The future of Lucene's MMapDirectory: *Why use it and what's coming with Java 16 and later?*

Uwe Schindler

Apache Software Foundation /
SD DataSolutions GmbH

thetaph1 – https://www.thetaphi.de

Overview

Memory Mapping (MMAP)

The “70s style” of file reads

Read `count` bytes from position `offset` of a file descriptor `fd`:

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
ssize_t read(int fd, void *buf, size_t count);
```

Problems

- Allocate local buffer in program
- 2 syscalls: seek to file position, read from file and copy data to local buffer
- Workflow behind scenes:
  - Kernel reads from disk into FS cache *(optional)*
  - Copy data across kernel/userspace border

Image: Julia Evans, Julia's drawings, the filesystem cache, https://drawings.jvns.ca/filesystem-cache/

- Allo
- 2 sy m file and
- Wor
  - K onal)
  - C r

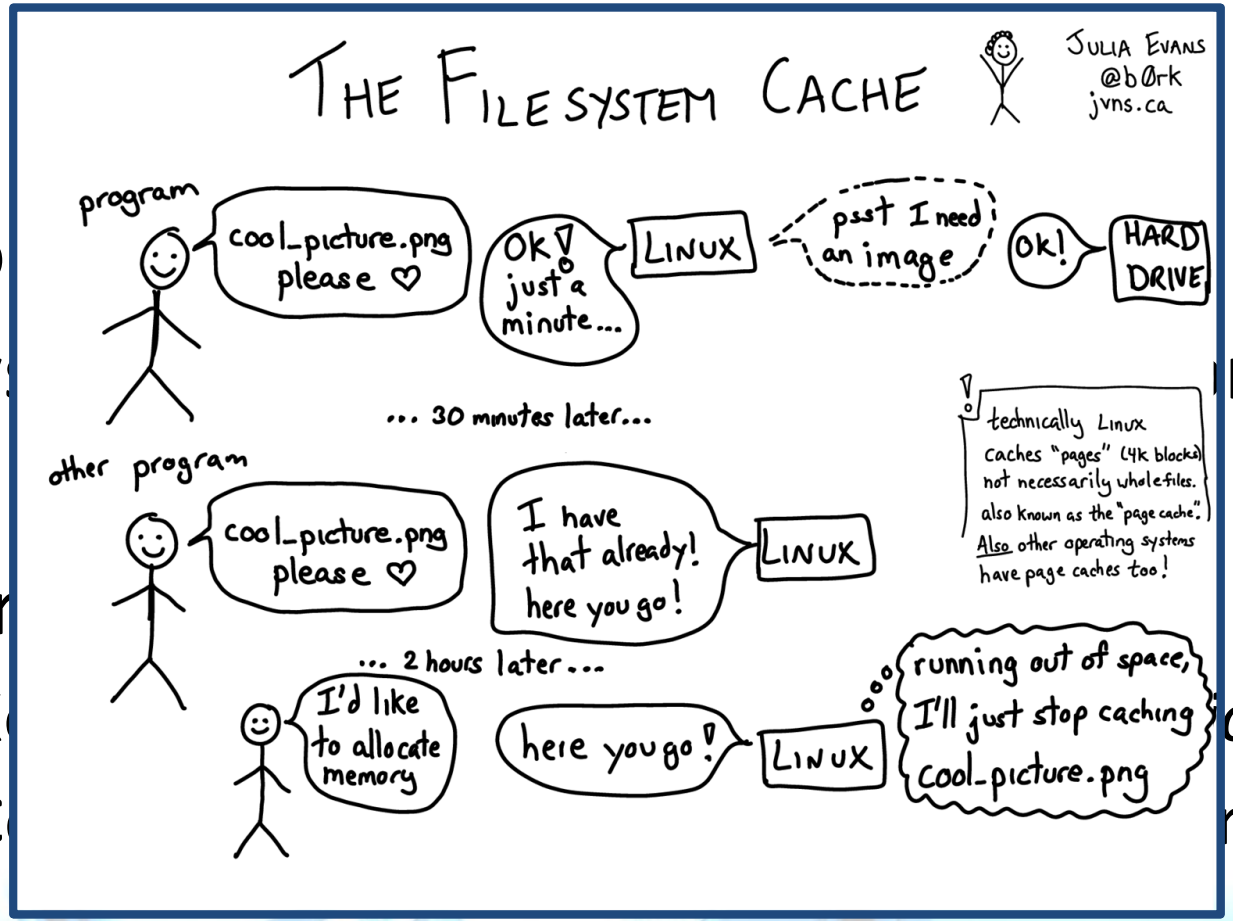


Image: Julia Evans, Julia's drawings, the filesystem cache, https://drawings.jvns.ca/filesystem-cache/

Problems

- Allocate local buffer in program
- 2 syscalls: seek to file position, read from file and copy data to local buffer
- Workflow behind scenes:
  - Kernel reads from disk into FS cache *(optional)*
  - Copy data across kernel/userspace border

Image: Julia Evans, Julia's drawings, the filesystem cache, https://drawings.jvns.ca/filesystem-cache/

With Java it gets worse!!!

- Allocate local buffer in JVM space (direct buffer)
- Application uses heap `byte[]`
- Copy: kernel -> direct buffer -> `byte[]`
- Apache Lucene: `SimpleFSDirectory`

...and why it's bad for Lucene:

Multithreading

- One file descriptor per index file in Lucene
- Seek / read must be synchronized!

The “90s style” of file reads

Read `count` bytes from position `offset` of a file descriptor `fd`:

```
#include <unistd.h>
ssize_t pread(int fd, void *buf, size_t count,
              off_t offset);
```

Pros / Cons

- Lucene implementation: `NIOFSDirectory`
- Prevents duplicate copy in most cases (native access to “direct” buffer outside heap)
- Reads in chunks; bad for random access (doc values)
- Works well on Linux, Java implementation on Windows is broken

Questions

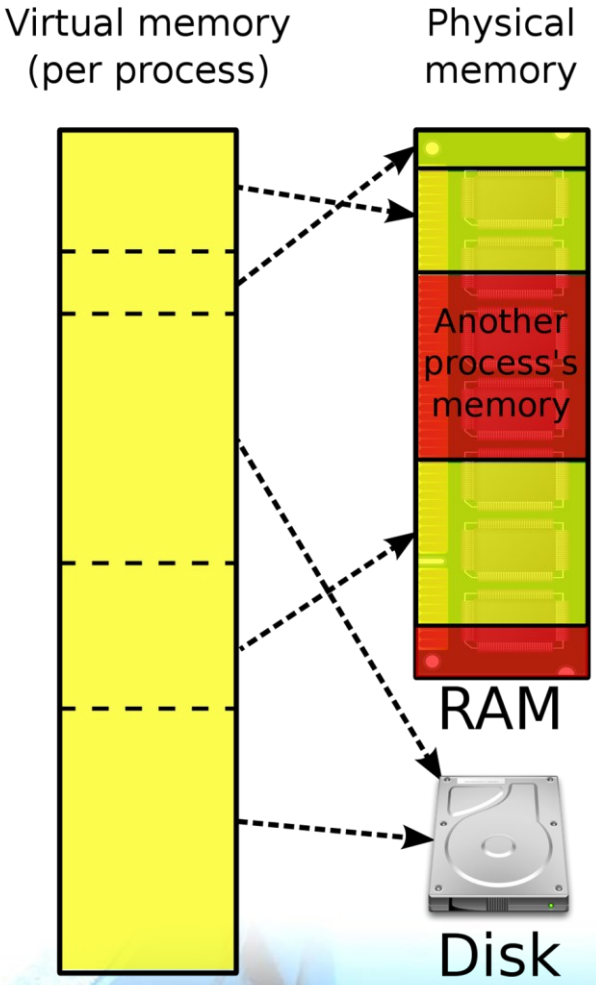
- Why do we copy data at all?
- If we know a page is in file system cache: Why not access it directly?
- Why do we need to ask kernel to load something from disk because we need it?

Memory Mapping: The “millenium style” of file reads

- Prevents copy of data between buffers (OS kernel, userspace, java heap)
- No syscalls
- Ideal use of FS cache
- Simple code *(at least theoretically)*

Foto: Supermicro X11DPi NT Memory Slots

Virtual Address Space



Virtual Address Space

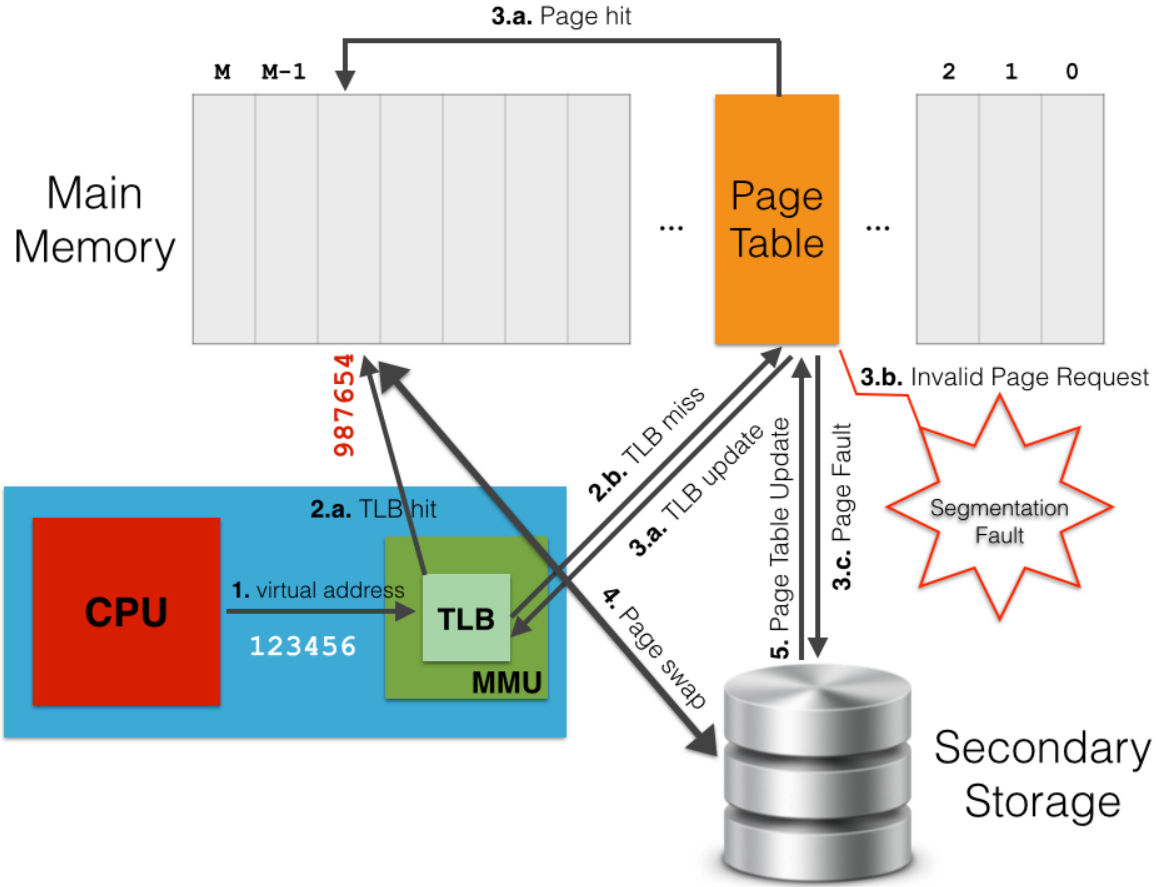
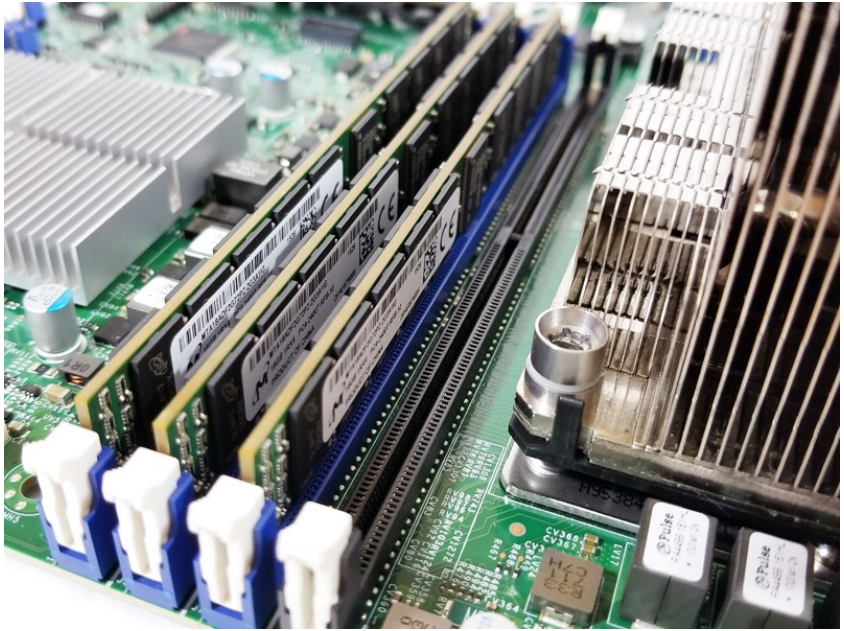


Image: Gabriele Tolomei, Virtual Memory, Paging, and Swapping, https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/virtual-memory-paging-and-swapping/

Advantages



Supermicro X11DPi NT Memory Slots

- Kernel manages free space, no program logic needed!
- Just add more “RUM”*) to speed up your index!

  (but don’t increase Java heap!)

*) Simon Willnauer

Apache Lucene and MMAP

Apache Lucene and MMAP

Apache Lucene preferred way to access fulltext indexes *(10th of Gigabytes, sometimes up to half of a Terabyte)* is via MMAP

`MMapDirectory` abstraction:

- `IndexOutput` (just writes to `Files.newOutputStream`)
- `IndexInput` (`MappedByteBuffer` in 1 GiB sized chunks, sequential access and random access using long pointers)

Why chunks of 1 GiB?

- Java's ByteBuffer has **32 bit (signed)** offsets and size
  - buffer's size must be <= $2^{31} - 1$
  - maximum offset/limit: $2^{31} - 2$
- For efficiency, mapping from long file offset to ByteBuffer chunk using bitshift (>>) and bitmask (&)
- Largest possible chunk size is 2^{30}
  - *damn the above minusone issue!!!*

Why chunks of 1 GiB?

```
295    @Override
296    public byte readByte(long pos) throws IOException {
297      try {
298        final int bi = (int) (pos >> chunkSizePower);
299        return guard.getByte(buffers[bi], (int) (pos & chunkSizeMask));
300      } catch (
301          @SuppressWarnings("unused")
302          IndexOutOfBoundsException ioobe) {
303        throw new EOFException("seek past EOF: " + this);
304      } catch (
305          @SuppressWarnings("unused")
306          NullPointerException npe) {
307        throw new AlreadyClosedException("Already closed: " + this);
308      }
309    }
```

) offsets and size

le offset to

and bitmask (&)

- *damn the above minusone issue!!!*

Why chunks of 1 GiB?

- Java's ByteBuffer has **32 bit (signed)** offsets and size
  - buffer's size must be <= $2^{31} - 1$
  - maximum offset/limit: $2^{31} - 2$
- For efficiency, mapping from long file offset to ByteBuffer chunk using bitshift (>>) and bitmask (&)
- Largest possible chunk size is 2^{30}
  - *damn the above minusone issue!!!*

Try-catch program logic

```
228     @Override
229     public final int readInt() throws IOException {
230       try {
231         return guard.getInt(curBuf);
232       } catch (
233           @SuppressWarnings("unused")
234           BufferUnderflowException e) {
235         return super.readInt();
236       } catch (
237           @SuppressWarnings("unused")
238           NullPointerException npe) {
239         throw new AlreadyClosedException("Already closed: " + this);
240       }
241     }
```

logic

```java
88    @Override
89    public final byte readByte() throws IOException {
90      try {
91        return guard.getByte(curBuf);
92      } catch (
93          @SuppressWarnings("unused")
94          BufferUnderflowException e) {
95        do {
96          curBufIndex++;
97          if (curBufIndex >= buffers.length) {
98            throw new EOFException("read past EOF: " + this);
99          }
100         setCurBuf(buffers[curBufIndex]);
101         curBuf.position(0);
102       } while (!curBuf.hasRemaining());
103       return guard.getByte(curBuf);
104     } catch (
105         @SuppressWarnings("unused")
106         NullPointerException npe) {
107       throw new AlreadyClosedException("Already closed: " + this);
108     }
109   }
```

```java
dInt() throws IOException {

Int(curBuf);

ngs("unused")
wException e) {
dInt();

ngs("unused")
eption npe) {
yClosedException("Already closed: " + this);
```

More Lucene I/O characteristics

- Files in Index are **write-once, read-many** ◎
- Highly concurrent: Every query execution (possibly parallelized) uses iterator-like APIs on “cloned” IndexInputs
  - duplicated ByteBuffers!
  - no synchronization!

The huge problem since early days

- Several threads work on the same memory-mapped area
- Is there a way to close file / unmap data?
  - Unmapping causes **SIGSEGV** on access *(fatal page fault)*
  - Every thread has non-volatile pointer to memory

The huge problem since early days

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00007f0b024734cd, pid=21947, tid=139676677560592
#
# JRE version: OpenJDK Runtime Environment (11.0.6+10) (build 11.0.6+10)
# Java VM: OpenJDK 64-Bit Server VM (11.0.6+10, mixed mode, tiered, linux-amd64)
# Problematic frame:
# V [libjvm.so+0xd55724]
#
# No core dump will be written. Core dumps have been disabled. To enable core
# dumping, try "ulimit -c unlimited" before starting Java again.
#
# If you would like to submit a bug report, please visit:
#   https://github.com/AdoptOpenJDK/openjdk-support/issues
#
```

– Every thread has non-volatile pointer to memory

The 19 year old feature request

https://bugs.openjdk.java.net/browse/JDK-4724038

(2002-07-31)

JDK-4724038: (fs) Add unmap method to MappedByteBuffer

Workaround

- Manual unmap with hacking into JDK internals:
  - Garbage collector calls some private method when `ByteBuffer` gets unreachable
  - Java 9 moved this method to `sun.misc.Unsafe` to work around module system restrictions
- **Risk:** Crushing the JVM when query is running and IndexReader is closed

How to use

- On **64 bit platforms** that allow the unmap hack to work => `FSDirectory.open()` automatically uses it
- **Elasticsearch / Opensearch** uses it by default for most index files
- **Solr** adapts Lucene factory

```
top - 16:25:17 up 2 days, 18:01,  1 user,  load average: 0.12, 0.09, 0.10
Tasks: 273 total,   1 running, 272 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.7 sy,  0.0 ni, 99.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  1.0 us,  0.3 sy,  0.0 ni, 98.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  3.7 us,  0.0 sy,  0.0 ni, 95.0 id,  1.0 wa,  0.0 hi,  0.3 si,  0.0 st
%Cpu3  :  0.0 us,  1.3 sy,  0.0 ni, 95.3 id,  3.4 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :  0.7 us,  0.7 sy,  0.0 ni, 98.0 id,  0.7 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :  4.7 us,  0.0 sy,  0.0 ni, 95.0 id,  0.3 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  32089.9 total,    233.1 free,  20275.1 used,  11581.6 buff/cache
MiB Swap:  20479.0 total,  18928.0 free,   1551.0 used.  11162.0 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    967 elastic+  20   0   73.6g   7.6g   2.0g S   0.7  24.1 234:50.22 java -Xms5g -Xm
```

```
# curl 'http://localhost:9200/_cat/indices?v=true'
health status index  uuid                   pri rep docs.count docs.deleted store.size pri.store.size
green  open   index1 chOaj1tLTBicG-DvrHgqGQ   1   0     368616        72223     202.4mb        202.4mb
green  open   index2 EaRSsYRwSMmf9weLCLl6cA   1   0    1409152       606915     972.5mb        972.5mb
green  open   index3 ceoTplPzS8GztuKIvqo3QA   1   0     214495            0     112.5mb        112.5mb
green  open   index4 UOun5qoAS8yu5L61p6QsCQ   2   0   15709471      3611144      47.3gb         47.3gb
green  open   index5 z3EKwUHdRZG2WHVwKR51Aw   5   0     403872         8392      15.4gb         15.4gb
#
```

Tuning

- Assign as less as possible JVM heap (max. 25% of physical RAM). **Trust me!**
- `sysctl -w vm.max_map_count=262144`
- Only run Lucene/Solr/Elasticsearch on node
- Don’t use Windows if you have huge indexes! (all indexes per node in total >> 1 TiB)

Photo: https://www.tuningblog.eu/kategorien/tuning-wiki/upside-down-car-269876/

Interconnecting JVM and native code

Project Panama

Project Panama

"Improving and enriching the connections between the Java virtual machine and well-defined but "foreign" (non-Java) APIs, including many interfaces commonly used by C programmers."

(https://openjdk.java.net/projects/panama/)

Key concepts

- Key concepts:
  - `MemorySegment` on top of `MemoryAddress`
  - `VarHandle` for access without overhead
- `MemorySegment` can cover any address space (on heap, off-heap, memory mapped) - just like `ByteBuffer`
- Positional API using `long`!

Project Panama
Incubation: JDK 14 (JEP 370)

Thread confinement: `MemorySegment` instance can only used by one thread!

- **From Lucene's point of view:** Open file, mmap file to segment, execute query, unmap segment in one thread (inside try-with-resources)
- **No way to use with Lucene!**

Project Panama
Incubation: JDK 15 (JEP 383)

Thread confinement update:

- Allow to create clones and transfer ownership of `MemorySegment` to other threads
- Mapped file and allocated address space is only released when last duplicate is **closed**!
- **Again: No way to use with Lucene**

State after JDK 15

Think of `MemorySegment` as 64 bits `ByteBuffer`

But you can use it only from one thread!

Of course, this fixes the unmap problem!

Thread confinement solution

https://bugs.openjdk.java.net/browse/JDK-4724038

Andrew Haley added a comment - 2016-12-12 06:58 - edited

It is possible to unmap a MappedByteBuffer safely with little effect on efficiency: my basic idea is to add an indirection which all callers (even those using derived buffers) have to use to access the memory, and then removing most uses of this indirection with a little compiler magic. A few will still remain, but these will be the minimum required to guarantee security and won't affect the speed of most operations.

I have a plan to implement this, but it requires changes to the Java Memory Model and to the compilers. I didn't get it done in time for JDK 9 because of being distracted by a ton of other things, but I hope I'll get it done by JDK 10.

I'm taking this bug; I hope that's OK.

Project Panama
Incubation: JDK 16 (JEP 393)

- Shared `MemorySegment`!
- Safely unmappable with public API: `close()`
- Other threads still using the segment get `IllegalStateException`

State after JDK 16

No slowdown during access to shared MemorySegment

Global JVM impact during unmapping / close of file: **All threads get stopped!**

We have to live with it!

Apache Lucene: First Pull Request

New Year: https://github.com/apache/lucene-solr/pull/2176

- Passes all tests 👍
- Similar code to `ByteBufferIndexInput`, but heavy cleanup: `MemorySegmentIndexInput`
- Default chunk size now 16 GiB
  - Most index files will be one segment only
  - Dramatic reduction of mappings
- Cleanup: https://github.com/apache/lucene/pull/173 *(endianness)*

Apache Lucene: First Pull Request

```java
import jdk.incubator.foreign.MemorySegment;

/**
 * Base IndexInput implementation that uses an array of MemorySegments to represent a file.
 *
 * <p>For efficiency, this class requires that the segment size are a power-of-two
 * (<code>chunkSizePower</code>).
 */
public abstract class MemorySegmentIndexInput extends IndexInput implements RandomAccessInput {
  // We pass 1L as alignment, because currently Lucene file formats are heavy unaligned: :(
  static final VarHandle VH_getByte = MemoryHandles.varHandle(byte.class, 1L, ByteOrder.BIG_ENDIAN).withInvokeExactBehavior();
  static final VarHandle VH_getShort = MemoryHandles.varHandle(short.class, 1L, ByteOrder.BIG_ENDIAN).withInvokeExactBehavior();
  static final VarHandle VH_getInt = MemoryHandles.varHandle(int.class, 1L, ByteOrder.BIG_ENDIAN).withInvokeExactBehavior();
  static final VarHandle VH_getLong = MemoryHandles.varHandle(long.class, 1L, ByteOrder.BIG_ENDIAN).withInvokeExactBehavior();

  static final boolean IS_LITTLE_ENDIAN = (ByteOrder.nativeOrder() == ByteOrder.LITTLE_ENDIAN);
```

  - Dramatic reduction of mappings
- Cleanup: https://github.com/apache/lucene/pull/173 *(endianness)*

Apache Lucen

```java
import jdk.incubator.foreign.MemorySegment;

/**
 * Base IndexInput implementation that uses an array
 *
 * <p>For efficiency, this class requires that the se
 * (<code>chunkSizePower</code>).
 */
public abstract class MemorySegmentIndexInput extends
  // We pass 1L as alignment, because currently Lucen
  static final VarHandle VH_getByte = MemoryHandles.v
  static final VarHandle VH_getShort = MemoryHandles.varHandle(short.class, 1L, ByteOrder.BIG_ENDIAN).withInvokeExactBehavior();
  static final VarHandle VH_getInt = MemoryHandles.varHandle(int.class, 1L, ByteOrder.BIG_ENDIAN).withInvokeExactBehavior();
  static final VarHandle VH_getLong = MemoryHandles.varHandle(long.class, 1L, ByteOrder.BIG_ENDIAN).withInvokeExactBehavior();

  static final boolean IS_LITTLE_ENDIAN = (ByteOrder.nativeOrder() == ByteOrder.LITTLE_ENDIAN);
```

```java
@Override
public final int readInt() throws IOException {
  try {
    final int v = (int) VH_getInt.get(curSegment, curPosition);
    curPosition += Integer.BYTES;
    return v;
  } catch (IndexOutOfBoundsException e) {
    return super.readInt();
  } catch (NullPointerException | IllegalStateException e) {
    throw wrapAlreadyClosedException(e);
  }
}
```

  - Dramatic reduction of mappings
- Cleanup: https://github.com/apache/lucene/pull/173 *(endianness)*

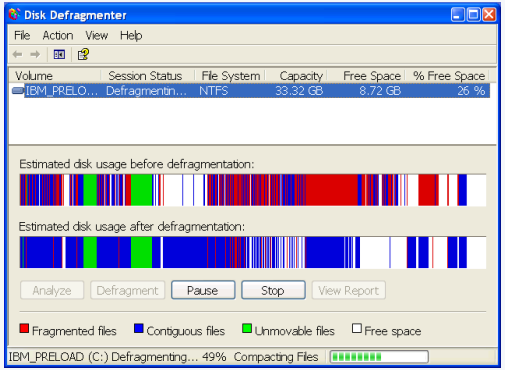
Apache Lucene: First Pull Request

New Year: https://github.com/apache/lucene-solr/pull/2176

- Passes all tests 👍
- Similar code to `ByteBufferIndexInput`, but heavy cleanup: `MemorySegmentIndexInput`
- Default chunk size now 16 GiB
  - Most index files will be one segment only
  - Dramatic reduction of mappings
- Cleanup: https://github.com/apache/lucene/pull/173 *(endianness)*

Why chunking?

Address space fragmentation!



It’s going away at some point with newer hardware and kernels anyways!

Project Panama
Incubation: JDK 17 (JEP 412)

- Just 2 weeks ago; since build 25 of JDK 17
- Incubation may end soon, API nice and stable
- Performance gets better!
- Memory / resource scopes:
  - `MemorySegment` no longe `AutoCloseable`
  - `ResourceScope` control when unmapping occurs

Apache Lucene: Third Pull Request

Last week: https://github.com/apache/lucene/pull/177

- Passes all tests 👍
- Still performs slower for certain workloads:
  - Direct access using `VarHandle` is fast
  - Bulk copies to Java arrays are slow. Cause known, fix in JDK is ongoing!

Possible future (Lucene 10+)

- **Use Panama API in Lucene!**
- Extend `Directory`/`IndexInput` API to let codec **lock slices of files into memory**
- Remove remaining parts of buffer copy
- Use **vector API** *(JEP 338)* in combination with Panama to run calculations off-heap!

Image: https://wallpapersafari.com/w/9pcfhV

THANK YOU!

Questions?

https://blog.thetaphi.de/2012/07/use-lucenes-mmapdirectory-on-64bit.html