Dumping a hash table with external chaining from the debugger

devblogs.microsoft.com/oldnewthing/20120824-00

August 24, 2012

Raymond Chen

I was doing some crash dump debugging, as I am often called upon to do, and one of the data structure I had to grovel through was something that operated basically like an atom table, so that's what I'll call it. Like an atom table, it manages a collection of strings. You can add a string to the table (getting a unique value back, which we will call an atom), and later you can hand it the atom and it will give you the string back. It looked something like this:

```
struct ENTRY
{
  ENTRY *next;
  UINT   atom;
  PCWSTR value;
};
#define ATOMTABLESIZE 19
struct ATOMTABLE
{
  ENTRY *buckets[ATOMTABLESIZE];
};
```

(It didn't actually look like this; I've reduced it to the smallest example that still illustrates my point.)

As part of my debugging, I had an atom and needed to look it up in the table. A program would do this by simply calling the "here is an atom, please give me the string" function, but since this was a crash dump, there's nothing around to execute anything. (Not that having a live machine would've made things much easier, because this was a kernel-mode crash, so you don't get any of this edit-and-continue froofy stuff. This is *real debugging*™.)

But even though the crashed system can't execute anything, the *debugger* can execute stuff, and the debugger engine used by `kd` comes with its own mini-programming language. Here's how I dumped the atom table:

```
// note: this was entered all on one line
// broken into two lines for readability
0: kd&gt .for (r $t0=0; @$t0 < 0n19; r $t0=@$t0+1)
         { dl poi (fffff8a0`06b69930+@$t0*8) 99 2 }
fffff8a0`06ad3b90  fffff8a0`037a3fc0 fffff8a0`0000000c \
fffff8a0`037a3fc0  fffff8a0`037a4ae0 00000000`00000025 | $t0=0
fffff8a0`037a4ae0  fffff8a0`02257580 00000000`00000036 |
fffff8a0`02257580  00000000`00000000 00000000`00000056 /
fffff8a0`06ad3b30  fffff8a0`06ad3ad0 a9e8a9d8`0000000d \
fffff8a0`06ad3ad0  fffff8a0`037a4700 000007fc`0000000e |
fffff8a0`037a4700  fffff8a0`01f96fb0 00000000`0000003f | $t0=1
fffff8a0`01f96fb0  fffff8a0`06cfa5d0 fffff8a0`00000044 |
fffff8a0`06cfa5d0  00000000`00000000 00181000`00000060 /
fffff8a0`033e7a70  fffff8a0`037a4770 00000020`00000023 \
fffff8a0`037a4770  fffff8a0`023b52f0 00000000`0000003e | $t0=2
fffff8a0`023b52f0  fffff8a0`03b6e020 006f0063`00000059 |
fffff8a0`03b6e020  00000000`00000000 00000000`00000075 /
fffff8a0`037a0670  fffff8a0`02b08870 fffff8a0`00000026 \ $t0=3
fffff8a0`03b9e390  00000000`00000000 00240000`00000071 /
...
```

Let's take that weirdo command apart one piece at a time.

The overall command is

```
.for (a; b; c) { d }
```

This operates the same as the C programming language. (Sorry, Delphi programmers, for yet another C-centric example.) In our case, we use the `$t0` pseudo-register as our loop control.

- `r $t0=0` — this sets `$t0` to zero
- `@$t0 < 0n19` — this stops once `$t0` reaches 19.
- `r $t0=@$t0+1` — this increments `$t0`.

Note that here as well as in the loop body, I prefix the register name with the `@` character when I want to obtain its value, in order to force it to be interpreted as a register. (Otherwise, the debugger will look for a symbol called `$t0`.)

The command being executed at each iteration is `{ dl poi (fffff8a0`06b69930+@$t0*8) 99 2 }`. Let's break this down, too:

- `dl` — this command dumps a singly-linked list.
- `poi (fffff8a0`06b69930+@$t0*8)` — this is the head of the linked list. In this example, `0xfffff8a0`06b69930` is the address of the `buckets` array, so we add the loop counter times the size of a pointer (8, in this case) to get the address of the `$t0`'th entry, and then dereference it (`poi`) to get the address of the head of the linked list.

- `99` — This is the maximum length of the linked list. I picked an arbitrary large-enough number. I like using 9's because it carries the most value per keypress. Other people like to use nice round numbers like `1000`, but `999` saves you a whole keypress and is just one less. (On the other hand, I don't use `fff` because that runs the risk of being misinterpreted as a symbol.)
- `2` — This is the number of pointer-sized objects to dump at the start of each node. For 32-bit code, I use 4, whereas for 64-bit code, I use 2. Why those values? Because those are the maximum number of pointer-sized objects that the debugger will print on one line.

Okay, so now I have that linked list dump. The value I'm looking for is the `atom` whose value is `0x3F`, so I skim down the last column looking for `0000003f`, and hey there it is.

```
fffff8a0`037a4700  fffff8a0`01f96fb0 00000000`0000003f
```

Now I have my `ENTRY`, and I can dump it to see what the corresponding value is.

```
0: kd> dt contoso!ENTRY fffff8a0`037a4700
    +0x000 next: 0xfffff8a0`01f96fb0
    +0x008 atom: 0x0000003f
    +0x010 value: 0xffff8a0`01f97e20 -> "foo"
```

Raymond Chen

Follow