UCARE Research

Ray Andrew

October 15, 2019

Abstract

The abstract text goes here.

Contents

0.1 Todos . . . 2
0.1.1 Subsect . . . 2
0.2 Summarization . . . 3
0.2.1 JVM Hotspot . . . 3
0.2.1.1 Oop Hierarchy . . . 3
0.2.1.2 Array Allocation . . . 3
0.2.1.3 Array Copy Mechanism . . . 3
0.2.1.4 Type Array Inheritance . . . 22
0.2.1.5 TLAB . . . 22
0.2.1.5.1 Inheritance . . . 22
0.2.1.5.2 Initialization . . . 22
0.2.1.5.3 Notes . . . 23
0.3 Notes . . . 24
0.3.1 Tuesday, 08 October 2019 . . . 25
0.3.1.1 Kafka Bug Study . . . 25
0.3.1.2 JVM Deduplication . . . 25
0.3.2 Friday, 04 October 2019 . . . 26
0.3.2.1 Allocation Classes . . . 26
0.3.2.2 Notes . . . 26
0.4 Resources . . . 27
0.4.1 Kernel Samepage Merging . . . 27

0.1 Todos

0.1.1 Subsect

Write your subsection text here.

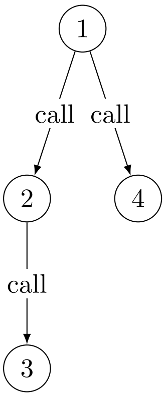
0.2 Summarization

0.2.1 JVM Hotspot

0.2.1.1 Oop Hierarchy

```
typedef class oopDesc*                            oop;
typedef class   instanceOopDesc*            instanceOop;
typedef class   arrayOopDesc*                    arrayOop;
typedef class     objArrayOopDesc*            objArrayOop;
typedef class     typeArrayOopDesc*            typeArrayOop;
```

0.2.1.2 Array Allocation



1. `void OptoRuntime::new_array_C(Klass* array_type, int len, JavaThread *thread)`
2. `typeArrayOop oopFactory::new_typeArray(BasicType type, int length, TRAPS)`
3. `TypeArrayKlass* TypeArrayKlass::allocate(ClassLoaderData* loader_data, BasicType type, Symbol* name, TRAPS)`
4. `TypeArrayKlass* TypeArrayKlass::allocate(ClassLoaderData* loader_data, BasicType type, Symbol* name, TRAPS)`
5. `objArrayOop oopFactory::new_objArray(Klass* klass, int length, TRAPS)`

0.2.1.3 Array Copy Mechanism

- hotspot/share/prism/jvm.cpp

```
JVM_ENTRY(void, JVM_ArrayCopy(JNIEnv *env, jclass ignored,
↪  jobject src, jint src_pos, jobject dst, jint dst_pos, jint
↪  length))
  JVMWrapper("JVM_ArrayCopy");
  // Check if we have null pointers
  if (src == NULL || dst == NULL) {
    THROW(vmSymbols::java_lang_NullPointerException());
  }

  arrayOop s = arrayOop(JNIHandles::resolve_non_null(src));
  arrayOop d = arrayOop(JNIHandles::resolve_non_null(dst));
  assert(oopDesc::is_oop(s), "JVM_ArrayCopy: src not an oop");
  assert(oopDesc::is_oop(d), "JVM_ArrayCopy: dst not an oop");
  // Do copy
  s->klass()->copy_array(s, src_pos, d, dst_pos, length,
   ↪  thread);
JVM_END
```

- hotspot/share/oops/arrayOopDesc.hpp
  based on the subsubsection 0.2.1.1, array oopDesc can be both typeArrayOopDesc or objArrayOopDesc

```
class arrayOopDesc : public oopDesc { ... }

// so basically oopDesc have attribute method call klass()
Klass* oopDesc::klass() const {
  if (UseCompressedClassPointers) {
    return
     ↪  Klass::decode_klass_not_null(_metadata._compressed_klass);
  } else {
    return _metadata._klass;
  }
}
```

- hotspot/share/oops/typeArrayKlass.hpp

```
void TypeArrayKlass::copy_array(arrayOop s, int src_pos,
 ↪  arrayOop d, int dst_pos, int length, TRAPS) {
  assert(s->is_typeArray(), "must be type array");

  // Check destination type.
  if (!d->is_typeArray()) {
    ResourceMark rm(THREAD);
    stringStream ss;
    if (d->is_objArray()) {
      ss.print("arraycopy: type mismatch: can not copy %s[]
       ↪  into object array[]",
       ↪  type2name_tab[ArrayKlass::cast(s->klass())->element_type()]);
    } else {
      ss.print("arraycopy: destination type %s is not an
       ↪  array", d->klass()->external_name());
    }

    THROW_MSG(vmSymbols::java_lang_ArrayStoreException(),
     ↪  ss.as_string());
```

```
}

if (element_type() !=
↪  TypeArrayKlass::cast(d->klass())->element_type()) {
  ResourceMark rm(THREAD);
  stringStream ss;
  ss.print("arraycopy: type mismatch: can not copy %s[] into
  ↪  %s[]",

  ↪  type2name_tab[ArrayKlass::cast(s->klass())->element_type()],

  ↪  type2name_tab[ArrayKlass::cast(d->klass())->element_type()]);
  THROW_MSG(vmSymbols::java_lang_ArrayStoreException(),
  ↪  ss.as_string());
}

// Check if all offsets and lengths are non negative.
if (src_pos < 0 || dst_pos < 0 || length < 0) {
  // Pass specific exception reason.
  ResourceMark rm(THREAD);
  stringStream ss;
  if (src_pos < 0) {
    ss.print("arraycopy: source index %d out of bounds for
    ↪  %s[%d]", src_pos,
    ↪  type2name_tab[ArrayKlass::cast(s->klass())->element_type()],
    ↪  s->length());
  } else if (dst_pos < 0) {
    ss.print("arraycopy: destination index %d out of bounds
    ↪  for %s[%d]", dst_pos,
    ↪  type2name_tab[ArrayKlass::cast(d->klass())->element_type()],
    ↪  d->length());
  } else {
    ss.print("arraycopy: length %d is negative", length);
  }

  ↪  THROW_MSG(vmSymbols::java_lang_ArrayIndexOutOfBoundsException(),
  ↪  ss.as_string());
}

// Check if the ranges are valid
if ((((unsigned int) length + (unsigned int) src_pos) >
↪  (unsigned int) s->length()) ||
(((unsigned int) length + (unsigned int) dst_pos) > (unsigned
↪  int) d->length())) {

  // Pass specific exception reason.
  ResourceMark rm(THREAD);
  stringStream ss;
  if (((unsigned int) length + (unsigned int) src_pos) >
  ↪  (unsigned int) s->length()) {
    ss.print("arraycopy: last source index %u out of bounds
    ↪  for %s[%d]", (unsigned int) length + (unsigned int)
    ↪  src_pos,
    ↪  type2name_tab[ArrayKlass::cast(s->klass())->element_type()],
    ↪  s->length());
  } else {
    ss.print("arraycopy: last destination index %u out of
    ↪  bounds for %s[%d]", (unsigned int) length + (unsigned
    ↪  int) dst_pos,
    ↪  type2name_tab[ArrayKlass::cast(d->klass())->element_type()],
    ↪  d->length());
  }


  ↪  THROW_MSG(vmSymbols::java_lang_ArrayIndexOutOfBoundsException(),
  ↪  ss.as_string());
}


// Check zero copy
if (length == 0)
  return;

// This is an attempt to make the copy_array fast.
int l2es = log2_element_size();
size_t src_offset =
    ↪  arrayOopDesc::base_offset_in_bytes(element_type()) +
    ↪  ((size_t)src_pos << l2es);
size_t dst_offset =
    ↪  arrayOopDesc::base_offset_in_bytes(element_type()) +
    ↪  ((size_t)dst_pos << l2es);
ArrayAccess<ARRAYCOPY_ATOMIC>::arraycopy<void>(s, src_offset,
↪  d, dst_offset, (size_t)length << l2es);
}
```

- hotspot/share/oops/access.hpp

```
// Helper for array access.
template <DecoratorSet decorators = INTERNAL_EMPTY>
class ArrayAccess: public HeapAccess<IS_ARRAY | decorators> {
 typedef HeapAccess<IS_ARRAY | decorators> AccessT;
 public:
 template <typename T>
 static inline void arraycopy(arrayOop src_obj, size_t
 ↪  src_offset_in_bytes, arrayOop dst_obj, size_t
 ↪  dst_offset_in_bytes, size_t length) {
  AccessT::arraycopy(src_obj, src_offset_in_bytes,
  ↪  reinterpret_cast<const T*>(NULL), dst_obj,
  ↪  dst_offset_in_bytes, reinterpret_cast<T*>(NULL),
  ↪  length);
 }

 template <typename T>
 static inline void arraycopy_to_native(arrayOop src_obj,
 ↪  size_t src_offset_in_bytes, T* dst, size_t length) {
  AccessT::arraycopy(src_obj, src_offset_in_bytes,
  ↪  reinterpret_cast<const T*>(NULL), NULL, 0, dst,
  ↪  length);
 }

 template <typename T>
 static inline void arraycopy_from_native(const T* src,
 ↪  arrayOop dst_obj, size_t dst_offset_in_bytes, size_t
 ↪  length) {
  AccessT::arraycopy(NULL, 0, src, dst_obj,
  ↪  dst_offset_in_bytes, reinterpret_cast<T*>(NULL),
  ↪  length);
 }

 static inline bool oop_arraycopy(arrayOop src_obj, size_t
 ↪  src_offset_in_bytes, arrayOop dst_obj, size_t
 ↪  dst_offset_in_bytes, size_t length) {
  return AccessT::oop_arraycopy(src_obj, src_offset_in_bytes,
  ↪  reinterpret_cast<const HeapWord*>(NULL), dst_obj,
  ↪  dst_offset_in_bytes, reinterpret_cast<HeapWord*>(NULL),
  ↪  length);
 }

 template <typename T>
 static inline bool oop_arraycopy_raw(T* src, T* dst, size_t
 ↪  length) {
  return AccessT::oop_arraycopy(NULL, 0, src, NULL, 0, dst,
  ↪  length);
 }

};
```

- hotspot/share/oops/access.hpp

```
template <DecoratorSet decorators = INTERNAL_EMPTY>
class Access: public AllStatic {
  // ...


  protected:
  template <typename T>
  static inline bool oop_arraycopy(arrayOop src_obj, size_t
  ↪  src_offset_in_bytes, const T* src_raw, arrayOop dst_obj,
  ↪  size_t dst_offset_in_bytes, T* dst_raw, size_t length) {
    verify_decorators<ARRAYCOPY_DECORATOR_MASK | IN_HEAP |
    ↪   AS_DECORATOR_MASK | IS_ARRAY |
    ↪   IS_DEST_UNINITIALIZED>();
    return AccessInternal::arraycopy<decorators |
    ↪   INTERNAL_VALUE_IS_OOP>(src_obj, src_offset_in_bytes,
    ↪   src_raw, dst_obj, dst_offset_in_bytes, dst_raw,
    ↪   length);
  }

  template <typename T>
  static inline void arraycopy(arrayOop src_obj, size_t
  ↪   src_offset_in_bytes, const T* src_raw, arrayOop dst_obj,
  ↪   size_t dst_offset_in_bytes, T* dst_raw, size_t length) {
    verify_decorators<ARRAYCOPY_DECORATOR_MASK | IN_HEAP |
    ↪   AS_DECORATOR_MASK | IS_ARRAY>();
    AccessInternal::arraycopy<decorators>(src_obj,
    ↪   src_offset_in_bytes, src_raw, dst_obj,
    ↪   dst_offset_in_bytes, dst_raw, length);
  }
}
```

- hotspot/share/oops/accessBackend.inline.hpp

```
class RawAccessBarrierArrayCopy: public AllStatic {
  template<typename T> struct IsHeapWordSized: public
  ↪   IntegralConstant<bool, sizeof(T) == HeapWordSize> { };
public:
  template <DecoratorSet decorators, typename T>
  static inline typename EnableIf<
    HasDecorator<decorators,
    ↪   INTERNAL_VALUE_IS_OOP>::value>::type
  arraycopy(arrayOop src_obj, size_t src_offset_in_bytes, T*
  ↪   src_raw,
            arrayOop dst_obj, size_t dst_offset_in_bytes, T*
            ↪   dst_raw,
            size_t length) {
    src_raw = arrayOopDesc::obj_offset_to_raw(src_obj,
    ↪   src_offset_in_bytes, src_raw);
    dst_raw = arrayOopDesc::obj_offset_to_raw(dst_obj,
    ↪   dst_offset_in_bytes, dst_raw);

    // We do not check for ARRAYCOPY_ATOMIC for oops, because
    ↪   they are unconditionally always atomic.
    if (HasDecorator<decorators, ARRAYCOPY_ARRAYOF>::value) {
      AccessInternal::arraycopy_arrayof_conjoint_oops(src_raw,
      ↪   dst_raw, length);
    } else {
      typedef typename HeapOopType<decorators>::type OopType;
  ↪ AccessInternal::arraycopy_conjoint_oops(reinterpret_cast<OopType*>(src_raw),
  ↪ reinterpret_cast<OopType*>(dst_raw), length);
 }
}

template <DecoratorSet decorators, typename T>
static inline typename EnableIf<
 !HasDecorator<decorators, INTERNAL_VALUE_IS_OOP>::value &&
 HasDecorator<decorators, ARRAYCOPY_ARRAYOF>::value>::type
arraycopy(arrayOop src_obj, size_t src_offset_in_bytes, T*
↪ src_raw,
     arrayOop dst_obj, size_t dst_offset_in_bytes, T*
     ↪ dst_raw,
     size_t length) {
 src_raw = arrayOopDesc::obj_offset_to_raw(src_obj,
 ↪ src_offset_in_bytes, src_raw);
 dst_raw = arrayOopDesc::obj_offset_to_raw(dst_obj,
 ↪ dst_offset_in_bytes, dst_raw);

 AccessInternal::arraycopy_arrayof_conjoint(src_raw,
 ↪ dst_raw, length);
}

template <DecoratorSet decorators, typename T>
static inline typename EnableIf<
 !HasDecorator<decorators, INTERNAL_VALUE_IS_OOP>::value &&
 HasDecorator<decorators, ARRAYCOPY_DISJOINT>::value &&
 ↪ IsHeapWordSized<T>::value>::type
arraycopy(arrayOop src_obj, size_t src_offset_in_bytes, T*
↪ src_raw,
     arrayOop dst_obj, size_t dst_offset_in_bytes, T*
     ↪ dst_raw,
     size_t length) {
 src_raw = arrayOopDesc::obj_offset_to_raw(src_obj,
 ↪ src_offset_in_bytes, src_raw);
 dst_raw = arrayOopDesc::obj_offset_to_raw(dst_obj,
 ↪ dst_offset_in_bytes, dst_raw);

 // There is only a disjoint optimization for word
 ↪ granularity copying
 if (HasDecorator<decorators, ARRAYCOPY_ATOMIC>::value) {
  AccessInternal::arraycopy_disjoint_words_atomic(src_raw,
  ↪ dst_raw, length);
 } else {
  AccessInternal::arraycopy_disjoint_words(src_raw,
  ↪ dst_raw, length);
 }
}
```

```
  template <DecoratorSet decorators, typename T>
  static inline typename EnableIf<
    !HasDecorator<decorators, INTERNAL_VALUE_IS_OOP>::value &&
    !(HasDecorator<decorators, ARRAYCOPY_DISJOINT>::value &&
    ↪  IsHeapWordSized<T>::value) &&
    !HasDecorator<decorators, ARRAYCOPY_ARRAYOF>::value &&
    !HasDecorator<decorators, ARRAYCOPY_ATOMIC>::value>::type
  arraycopy(arrayOop src_obj, size_t src_offset_in_bytes, T*
  ↪  src_raw,
            arrayOop dst_obj, size_t dst_offset_in_bytes, T*
            ↪  dst_raw,
            size_t length) {
    src_raw = arrayOopDesc::obj_offset_to_raw(src_obj,
    ↪  src_offset_in_bytes, src_raw);
    dst_raw = arrayOopDesc::obj_offset_to_raw(dst_obj,
    ↪  dst_offset_in_bytes, dst_raw);

    AccessInternal::arraycopy_conjoint(src_raw, dst_raw,
    ↪  length);
  }

  template <DecoratorSet decorators, typename T>
  static inline typename EnableIf<
    !HasDecorator<decorators, INTERNAL_VALUE_IS_OOP>::value &&
    !(HasDecorator<decorators, ARRAYCOPY_DISJOINT>::value &&
    ↪  IsHeapWordSized<T>::value) &&
    !HasDecorator<decorators, ARRAYCOPY_ARRAYOF>::value &&
    HasDecorator<decorators, ARRAYCOPY_ATOMIC>::value>::type
  arraycopy(arrayOop src_obj, size_t src_offset_in_bytes, T*
  ↪  src_raw,
            arrayOop dst_obj, size_t dst_offset_in_bytes, T*
            ↪  dst_raw,
            size_t length) {
    src_raw = arrayOopDesc::obj_offset_to_raw(src_obj,
    ↪  src_offset_in_bytes, src_raw);
    dst_raw = arrayOopDesc::obj_offset_to_raw(dst_obj,
    ↪  dst_offset_in_bytes, dst_raw);

    AccessInternal::arraycopy_conjoint_atomic(src_raw, dst_raw,
    ↪  length);
  }
};
```

- hotspot/share/oops/accessBackend.cpp

```
// These forward copying calls to Copy without exposing the
↪  Copy type in headers unnecessarily

void arraycopy_arrayof_conjoint_oops(void* src, void* dst,
↪  size_t length) {
  Copy::arrayof_conjoint_oops(reinterpret_cast<HeapWord*>(src),
   ↪  reinterpret_cast<HeapWord*>(dst), length);
}

void arraycopy_conjoint_oops(oop* src, oop* dst, size_t length)
 ↪  {
  Copy::conjoint_oops_atomic(src, dst, length);
}

void arraycopy_conjoint_oops(narrowOop* src, narrowOop* dst,
 ↪  size_t length) {
  Copy::conjoint_oops_atomic(src, dst, length);
}

void arraycopy_disjoint_words(void* src, void* dst, size_t
 ↪  length) {
  Copy::disjoint_words(reinterpret_cast<HeapWord*>(src),
   ↪  reinterpret_cast<HeapWord*>(dst), length);
}

void arraycopy_disjoint_words_atomic(void* src, void* dst,
 ↪  size_t length) {
  Copy::disjoint_words_atomic(reinterpret_cast<HeapWord*>(src),
   ↪  reinterpret_cast<HeapWord*>(dst), length);
}

template<>
void arraycopy_conjoint<jboolean>(jboolean* src, jboolean* dst,
 ↪  size_t length) {
  Copy::conjoint_jbytes(reinterpret_cast<jbyte*>(src),
   ↪  reinterpret_cast<jbyte*>(dst), length);
}

template<>
void arraycopy_conjoint<jbyte>(jbyte* src, jbyte* dst, size_t
 ↪  length) {
  Copy::conjoint_jbytes(src, dst, length);
}

template<>
void arraycopy_conjoint<jchar>(jchar* src, jchar* dst, size_t
 ↪  length) {
  Copy::conjoint_jshorts_atomic(reinterpret_cast<jshort*>(src),
   ↪  reinterpret_cast<jshort*>(dst), length);
}

template<>
void arraycopy_conjoint<jshort>(jshort* src, jshort* dst,
 ↪  size_t length) {
  Copy::conjoint_jshorts_atomic(src, dst, length);
```

```
}

template<>
void arraycopy_conjoint<jint>(jint* src, jint* dst, size_t
↪  length) {
  Copy::conjoint_jints_atomic(src, dst, length);
}

template<>
void arraycopy_conjoint<jfloat>(jfloat* src, jfloat* dst,
↪  size_t length) {
  Copy::conjoint_jints_atomic(reinterpret_cast<jint*>(src),
  ↪  reinterpret_cast<jint*>(dst), length);
}

template<>
void arraycopy_conjoint<jlong>(jlong* src, jlong* dst, size_t
↪  length) {
  Copy::conjoint_jlongs_atomic(src, dst, length);
}

template<>
void arraycopy_conjoint<jdouble>(jdouble* src, jdouble* dst,
↪  size_t length) {
  Copy::conjoint_jlongs_atomic(reinterpret_cast<jlong*>(src),
  ↪  reinterpret_cast<jlong*>(dst), length);
}

template<>
void arraycopy_arrayof_conjoint<jbyte>(jbyte* src, jbyte* dst,
↪  size_t length) {

  ↪  Copy::arrayof_conjoint_jbytes(reinterpret_cast<HeapWord*>(src),
  ↪  reinterpret_cast<HeapWord*>(dst), length);
}

template<>
void arraycopy_arrayof_conjoint<jshort>(jshort* src, jshort*
↪  dst, size_t length) {

  ↪  Copy::arrayof_conjoint_jshorts(reinterpret_cast<HeapWord*>(src),
  reinterpret_cast<HeapWord*>(dst),
  length);
}

template<>
void arraycopy_arrayof_conjoint<jint>(jint* src, jint* dst,
↪  size_t length) {
   ↪  Copy::arrayof_conjoint_jints(reinterpret_cast<HeapWord*>(src),
   ↪  reinterpret_cast<HeapWord*>(dst), length);
}

template<>
void arraycopy_arrayof_conjoint<jlong>(jlong* src, jlong* dst,
 ↪  size_t length) {

   ↪  Copy::arrayof_conjoint_jlongs(reinterpret_cast<HeapWord*>(src),
   ↪  reinterpret_cast<HeapWord*>(dst), length);
}

template<>
void arraycopy_conjoint<void>(void* src, void* dst, size_t
 ↪  length) {
  Copy::conjoint_jbytes(reinterpret_cast<jbyte*>(src),
   ↪  reinterpret_cast<jbyte*>(dst), length);
}

template<>
void arraycopy_conjoint_atomic<jbyte>(jbyte* src, jbyte* dst,
 ↪  size_t length) {
  Copy::conjoint_jbytes_atomic(src, dst, length);
}

template<>
void arraycopy_conjoint_atomic<jshort>(jshort* src, jshort*
 ↪  dst, size_t length) {
  Copy::conjoint_jshorts_atomic(src, dst, length);
}

template<>
void arraycopy_conjoint_atomic<jint>(jint* src, jint* dst,
 ↪  size_t length) {
  Copy::conjoint_jints_atomic(src, dst, length);
}

template<>
void arraycopy_conjoint_atomic<jlong>(jlong* src, jlong* dst,
 ↪  size_t length) {
  Copy::conjoint_jlongs_atomic(src, dst, length);
}

template<>
void arraycopy_conjoint_atomic<void>(void* src, void* dst,
 ↪  size_t length) {
  Copy::conjoint_memory_atomic(src, dst, length);
}
```

- hotspot/share/utilities/copy.hpp

```
class Copy : AllStatic {
 public:
  // Block copy methods have four attributes.  We don't define
   ↪  all possibilities.
  //   alignment: aligned to BytesPerLong
  //   arrayof:   arraycopy operation with both operands
   ↪  aligned on the same
  //              boundary as the first element of an array of
   ↪  the copy unit.
  //              This is currently a HeapWord boundary on all
   ↪  platforms, except
  //              for long and double arrays, which are aligned
   ↪  on an 8-byte
  //              boundary on all platforms.
  //              arraycopy operations are implicitly atomic on
   ↪  each array element.
  //   overlap:   disjoint or conjoint.
  //   copy unit: bytes or words (i.e., HeapWords) or oops
   ↪  (i.e., pointers).
  //   atomicity: atomic or non-atomic on the copy unit.
  //
  // Names are constructed thusly:
  //
  //     [ 'aligned_' | 'arrayof_' ]
  //     ('conjoint_' | 'disjoint_')
  //     ('words' | 'bytes' | 'jshorts' | 'jints' | 'jlongs' |
   ↪  'oops')
  //     [ '_atomic' ]
  //
  // Except in the arrayof case, whatever the alignment is, we
   ↪  assume we can copy
  // whole alignment units.  E.g., if BytesPerLong is 2x word
   ↪  alignment, an odd
  // count may copy an extra word.  In the arrayof case, we are
   ↪  allowed to copy
  // only the number of copy units specified.
  //
  // All callees check count for 0.
  //

  // HeapWords

  // Word-aligned words,    conjoint, not atomic on each word
  static void conjoint_words(const HeapWord* from, HeapWord*
   ↪  to, size_t count) {
    assert_params_ok(from, to, HeapWordSize);
    pd_conjoint_words(from, to, count);
}

// Word-aligned words,    disjoint, not atomic on each word
static void disjoint_words(const HeapWord* from, HeapWord*
↪  to, size_t count) {
  assert_params_ok(from, to, HeapWordSize);
  assert_disjoint(from, to, count);
  pd_disjoint_words(from, to, count);
}

// Word-aligned words,    disjoint, atomic on each word
static void disjoint_words_atomic(const HeapWord* from,
↪  HeapWord* to, size_t count) {
  assert_params_ok(from, to, HeapWordSize);
  assert_disjoint(from, to, count);
  pd_disjoint_words_atomic(from, to, count);
}

// Object-aligned words,  conjoint, not atomic on each word
static void aligned_conjoint_words(const HeapWord* from,
↪  HeapWord* to, size_t count) {
  assert_params_aligned(from, to);
  pd_aligned_conjoint_words(from, to, count);
}

// Object-aligned words,  disjoint, not atomic on each word
static void aligned_disjoint_words(const HeapWord* from,
↪  HeapWord* to, size_t count) {
  assert_params_aligned(from, to);
  assert_disjoint(from, to, count);
  pd_aligned_disjoint_words(from, to, count);
}

// bytes, jshorts, jints, jlongs, oops

// bytes,                 conjoint, not atomic on each byte
↪  (not that it matters)
static void conjoint_jbytes(const void* from, void* to,
↪  size_t count) {
  pd_conjoint_bytes(from, to, count);
}

// bytes,                 conjoint, atomic on each byte (not
↪  that it matters)
static void conjoint_jbytes_atomic(const void* from, void*
↪  to, size_t count) {
  pd_conjoint_bytes(from, to, count);
}

// jshorts,               conjoint, atomic on each jshort
```

```
static void conjoint_jshorts_atomic(const jshort* from,
↪   jshort* to, size_t count) {
  assert_params_ok(from, to, BytesPerShort);
  pd_conjoint_jshorts_atomic(from, to, count);
}

// jints,                 conjoint, atomic on each jint
static void conjoint_jints_atomic(const jint* from, jint* to,
↪   size_t count) {
  assert_params_ok(from, to, BytesPerInt);
  pd_conjoint_jints_atomic(from, to, count);
}

// jlongs,                conjoint, atomic on each jlong
static void conjoint_jlongs_atomic(const jlong* from, jlong*
↪   to, size_t count) {
  assert_params_ok(from, to, BytesPerLong);
  pd_conjoint_jlongs_atomic(from, to, count);
}

// oops,                  conjoint, atomic on each oop
static void conjoint_oops_atomic(const oop* from, oop* to,
↪   size_t count) {
  assert_params_ok(from, to, BytesPerHeapOop);
  pd_conjoint_oops_atomic(from, to, count);
}

// overloaded for UseCompressedOops
static void conjoint_oops_atomic(const narrowOop* from,
↪   narrowOop* to, size_t count) {
  assert(sizeof(narrowOop) == sizeof(jint), "this cast is
   ↪   wrong");
  assert_params_ok(from, to, BytesPerInt);
  pd_conjoint_jints_atomic((const jint*)from, (jint*)to,
   ↪   count);
}

// Copy a span of memory.  If the span is an integral number
↪   of aligned
// longs, words, or ints, copy those units atomically.
// The largest atomic transfer unit is 8 bytes, or the
↪   largest power
// of two which divides all of from, to, and size, whichever
↪   is smaller.
static void conjoint_memory_atomic(const void* from, void*
↪   to, size_t size);

// bytes,                 conjoint array, atomic on each byte
↪   (not that it matters)
```

```
static void arrayof_conjoint_jbytes(const HeapWord* from,
↪   HeapWord* to, size_t count) {
  pd_arrayof_conjoint_bytes(from, to, count);
}

// jshorts,               conjoint array, atomic on each
↪   jshort
static void arrayof_conjoint_jshorts(const HeapWord* from,
↪   HeapWord* to, size_t count) {
  assert_params_ok(from, to, BytesPerShort);
  pd_arrayof_conjoint_jshorts(from, to, count);
}

// jints,                 conjoint array, atomic on each jint
static void arrayof_conjoint_jints(const HeapWord* from,
↪   HeapWord* to, size_t count) {
  assert_params_ok(from, to, BytesPerInt);
  pd_arrayof_conjoint_jints(from, to, count);
}

// jlongs,                conjoint array, atomic on each
↪   jlong
static void arrayof_conjoint_jlongs(const HeapWord* from,
↪   HeapWord* to, size_t count) {
  assert_params_ok(from, to, BytesPerLong);
  pd_arrayof_conjoint_jlongs(from, to, count);
}

// oops,                  conjoint array, atomic on each oop
static void arrayof_conjoint_oops(const HeapWord* from,
↪   HeapWord* to, size_t count) {
  assert_params_ok(from, to, BytesPerHeapOop);
  pd_arrayof_conjoint_oops(from, to, count);
}

// Known overlap methods

// Copy word-aligned words from higher to lower addresses,
↪   not atomic on each word
inline static void conjoint_words_to_lower(const HeapWord*
↪   from, HeapWord* to, size_t byte_count) {
  // byte_count is in bytes to check its alignment
  assert_params_ok(from, to, HeapWordSize);
  assert_byte_count_ok(byte_count, HeapWordSize);

  size_t count = align_up(byte_count, HeapWordSize) >>
  ↪   LogHeapWordSize;
  assert(to <= from || from + count <= to, "do not overwrite
  ↪   source data");
  while (count-- > 0) {
    *to++ = *from++;
  }
}

// Copy word-aligned words from lower to higher addresses,
↪  not atomic on each word
inline static void conjoint_words_to_higher(const HeapWord*
↪  from, HeapWord* to, size_t byte_count) {
  // byte_count is in bytes to check its alignment
  assert_params_ok(from, to, HeapWordSize);
  assert_byte_count_ok(byte_count, HeapWordSize);

  size_t count = align_up(byte_count, HeapWordSize) >>
  ↪  LogHeapWordSize;
  assert(from <= to || to + count <= from, "do not overwrite
  ↪  source data");

  from += count - 1;
  to   += count - 1;
  while (count-- > 0) {
    *to-- = *from--;
  }
}

/**
 * Copy elements
 *
 * @param src address of source
 * @param dst address of destination
 * @param byte_count number of bytes to copy
 * @param elem_size size of the elements to copy-swap
 */
static void conjoint_copy(const void* src, void* dst, size_t
↪  byte_count, size_t elem_size);

/**
 * Copy and *unconditionally* byte swap elements
 *
 * @param src address of source
 * @param dst address of destination
 * @param byte_count number of bytes to copy
 * @param elem_size size of the elements to copy-swap
 */
static void conjoint_swap(const void* src, void* dst, size_t
↪  byte_count, size_t elem_size);

/**
 * Copy and byte swap elements from the specified endian to
↪  the native (cpu) endian if needed (if they differ)
```

```
 *
 * @param src address of source
 * @param dst address of destination
 * @param byte_count number of bytes to copy
 * @param elem_size size of the elements to copy-swap
 */
template <Endian::Order endian>
static void conjoint_swap_if_needed(const void* src, void*
↪  dst, size_t byte_count, size_t elem_size) {
  if (Endian::NATIVE != endian) {
    conjoint_swap(src, dst, byte_count, elem_size);
  } else {
    conjoint_copy(src, dst, byte_count, elem_size);
  }
}

// Fill methods

// Fill word-aligned words, not atomic on each word
// set_words
static void fill_to_words(HeapWord* to, size_t count, juint
↪  value = 0) {
  assert_params_ok(to, HeapWordSize);
  pd_fill_to_words(to, count, value);
}

static void fill_to_aligned_words(HeapWord* to, size_t count,
↪  juint value = 0) {
  assert_params_aligned(to);
  pd_fill_to_aligned_words(to, count, value);
}

// Fill bytes
static void fill_to_bytes(void* to, size_t count, jubyte
↪  value = 0) {
  pd_fill_to_bytes(to, count, value);
}

// Fill a span of memory.  If the span is an integral number
↪  of aligned
// longs, words, or ints, store to those units atomically.
// The largest atomic transfer unit is 8 bytes, or the
↪  largest power
// of two which divides both to and size, whichever is
↪  smaller.
static void fill_to_memory_atomic(void* to, size_t size,
↪  jubyte value = 0);

// Zero-fill methods
```

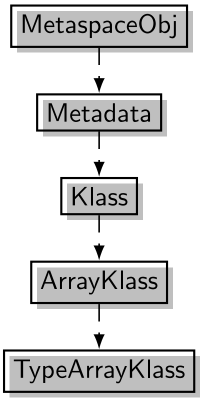
```
 // Zero word-aligned words, not atomic on each word
 static void zero_to_words(HeapWord* to, size_t count) {
  assert_params_ok(to, HeapWordSize);
  pd_zero_to_words(to, count);
 }

 // Zero bytes
 static void zero_to_bytes(void* to, size_t count) {
  pd_zero_to_bytes(to, count);
 }

private:
 static bool params_disjoint(const HeapWord* from, HeapWord*
 ↪  to, size_t count) {
  if (from < to) {
   return pointer_delta(to, from) >= count;
  }
  return pointer_delta(from, to) >= count;
 }

 // These methods raise a fatal if they detect a problem.

 static void assert_disjoint(const HeapWord* from, HeapWord*
 ↪  to, size_t count) {
  assert(params_disjoint(from, to, count), "source and dest
  ↪  overlap");
 }

 static void assert_params_ok(const void* from, void* to,
 ↪  intptr_t alignment) {
  assert(is_aligned(from, alignment), "must be aligned: "
  ↪  INTPTR_FORMAT, p2i(from));
  assert(is_aligned(to, alignment),   "must be aligned: "
  ↪  INTPTR_FORMAT, p2i(to));
 }

 static void assert_params_ok(HeapWord* to, intptr_t
 ↪  alignment) {
  assert(is_aligned(to, alignment), "must be aligned: "
  ↪  INTPTR_FORMAT, p2i(to));
 }

 static void assert_params_aligned(const HeapWord* from,
 ↪  HeapWord* to) {
  assert(is_aligned(from, BytesPerLong), "must be aligned: "
  ↪  INTPTR_FORMAT, p2i(from));
  assert(is_aligned(to, BytesPerLong),   "must be aligned: "
  ↪  INTPTR_FORMAT, p2i(to));
 }
```

```
  static void assert_params_aligned(HeapWord* to) {
    assert(is_aligned(to, BytesPerLong), "must be aligned: "
     ↪  INTPTR_FORMAT, p2i(to));
  }

  static void assert_byte_count_ok(size_t byte_count, size_t
   ↪  unit_size) {
    assert(is_aligned(byte_count, unit_size), "byte count must
     ↪  be aligned");
  }

  // Platform dependent implementations of the above methods.
#include CPU_HEADER(copy)

};
```

- hotspot/cpu/ppc/copy_ppc.hpp

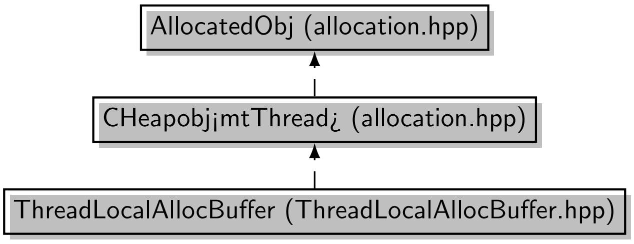
```
// Inline functions for memory copy and fill.

static void pd_conjoint_words(const HeapWord* from, HeapWord*
 ↪  to, size_t count) {
  (void)memmove(to, from, count * HeapWordSize);
}

// Template for atomic, element-wise copy.
template <class T>
static void copy_conjoint_atomic(const T* from, T* to, size_t
 ↪  count) {
  if (from > to) {
    while (count-- > 0) {
      // Copy forwards
      *to++ = *from++;
    }
  } else {
    from += count - 1;
    to   += count - 1;
    while (count-- > 0) {
      // Copy backwards
      *to-- = *from--;
    }
  }
}

// ...
```

0.2.1.4 Type Array Inheritance

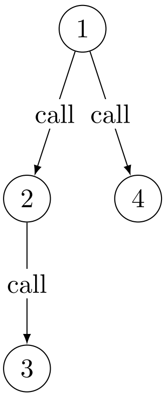


0.2.1.5 TLAB

0.2.1.5.1 Inheritance


AllocatedObj (allocation.hpp)
CHeapobj¡mtThread¿ (allocation.hpp)
ThreadLocalAllocBuffer (ThreadLocalAllocBuffer.hpp)

0.2.1.5.2 Initialization



1. `void OptoRuntime::new_array_C(Klass* array_type, int len, JavaThread *thread)`

2. `typeArrayOop oopFactory::new_typeArray(BasicType type, int length, TRAPS)`

3. `TypeArrayKlass* TypeArrayKlass::allocate(ClassLoaderData* loader_data, BasicType type, Symbol* name, TRAPS)`

4. `TypeArrayKlass* TypeArrayKlass::allocate(ClassLoaderData* loader_data, BasicType type, Symbol* name, TRAPS)`
5. `objArrayOop oopFactory::new_objArray(Klass* klass, int length, TRAPS)`

0.2.1.5.3 Notes

The virtual machine must never call one of the implicitly declared global allocation or deletion functions. (Such calls may result in link-time or run-time errors.) For convenience and documentation of intended use, classes in the virtual machine may be derived from one of the following allocation classes, some of which define allocation and deletion functions. Note: `std::malloc` and `std::free` should never called directly.
For objects allocated in the resource area (see resourceArea.hpp). - ResourceObj
For objects allocated in the C-heap (managed by: free & malloc and tracked with NMT) - CHeapObj
For objects allocated on the stack. - StackObj
For classes used as name spaces. - AllStatic
For classes in Metaspace (class data) - MetaspaceObj
The printable subclasses are used for debugging and define virtual member functions for printing. Classes that avoid allocating the vtbl entries in the objects should therefore not be the printable subclasses.
The following macros and function should be used to allocate memory directly in the resource area or in the C-heap, The _OBJ variants of the NEW/FREE_C_HEAP macros are used for alloc/dealloc simple objects which are not inherited from CHeapObj, note constructor and destructor are not called. The preferable way to allocate objects is using the new operator.
WARNING: The array variant must only be used for a homogenous array where all objects are of the exact type specified. If subtypes are stored in the array then must pay attention to calling destructors at needed.

- `NEW_RESOURCE_ARRAY(type, size)`
- `NEW_RESOURCE_OBJ(type)`
- `NEW_C_HEAP_ARRAY(type, size)`
- `NEW_C_HEAP_OBJ(type, memflags)`
- `FREE_C_HEAP_ARRAY(type, old)`
- `FREE_C_HEAP_OBJ(objname, type, memflags)`
- `char* AllocateHeap(size_t size, const char* name);`
- `void FreeHeap(void* p);`

In non product mode we introduce a super class for all allocation classes that supports printing. We avoid the superclass in product mode to save space.

0.3 Notes

0.3.1 Tuesday, 08 October 2019

0.3.1.1 Kafka Bug Study

- Distributed Concurrency Bugs

  1. [KAFKA-1183] - DefaultEventHandler causes unbalanced distribution of messages across partitions
  2. [KAFKA-1154] - replicas may not have consistent data after becoming follower
  3. [KAFKA-1124] - Sending to a new topic (with auto.create.topics.enable) returns ERROR
  4. [KAFKA-1097] - Race condition while reassigning low throughput partition leads to incorrect ISR information in zookeeper

- Scalability Bugs

  1. [KAFKA-1228] - Socket Leak on ReplicaFetcherThread

0.3.1.2 JVM Deduplication

- Trying to implement the madvise call inside the TLAB allocation (ThreadLocalAllocBuffer::allocate)
  Only saving 44kb for two JVM process and array 139 with its elements 42

0.3.2 Friday, 04 October 2019

0.3.2.1 Allocation Classes

1. Allocation, consists of:
1.1. AllocatedObj (Abstract Class)
1.1.1. CHeapObj (Abstract Class)
1.1.1.1. CollectedHeap
1.1.1.1.1. GenCollectedHeap
1.1.1.1.1.1. SerialHeap
1.1.1.1.1.2. CMSHeap
1.1.1.1.2. G1CollectedHeap
1.1.1.1.3. ParallelScavengeHeap
1.1.1.1.4. ZCollectedHeap
1.1.2. StackObj (Abstract Class)
1.1.2.1. MemAllocator
1.1.2.1.1. ObjAllocator
1.1.2.1.2. ObjArrayAllocator
1.1.2.1.3. ClassAllocator
1.1.3. ResourceObj (Abstract Class)
1.2. MetaspaceObj (Abstract Class)
1.3. AllStatic (Abstract Class)
1.3.1. ArrayAllocator
1.3.2. MmapArrayAllocator
1.4. MallocArrayAllocator

0.3.2.2 Notes

- All of the Klass is being instantiated in Metaspace
- EpsilonGC is not creating its own heap

0.4 Resources

0.4.1 Kernel Samepage Merging

1. Documentation

   (a) Madvise

   (b) KSM