Bug: even though ByteBuffer.allocateDirect is said to use out-of-heap memory when needed, it can cause OOM for the heap memory

STATUS UPDATE No update yet. Edit

DESCRIPTION lb...@gmail.com created issue #1

https://issuetracker.google.com/issues/156102344

Still occurs, including on Android 11 Pixel 4.

deleted
0 B

deleted
0 B

deleted
0 B

✓ Mentioned issues (2) ✓ Links (3)

Mentioned issues (2)

P3 Bug: even though ByteBuffer.allocateDirect is said to use out-of-heap memory when needed, it can cause OOM for the heap memory “https://issuetracker.google.com/156102344”

P3 Bug: ZipInputStream can't open a zip file that ZipFile can, making it impossible to use SAF for this matter “https://issuetracker.google.com/152802775”

Links (3)

“In the meantime, apps can allocate direct buffers in native code using NewDirectByteBuffer , but they take responsibility for the allocating the memory and managing it.”
“If the memory is being allocated for data that is coming from (or going) files, then FileChannel.map() will memory map those files without touching the heap.”
“ … to create out-of-heap allocations, consider instead just requesting a larger heap by setting android:largeHeap="true" in your application's manifest. See https://developer.android.com/guide/topics/

COMMENTS

ja...@google.com <ja...@google.com> #2

Assigned to ja...@google.com.

Thank you for reporting this issue. We've shared this with our product and engineering teams and will continue to provide updates as more information becomes available.

ja...@google.com <ja...@google.com> #3

Status: Won't Fix (Intended Behavior)

Thanks again for the feedback! Our product and engineering teams have evaluated the issue and responded:

> We do not intend the statement that "The contents of direct buffers may reside outside of the normal garbage-collected heap" to imply that allocations exceeding the Java heap size limit w
> allocated in a non-movable section of the Java heap. It does count towards Java heap space. The specification gives the implementation the option of either doing this or of not including i
> either. Thus we view the current behavior as "working as intended".

> Note that in AOSP ToT, we no longer guarantee that the heap is 100% exhausted before returning OOME. (This avoids an almost-always-useless GC storm shortly before reporting OOME; th
> There's also a possibility of allocation failing early due to fragmentation. I don't think this is a factor here, but we urge caution when it comes to reasoning about remaining heap capacity.

lb...@gmail.com <lb...@gmail.com> #4

@3 So is there a good class that does it, then?
A nice class that allows to easily store some stuff into out-of-heap memory?
Is there a way to tell the DirectByteBuffer to be there?
Or we have to rely purely on JNI ?

ja...@google.com <ja...@google.com> #5

Thanks for the update. We've shared this with our product and engineering teams and will continue to provide updates as more information becomes available.

In the meantime, apps can allocate direct buffers in native code using NewDirectByteBuffer, but they take responsibility for the allocating the memory and managing it.

lb...@gmail.com <lb...@gmail.com> #6

@5 Is there a nice way to reach it via Java/Kotlin?
Can you please consider adding such a thing, even as a new dependency of android-x ?
Maybe call it "memory allocation" ?

ja...@google.com <ja...@google.com> #7

Thanks again for the feedback! Our product and engineering teams have evaluated the issue and responded:

Thanks for the suggestion. In the meantime...Android does not offer any utilities here. The core library is based on OpenJDK 8.

The `allocateDirect()` allocations are in the non-movable heap.

If the memory is being allocated for data that is coming from (or going) files, then FileChannel.map() will memory map those files without touching the heap.

Otherwise, the only support option is the JNI method NewDirectByteBuffer() and this can be wrapped appropriately for your own scenarios. It seems unlikely there is one-size suits all solutio

ja...@google.com <ja...@google.com> #8

Thanks again for the feedback! Our product and engineering teams have evaluated the issue and responded:

> Instead of trying to create out-of-heap allocations, consider instead just requesting a larger heap by setting android:largeHeap="true" in your application's manifest. See https://developer.ar

lb...@gmail.com <lb...@gmail.com> #9

@8 No, because:
1. Nothing is guaranteed out of it
2. You will almost always just get much less than the real free RAM.
3. It is fixed in size. Meaning that if you got X amount RAM, it will stay this way. No way to get more. You can't even specify how much you need. It's either true or false.
4. It even says you should avoid using it, as it causes performance to degrade due to larger work for GC.

So, for example, if I want to cache a large file into RAM to perform some operation on it, this is not a good way to do it.
I've actually had such a case. Some ZIP content (that you get from Uri as InputStream) are created differently than others, so you can't use ZipInputStream on them (of the framework), but wl
byte array (The Android frameworks ZipFile accepts only a file path, so it can't always be used).
Sadly though, since byte array is in the heap, and the max RAM you can get for the heap is determined by the OS and is fixed, it can't always work
If you got 200MB for example (with or without the largeHeap flag), you won't be able to open a 300MB ZIP file using this approach.

If you want, you can read about this, here's an example of Google's comment on this exact matter:
https://issuetracker.google.com/issues/152802775#comment5

Google explains that we can't use some amount of RAM just like that for heap. I was told that Apache can handle it, but again, it needs a large heap to handle the file content...
So what I did is to get it via JNI in this case (someone else helped me on this, as I'm very rusty with JNI), which is quite a weird workaround.
Another solution I got is that instead of caching anything, I just let it go through the stream and re-create it when needed. It's more memory efficient, but performance-wise it might be quite b

ja...@google.com <ja...@google.com> #10

Thanks for the update. We've shared this with our product and engineering teams and will continue to provide updates as more information becomes available.