READ MODE · ANSWER FIRST

How Java Off-Heap Memory Works

An interactive, visual explanation of How Java Off-Heap Memory Works, from setup through the complete system.

CHEAT SHEET · 8 ESSENTIAL IDEAS

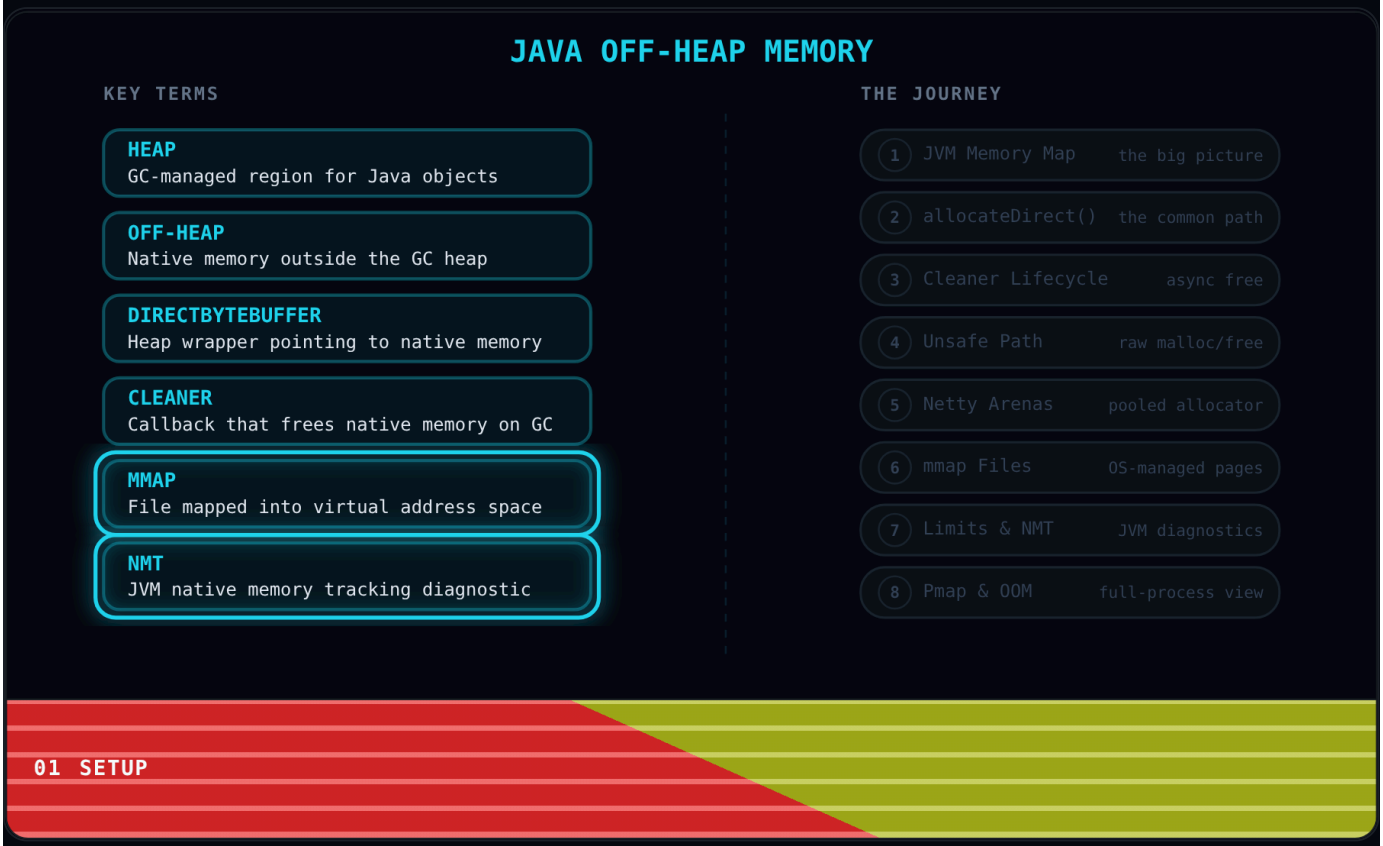
The whole story in 8 lines

How Java Off-Heap Memory Works becomes easier to reason about when every stage is connected as one system.

1. Structural containment scene. Show the process RSS as a large rectangle containing heap, metaspace, thread stacks, code cache, GC, and...
2. Pipeline flow: Java code → DirectByteBuffer constructor → Bits.reserveMemory → malloc → native segment.
3. Flow scene showing GC → phantom ref enqueue → Cleaner thread → Deallocator.free().
4. Lanes layout comparing the managed DirectByteBuffer path (left) against the raw Unsafe path (right).
5. Structural containment scene. Show the Arena → Chunk → Page hierarchy with thread-local cache flow.
6. Two-tier: disk file at bottom, virtual address space in middle, physical RAM pages at top. Show page faults triggering lazy loading.
7. Gauge bars for direct memory usage approaching the limit, NMT zone breakdown, and the System.gc() trigger mechanism.
8. Data representation scene. Horizontal bars showing pmap regions color-coded by type, with an RSS vs NMT gap indicator.

STAGE 1 OF 10

Setup



When your Java program runs, the JVM claims a chunk of memory from the operating system. Most developers only think about the heap, the region managed by the garbage collector. But there is a whole world of memory outside the heap that the JVM and your code can use directly. This explainer teaches you how that off-heap world works.

The heap is the GC-managed region where most Java objects live. Off-heap (or native memory) is everything the JVM process uses outside the heap: thread stacks, JIT-compiled code, class metadata, and memory you allocate explicitly via NIO buffers or Unsafe.

A DirectByteBuffer is the main API for allocating off-heap memory from Java code. It creates a small wrapper object on the heap that points to a large native memory segment obtained via malloc. A Cleaner is the mechanism that frees that native memory when the GC collects the wrapper.

mmap (memory-mapped I/O) maps a file directly into the process virtual address space, letting you read file data as if it were ordinary memory. NMT (Native Memory Tracking) is the JVM diagnostic that accounts for native memory usage by JVM subsystem, though it has blind spots we will explore.

These six terms will thread through every stage. Notice how they connect: DirectByteBuffers use native memory that Cleaners free, mmap bypasses both, and NMT tries to track it all. Together they form the machinery of Java off-heap memory.

Over the next eight stages we will walk through the full landscape: the JVM memory map, allocateDirect internals, the Cleaner lifecycle, the raw Unsafe path, Netty arena-based pooling, memory-mapped files, NMT diagnostics, and pmap-based OOM diagnosis. Let us start with the big picture.

STAGE 2 OF 10

JVM Memory Map

Here is the JVM process as the operating system sees it: one block of resident memory. Everything your Java program uses, whether heap objects, compiled code, or raw native buffers, lives inside this single process address space.

The Java Heap is the region most developers think about. It is where new Object() goes, where your application data lives, and where the garbage collector operates. But notice how much space remains outside it.

Metaspace holds class definitions, method metadata, and constant pools. Before Java 8 this was called PermGen and lived inside the heap. Now it uses native memory and can grow until the OS runs out.

Thread stacks consume native memory too. Each thread gets a fixed-size stack (usually 512 KB to 1 MB). An application with 500 threads can easily consume 500 MB of off-heap memory just for stacks.

The JIT compiler generates optimized machine code and stores it in the Code Cache. The GC itself needs bookkeeping structures like remembered sets and card tables. Both are off-heap.

Finally, the "Other" or "Internal" zone. This is where DirectByteBuffers, Unsafe allocations, and JNI native allocations live. This is the region we will focus on for the rest of this explainer. Try toggling AlwaysPreTouch in the sidebar to see how it affects the RSS commitment pattern.

The key insight: the heap is just one slice of a much larger memory footprint. When your container gets OOMKilled but the heap looks fine, the answer is almost always hiding in these off-heap regions. Next, let us see how ByteBuffer.allocateDirect taps into that native zone.

STAGE 3 OF 10

allocateDirect()

03 ALLOCATEDIRECT()

Now that we have seen the memory map, let us zoom into the most common off-heap API: ByteBuffer.allocateDirect(). Your code calls this with a size in bytes. Watch how the request flows through the JVM internals to produce a usable native buffer.

The first thing the JVM does is check the direct memory reservation. The class Bits.reserveMemory tracks how much direct memory is currently in use. If the new allocation would exceed MaxDirectMemorySize, it triggers a System.gc() and retries. If it still exceeds the limit, an OutOfMemoryError is thrown.

Once the reservation succeeds, the JVM calls Unsafe.allocateMemory(size) internally. This is a thin JNI wrapper around the C malloc function. The operating system returns a pointer to a block of virtual memory pages.

Now two things exist: a DirectByteBuffer object on the Java heap (about 72 bytes, holding the native address, capacity, and position) and the actual native memory segment off-heap. The heap object is the handle. The native segment is the payload. Try changing Buffer Size in the sidebar. Notice how the heap wrapper stays the same size while the native segment grows.

The constructor also registers a Cleaner. This is a callback (a Deallocator) that will call Unsafe.freeMemory when the GC collects the DirectByteBuffer wrapper. Without this Cleaner, the native memory would leak forever. We will explore this lifecycle in detail in the next stage.

The returned ByteBuffer is ready. Your code reads and writes through the heap wrapper, but every get() and put() operates directly on native memory, no copying through the heap. This is why NIO channels prefer direct buffers for I/O: the kernel can DMA straight from that native segment without an intermediate copy.

STAGE 4 OF 10

Cleaner Lifecycle

We just saw how allocateDirect registers a Cleaner. Now let us trace what happens when that DirectByteBuffer is no longer needed. The deallocation path is asynchronous and depends entirely on the garbage collector running at the right time.

Your application drops all references to the DirectByteBuffer. The object is now unreachable, but nothing has happened yet. The native memory is still allocated. This is the critical window: the heap wrapper is garbage, but the off-heap segment persists.

A GC cycle runs. The collector discovers the DirectByteBuffer is unreachable. It does not free it immediately. Instead, it enqueues the associated PhantomReference into a ReferenceQueue. Think of this as the GC leaving a note: "this object is dead, someone should clean up after it."

The Reference Handler thread (a JVM-internal daemon) picks up the enqueued reference and passes it to the Cleaner. The Cleaner invokes the Deallocator, which is the callback that was registered during allocation.

The Deallocator calls Unsafe.freeMemory(address), which calls the C free() function. The native memory is finally returned to the OS. The Bits.reserveMemory counter is decremented, making room for new direct buffers.

Switch GC Timing to Delayed in the sidebar. Notice how the native memory stays allocated longer when GC is delayed. In a real system, if your application allocates direct buffers faster than GC collects them, native memory pressure builds up until an OOM or a forced System.gc() saves you.

This is the fundamental tension of direct buffers: allocation is explicit (you call allocateDirect), but deallocation is implicit (it depends on GC). If GC does not run, native memory leaks. This is why System.gc() calls inside Bits.reserveMemory exist. Next, let us see the escape hatch: Unsafe.allocateMemory, where you control both sides.

STAGE 5 OF 10

Unsafe Path

The Cleaner lifecycle we just learned about works, but it has a fundamental cost: deallocation timing is at the mercy of the GC. Libraries that need predictable, high-throughput memory management take a different route entirely: sun.misc.Unsafe.

On the left is the managed path you already know. ByteBuffer.allocateDirect checks MaxDirectMemorySize via Bits.reserveMemory, allocates native memory, registers a Cleaner, and trusts GC for cleanup. On the right is the raw Unsafe path.

Unsafe.allocateMemory(size) is a direct JNI call to malloc. It returns a raw native address as a long. No limit checking, no Bits reservation, no Cleaner registration. The JVM has no idea this memory exists. NMT tracks it in the "Internal" category only in its own bookkeeping, not against MaxDirectMemorySize.

Reading and writing use Unsafe.putByte(address, value) and Unsafe.getByte(address). There is no bounds checking. Write past the end of your allocation and you corrupt the process heap. Read from a freed address and you get a segfault or garbage data.

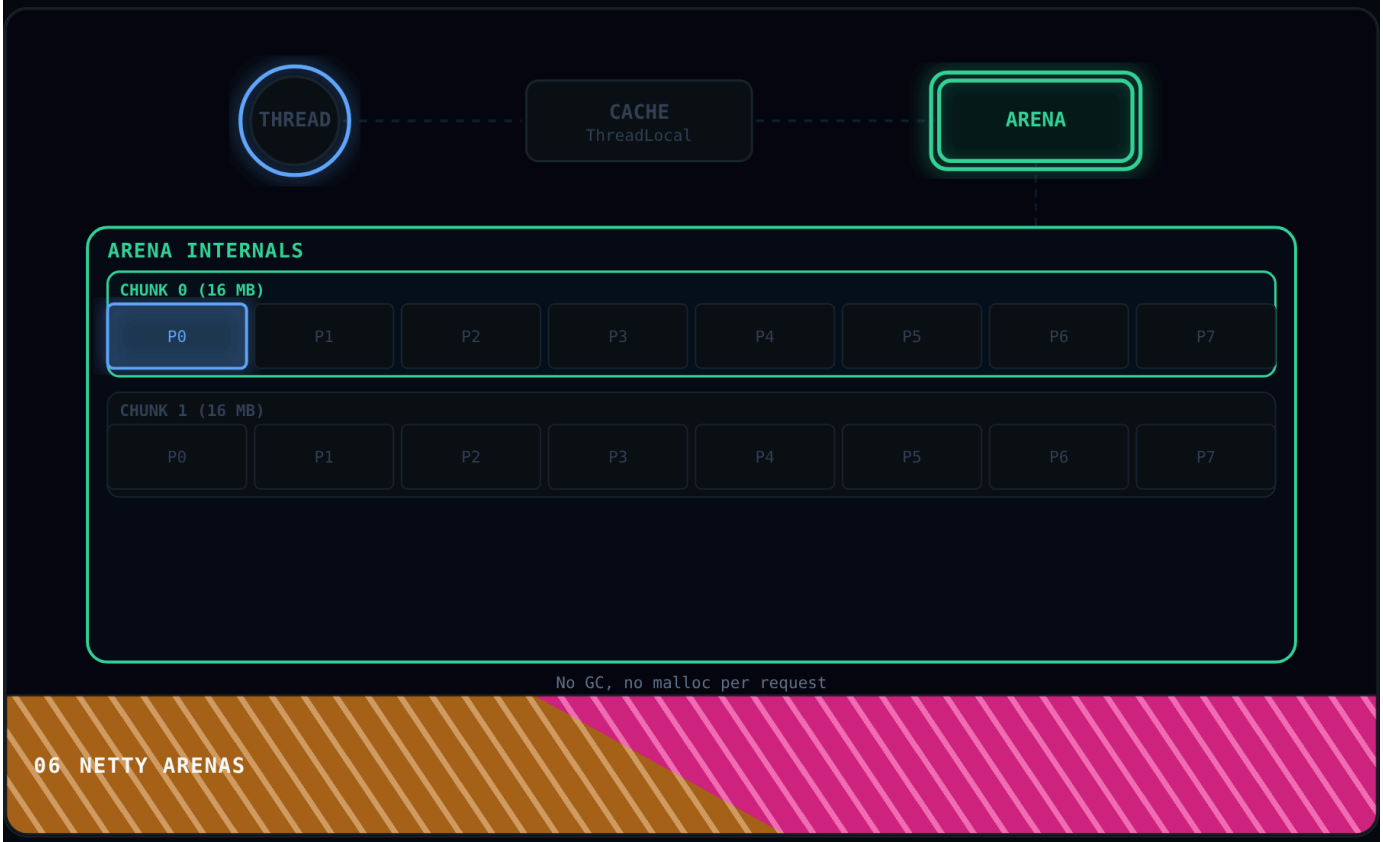
Deallocation is Unsafe.freeMemory(address). You must call this yourself. Forget it and the memory leaks permanently. Call it twice and you get a double-free crash. This is C-style manual memory management inside a Java process.

Switch Allocator to Pooled in the sidebar. Libraries like Netty build pooled allocators on top of Unsafe. They allocate large chunks (arenas) once, carve them into smaller slabs, and recycle them without ever going back to malloc. This eliminates GC pressure entirely and gives deterministic allocation performance.

The tradeoff is clear: the managed path is safe but couples deallocation to GC timing. The Unsafe path is fast and deterministic but shifts all responsibility to the programmer. Most production systems that handle high-throughput I/O (Kafka, Cassandra, Netty) choose the Unsafe path. Netty, in particular, builds an entire arena-based allocator on top of Unsafe. Let us see how that works next.

STAGE 6 OF 10

Netty Arenas



06 NETTY ARENAS

In the last stage we saw how Netty and similar libraries choose Unsafe for deterministic allocation. But Netty does not just call malloc and free repeatedly. It builds a sophisticated arena-based allocator on top of Unsafe that eliminates per-allocation system calls entirely.

The PooledByteBufAllocator creates a fixed number of Arenas at startup, typically two per CPU core. Each Arena pre-allocates large Chunks of native memory (16 MB each) via Unsafe.allocateMemory. These chunks are the raw material from which all buffers are carved.

Each application thread is bound to one Arena in round-robin fashion. But before touching the Arena at all, the thread checks its own PoolThreadCache. This is a thread-local free list of recently released buffers, organized by size class. A cache hit means zero contention and zero locking.

On a cache miss, the Arena locks and uses a buddy algorithm to find free pages in one of its Chunks. For a page-sized request (8 KB), it finds one free page. For larger requests, it merges adjacent pages. For small requests under 8 KB, it subdivides a page into fixed-size slots using a bitmap. Try changing Alloc Size in the sidebar to see how the allocation tier changes.

The allocated buffer is returned to the thread. The thread reads and writes directly to the native memory through the buffer handle. No heap allocation occurs beyond the thin wrapper object. No GC pressure is generated by the data itself.

When the thread releases the buffer, it does not go back to the OS. It returns to the PoolThreadCache for immediate reuse. Only when the cache exceeds its size limit does it return buffers to the Arena free list. Memory is recycled, never freed. This is why Netty handles millions of allocations per second.

The tradeoff: Netty trades memory overhead for speed and predictability. Your process RSS stays high even when traffic drops, because the Arena keeps its Chunks allocated. Monitoring Netty memory requires its own metrics (PooledByteBufAllocator.metric()) since NMT only sees the raw Unsafe allocations. Next, let us explore a completely different path: memory-mapped files.

mmap Files

07 MMAP FILES

We have seen three ways to get off-heap memory: DirectByteBuffer (managed), Unsafe (raw), and Netty arenas (pooled). There is a fourth path that is fundamentally different: memory-mapped files. Instead of allocating anonymous memory, you map a file from disk directly into the process address space.

FileChannel.map(MapMode.READ_ONLY, 0, size) calls the POSIX mmap system call. The OS kernel creates a virtual memory mapping: a range of addresses in your process that correspond to byte offsets in the file. At this point, no physical RAM has been consumed.

When your code reads from the MappedByteBuffer, the CPU translates the virtual address to a physical address via the page table. If the page is not in RAM yet, a page fault fires. The kernel handles the fault by reading the file page from disk into a physical RAM page, then resumes your code.

This is lazy loading at the OS level. Only the pages you actually touch consume physical memory. A 50 GB file can be mapped into a process with only 100 MB of RAM. The OS evicts cold pages under memory pressure and re-faults them if needed. Change File Pages in the sidebar to see how the mapping grows.

Here is the critical diagnostic blind spot: NMT does not track memory-mapped files. The mapping is a kernel-level operation, and the JVM has no visibility into it. To see mapped memory, you need OS tools like pmap or /proc/pid/smaps. Entries with a non-zero inode are file-backed mappings.

MappedByteBuffers also lack explicit unmap. The mapping persists until GC collects the MappedByteBuffer object, which triggers an internal Cleaner similar to DirectByteBuffer. On Windows, this means the file remains locked until GC runs. Switch Access to Random and watch how page faults scatter across the file instead of streaming sequentially.

Memory-mapped files are the backbone of databases (RocksDB, LMDB), search engines (Lucene), and message brokers (Kafka log segments). They trade explicit memory management for OS-level paging, which is efficient for read-heavy, sequential workloads but unpredictable under memory pressure. Now let us look at how the JVM tracks and limits all of these off-heap paths.

STAGE 8 OF 10

Limits & NMT

We have explored three allocation paths: DirectByteBuffer, Unsafe, and mmap. Now let us see how the JVM keeps things from going off the rails. The primary safety mechanism is MaxDirectMemorySize, which caps how much direct buffer memory Bits.reserveMemory will allow.

By default, MaxDirectMemorySize is approximately equal to the heap size. When your code calls allocateDirect and Bits.reserveMemory sees that the new allocation would breach this limit, it does something surprising: it calls System.gc(). This is a full-stop hint to the GC to run immediately.

After triggering GC, Bits.reserveMemory retries the reservation. If dead DirectByteBuffers were collected and their Cleaners freed native memory, the new allocation succeeds. If it still exceeds the limit, an OutOfMemoryError: Direct buffer memory is thrown. This is why disabling System.gc() with -XX:+DisableExplicitGC can cause mysterious direct buffer OOMs.

Native Memory Tracking is the JVM diagnostic that shows where native memory goes. Enable it with -XX:NativeMemoryTracking=summary. It breaks down usage into zones: Java Heap, Class (Metaspace), Thread, Code, GC, Internal, and Other. Change NMT Mode in the sidebar to see what tracking reveals.

DirectByteBuffers appear in the "Other" zone (Java 11+, previously Internal). Unsafe allocations show up in "Internal". But here is the key blind spot: NMT does not track memory-mapped files or allocations made by native JNI libraries. If pmap shows your process using 8 GB but NMT only accounts for 5 GB, the gap is likely mmap or native library allocations.

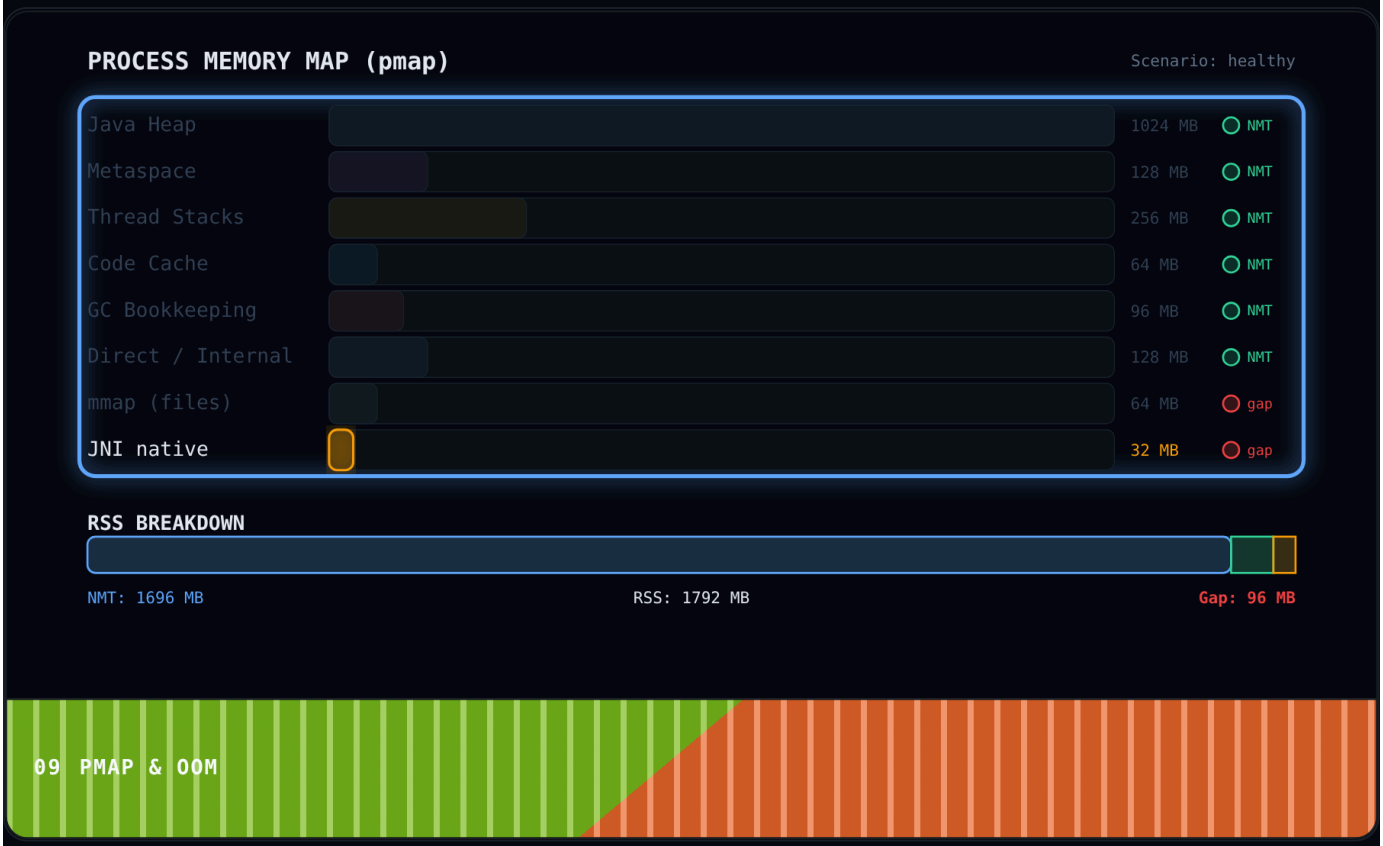
The performance overhead of NMT in summary mode is negligible for most workloads: roughly 5-10% additional memory for the tracking metadata itself, and no measurable CPU cost. Detailed mode adds more overhead and tracks individual call sites, useful for debugging but not for production.

The diagnostic toolkit is now clear: NMT for JVM-tracked native memory, pmap or /proc/pid/smaps for the complete picture including mmap, and jcmd VM.native_memory to get live NMT reports. When your

container gets OOMKilled, start with NMT to see what the JVM knows, then pmap to find what it does not. Let us look at pmap in detail next and see how to actually diagnose mmap and JNI-based OOMs.

STAGE 9 OF 10

Pmap & OOM



We just learned that NMT has blind spots: it cannot track mmap regions or memory allocated by native JNI libraries. When your container gets OOMKilled but NMT looks healthy, pmap is the tool that shows you what NMT cannot see.

pmap -x <pid> prints every memory region in the process address space. Each line shows an address range, its size in KB, resident pages (RSS), permissions, and a mapping name. File-backed regions show the file path. Anonymous regions show [anon] or [heap].

Start by identifying the NMT-tracked regions. The Java Heap is a large anonymous mapping, often the biggest single entry. Metaspace, thread stacks, code cache, and GC structures all appear as anonymous mappings that NMT accounts for. These are the blue bars in the visualization.

File-backed mappings come from mmap. They show a file path like /data/index.db or a shared library like libjvm.so. Large file-backed entries with high RSS are your mmap memory. If these are growing over time, you have a MappedByteBuffer leak or a file mapping that was never unmapped. Switch Scenario to mmap Leak in the sidebar to see how these regions dominate the gap.

Anonymous mappings not accounted for by NMT are typically JNI native library allocations. Libraries like RocksDB JNI, LevelDB, or custom C libraries allocate memory via their own malloc. NMT sees none of it. Switch Scenario to JNI Leak to see how these regions inflate the RSS gap.

The diagnostic workflow: run jcmd <pid> VM.native_memory summary to get the NMT total. Run pmap -x <pid> and sum the RSS column for the total process RSS. The gap between them is your investigation target. The stacked bar at the bottom breaks RSS into NMT-tracked, mmap, and JNI portions.

For mmap leaks, check /proc/<pid>/smaps for entries with high Rss and a file inode. For JNI leaks, use jemalloc profiling or async-profiler native memory mode. The key lesson: NMT is necessary but not sufficient. A complete memory audit always combines NMT, pmap, and knowledge of which native libraries your application loads. Now let us step back and see the whole picture together.

STAGE 10 OF 10

Recap

10 RECAP

We started with the JVM Memory Map: the heap is just one slice of a much larger process footprint. Thread stacks, metaspace, code cache, GC structures, and native allocations all live outside the heap and contribute to your process RSS.

Then we traced ByteBuffer.allocateDirect through its internals: Bits.reserveMemory checks the limit, Unsafe.allocateMemory calls malloc, and the result is a small heap wrapper pointing to a large native segment. The heap object is the handle; the native memory is the payload.

We followed the Cleaner lifecycle: when GC collects the DirectByteBuffer wrapper, a PhantomReference is enqueued, the Cleaner thread invokes the Deallocator, and Unsafe.freeMemory returns the native segment to the OS. Deallocation is implicit and depends on GC timing.

We explored the Unsafe escape hatch: raw malloc and free with no GC interaction, no limit tracking, and no safety net. This is the foundation high-performance libraries build on.

We zoomed into Netty arenas: a full pooled allocator built on Unsafe with pre-allocated chunks, buddy page allocation, thread-local caches, and buffer recycling. No GC pressure, no per-request system calls, and millions of allocations per second.

We mapped memory-mapped files: mmap gives you OS-managed, lazily-loaded, file-backed memory that is invisible to both the GC and NMT. Page faults load data on demand; the OS evicts under pressure.

We covered limits and diagnostics: MaxDirectMemorySize caps managed direct buffers, System.gc() is the safety valve, and NMT tracks JVM-internal zones by subsystem.

And we closed with pmap: the OS-level tool that shows the full process memory map, reveals mmap and JNI allocations invisible to NMT, and exposes the RSS gap that explains container OOMs.

The unified takeaway: Java off-heap memory is not one thing. It is a spectrum from fully managed (DirectByteBuffer + Cleaner) through pooled (Netty arenas) and OS-managed (mmap) to fully manual (Unsafe). Understanding which path your libraries use, and knowing which tool reveals each path, is the key to diagnosing memory issues in production.