NMT for all - Extending NMT beyond Hotspot

Johan Sjölén
Oracle JPG

Native memory tracking – what is it?

- Native memory tracking
  - Groups native allocations into categories
  - Presents statistics regarding these allocations

Native memory tracking – what is it?

```
Total:  reserved=664192KB,  committed=253120KB                          <--- total memory tracked by
Native Memory Tracking

-                 Java Heap (reserved=516096KB, committed=204800KB)     <--- Java Heap
                            (mmap: reserved=516096KB, committed=204800KB)

-                     Class (reserved=6568KB, committed=4140KB)         <--- class metadata
                            (classes #665)                            <--- number of loaded classes
                            (malloc=424KB, #1000)                     <--- malloc'd memory, #number
of malloc
                            (mmap: reserved=6144KB, committed=3716KB)
```

Native memory tracking today

- ✅ Hotspot – The Java Virtual Machine
- ❌ Core libraries
- ❌ FFM, the Foreign Function & Memory API
- ❌ Third party libraries

Native memory tracking today

- ✅ Hotspot
- ❌ Core libraries – Libraries implemented in C and exposed to Java via JNI
- ❌ FFM, the Foreign Function & Memory API
- ❌ Third party libraries

Native memory tracking today

- ✅ Hotspot
- ❌ Core libraries
- ❌ FFM, the Foreign Function & Memory API – C/Java interop without JNI
- ❌ Third party libraries

Native memory tracking today

- ✅ Hotspot
- ❌ Core libraries
- ❌ FFM, the Foreign Function & Memory API
- ❌ Third party libraries – C libraries whose source we do not control

Native memory tracking tomorrow?

- ✅ Hotspot
- ✅ Core libraries
- ✅ FFM, the Foreign Function & Memory API
- ❌ Third party libraries

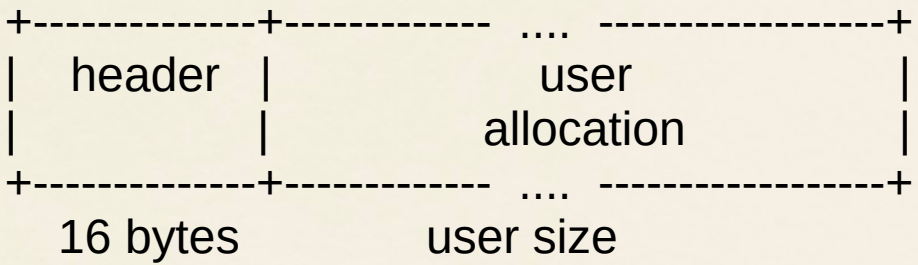
Native memory tracking tomorrow?

- ✅ Hotspot
- ✅ Core libraries
- ✅ FFM, the Foreign Function & Memory API
- ❌ Third party libraries

What’s the minimal change set required for this change?

A look at the internals

- NMT has MemTags
- Statistics done via atomically changing entries in a statically sized array
- MemTags are put into each malloc at the start via a header datastructure



A look at the internals

- NMT has MemTags
- Statistics done via atomically changing entries in a statically sized array
- MemTags are put into each malloc at the start via a header datastructure
- MemTags are defined statically in an enum

A look at the internals

- NMT has MemTags
- Statistics done via atomically changing entries in a statically sized array
- MemTags are put into each malloc at the start via a header datastructure
- MemTags are defined statically in an enum
- All functionality in NMT’s summary mode depends on handing out and receiving MemTags

A look at the internals

- NMT has MemTags
- Statistics done via atomically changing entries in a statically sized array
- MemTags are put into each malloc at the start via a header datastructure
- MemTags are defined statically in an enum
- All functionality in NMT's summary mode depends on handing out and receiving MemTags

So just give Java and C access to MemTags?

A plan for tackling the problem!

- Ditch MemTags as enum members
  - Make them dynamically creatable!
- Then we can expose them via an interface in jvm.h for the native libraries
- And a Java interface for FFM

Let’s enact the plan!

Dynamically creatable MemTags

- We need:
  - Lock free access to the memory tag accounting
  - Dynamically adding MemTags
  - Only use as much memory as is needed to support the use case

Dynamically creatable MemTags

- We need:
  - Lock free access to the memory tag accounting
  - Dynamically adding MemTags
  - Only use as much memory as is needed to support the use case
- So we:
  - Make MemTag 4 bytes long instead of 1 (because 256 MemTags is too few, 2**32 is definitely enough)
  - Add a linear allocator using mmap to page in new memory as needed
    - Allows for ‘resizing’ without moving any memory
    - Allows for graceful failure if out of memory
  - Add a MemTag factory which takes strings and returns MemTags

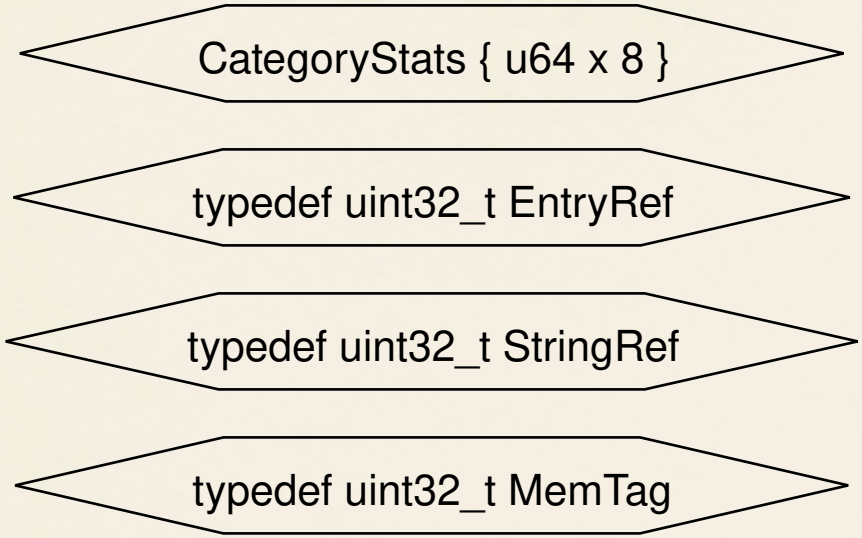
Dynamically creatable MemTags

So we:

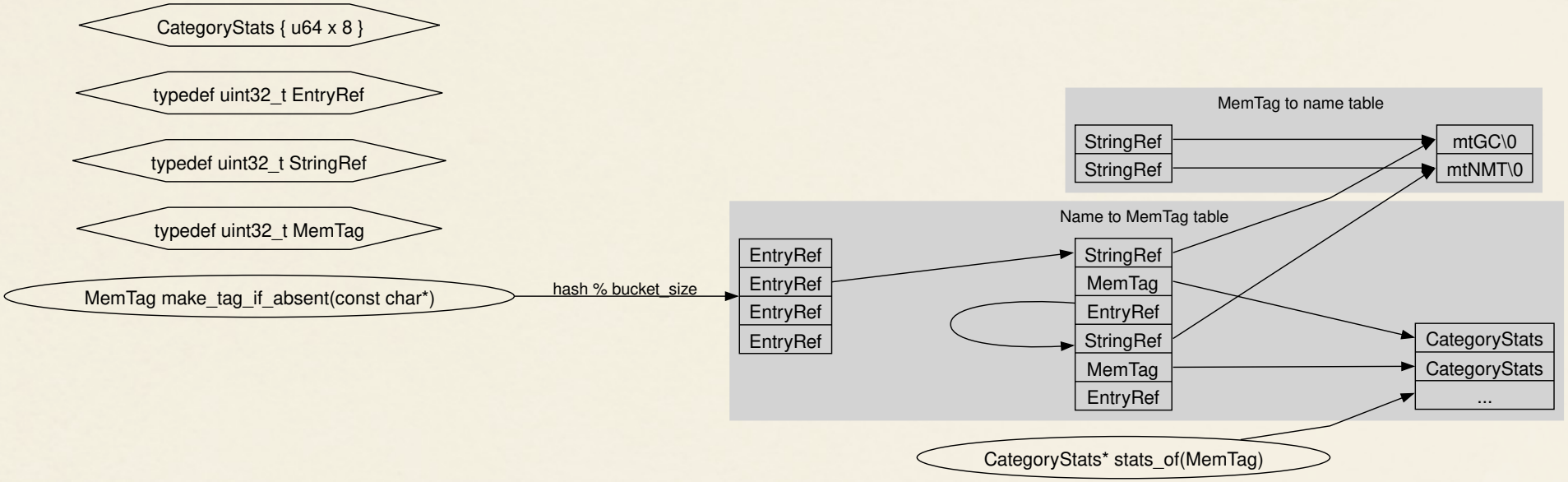
- Make MemTag 4 bytes long instead of 1 (because 256 MemTags is too few, 2**32 is definitely enough)
- Add a linear allocator using mmap to page in new memory as needed
  - Allows for ‘resizing’ without moving any memory
  - Allows for graceful failure if out of memory
- Add a MemTag factory which takes strings and returns MemTags
  - A dual table: Name to MemTag/MemTag to name
    - Simple closed hashtables, with some space saving tricks applied
  - Accessing these can be done under a lock – it’s a much rarer operation

Data structure layout

- CategoryStats 64 bytes
  - Exactly 1 cache line (avoids false sharing)
- Use 4-byte indices instead of pointers
  - base + index = address
  - Save 4 bytes per reference
  - No necessity to update pointers
  - Con: 4GiB limit
    - Unlikely to be issue



Data structure layout



Exporting this to C

JVM API

- JVM.h API
  - Arena-style API
    - Allocations ‘belong’ to an arena. In other words: They belong to a MemTag
  - Devs familiar with thinking about allocations belonging to an arena from the FFM API

JVM API

```
typedef uint32_t arena_t;
arena_t JNICALL JVM_MakeArena(const char *name);
void *JVM_ArenaAlloc(size_t size, arena_t a);
void  JVM_ArenaFree(void *ptr);
void *JVM_ArenaRealloc(void *p, size_t size, arena_t a);
void *JVM_ArenaCalloc(size_t numelems, size_t elemsize, arena_t a);

/* Virtual memory operations such as reserve_memory omitted */
```

Let’s use it!

Example C program

```
arena_t deflate_arena;
jlong Java_java_util_zip_Deflater_init(…) {
 deflate_arena = JVM_MakeArena(“java.util.zip.Deflater”);
}
static voidpf local_allocation(voidpf opaque, uInt items, uInt size) {
 return JVM_ArenaCalloc(items, size, deflate_arena);
}
```

Example C program

```
arena_t zip_arena = 0;
bool initialized = 0;
const char* arena_name = "java.util.zip";
arena_t arena() {
 if (!initialized) {
  zip_arena = JVM_MakeArena(arena_name); initialized = 1;
 }

 return zip_arena;
}
```

What about FFM?

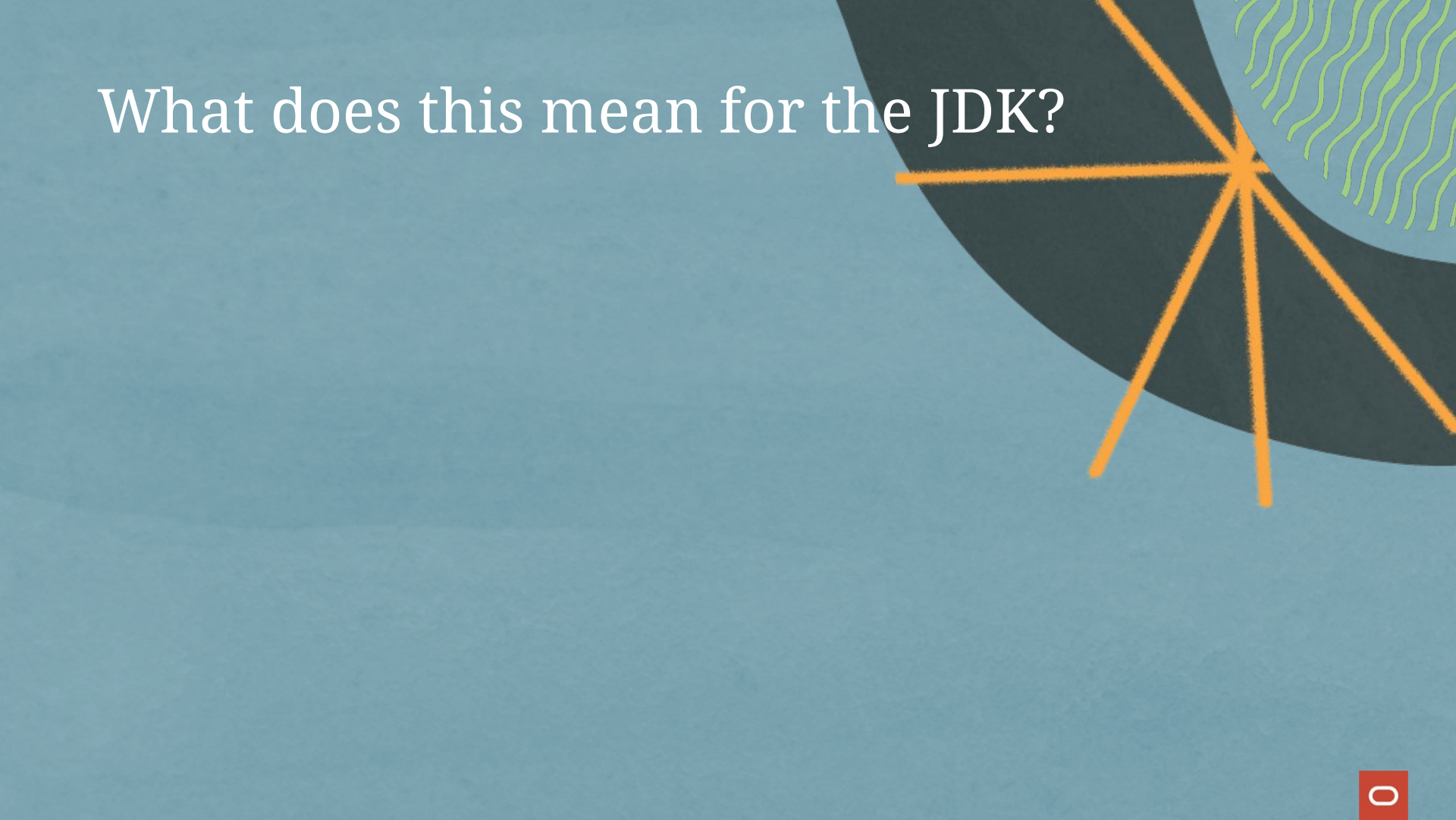
FFM

- Expose NMT via JNI – A set of native methods corresponding to the C API
- Replace the usage of sun.misc.Unsafe, instead use NMT
- Equip Arenas with constructors taking strings as names

FFM

- Expose NMT via JNI – A set of native methods corresponding to the C API
- Replace the usage of sun.misc.Unsafe, instead use NMT
- Equip Arena’s with constructors taking strings as names
  - Ouch! A lot of indirection in the Java code makes this a bit painful to implement
  - But really… It all boils down to Unsafe.allocatememory0

What does this mean for the JDK?



What does this mean for the JDK?

- May change how we do things in the VM
  - Tags are so cheap and dynamic that:
    - We can have namespacing for NMT
      - GrowableArray(MemTag super_group);
      - MemTag gcCardTable = make_memtag(“gc.CardTable”);
      - Today: GrowableArray<MemTag MT>; Tomorrow: No answer :-(
      - Requires some sort of consistency rules

What does this mean for the JDK?

- Better analysis of memory issues
  - Much easier to determine if it’s a JVM, Java core library, or a user application issue
  - Easier FFM => More C interop => Good with better memory analysis?
  - With more readily parsable output we can use this for all kinds of cool analysis

Thank you

Johan Sjölén

johan.sjolen@oracle.com