User Interface - Bug #2662

memory leak in the native NCURSES client

08/25/2015 10:16 AM - Greg Shah

Status:	Closed	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:		**% Done:**	100%
Category:		**Estimated time:**	0.00 hour
Target version:	Cleanup and Stablization for Server Features		
billable:	No	**case_num:**	
vendor_id:	GCD		

Description

Related issues:

Related to Base Language - Bug #2657: appserver agent context reset does not ...	**Closed**

History

#1 - 08/25/2015 10:18 AM - Greg Shah

Initial identification of the problem was from this report by a customer:

```
I've been watching memory usage (just using top) whilst running the full search tests.

I'm seeing overall usage of just under 19gb - so there is no swapping at all with my 20gb allocation. Remember
 that my environment is much more sensitive to swapping than yours (1gb swap partition only; virtualisation).

The server jvm process is showing a virtual memory usage of about 14.3g.
Some of the client jvm processes are showing approx 1.6g of virtual memory usage (if I'm doing my sums right):
1638316 / 1024 / 1024 = 1.56gb.
(top reports '1638316' - which should be in kb).

So, one question is, how do we estimate client jvm process memory utilisation? (and why is there such a large
difference between virtual and resident memory for the client processes - is there some use of off-heap memory
 usage, perhaps in the native code?)

Example recent top output:

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 5947 vagrant   20   0 14.323g 0.010t  53592 S 148.3 54.1 577:46.99 java
 5927 vagrant   20   0 4926096 999.0m  22444 S   2.7  5.0   6:04.41 java
 6872 vagrant   20   0 1703852 114152  17476 S   0.3  0.6   0:27.53 java
 7244 vagrant   20   0 1638324  90824  17492 S   0.3  0.4   0:21.45 java
 7403 vagrant   20   0 1638304  74332  17432 S   0.3  0.4   0:13.93 java
 7509 vagrant   20   0 1638304 103288  17464 S   0.3  0.5   0:24.43 java
 7657 vagrant   20   0 1638312 106436  17472 S   0.3  0.5   0:23.08 java
 7710 vagrant   20   0 1638328  87112  17496 S   0.3  0.4   0:19.99 java
12636 vagrant   20   0   23652   1620   1148 R   0.3  0.0   0:00.07 top
23298 vagrant   20   0 1638320  90956  17680 S   0.3  0.4   0:19.06 java
23475 vagrant   20   0 1638320  98068  17680 S   0.3  0.5   0:20.92 java
```

#2 - 08/25/2015 10:19 AM - Greg Shah

This link is useful in understanding things:

http://stackoverflow.com/questions/561245/virtual-memory-usage-from-java-under-linux-too-much-memory-used

#3 - 08/25/2015 10:20 AM - Greg Shah

-------- Forwarded Message --------
Subject: Re: memory usage
Date: Thu, 20 Aug 2015 08:46:21 -0400
From: Greg Shah <ges@goldencode.com>
Reply-To: ges@goldencode.com
To: Eric Faulhaber <ecf@goldencode.com>, ...

The stackoverflow link is a pretty good description of what is going on.

1. The virtual memory of the process is what is **allocated** at the OS level. Each allocation is made in one or more pages, the size of which are dependent upon the OS configuration and "bitness". In older 32-bit environments, each page was always 4k. In newer 64-bit environments and 32-bit PAE environments, the page size can be bigger. Your system looks like it is using 4k pages (every memory entry is a multiple of 4). These pages are allocated in large chunks based on what may ever be needed. When such memory is allocated, it is just marked as valid for access in the page tables. There is not any physical memory involved until the addresses are actually used. For example, when actual data for a shared library is loaded into memory from disk, there are real physical pages (4k) of memory that are now dedicated to the process. That is the resident memory for the process, and it really does matter. When the collective resident memory for all processes on the system passes a threshold as defined by configuration in the OS, the kernel will start to swap pages to disk. Until that point, allocated pages will keep getting used over time, turning into resident memory. Once a steady state has been reached (if there is one for that code), that is the best way to gauge the application's average memory usage. Peak resident memory usage may also be an important stat, if enough of the cumulative peak usage of all apps can ever be reached simultaneously such that the system would be forced to swap.

2. The heap is just one of many allocations in the JVM. It often happens to be the largest single allocation. But having gigabytes of additional allocations is perfectly reasonable, especially if there are things going on like off-heap caching of database records. The normal JVM footprint (resident memory) of shared libraries, memory mapped jars, JVM internal memory and so forth should be measured in megabytes. On 64-bit JVMs this will be larger than for 32-bit JVMs. On 32-bit JVMs we can get away with 16MB or 32MB heaps + the JVM footprint which is usually in the 8-12MB range, if I remember correctly. I haven't examined the average footprint of the 64-bit JVMs. What is the heap size you are specifying for the client JVM processes? Considering that the resident memory is being reported as 90-100MB for most of your java processes, I expect that the used portion of your heap is probably at least 64MB. The rest could easily be in use as the JVM footprint. It does seem a bit large and I expect that the heap may be oversized.

...

Thanks,
Greg

#4 - 08/25/2015 10:22 AM - Greg Shah

-------- Forwarded Message --------
Subject: Re: re-running full-search tests.
Date: Tue, 25 Aug 2015 12:57:55 +0300
From: Constantin Asofiei <ca@goldencode.com>
Reply-To: ca@goldencode.com
To: Eric Faulhaber <ecf@goldencode.com>
CC: Greg Shah (GC) <ges@goldencode.com>

Greg/Eric,

I've checked a simple test (just a "DISPLAY i" in a very long loop) and
the RES memory raises constantly:

- without -Xmx32m it reaches ~300MB when the loop ends after 1 million iterations.
- with -Xmx32m the RES mem raises at a lower pace, and it finishes with ~100MB.

In pmap, the largest blocks are anon blocks.

Thanks,
Constantin

#5 - 08/25/2015 10:22 AM - Greg Shah

-------- Forwarded Message --------
Subject: Re: re-running full-search tests.
Date: Tue, 25 Aug 2015 07:53:44 -0400
From: Greg Shah <ges@goldencode.com>
Reply-To: ges@goldencode.com
To: ca@goldencode.com, Eric Faulhaber <ecf@goldencode.com>

Constantin,

Interesting finding! I took a quick look at the terminal.c and terminal_linux.c which are where we implement our NCURSES native methods. The only memory leak I see there is when we throw an exception. In each thrown exception we malloc for the error text and never free.

But it seems unlikely that we are throwing millions of exceptions at that layer. It would need to be in that range to create the size of leak that we have.

It is possible that there is some nuance of how we are accessing the NCURSES functions such that they require some cleanup that we are not doing. Or it may be a leak in NCURSES.

Can you think of a different client-side off-heap memory use that exists during DISPLAY?

Thanks,
Greg

#6 - 08/25/2015 10:22 AM - Greg Shah

-------- Forwarded Message --------
Subject: Re: re-running full-search tests.
Date: Tue, 25 Aug 2015 08:35:14 -0400
From: Greg Shah <ges@goldencode.com>
Reply-To: ges@goldencode.com
To: ca@goldencode.com, Eric Faulhaber <ecf@goldencode.com>

Constantin,

Please see this:

http://www.gnu.org/software/libc/manual/html_node/Allocation-Debugging.html

Use of the glibc memory allocation tracing function mtrace() may very well give us the answer we need. Could you give it a quick try to see what the output is?

Thanks,
Greg

#7 - 08/25/2015 10:44 AM - Constantin Asofiei

After using mtrace() function, it reports lots of still allocated addresses. Unfortunately I haven't found a way to specify the program name to mtrace tool, but I managed to extract the allocators of these addresses:

```
      1 @ /lib64/ld-linux-x86-64.so.2:[0x7fdc86cc759d]
      8 @ /lib64/ld-linux-x86-64.so.2:[0x7fdc86ccae37]
      8 @ /lib64/ld-linux-x86-64.so.2:[0x7fdc86ccb140]
     68 @ /lib64/ld-linux-x86-64.so.2:[0x7fdc86ccf220]
      8 @ /lib64/ld-linux-x86-64.so.2:[0x7fdc86cd0d2e]
      1 @ /lib64/ld-linux-x86-64.so.2:[0x7fdc86cd49f7]
     10 @ /lib64/ld-linux-x86-64.so.2:[0x7fdc86cd9d4a]
      5 @ /lib64/ld-linux-x86-64.so.2:(_dl_allocate_tls
    105 @ /lib/x86_64-linux-gnu/libc.so.6:(qsort_r
      5 @ /lib/x86_64-linux-gnu/libc.so.6:(__strdup
      6 @ /lib/x86_64-linux-gnu/libglib-2.0.so.0:[0x7fdc73973c28]
      1 @ /lib/x86_64-linux-gnu/libglib-2.0.so.0:[0x7fdc7399be7b]
     28 @ /lib/x86_64-linux-gnu/libglib-2.0.so.0:(g_malloc
    161 @ /lib/x86_64-linux-gnu/libglib-2.0.so.0:(g_malloc0
      1 @ /lib/x86_64-linux-gnu/libglib-2.0.so.0:(g_private_get
    110 @ /lib/x86_64-linux-gnu/libglib-2.0.so.0:(g_realloc
    158 @ /lib/x86_64-linux-gnu/libncurses.so.5:(_nc_add_to_try
      4 @ /lib/x86_64-linux-gnu/libncurses.so.5:(_nc_doalloc
      2 @ /lib/x86_64-linux-gnu/libncurses.so.5:(_nc_first_db
      3 @ /lib/x86_64-linux-gnu/libncurses.so.5:(_nc_hash_map
      1 @ /lib/x86_64-linux-gnu/libncurses.so.5:(_nc_home_terminfo
      6 @ /lib/x86_64-linux-gnu/libncurses.so.5:(_nc_makenew
      4 @ /lib/x86_64-linux-gnu/libncurses.so.5:(_nc_read_termtype
      1 @ /lib/x86_64-linux-gnu/libncurses.so.5:(_nc_scroll_optimize
      5 @ /lib/x86_64-linux-gnu/libncurses.so.5:(_nc_setupscreen
      1 @ /lib/x86_64-linux-gnu/libncurses.so.5:(_nc_setupterm
    198 @ /lib/x86_64-linux-gnu/libncurses.so.5:(newwin
      2 @ /lib/x86_64-linux-gnu/libncurses.so.5:(start_color
      1 @ /usr/lib/jvm/java-7-openjdk-amd64/jre/lib/amd64/libnet.so:[0x7fdc78315705]
      1 @ /usr/lib/jvm/java-7-openjdk-amd64/jre/lib/amd64/libnet.so:[0x7fdc7831ae95]
      1 @ /usr/lib/jvm/java-7-openjdk-amd64/jre/lib/amd64/libnet.so:[0x7fdc7831b615]
```

```
      1 @ /usr/lib/jvm/java-7-openjdk-amd64/jre/lib/amd64/libzip.so:[0x7fdc7f5bf5d2]
      1 @ /usr/lib/jvm/java-7-openjdk-amd64/jre/lib/amd64/libzip.so:[0x7fdc7f5bf604]
      1 @ /usr/lib/jvm/java-7-openjdk-amd64/jre/lib/amd64/libzip.so:[0x7fdc7f5bfc5f]
      2 @ /usr/lib/jvm/java-7-openjdk-amd64/jre/lib/amd64/libzip.so:[0x7fdc7f5bfc90]
    406 @ /usr/lib/jvm/java-7-openjdk-amd64/jre/lib/amd64/libzip.so:[0x7fdc7f5bff64]
    406 @ /usr/lib/jvm/java-7-openjdk-amd64/jre/lib/amd64/libzip.so:[0x7fdc7f5c00e9]
      1 @ /usr/lib/jvm/java-7-openjdk-amd64/jre/lib/amd64/libzip.so:[0x7fdc7f5c0905]
 377403 @ /usr/lib/jvm/java-7-openjdk-amd64/jre/lib/amd64/server/libjvm.so:[0x7fdc85a59f8a]
      8 @ /usr/lib/x86_64-linux-gnu/libstdc
```

and the number of unallocated addresses for each one:

```
     25 0x7fdc7395d79a
    151 0x7fdc7395d7f2
     51 0x7fdc7395d85e
      6 0x7fdc73973c28
      1 0x7fdc7399be7b
      1 0x7fdc7399c269
      1 0x7fdc78315705
      1 0x7fdc7831ae95
      1 0x7fdc7831b615
      1 0x7fdc78b4351f
      1 0x7fdc78b43e48
      1 0x7fdc78b442b4
      1 0x7fdc78b442e5
      1 0x7fdc78b464be
      1 0x7fdc78b464de
      3 0x7fdc78b4d387
      3 0x7fdc78b4d3b6
    198 0x7fdc78b4d63d
      1 0x7fdc78b4f81f
      1 0x7fdc78b4f844
      1 0x7fdc78b4f863
      1 0x7fdc78b4f896
      1 0x7fdc78b4f955
     70 0x7fdc78b5a6e6
     87 0x7fdc78b5a74b
      1 0x7fdc78b5a7bb
      1 0x7fdc78b5c64c
      1 0x7fdc78b5c6e4
      2 0x7fdc78b5ca33
      1 0x7fdc78b5cd65
      1 0x7fdc78b5ef3b
      1 0x7fdc78b6448f
      1 0x7fdc78b6456b
      1 0x7fdc78b645d0
      1 0x7fdc78b64630
      1 0x7fdc7f5bf5d2
      1 0x7fdc7f5bf604
      1 0x7fdc7f5bfc5f
      2 0x7fdc7f5bfc90
      3 0x7fdc7f5bff64
      3 0x7fdc7f5c00e9
      1 0x7fdc7f5c0905
      3 0x7fdc84fb4698
  19016 0x7fdc85a59f8a
      2 0x7fdc867201c9
      3 0x7fdc8677242a
      1 0x7fdc86cc759d
      8 0x7fdc86ccae37
      8 0x7fdc86ccb140
      1 0x7fdc86ccf220
      8 0x7fdc86cd0d2e
      1 0x7fdc86cd24b5
      1 0x7fdc86cd49f7
      8 0x7fdc86cd9d4a
```

From these lists, the most unallocated addresses were allocated by libjvm.so...

The commands I've used are these:

1. changed init.c to call mtrace() in Java_com_goldencode_p2j_main_ClientCore_processInit
2. export MALLOC_TRACE=~/workspace/p2j/testcases/simple/client/mtrace.log to set the log file name
3. call mtrace to analyze it: mtrace mtrace.log > mtrace2.log
4. extract the sources of the unallocated addresses: cat mtrace2.log | cut -b 33- | sort | uniq > mtrace3.log
5. edit the mtrace3.log file manually to leave only the addresses
6. get the users of these addresses: cat mtrace3.log | while read line ; do lline=`echo $line | sed -e 's/^[ \t]*//'` ; grep $lline mtrace.log ; done | cut -d '+' -f 1 | sort | uniq -c > mtrace4.log which produces the output above.

#8 - 08/25/2015 10:54 AM - Constantin Asofiei

Next test was a long loop which just writes to a file numbers from 1 to 1 million: the RES memory stays constant at ~66MB. So the leak might be somewhere in the JNI code.

#9 - 08/25/2015 11:17 AM - Greg Shah

I've been digging into the NCURSES API usage.

In initConsole() we call newterm(), which allocates a SCREEN data structure and returns the pointer. Although we do call endwin() both at process exit and on any suspend(), we never call delscreen() to delete that memory. This may not be a big deal if we only ever call newterm() once. But if we re-initialize the client during context reset, then it could be a cause of a leak. Do we reset the client in that case? If not, we possibly should but we would need to fix this leak.

Of course, we do re-initialize the client/driver on CTRL-C. We almost certainly have a memory leak in that case.

I can't tell from the NCURSES docs if restartterminal() is the equivalent of the newterm(). It would only be called when the 4GL code TERM = ... gets executed, but if that is done in a ChUI app, then it may be another source of a leak.

Your results do show 198 allocations from newwin(), which makes me wonder if that is connected to the use of newterm().

#10 - 08/25/2015 11:30 AM - Constantin Asofiei

Greg, something else in terminal.c - the Java_com_goldencode_p2j_ui_client_chui_driver_console_ConsoleHelper_addArrayNative is defined using jbyteArray instead of jintArray for attrs and colors parameters. Also, I've tried calling DeleteLocalRef for the array arguments but it didn't help.

I wonder if it isn't some flaw in some native array management at the JVM level.

#11 - 08/25/2015 11:43 AM - Greg Shah

Created task branch 2662a from trunk revision 10928.

#12 - 08/25/2015 11:53 AM - Greg Shah

I have committed a change (rev 10929) to delete the screen resources when the driver is re-initialized (and also at process exit).

Constantin: please add your changes to this branch.

#13 - 08/25/2015 11:55 AM - Greg Shah

> is defined using jbyteArray instead of jintArray for attrs and colors parameters

This is definitely wrong. I assume you tested with this fixed and it doesn't make any noticeable difference?

#14 - 08/26/2015 03:35 AM - Constantin Asofiei

Greg Shah wrote:

> > is defined using jbyteArray instead of jintArray for attrs and colors parameters
>
> This is definitely wrong. I assume you tested with this fixed and it doesn't make any noticeable difference?

Correct, there was no difference in not freed addresses/mem usage.

My changes are in rev 10930.

#15 - 08/30/2015 01:09 PM - Eric Faulhaber

Rebased task branch 2662a to trunk rev. 10929; task branch is now 10933.

Merged 2662a/10933 to trunk. 2662a was archived. Trunk revision is now at 10930. Team was notified via email.

#16 - 04/14/2016 11:58 AM - Eric Faulhaber

AFAICT from the above notes, all the known memory leaks around NCURSES use have been addressed. Does anyone else know of any ideas not yet followed up? Are we assuming any remaining client leaks are related to #2657?

I think I had forgotten to close out this issue after the fix was merged because the work on this issue and #2657 had combined, and there were still open questions in that task. If there is nothing left to do on this issue, I'd like to close it.

#17 - 04/16/2016 01:31 PM - Eric Faulhaber

Greg, Constantin: is there anything you know of that still needs to be addressed for this issue?

#18 - 04/16/2016 01:32 PM - Constantin Asofiei

Eric Faulhaber wrote:

> Greg, Constantin: is there anything you know of that still needs to be addressed for this issue?

I'm not aware of any other issues in this task.

#19 - 04/19/2016 03:19 PM - Eric Faulhaber

- % Done changed from 0 to 100

- Status changed from New to Closed

#20 - 11/16/2016 12:06 PM - Greg Shah

- Target version changed from Milestone 11 to Cleanup and Stablization for Server Features