILE bucket?

As we've seen, apr_bucket_read() always gives us back a pointer to some in-memory data. But in some bucket types, the data isn't in memory. FILE buckets are a perfect example. There we have a bucket that just contains an apr_file_t pointer, which in turn contains a handle to some open file. That's it. So obviously if we call apr_bucket_read() on that FILE bucket, we're going to have to call apr_file_read() or something to get the data out of the file and into memory. Once we've done that, though, the data isn't really in a file anymore, it's in a buffer; let's say that that buffer is on the heap. Well, it doesn't make much sense to keep the bucket as a FILE bucket, because then the FILE bucket code would have to have a bunch of special cases for whether the data has been read yet or not, and so on, or it would have to do a bunch of redundant apr_file_read() calls, which would also be bad. Instead, it makes a lot more sense to just hand the data off to the HEAP buckets code, which already knows how best to deal with data on the heap. But there are a few catches. The function that called apr_bucket_read() will probably have a pointer to the original bucket and will expect that pointer to still be valid once apr_bucket_read() is done, so we can't just destroy the FILE bucket and link a new HEAP bucket into the brigade in its place. Furthermore, the file could be really really big, so we don't want to read the *whole* file into a buffer all at the same time.

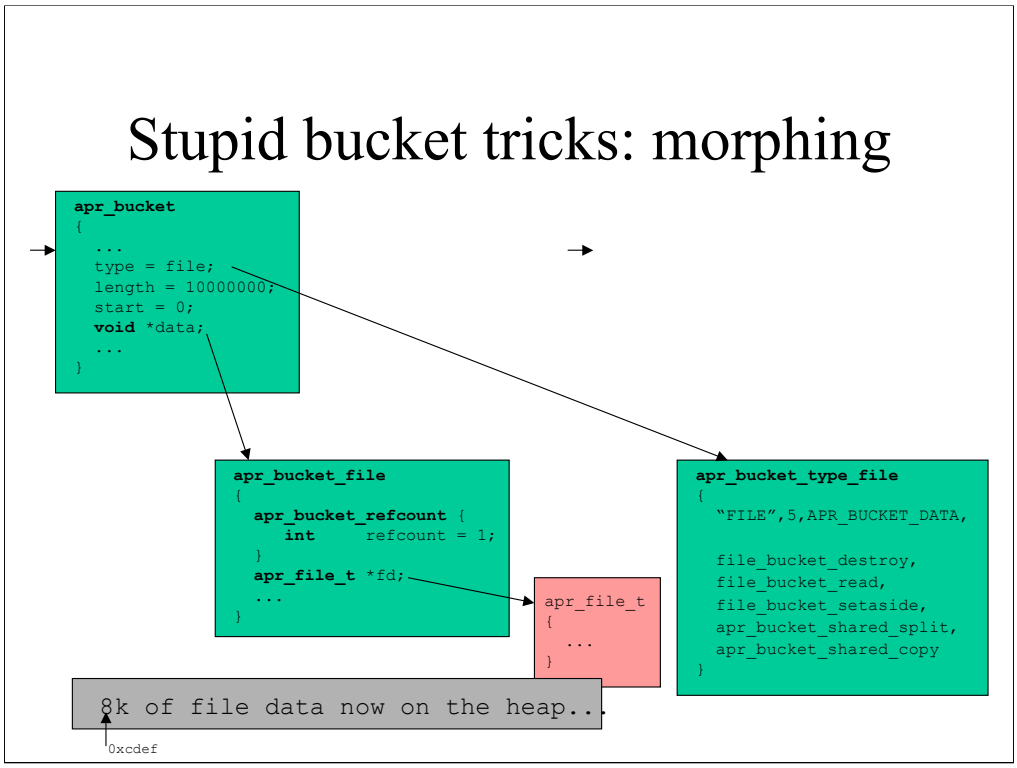
Stupid bucket tricks: morphing



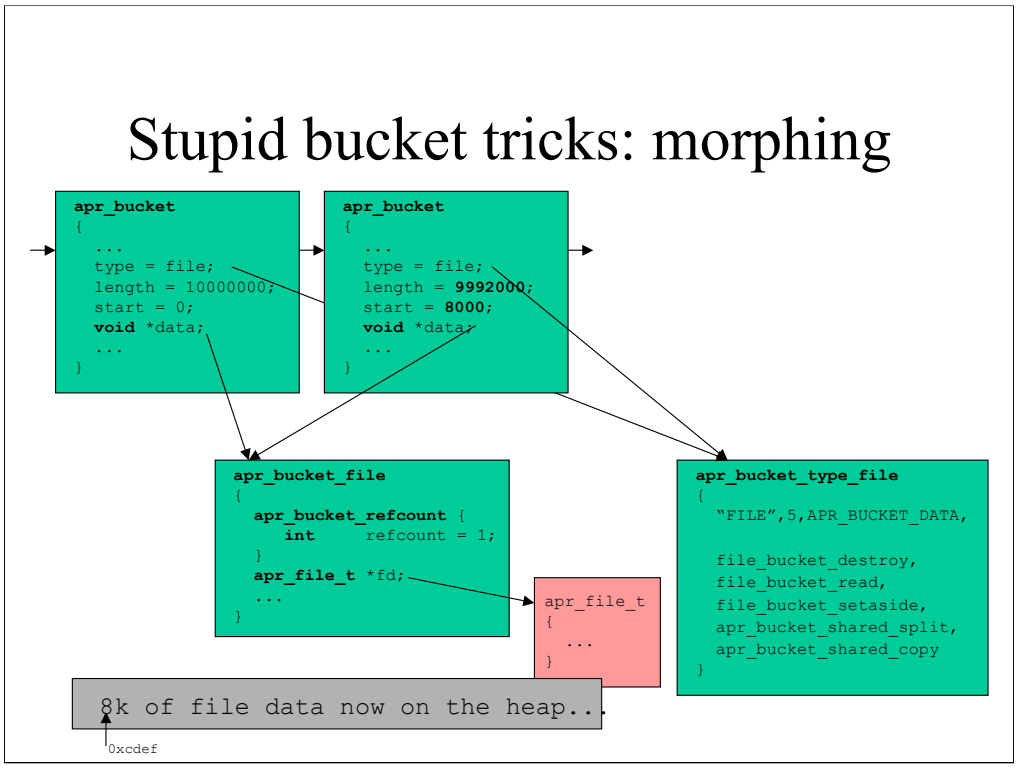
So when we start we have a plain old FILE bucket.



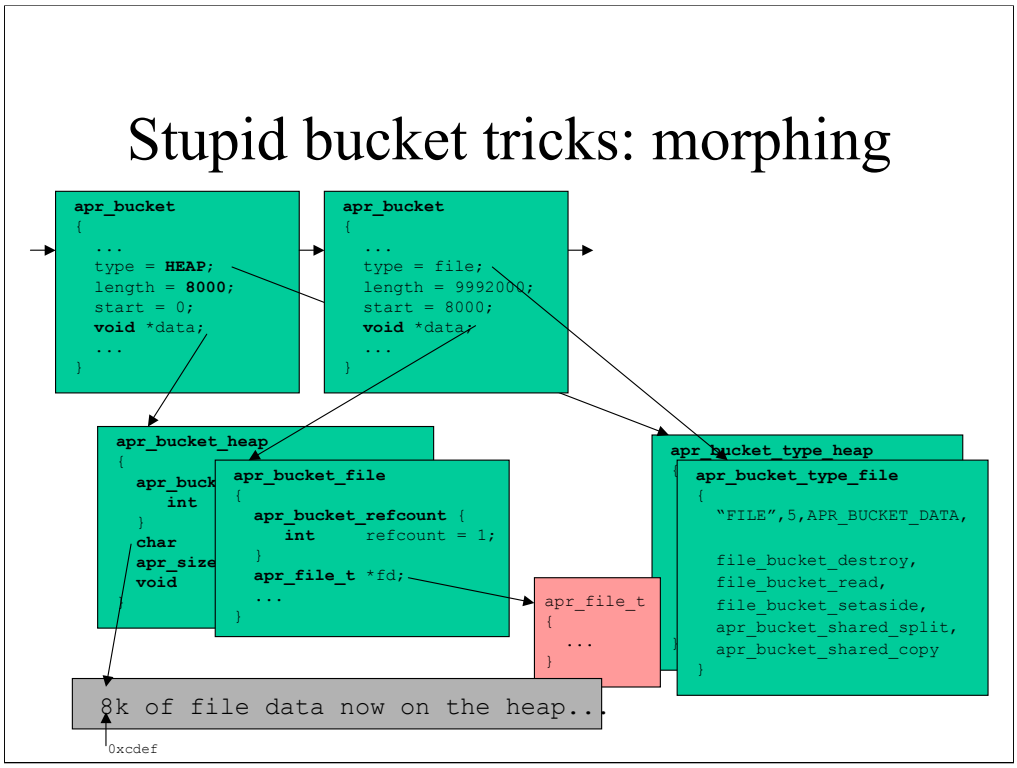
We then read 8k of data from the file and dump it in a buffer on the heap.



But we have to do something with the rest of the data in the file, so we'll insert another FILE bucket after this one in the brigade…

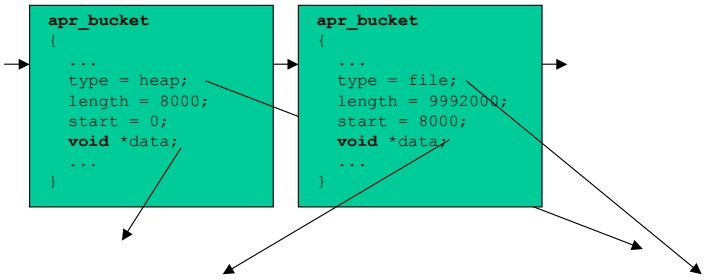


Like so. We adjust the start and length on the new bucket and have it point to the same apr_bucket_file. We take a shortcut and don't update the reference count because it's about to go back down to 1 anyway…



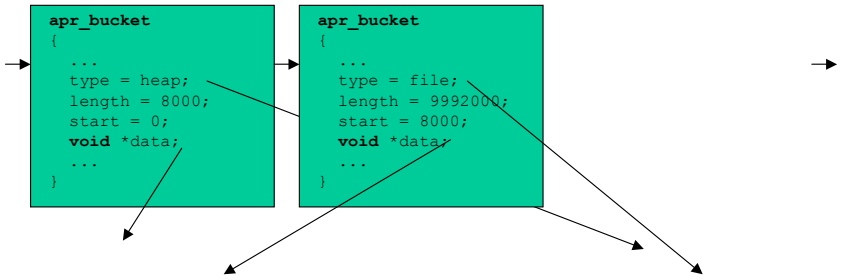
since the next step is to take the first bucket and call apr_bucket_heap_make() on it and the buffer we copied the data into a while back. That creates a new apr_bucket_heap structure pointing to that buffer (with a fixed length value), and the refcount is right once again.

Stupid bucket tricks: morphing



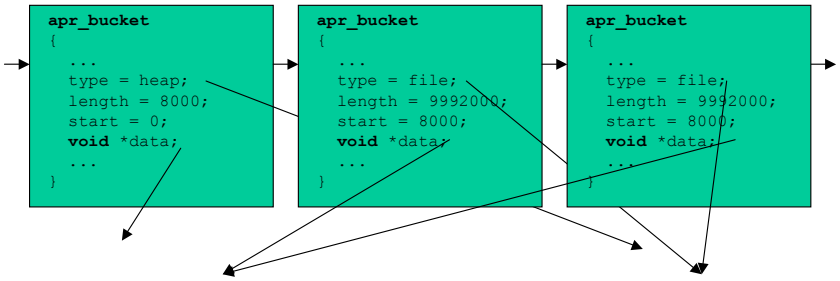
So then let's see what happens when we call apr_bucket_read() the second time (on the "remainder" file bucket). I've chopped out all the clutter at the bottom of the slide, but pretend it's still there.

Stupid bucket tricks: morphing



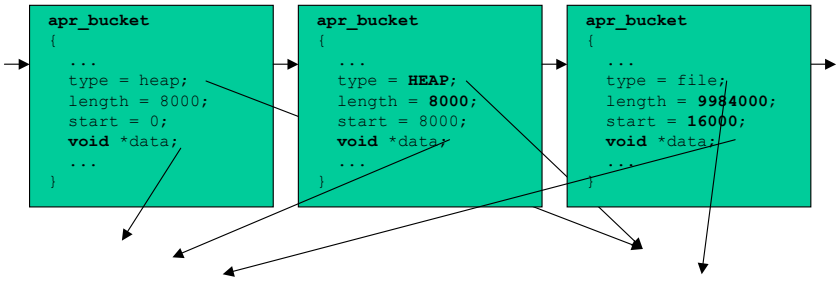
Again we insert a new bucket after…

Stupid bucket tricks: morphing



And set it up as a copy of the one before it.

Stupid bucket tricks: morphing



Again we morph the first one into a heap bucket and adjust all the start and length values. And then we're done!

The big thing to note, though, is that this process alone would still cause us to EVENTUALLY read the entire file into memory all at the same time if we just called apr_bucket_read() in a loop. Sure we'd read it into a bunch of 8kb HEAP buckets rather than one huge HEAP bucket, but it would still be all in memory at the same time. So you have to combine this "stepwise" reading technique with some smarts on the part of the caller, such that the caller only calls read until it has read a certain total number of bytes max (say 8kb). It then passes the buckets it's read along and frees them before moving on to the "remainder" buckets, which it then reads and repeats the process. That way the "remainder" bucket contains the bulk of the data in a not-in-memory form, and we have an upper bound on the amount of memory in use at any one time.

Thanks!

- Contact info:

  Cliff Woolley
  jwoolley@apache.org

  APR Development List
  dev@apr.apache.org

  Website and API documentation
  http://apr.apache.org/