Pablo Zurita 1:54 pm on June 29, 2015

Tags: allocator, Memory allocation

Memory stomp allocator for Unreal Engine 4.

As I have been working on the memory tracking feature one of the issues I had to deal with is memory stomps. They are hard to track even with the debug allocator provided by Unreal Engine 4. I will go over what are the cases to handle and the actual implementation.

Symptoms of a memory stomp.

The symptoms of a memory stomp could be clearly evident as an explicit error about a corrupted heap, or as complex as unexpected behavior without any crash at all which is why they are so hard to catch. This is exacerbated by the fact that any performance-aware allocator won't actually request pages to the OS on every allocation but rather request multiple pages at once and assign addresses as necessary. Only when they run out of available pages will they request new pages to the OS. In that situation the OS won't be able to do anything to let us know that we messed up (by for example, throwing an access violation exception). Instead execution continues as usual but behavior isn't what is expected since we could be effectively be operating with memory that is unrelated to what we want to read or write. Just as an example, I had to deal with the case where some CPU particles were behaving in a way that they would end up at origin and color change every now and then. After looking for a logic error, I was able to determine that the issue had nothing to do with the code changing the color or transform but rather a memory overrun on unrelated code which ended up in the same pool in the binned allocator. Another example is when pointers get overwritten. If you don't see that the pointer itself was written somewhere else, rather than having the data that is being pointed to corrupt, you may waste some time. Depending on the nature of the code accessing the data pointer it may or may not crash. Still, the symptom would be similar to that of overwritten data.

Types of memory stomps.

A memory stomp could be defined as doing different type of operations with memory that are invalid. All these invalid operations are hard to catch due to their nature. They are:

- **Memory overrun.** Reading or writing off the end of an allocation.

```
static const size_t NumBytes = 1U << 6U;
uint8_t * const TestBytes = new uint8_t[NumBytes];
// Memory overrun:
static const size_t SomeVariable = 7U;
TestBytes[1U << SomeVariable] = 1U;
delete [] TestBytes;
```

- **Memory underrun.** Reading or writing off the beginning of an allocation.

```
static const size_t NumBytes = 1U << 6U;
uint8_t * const TestBytes = new uint8_t[NumBytes];
// Memory underrun:
TestBytes[-128] = 1U;
delete [] TestBytes;
```

- **Operation after free.** Reading or writing an allocation that was already free.

```
static const size_t NumBytes = 1U << 6U;
uint8_t * const TestBytes = new uint8_t[NumBytes];
delete [] TestBytes;
// Operation after free:
TestBytes[0] = 1U;
```

One confusion that does raise up is if dereferencing a null pointer could be considered a memory stomp. Given that the behavior when dereferencing null is undefined, it wouldn't be possible to say that it is effectively a memory stomp. So to keep things simple, I rely on the target-platform OS to deal with that case.

How it works.

The stomp allocator work by using the memory protection features in the different operating systems. They allow us to tag different memory pages (which are usually 4KiB each) with different access modes. The access modes are OS-dependent but they all offer four basic protection modes: execute, read, write, and no-access. Based on that the stomp allocator allocates at least two pages, one page where the actual allocation lives, and an extra page that will be used to detect the memory stomp. The allocation requested is pushed to the end of the last valid page in such way that any read or write beyond that point would cause a segmentation fault since you would be trying to access a protected page. Since people say that a picture is worth a thousand words, here is a reference diagram:

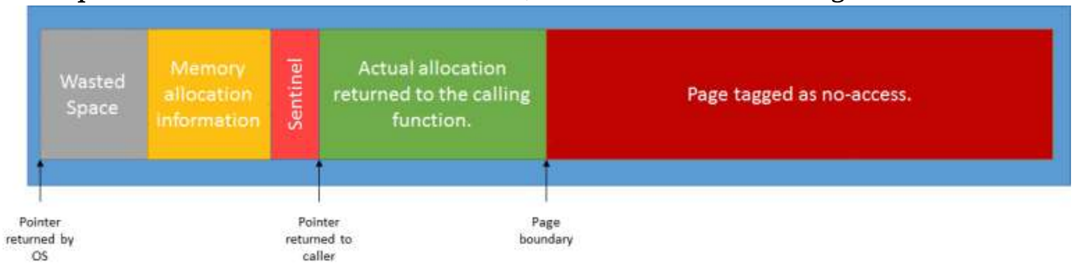
Wasted Space
Memory allocation information
Sentinel
Actual allocation returned to the calling function.
Page tagged as no-access.
Pointer returned by OS
Pointer returned to caller
Page boundary

Diagram not to scale. The memory allocation information + the sentinel only accounts for 0.39% of the two pages shown.

As it is visible, there is a waste of space due to the fact that we have to deal with full pages. That is the price we have to pay. But the waste is limited to a full page plus the waste in the space of the first valid page. So the stomp allocator isn't something that you would have enabled by default, but it is a tool to help you find those hard to catch issues. Another nice benefit is that this approach should work fine on many platforms. As a pull request to Epic I'm providing the implementation for Windows, Xbox One, Linux and Mac. But it can be implemented on PlayStation 4 (details under NDA) and perhaps even on mobile platforms as long as they provide functionality similar to what's provided with mprotect or VirtualProtect. Another aspect for safety is the use of a sentinel to detect an underrun. The sentinel is the last member of the AllocationData which is shown as "Memory allocation information" which is necessary to be able to deallocate the full allocation on free, and to see how much information to copy on Realloc. When an allocation is freed then the sentinel is checked to see if it is the value expected (which in my code it is 0xdeadbeef or 0xdeadbeefdeadbeef depending if it is a 32-bit or 64-bit build). If the sentinel doesn't match the expected value then there was an underrun detected and the debugger will be stopped. But that will only work for very specific cases where the sentinel is overwritten. So changing the layout with a different mode would help with this issue. Here is the diagram:

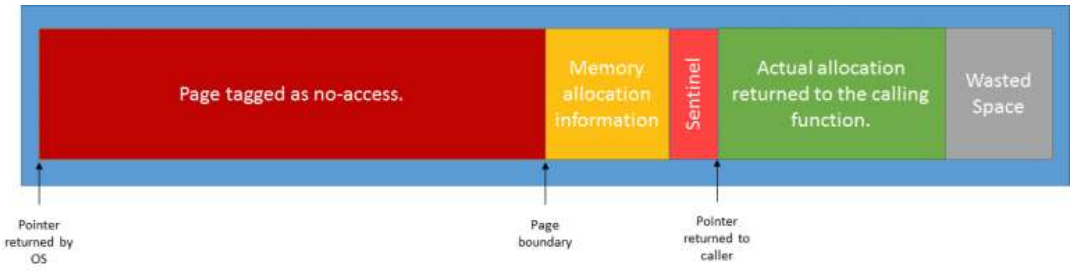
Page tagged as no-access.
Memory allocation information
Sentinel
Actual allocation returned to the calling function.
Wasted Space
Pointer returned by OS
Page boundary
Pointer returned to caller

Diagram not to scale.

This basically flips where the no-access page is set. This allows finding underruns that happen as long as it writes before the "Memory allocation information" shown in the diagram. That piece of data is small (the size depends on the architecture used to compile). The sentinel is still present to deal with small underrun errors. The underrun errors that manage to go below that point will actually hit the no-access page which will trigger the response we want from the OS. The last aspect is that any attempt to read or write from memory already free will fail as it happens. A performance-aware allocator (such as the binned allocator) would add those pages to a free list and return them when someone else request memory without actually making a request to the OS for new pages. That is one of the reasons why they are fast. The stomp allocator will actually decommit the freed pages which makes any read or write operation in those pages invalid.

Do you have any idea how much more memory this causes the engine to use than normal? (with the extra info per alloc) I recall having trouble when enabling page guards on 32-bit machines with projects that already were close to the OS heap limit. Maybe just run on lower quality/shut down unnecessary subsystems? (that's what I did, of course making sure you can still repro the issue afterwards is the key)

By the way – this article describes the solution very well, thanks for sharing!

That depends on the number of allocations done rather than anything else. Multiple small allocations are worse than few big ones since the cost is almost fixed per allocation. The worst case scenario is 2 pages per allocation in the case that an allocation takes exactly a multiple of the page size. Given some of the memory tracking stats I got loading the

Elements demo, the increase would be around 2.5GBs given that there are 605016 allocations alive. And that's including all the subsystems.

Ah, that makes sense. Thank you!

Hey there. This looks really useful. One thing though, in the case of the underrun detection, why not put the allocation information into the page prior to the data returned to the caller, and then change the page access rights? You could get rid of the sentinel too as it's not required, the page protection is the sentinel.

That's a very good question and the truth is that everything comes down to tradeoffs.

If you don't care about catching read-access errors then what you mentioned works just fine. From my point of view it is better to have the read-access protection even though it will not catch the reads happening on the 32 bytes (on 64-bit builds) or 16 bytes (on 32-bit builds) just before the pointer returned. If read underruns are no concern then this is definitely the way to go.

Another possible solution is to store the pointer returned by the OS somewhere else so that it is possible to change the page protection to access the allocation data when needed. That would be needed on Realloc() and Free(). I don't like that solution much because it spreads data around, far away from the actual allocation, and pollutes the data cache. Another issue is performance which I have seen degrade greatly (at the very least on OSX) when repeatedly changing page protection modes. UnrealEd becomes unusable when you try to do something like that. But depending on your situation it might be valid.

Sorry for not getting back sooner! What you say makes a lot of sense – I wasn't thinking of out of bound reads particularly, only writes, because that's what tends to bite us hardest. Very good point! 🙂

So the name "stomp allocator" is a misonomer. It's an anti-stomp allocator 🙂

For your second layout to find underruns, couldn't you put the alloc info to the end of the page? That way you can catch any underruns? Like that:

[protected_page][actual_allocation | sentinel | allocation_info]

Regards Tarantula

← Streaming on Twitch.

Adding memory tracking features to Unreal Engine 4. →