Subject: Re: IDL Memory Leaks
Posted by John-David T. Smith on Mon, 05 Nov 2001 22:22:19 GMT
View Forum Message <> Reply to Message

David Fanning wrote:
>
> Myron Brown (brownmz1@jhuapl.edu) writes:
>
>> 4. The code includes repetitive reassignments of variables to large arrays,
>> even within a routine (still in scope). I tried to assign variables to zero
>> each time, but it doesn't seem to make a difference. Should this really be
>> necessary?
>
> I'd like to see how you are doing this (I hope you
> are using the TEMPORARY function!), as this is more
> than likely the source of your problem. You don't say
> how you know you are "leaking memory", but this is the
> kind of thing that will eat up process memory if you
> are not careful.
>
> If you have the same variable on both sides of an equal
> sign, you really need something like this:
>
> image = Temporary(image) * 5
>
> And when you are done with it:
>
> Undefine, image
>
> (The UNDEFINE program can be found on my web page.)
>
> Otherwise, you continually get more and more process
> memory (with malloc), and there is no way to give it
> back.
>
> "Is it necessary?". Yes, it is very, very necessary. :-)
>

While I agree that managing memory leaks can be challenging, I'm not sure these Draconian measures are indicated. Indeed:

image = image * 5

does waste some memory: exactly as much as "image" occupies, since it is made into a temporary variable on the right, then modified, and copied to the variable of the same size on the left. As soon as the temporary variable goes out of scope on the right (i.e. after this call), it's memory is returned. It is true that IDL doesn't give back to the system

memory it has allocated. It *does*, however, recycle the memory it already has, and only ask for more memory if it really, really needs it.

Consider:

```
pro mem_assign_leak
  print,'   ===>Startup:'
  help,/memory
  a=dist(1024)
  print,'   ===>First allocation'
  help,/memory
  print,'   ===>Assignment'
  for i=0,4 do begin
     a=a*5
     help,/memory
  endfor
end

IDL> mem_assign_leak
   ===>Startup:
heap memory used:     403269, max:    4597588, gets:     1575,
frees:     1201
   ===>First allocation
heap memory used:    4597645, max:    4610149, gets:     2091,
frees:     1716
   ===>Assignment
heap memory used:    4597645, max:    8792021, gets:     2092,
frees:     1717
heap memory used:    4597645, max:    8792021, gets:     2093,
frees:     1718
heap memory used:    4597645, max:    8792021, gets:     2094,
frees:     1719
heap memory used:    4597645, max:    8792021, gets:     2095,
frees:     1720
heap memory used:    4597645, max:    8792021, gets:     2096,
frees:     1721
```

Here we see around 4MB allocated for the large array (the 400KB or so just represents IDL startup overheads). Notice that on each subsequent iteration, no additional memory is on the heap at all. Also notice that *max* jumps after the first iteration to 8MB... exactly the amount required for that temporary copy of "a" which was made on the right hand side, and discarded. This 8MB is enough to hold the entire calculation, which is then performed in subsequent iterations in the same memory IDL had already allocated from the system. After the first iteration, no new memory is requested.

Now try it from a fresh session with "a=temporary(a)*5" instead:

```
IDL> mem_assign_leak
  ===>Startup:
heap memory used:     400431, max:     420977, gets:      901,
frees:      653
  ===>First allocation
heap memory used:    4595719, max:    4608223, gets:     1424,
frees:     1172
  ===>Assignment
heap memory used:    4595719, max:    4595719, gets:     1424,
frees:     1172
heap memory used:    4595719, max:    4595719, gets:     1424,
frees:     1172
heap memory used:    4595719, max:    4595719, gets:     1424,
frees:     1172
heap memory used:    4595719, max:    4595719, gets:     1424,
frees:     1172
heap memory used:    4595719, max:    4595719, gets:     1424,
frees:     1172
```

Here we see the same usage (more or less), with the difference that even the *max* usage remained at 4MB, since no temporary copy of "a" was made in this case (it's simply modified "in place"). There's one more difference. Notice the "gets" and "frees". In this case, after the first allocation, no more memory is "gotten" or "freed". In the former case, everytime through the loop, both "gets" and "frees" are incremented by exactly one: the memory required for that temporary array is gotten and then freed, leaving the max memory constant: i.e. no new system memory is obtained, despite 4 more "gets" and "frees" of memory.

So, with temporary, you can at most halve the max usage (aka system memory being eaten), but you can't correct for an error which grabs and hangs onto more memory within each new iteration. That's a memory leak, and it can be your fault (most common), or a subtle bug in an IDL builtin; hopefully you can find out which.

JD
