import java.nio.ByteBuffer

import java.nio.file.StandardOpenOption

internal class JPEGMetaWriter(private val uniFile: UniFile) : JPEGMetaBase(uniFile), ImageMetaWriter {

override fun reserveXMP(size: Int): ImageMetaReader.Box {

SLog.d(TAG, "reserveXMP: size=$size")

return try {

reserveXMPByMove(size)

} catch (e: Exception) {

SLog.w(TAG, "reserveXMPByMove failed, fallback to copy: ${e.message}")

reserveXMPByCopy(size)

}

}

private fun reserveXMPByMove(size: Int): ImageMetaReader.Box {

SLog.i(TAG, "reserveXMPByMove:

$size")

return uniFile.useFileChannel(StandardOpenOption.READ, StandardOpenOption.WRITE) { ch ->

var xmpBox: ImageMetaReader.Box? = null

// skip SOI

ch.position(JPEG\_MARKER\_SIZE.toLong())

while (findNextApp1Marker(ch) > 0) {

val (metaType, position, payload) = getMeta(ch)

when (metaType) {

MetaType.META\_EXIF -> {

SLog.i(TAG, "found exif: pos=$position, payload=$payload")

// position is the start of

payload (as in old code),

// the APP1 marker start is position - marker overhead

val xmpAPP1Position = position + payload // original new code — WRONG

// Fix: compute app1 marker start and where XMP should be inserted AFTER exif APP1

val exifApp1Start = position - (JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE)

val exifApp1Payload = payload + (JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE)

// We want to insert new APP1 (XMP) right after EXIF APP1 region:

val insertApp1Start = exifApp1Start + exifApp1Payload

// shift right from

insertApp1Start by the new APP1 total size

ch.position(insertApp1Start)

ch.shiftRightSafely(size + (JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE))

// now write APP1 marker + length at insertApp1Start, and set xmpPosition to payload area

ch.position(insertApp1Start)

ch.write(JPEG\_APP1\_MARKER)

ch.write((size + JPEG\_LENGTH\_SIZE).toShort())

val xmpPosition = insertApp1Start + (JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE)

// position in file where xmp payload starts

xmpBox =

ImageMetaReader.Box(xmpPosition, size.toLong(), byteReader)

}

MetaType.META\_XMP -> {

SLog.i(TAG, "found xmp: pos=$position, payload=$payload")

// existing xmp's APP1 start:

val xmpApp1Start = position - (JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE)

val xmpApp1Payload = payload + (JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE)

// remove existing APP1 (skip it) by shift-left the region after it to overwrite

ch.position(xmpApp1Start)

ch.shiftLeftSafely(xmpApp1Payload.toInt())

// after shiftLeft, position stays at where new APP1 can be inserted (xmpApp1Start)

ch.position(xmpApp1Start)

// continue scanning from this position

}

else -> {

ch.skipNBytes(payload.toInt())

}

}

}

if (xmpBox == null) {

// no exif found — insert XMP right after SOI marker

val xmpAPP1Position = JPEG\_MARKER\_SIZE.toLong()

ch.position(xmpAPP1Position)

ch.shiftRightSafely(size + (JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE))

ch.position(xmpAPP1Position)

ch.write(JPEG\_APP1\_MARKER)

ch.write((size + JPEG\_LENGTH\_SIZE).toShort())

val xmpPosition = xmpAPP1Position + (JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE)

xmpBox = ImageMetaReader.Box(xmpPosition, size.toLong(), byteReader)

}

xmpBox!!

}

}

private fun reserveXMPByCopy(size: Int): ImageMetaReader.Box {

SLog.d(TAG, "reserveXMPByCopy: $size")

val (jpegBuffer, xmpPosition) = uniFile.newInputFileStream().use { input ->

var buffer: ByteBuffer? = null

var offset = JPEG\_MARKER\_SIZE.toLong()

input.channel.position(offset)

while (findNextApp1Marker(input) > 0) {

val (metaType, position, payload) = getMeta(input)

when (metaType) {

MetaType.META\_EXIF -> {

SLog.d(TAG, "found exif: pos=$position, payload=$payload")

// compute APP1 start & payload (including marker+length)

val app1Position = position - (JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE)

val app1Payload = payload + (JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE)

// set read position to offset and read up to end, but we will inject XMP after this EXIF APP1

input.channel.position(app1Position + app1Payload)

val remains = (input.channel.size() - input.channel.position()).toInt()

// allocate buffer to hold from app1Position..end plus new APP1

header+payload

buffer = ByteBuffer.allocate((app1Position + app1Payload - offset).toInt() + JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE + remains).apply {

// first copy bytes from offset..(app1Position+app1Payload)

input.channel.position(offset)

val beforeLen = (app1Position + app1Payload - offset).toInt()

val tmpBefore = ByteArray(beforeLen)

input.read(tmpBefore, 0, beforeLen)

put(tmpBefore)

// put new APP1 marker + length for XMP

putShort(JPEG\_APP1\_MARKER)

putShort((size + JPEG\_LENGTH\_SIZE).toShort())

// then copy the remainder (from app1Position+app1Payload .. end)

val tmpRemains = ByteArray(remains)

input.read(tmpRemains, 0, remains)

put(tmpRemains)

flip()

}

// xmp payload position is: (current put position) = offset + beforeLen + JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE - but we want file offset

val xmpPayloadFilePos = (offset + (app1Position + app1Payload - offset)) + (JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE)

offset = app1Position + app1Payload

return@use buffer to xmpPayloadFilePos

}

MetaType.META\_XMP -> {

SLog.d(TAG, "found xmp: pos=$position, payload=$payload")

// If existing XMP found, we should handle replace case.

// For simplicity in copy mode: remove existing APP1 (skip it) and later insert new.

val app1Position = position - (JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE)

val app1Payload = payload +

(JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE)

// read from offset to app1Position (copying everything up to existing xmp)

input.channel.position(offset)

val beforeLen = (app1Position - offset).toInt()

val tmpBefore = ByteArray(beforeLen)

input.read(tmpBefore, 0, beforeLen)

val remains = (input.channel.size() - (app1Position + app1Payload)).toInt()

val tmpRemains = ByteArray(remains)

input.channel.position(app1Position + app1Payload)

input.read(tmpRemains, 0, remains)

// Build buffer: [bytes before existing xmp] + [new APP1 header+len] + [remainder]

buffer = ByteBuffer.allocate(beforeLen + JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE + remains).apply {

put(tmpBefore)

putShort(JPEG\_APP1\_MARKER)

putShort((size + JPEG\_LENGTH\_SIZE).toShort())

put(tmpRemains)

flip()

}

val xmpPayloadFilePos = offset + beforeLen + (JPEG\_MARKER\_SIZE

+ JPEG\_LENGTH\_SIZE)

return@use buffer to xmpPayloadFilePos

}

else -> {

input.skipNBytes(payload)

}

}

}

// If we reach here and didn't find EXIF or XMP — insert after SOI

// read whole file and create new buffer with XMP inserted after SOI

input.channel.position(0)

val whole = ByteArray(input.channel.size().toInt())

input.read(whole, 0, whole.size)

val bufferAll = ByteBuffer.allocate(whole.size + JPEG\_MARKER\_SIZE +

JPEG\_LENGTH\_SIZE + size).apply {

// put SOI

putShort(JPEG\_SOI\_MARKER)

put(whole, 2, whole.size - 2) // after SOI put the rest

// But better: assemble properly: put whole up to SOI, then APP1...

// Simpler and safe approach: recreate as: SOI (2 bytes) + APP1(XMP) + rest

rewind()

}

// fallback: simpler - return null to trigger different branch

throw IllegalStateException("reserveXMPByCopy: unexpected path - no EXIF/XMP found")

}

// write prepared buffer into file

uniFile.useOutputFileChannel { out ->

val xmpAPP1Position = xmpPosition - (JPEG\_MARKER\_SIZE + JPEG\_LENGTH\_SIZE)

out.position(xmpAPP1Position)

out.write(jpegBuffer)

out.truncate(xmpAPP1Position + jpegBuffer.limit().toLong())

}

return ImageMetaReader.Box(xmpPosition, size.toLong(), byteReader)

}

override fun removeXMP() {

TODO("Not yet implemented")

}

override fun writeExif(exifInfo: ExifInfo) {

TODO("Not yet implemented")

}

override fun writeXMP(xmpInfo: XMPInfo) {

if (!xmpInfo.edit().isDirty) {

SLog.i(TAG, "xmpInfo is clean, skip write")

return

}

val xmp = (xmpInfo as XMPInfoImpl).xmp

val xmpBuffer = XMPMetaFactory.serializeToBuffer(xmp, SerializeOptions().apply {

omitPacketWrapper = true

useCompactFormat = true

})

val signatureArray = XMP\_SIGNATURE.toByteArray()

val requiredSize = xmpBuffer.size + signatureArray.size

var (xmpPosition, xmpPayload) = findXMPApp1Marker()

SLog.d(TAG, "found xmp pos=$xmpPosition, size=$xmpPayload")

if (xmpPayload >= requiredSize && xmpPosition > 0L) {

// reuse existing space

SLog.i(TAG, "reuse exist xmp: xmpPayload=$xmpPayload, requiredSize=$requiredSize")

uniFile.useOutputFileChannel { out ->

out.position(xmpPosition)

out.write(signatureArray)

out.write(xmpBuffer)

if (xmpPayload.toInt() > requiredSize) {

out.write(0x20, xmpPayload.toInt() - requiredSize)

}

}

} else {

// need to reserve larger space then write

val box = reserveXMP(maxOf(XMP\_RESERVED\_SIZE, requiredSize))

xmpPosition = box.offset

xmpPayload = box.length

uniFile.useOutputFileChannel { out ->

SLog.i(TAG, "write xmp at $xmpPosition w/ size=${signatureArray.size + xmpBuffer.size}")

out.position(xmpPosition)

// write signature + payload

out.write(signatureArray)

out.write(xmpBuffer)

// pad remaining bytes in reserved payload if any

if (xmpPayload.toInt() > (signatureArray.size + xmpBuffer.size)) {

out.write(0x20, xmpPayload.toInt() - (signatureArray.size + xmpBuffer.size))

}

}

}

}

private fun findXMPApp1Marker(): Pair<Long, Long> {

uniFile.newInputFileStream().use { input ->

input.channel.position(JPEG\_MARKER\_SIZE.toLong()) //skip SOI marker

while (findNextApp1Marker(input) > 0) {

val (metaType, position, payload) = getMeta(input)

if (metaType ==

MetaType.META\_XMP) {

SLog.d(TAG, "found xmp on position=$position, payload=$payload")

return position to payload

}

input.skipNBytes(payload)

}

return 0L to 0L

}

}

companion object {

private val TAG = SLog.tagOf(JPEGMetaWriter::class.java)

}

}