****
129608 Fri Jul 13 15:35:50 2012
new/src/share/vm/classfile/javaClasses.cpp
****
_____***unchanged_portion_omitted_***

```
2731 // Support for java_lang_invoke_CallSite

2733 int java_lang_invoke_CallSite::_target_offset;

2735 void java_lang_invoke_CallSite::compute_offsets() {
2736   if (!EnableInvokeDynamic)  return;
2737   klassOop k = SystemDictionary::CallSite_klass();
2738   if (k != NULL) {
2739     compute_offset(_target_offset, k, vmSymbols::target_name(), vmSymbols::java_
2740   }

2742   // Disallow compilation of CallSite.setTargetNormal and CallSite.setTargetVola
2743   // (For C2:  keep this until we have throttling logic for uncommon traps.)
2744   if (k != NULL) {
2745     instanceKlass* ik = instanceKlass::cast(k);
2746     methodOop m_normal   = ik->lookup_method(vmSymbols::setTargetNormal_name(),
2747     methodOop m_volatile = ik->lookup_method(vmSymbols::setTargetVolatile_name()
2748     guarantee(m_normal != NULL && m_volatile != NULL, "must exist");
2749     m_normal->set_not_compilable_quietly();
2750     m_volatile->set_not_compilable_quietly();
2751   }
2741 }
```

_____***unchanged_portion_omitted_***

****
50967 Fri Jul 13 15:35:51 2012
new/src/share/vm/interpreter/interpreterRuntime.cpp
****
_____unchanged_portion_omitted_

```
 296 // Special handling for stack overflow: since we don't have any (java) stack
 297 // space left we use the pre-allocated & pre-initialized StackOverflowError
 298 // klass to create an stack overflow error instance.  We do not call its
 299 // constructor for the same reason (it is empty, anyway).
 300 IRT_ENTRY(void, InterpreterRuntime::throw_StackOverflowError(JavaThread* thread)
 301   Handle exception = get_preinitialized_exception(
 302                                 SystemDictionary::StackOverflowError_klass(),
 303                                 CHECK);
 304   THROW_HANDLE(exception);
 305 IRT_END


 308 IRT_ENTRY(void, InterpreterRuntime::create_exception(JavaThread* thread, char* n
 309   // lookup exception klass
 310   TempNewSymbol s = SymbolTable::new_symbol(name, CHECK);
 311   if (ProfileTraps) {
 312     if (s == vmSymbols::java_lang_ArithmeticException()) {
 313       note_trap(thread, Deoptimization::Reason_div0_check, CHECK);
 314     } else if (s == vmSymbols::java_lang_NullPointerException()) {
 315       note_trap(thread, Deoptimization::Reason_null_check, CHECK);
 316     }
 317   }
 318   // create exception
 319   Handle exception = Exceptions::new_exception(thread, s, message);
 320   thread->set_vm_result(exception());
 321 IRT_END


 324 IRT_ENTRY(void, InterpreterRuntime::create_klass_exception(JavaThread* thread, c
 325   ResourceMark rm(thread);
 326   const char* klass_name = Klass::cast(obj->klass())->external_name();
 327   // lookup exception klass
 328   TempNewSymbol s = SymbolTable::new_symbol(name, CHECK);
 329   if (ProfileTraps) {
 330     note_trap(thread, Deoptimization::Reason_class_check, CHECK);
 331   }
 332   // create exception, with klass name as detail message
 333   Handle exception = Exceptions::new_exception(thread, s, klass_name);
 334   thread->set_vm_result(exception());
 335 IRT_END


 338 IRT_ENTRY(void, InterpreterRuntime::throw_ArrayIndexOutOfBoundsException(JavaThr
 339   char message[jintAsStringSize];
 340   // lookup exception klass
 341   TempNewSymbol s = SymbolTable::new_symbol(name, CHECK);
 342   if (ProfileTraps) {
 343     note_trap(thread, Deoptimization::Reason_range_check, CHECK);
 344   }
 345   // create exception
 346   sprintf(message, "%d", index);
 347   THROW_MSG(s, message);
 348 IRT_END

 350 IRT_ENTRY(void, InterpreterRuntime::throw_ClassCastException(
 351   JavaThread* thread, oopDesc* obj))

 353   ResourceMark rm(thread);
 354   char* message = SharedRuntime::generate_class_cast_message(
 355     thread, Klass::cast(obj->klass())->external_name());
```



```
 357   if (ProfileTraps) {
 358     note_trap(thread, Deoptimization::Reason_class_check, CHECK);
 359   }

 361   // create exception
 362   THROW_MSG(vmSymbols::java_lang_ClassCastException(), message);
 363 IRT_END

 365 // exception_handler_for_exception(...) returns the continuation address,
 366 // the exception oop (via TLS) and sets the bci/bcp for the continuation.
 367 // The exception oop is returned to make sure it is preserved over GC (it
 368 // is only on the stack if the exception was thrown explicitly via athrow).
 369 // During this operation, the expression stack contains the values for the
 370 // bci where the exception happened. If the exception was propagated back
 371 // from a call, the expression stack contains the values for the bci at the
 372 // invoke w/o arguments (i.e., as if one were inside the call).
 373 IRT_ENTRY(address, InterpreterRuntime::exception_handler_for_exception(JavaThrea

 375   Handle             h_exception(thread, exception);
 376   methodHandle       h_method   (thread, method(thread));
 377   constantPoolHandle h_constants(thread, h_method->constants());
 378   typeArrayHandle    h_extable  (thread, h_method->exception_table());
 379   bool               should_repeat;
 380   int                handler_bci;
 381   int                current_bci = bci(thread);

 383   // Need to do this check first since when _do_not_unlock_if_synchronized
 384   // is set, we don't want to trigger any classloading which may make calls
 385   // into java, or surprisingly find a matching exception handler for bci 0
 386   // since at this moment the method hasn't been "officially" entered yet.
 387   if (thread->do_not_unlock_if_synchronized()) {
 388     ResourceMark rm;
 389     assert(current_bci == 0,  "bci isn't zero for do_not_unlock_if_synchronized"
 390     thread->set_vm_result(exception);
 391 #ifdef CC_INTERP
 392     return (address) -1;
 393 #else
 394     return Interpreter::remove_activation_entry();
 395 #endif
 396   }

 398   do {
 399     should_repeat = false;

 401     // assertions
 402 #ifdef ASSERT
 403     assert(h_exception.not_null(), "NULL exceptions should be handled by athrow"
 404     assert(h_exception->is_oop(), "just checking");
 405     // Check that exception is a subclass of Throwable, otherwise we have a Veri
 406     if (!(h_exception->is_a(SystemDictionary::Throwable_klass()))) {
 407       if (ExitVMOnVerifyError) vm_exit(-1);
 408       ShouldNotReachHere();
 409     }
 410 #endif

 412     // tracing
 413     if (TraceExceptions) {
 414       ttyLocker ttyl;
 415       ResourceMark rm(thread);
 416       tty->print_cr("Exception <%s> (" INTPTR_FORMAT ")", h_exception->print_val
 417       tty->print_cr(" thrown in interpreter method <%s>", h_method->print_value_
 418       tty->print_cr(" at bci %d for thread " INTPTR_FORMAT, current_bci, thread)
 419     }
 420 // Don't go paging in something which won't be used.
 421 //     else if (h_extable->length() == 0) {
```

```
422 //       // disabled for now - interpreter is not using shortcut yet
423 //       // (shortcut is not to call runtime if we have no exception handlers)
424 //       // warning("performance bug: should not call runtime if method has no e
425 //     }
426     // for AbortVMOnException flag
427     NOT_PRODUCT(Exceptions::debug_check_abort(h_exception));
428
429     // exception handler lookup
430     KlassHandle h_klass(THREAD, h_exception->klass());
431     handler_bci = h_method->fast_exception_handler_bci_for(h_klass, current_bci,
432     if (HAS_PENDING_EXCEPTION) {
433       // We threw an exception while trying to find the exception handler.
434       // Transfer the new exception to the exception handle which will
435       // be set into thread local storage, and do another lookup for an
436       // exception handler for this exception, this time starting at the
437       // BCI of the exception handler which caused the exception to be
438       // thrown (bug 4307310).
439       h_exception = Handle(THREAD, PENDING_EXCEPTION);
440       CLEAR_PENDING_EXCEPTION;
441       if (handler_bci >= 0) {
442         current_bci = handler_bci;
443         should_repeat = true;
444       }
445     }
446   } while (should_repeat == true);
447
448   // notify JVMTI of an exception throw; JVMTI will detect if this is a first
449   // time throw or a stack unwinding throw and accordingly notify the debugger
450   if (JvmtiExport::can_post_on_exceptions()) {
451     JvmtiExport::post_exception_throw(thread, h_method(), bcp(thread), h_excepti
452   }
453
454 #ifdef CC_INTERP
455   address continuation = (address)(intptr_t) handler_bci;
456 #else
457   address continuation = NULL;
458 #endif
459   address handler_pc = NULL;
460   if (handler_bci < 0 || !thread->reguard_stack((address) &continuation)) {
461     // Forward exception to callee (leaving bci/bcp untouched) because (a) no
462     // handler in this method, or (b) after a stack overflow there is not yet
463     // enough stack space available to reprotect the stack.
464 #ifndef CC_INTERP
465     continuation = Interpreter::remove_activation_entry();
466 #endif
467     // Count this for compilation purposes
468     h_method->interpreter_throwout_increment();
469   } else {
470     // handler in this method => change bci/bcp to handler bci/bcp and continue
471     handler_pc = h_method->code_base() + handler_bci;
472 #ifndef CC_INTERP
473     set_bcp_and_mdp(handler_pc, thread);
474     continuation = Interpreter::dispatch_table(vtos)[*handler_pc];
475 #endif
476   }
477   // notify debugger of an exception catch
478   // (this is good for exceptions caught in native methods as well)
479   if (JvmtiExport::can_post_on_exceptions()) {
480     JvmtiExport::notice_unwind_due_to_exception(thread, h_method(), handler_pc,
481   }
482
483   thread->set_vm_result(h_exception());
484   return continuation;
485 IRT_END
```



```
488 IRT_ENTRY(void, InterpreterRuntime::throw_pending_exception(JavaThread* thread))
489   assert(thread->has_pending_exception(), "must only ne called if there's an exc
490   // nothing to do - eventually we should remove this code entirely (see comment
491 IRT_END
492
493
494 IRT_ENTRY(void, InterpreterRuntime::throw_AbstractMethodError(JavaThread* thread
495   THROW(vmSymbols::java_lang_AbstractMethodError());
496 IRT_END
497
498
499 IRT_ENTRY(void, InterpreterRuntime::throw_IncompatibleClassChangeError(JavaThrea
500   THROW(vmSymbols::java_lang_IncompatibleClassChangeError());
501 IRT_END
502
503
504 //------------------------------------------------------------------------------
505 // Fields
506 //
507
508 IRT_ENTRY(void, InterpreterRuntime::resolve_get_put(JavaThread* thread, Bytecode
509   // resolve field
510   FieldAccessInfo info;
511   constantPoolHandle pool(thread, method(thread)->constants());
512   bool is_put    = (bytecode == Bytecodes::_putfield  || bytecode == Bytecodes::
513   bool is_static = (bytecode == Bytecodes::_getstatic || bytecode == Bytecodes::
514
515   {
516     JvmtiHideSingleStepping jhss(thread);
517     LinkResolver::resolve_field(info, pool, get_index_u2_cpcache(thread, bytecod
518                                 bytecode, false, CHECK);
519   } // end JvmtiHideSingleStepping
520
521   // check if link resolution caused cpCache to be updated
522   if (already_resolved(thread)) return;
523
524   // compute auxiliary field attributes
525   TosState state  = as_TosState(info.field_type());
526
527   // We need to delay resolving put instructions on final fields
528   // until we actually invoke one. This is required so we throw
529   // exceptions at the correct place. If we do not resolve completely
530   // in the current pass, leaving the put_code set to zero will
531   // cause the next put instruction to reresolve.
532   Bytecodes::Code put_code = (Bytecodes::Code)0;
533
534   // We also need to delay resolving getstatic instructions until the
535   // class is intitialized.  This is required so that access to the static
536   // field will call the initialization function every time until the class
537   // is completely initialized ala. in 2.17.5 in JVM Specification.
538   instanceKlass *klass = instanceKlass::cast(info.klass()->as_klassOop());
539   bool uninitialized_static = ((bytecode == Bytecodes::_getstatic || bytecode ==
540                                !klass->is_initialized());
541   Bytecodes::Code get_code = (Bytecodes::Code)0;
542
543   if (!uninitialized_static) {
544     get_code = ((is_static) ? Bytecodes::_getstatic : Bytecodes::_getfield);
545     if (is_put || !info.access_flags().is_final()) {
546       put_code = ((is_static) ? Bytecodes::_putstatic : Bytecodes::_putfield);
547     }
548   }
549
550   if (is_put && !is_static && klass->is_subclass_of(SystemDictionary::CallSite_k
551     const jint direction = frame::interpreter_frame_expression_stack_direction()
552     Handle call_site    (THREAD, *((oop*) thread->last_frame().interpreter_frame
553     Handle method_handle(THREAD, *((oop*) thread->last_frame().interpreter_frame
```

```
 554     assert(call_site    ->is_a(SystemDictionary::CallSite_klass()),     "must be
 555     assert(method_handle->is_a(SystemDictionary::MethodHandle_klass()), "must be

 557     {
 558       // Walk all nmethods depending on this call site.
 559       MutexLocker mu(Compile_lock, thread);
 560       Universe::flush_dependents_on(call_site, method_handle);
 561     }

 563     // Don't allow fast path for setting CallSite.target and sub-classes.
 564     put_code = (Bytecodes::Code) 0;
 565   }
```

```
 550   cache_entry(thread)->set_field(
 551     get_code,
 552     put_code,
 553     info.klass(),
 554     info.field_index(),
 555     info.field_offset(),
 556     state,
 557     info.access_flags().is_final(),
 558     info.access_flags().is_volatile()
 559   );
 560 IRT_END


 563 //------------------------------------------------------------------------------
 564 // Synchronization
 565 //
 566 // The interpreter's synchronization code is factored out so that it can
 567 // be shared by method invocation and synchronized blocks.
 568 //%note synchronization_3

 570 static void trace_locking(Handle& h_locking_obj, bool is_locking) {
 571   ObjectSynchronizer::trace_locking(h_locking_obj, false, true, is_locking);
 572 }
```

_____***unchanged_portion_omitted_***

```
*********************************************************
  133834 Fri Jul 13 15:35:53 2012
new/src/share/vm/prims/methodHandles.cpp
*********************************************************
______unchanged_portion_omitted_

3161 // This one function is exported, used by NativeLookup.

3163 JVM_ENTRY(void, JVM_RegisterMethodHandleMethods(JNIEnv *env, jclass MHN_class))
3164   assert(MethodHandles::spot_check_entry_names(), "entry enum is OK");

3166   if (!EnableInvokeDynamic) {
3167     warning("JSR 292 is disabled in this JVM.  Use -XX:+UnlockDiagnosticVMOption
3168     return;  // bind nothing
3169   }

3171   assert(!MethodHandles::enabled(), "must not be enabled");
3172   bool enable_MH = true;

3174   {
3175     ThreadToNativeFromVM ttnfv(thread);
3176     int status = env->RegisterNatives(MHN_class, methods, sizeof(methods)/sizeof
3177     if (!env->ExceptionOccurred()) {
3178       const char* L_MH_name = (JLINV "MethodHandle");
3179       const char* MH_name = L_MH_name+1;
3180       jclass MH_class = env->FindClass(MH_name);
3181       status = env->RegisterNatives(MH_class, invoke_methods, sizeof(invoke_meth
3182     }
3183     if (!env->ExceptionOccurred()) {
3184       status = env->RegisterNatives(MHN_class, call_site_methods, sizeof(call_si
3185     }
3186 #endif /* ! codereview */
3187     if (env->ExceptionOccurred()) {
3188       warning("JSR 292 method handle code is mismatched to this JVM.  Disabling
3189       enable_MH = false;
3190       env->ExceptionClear();
3191     }

3183     status = env->RegisterNatives(MHN_class, call_site_methods, sizeof(call_site
3184     if (env->ExceptionOccurred()) {
3185       // Exception is okay until 7087357
3186       env->ExceptionClear();
3187     }
3193   }

3195   if (enable_MH) {
3196     methodOop raiseException_method = MethodHandles::resolve_raise_exception_met
3197     if (raiseException_method != NULL) {
3198       MethodHandles::set_raise_exception_method(raiseException_method);
3199     } else {
3200       warning("JSR 292 method handle code is mismatched to this JVM.  Disabling
3201       enable_MH = false;
3202     }
3203   }

3205   if (enable_MH) {
3206     MethodHandles::generate_adapters();
3207     MethodHandles::set_enabled(true);
3208   }
3209 }
______unchanged_portion_omitted_
```

```
**********************************************************
   64781 Fri Jul 13 15:35:54 2012
new/src/share/vm/prims/unsafe.cpp
**********************************************************
______unchanged_portion_omitted_


 151 ///// Data in the Java heap.

 153 #define GET_FIELD(obj, offset, type_name, v) \
 154   oop p = JNIHandles::resolve(obj); \
 155   type_name v = *(type_name*)index_oop_from_field_offset_long(p, offset)

 157 #define SET_FIELD(obj, offset, type_name, x) \
 158   oop p = JNIHandles::resolve(obj); \
 159   *(type_name*)index_oop_from_field_offset_long(p, offset) = x

 161 #define GET_FIELD_VOLATILE(obj, offset, type_name, v) \
 162   oop p = JNIHandles::resolve(obj); \
 163   volatile type_name v = OrderAccess::load_acquire((volatile type_name*)index_oo

 165 #define SET_FIELD_VOLATILE(obj, offset, type_name, x) \
 166   oop p = JNIHandles::resolve(obj); \
 167   OrderAccess::release_store_fence((volatile type_name*)index_oop_from_field_off

 169 // Macros for oops that check UseCompressedOops

 171 #define GET_OOP_FIELD(obj, offset, v) \
 172   oop p = JNIHandles::resolve(obj);   \
 173   oop v;                              \
 174   if (UseCompressedOops) {            \
 175     narrowOop n = *(narrowOop*)index_oop_from_field_offset_long(p, offset); \
 176     v = oopDesc::decode_heap_oop(n);                                         \
 177   } else {                            \
 178     v = *(oop*)index_oop_from_field_offset_long(p, offset);                \
 179   }

 181 #define GET_OOP_FIELD_VOLATILE(obj, offset, v) \
 182   oop p = JNIHandles::resolve(obj);   \
 183   volatile oop v;                     \
 184   if (UseCompressedOops) {            \
 185     volatile narrowOop n = *(volatile narrowOop*)index_oop_from_field_offset_lon
 186     v = oopDesc::decode_heap_oop(n);                                         \
 187   } else {                            \
 188     v = *(volatile oop*)index_oop_from_field_offset_long(p, offset);       \
 189   } \
 190   OrderAccess::acquire();

 182 // Get/SetObject must be special-cased, since it works with handles.

 184 // The xxx140 variants for backward compatibility do not allow a full-width offs
 185 UNSAFE_ENTRY(jobject, Unsafe_GetObject140(JNIEnv *env, jobject unsafe, jobject o
 186   UnsafeWrapper("Unsafe_GetObject");
 187   if (obj == NULL)  THROW_0(vmSymbols::java_lang_NullPointerException());
 188   GET_OOP_FIELD(obj, offset, v)
 189   jobject ret = JNIHandles::make_local(env, v);
 190 #ifndef SERIALGC
 191   // We could be accessing the referent field in a reference
 192   // object. If G1 is enabled then we need to register a non-null
 193   // referent with the SATB barrier.
 194   if (UseG1GC) {
 195     bool needs_barrier = false;

 197     if (ret != NULL) {
 198       if (offset == java_lang_ref_Reference::referent_offset) {
```



```
 199         oop o = JNIHandles::resolve_non_null(obj);
 200         klassOop k = o->klass();
 201         if (instanceKlass::cast(k)->reference_type() != REF_NONE) {
 202           assert(instanceKlass::cast(k)->is_subclass_of(SystemDictionary::Refere
 203           needs_barrier = true;
 204         }
 205       }
 206     }

 208     if (needs_barrier) {
 209       oop referent = JNIHandles::resolve(ret);
 210       G1SATBCardTableModRefBS::enqueue(referent);
 211     }
 212   }
 213 #endif // SERIALGC
 214   return ret;
 215 UNSAFE_END

 217 UNSAFE_ENTRY(void, Unsafe_SetObject140(JNIEnv *env, jobject unsafe, jobject obj,
 218   UnsafeWrapper("Unsafe_SetObject");
 219   if (obj == NULL)  THROW(vmSymbols::java_lang_NullPointerException());
 220   oop x = JNIHandles::resolve(x_h);
 221   //SET_FIELD(obj, offset, oop, x);
 222   oop p = JNIHandles::resolve(obj);
 223   if (UseCompressedOops) {
 224     if (x != NULL) {
 225       // If there is a heap base pointer, we are obliged to emit a store barrier
 226       oop_store((narrowOop*)index_oop_from_field_offset_long(p, offset), x);
 227     } else {
 228       narrowOop n = oopDesc::encode_heap_oop_not_null(x);
 229       *(narrowOop*)index_oop_from_field_offset_long(p, offset) = n;
 230     }
 231   } else {
 232     if (x != NULL) {
 233       // If there is a heap base pointer, we are obliged to emit a store barrier
 234       oop_store((oop*)index_oop_from_field_offset_long(p, offset), x);
 235     } else {
 236       *(oop*)index_oop_from_field_offset_long(p, offset) = x;
 237     }
 238   }
 239 UNSAFE_END

 241 // The normal variants allow a null base pointer with an arbitrary address.
 242 // But if the base pointer is non-null, the offset should make some sense.
 243 // That is, it should be in the range [0, MAX_OBJECT_SIZE].
 244 UNSAFE_ENTRY(jobject, Unsafe_GetObject(JNIEnv *env, jobject unsafe, jobject obj,
 245   UnsafeWrapper("Unsafe_GetObject");
 246   GET_OOP_FIELD(obj, offset, v)
 247   jobject ret = JNIHandles::make_local(env, v);
 248 #ifndef SERIALGC
 249   // We could be accessing the referent field in a reference
 250   // object. If G1 is enabled then we need to register non-null
 251   // referent with the SATB barrier.
 252   if (UseG1GC) {
 253     bool needs_barrier = false;

 255     if (ret != NULL) {
 256       if (offset == java_lang_ref_Reference::referent_offset && obj != NULL) {
 257         oop o = JNIHandles::resolve(obj);
 258         klassOop k = o->klass();
 259         if (instanceKlass::cast(k)->reference_type() != REF_NONE) {
 260           assert(instanceKlass::cast(k)->is_subclass_of(SystemDictionary::Refere
 261           needs_barrier = true;
 262         }
 263       }
 264     }
```

```
266     if (needs_barrier) {
267       oop referent = JNIHandles::resolve(ret);
268       G1SATBCardTableModRefBS::enqueue(referent);
269     }
270   }
271 #endif // SERIALGC
272   return ret;
273 UNSAFE_END

275 UNSAFE_ENTRY(void, Unsafe_SetObject(JNIEnv *env, jobject unsafe, jobject obj, jl
276   UnsafeWrapper("Unsafe_SetObject");
277   oop x = JNIHandles::resolve(x_h);
278   oop p = JNIHandles::resolve(obj);
279   if (UseCompressedOops) {
280     oop_store((narrowOop*)index_oop_from_field_offset_long(p, offset), x);
281   } else {
282     oop_store((oop*)index_oop_from_field_offset_long(p, offset), x);
283   }
284 UNSAFE_END

286 UNSAFE_ENTRY(jobject, Unsafe_GetObjectVolatile(JNIEnv *env, jobject unsafe, jobj
287   UnsafeWrapper("Unsafe_GetObjectVolatile");
288   oop p = JNIHandles::resolve(obj);
289   void* addr = index_oop_from_field_offset_long(p, offset);
290   volatile oop v;
291   if (UseCompressedOops) {
292     volatile narrowOop n = *(volatile narrowOop*) addr;
293     v = oopDesc::decode_heap_oop(n);
294   } else {
295     v = *(volatile oop*) addr;
296   }
297   OrderAccess::acquire();
299   GET_OOP_FIELD_VOLATILE(obj, offset, v)
298   return JNIHandles::make_local(env, v);
299 UNSAFE_END

301 UNSAFE_ENTRY(void, Unsafe_SetObjectVolatile(JNIEnv *env, jobject unsafe, jobject
302   UnsafeWrapper("Unsafe_SetObjectVolatile");
305   {
306     // Catch VolatileCallSite.target stores (via
307     // CallSite.setTargetVolatile) and check call site dependencies.
308     oop p = JNIHandles::resolve(obj);
309     if ((offset == java_lang_invoke_CallSite::target_offset_in_bytes()) && p->is
310       Handle call_site    (THREAD, p);
311       Handle method_handle(THREAD, JNIHandles::resolve(x_h));
312       assert(call_site    ->is_a(SystemDictionary::CallSite_klass()),     "must
313       assert(method_handle->is_a(SystemDictionary::MethodHandle_klass()), "must
314       {
315         // Walk all nmethods depending on this call site.
316         MutexLocker mu(Compile_lock, thread);
317         Universe::flush_dependents_on(call_site(), method_handle());
318       }
319     }
320   }
303   oop x = JNIHandles::resolve(x_h);
304   oop p = JNIHandles::resolve(obj);
305   void* addr = index_oop_from_field_offset_long(p, offset);
306   OrderAccess::release();
307   if (UseCompressedOops) {
308     oop_store((narrowOop*)addr, x);
309   } else {
310     oop_store((oop*)addr, x);
311   }
312   OrderAccess::fence();
313 UNSAFE_END
```



```
315 #if defined(SPARC) || defined(X86)
316 // Sparc and X86 have atomic jlong (8 bytes) instructions

318 #else
319 // Keep old code for platforms which may not have atomic jlong (8 bytes) instruc

321 // Volatile long versions must use locks if !VM_Version::supports_cx8().
322 // support_cx8 is a surrogate for 'supports atomic long memory ops'.

324 UNSAFE_ENTRY(jlong, Unsafe_GetLongVolatile(JNIEnv *env, jobject unsafe, jobject
325   UnsafeWrapper("Unsafe_GetLongVolatile");
326   {
327     if (VM_Version::supports_cx8()) {
328       GET_FIELD_VOLATILE(obj, offset, jlong, v);
329       return v;
330     }
331     else {
332       Handle p (THREAD, JNIHandles::resolve(obj));
333       jlong* addr = (jlong*)(index_oop_from_field_offset_long(p(), offset));
334       ObjectLocker ol(p, THREAD);
335       jlong value = *addr;
336       return value;
337     }
338   }
339 UNSAFE_END

341 UNSAFE_ENTRY(void, Unsafe_SetLongVolatile(JNIEnv *env, jobject unsafe, jobject o
342   UnsafeWrapper("Unsafe_SetLongVolatile");
343   {
344     if (VM_Version::supports_cx8()) {
345       SET_FIELD_VOLATILE(obj, offset, jlong, x);
346     }
347     else {
348       Handle p (THREAD, JNIHandles::resolve(obj));
349       jlong* addr = (jlong*)(index_oop_from_field_offset_long(p(), offset));
350       ObjectLocker ol(p, THREAD);
351       *addr = x;
352     }
353   }
354 UNSAFE_END

356 #endif // not SPARC and not X86

358 #define DEFINE_GETSETOOP(jboolean, Boolean) \
359  \
360 UNSAFE_ENTRY(jboolean, Unsafe_Get##Boolean##140(JNIEnv *env, jobject unsafe, job
361   UnsafeWrapper("Unsafe_Get"#Boolean); \
362   if (obj == NULL)  THROW_0(vmSymbols::java_lang_NullPointerException()); \
363   GET_FIELD(obj, offset, jboolean, v); \
364   return v; \
365 UNSAFE_END \
366  \
367 UNSAFE_ENTRY(void, Unsafe_Set##Boolean##140(JNIEnv *env, jobject unsafe, jobject
368   UnsafeWrapper("Unsafe_Set"#Boolean); \
369   if (obj == NULL)  THROW(vmSymbols::java_lang_NullPointerException()); \
370   SET_FIELD(obj, offset, jboolean, x); \
371 UNSAFE_END \
372  \
373 UNSAFE_ENTRY(jboolean, Unsafe_Get##Boolean(JNIEnv *env, jobject unsafe, jobject
374   UnsafeWrapper("Unsafe_Get"#Boolean); \
375   GET_FIELD(obj, offset, jboolean, v); \
376   return v; \
377 UNSAFE_END \
378  \
379 UNSAFE_ENTRY(void, Unsafe_Set##Boolean(JNIEnv *env, jobject unsafe, jobject obj,
```

```
380   UnsafeWrapper("Unsafe_Set"#Boolean); \
381   SET_FIELD(obj, offset, jboolean, x); \
382 UNSAFE_END \
383  \
384 // END DEFINE_GETSETOOP.

386 DEFINE_GETSETOOP(jboolean, Boolean)
387 DEFINE_GETSETOOP(jbyte, Byte)
388 DEFINE_GETSETOOP(jshort, Short);
389 DEFINE_GETSETOOP(jchar, Char);
390 DEFINE_GETSETOOP(jint, Int);
391 DEFINE_GETSETOOP(jlong, Long);
392 DEFINE_GETSETOOP(jfloat, Float);
393 DEFINE_GETSETOOP(jdouble, Double);

395 #undef DEFINE_GETSETOOP

397 #define DEFINE_GETSETOOP_VOLATILE(jboolean, Boolean) \
398  \
399 UNSAFE_ENTRY(jboolean, Unsafe_Get##Boolean##Volatile(JNIEnv *env, jobject unsafe
400   UnsafeWrapper("Unsafe_Get"#Boolean); \
401   GET_FIELD_VOLATILE(obj, offset, jboolean, v); \
402   return v; \
403 UNSAFE_END \
404  \
405 UNSAFE_ENTRY(void, Unsafe_Set##Boolean##Volatile(JNIEnv *env, jobject unsafe, jo
406   UnsafeWrapper("Unsafe_Set"#Boolean); \
407   SET_FIELD_VOLATILE(obj, offset, jboolean, x); \
408 UNSAFE_END \
409  \
410 // END DEFINE_GETSETOOP_VOLATILE.

412 DEFINE_GETSETOOP_VOLATILE(jboolean, Boolean)
413 DEFINE_GETSETOOP_VOLATILE(jbyte, Byte)
414 DEFINE_GETSETOOP_VOLATILE(jshort, Short);
415 DEFINE_GETSETOOP_VOLATILE(jchar, Char);
416 DEFINE_GETSETOOP_VOLATILE(jint, Int);
417 DEFINE_GETSETOOP_VOLATILE(jfloat, Float);
418 DEFINE_GETSETOOP_VOLATILE(jdouble, Double);

420 #if defined(SPARC) || defined(X86)
421 // Sparc and X86 have atomic jlong (8 bytes) instructions
422 DEFINE_GETSETOOP_VOLATILE(jlong, Long);
423 #endif

425 #undef DEFINE_GETSETOOP_VOLATILE

427 // The non-intrinsified versions of setOrdered just use setVolatile

429 UNSAFE_ENTRY(void, Unsafe_SetOrderedInt(JNIEnv *env, jobject unsafe, jobject obj
430   UnsafeWrapper("Unsafe_SetOrderedInt");
431   SET_FIELD_VOLATILE(obj, offset, jint, x);
432 UNSAFE_END

434 UNSAFE_ENTRY(void, Unsafe_SetOrderedObject(JNIEnv *env, jobject unsafe, jobject
435   UnsafeWrapper("Unsafe_SetOrderedObject");
436   oop x = JNIHandles::resolve(x_h);
437   oop p = JNIHandles::resolve(obj);
438   void* addr = index_oop_from_field_offset_long(p, offset);
439   OrderAccess::release();
440   if (UseCompressedOops) {
441     oop_store((narrowOop*)addr, x);
442   } else {
443     oop_store((oop*)addr, x);
444   }
445   OrderAccess::fence();
```



```
446 UNSAFE_END

448 UNSAFE_ENTRY(void, Unsafe_SetOrderedLong(JNIEnv *env, jobject unsafe, jobject ob
449   UnsafeWrapper("Unsafe_SetOrderedLong");
450 #if defined(SPARC) || defined(X86)
451   // Sparc and X86 have atomic jlong (8 bytes) instructions
452   SET_FIELD_VOLATILE(obj, offset, jlong, x);
453 #else
454   // Keep old code for platforms which may not have atomic long (8 bytes) instru
455   {
456     if (VM_Version::supports_cx8()) {
457       SET_FIELD_VOLATILE(obj, offset, jlong, x);
458     }
459     else {
460       Handle p (THREAD, JNIHandles::resolve(obj));
461       jlong* addr = (jlong*)(index_oop_from_field_offset_long(p(), offset));
462       ObjectLocker ol(p, THREAD);
463       *addr = x;
464     }
465   }
466 #endif
467 UNSAFE_END

469 ////// Data in the C heap.

471 // Note:  These do not throw NullPointerException for bad pointers.
472 // They just crash.  Only a oop base pointer can generate a NullPointerException
473 //
474 #define DEFINE_GETSETNATIVE(java_type, Type, native_type) \
475  \
476 UNSAFE_ENTRY(java_type, Unsafe_GetNative##Type(JNIEnv *env, jobject unsafe, jlon
477   UnsafeWrapper("Unsafe_GetNative"#Type); \
478   void* p = addr_from_java(addr); \
479   JavaThread* t = JavaThread::current(); \
480   t->set_doing_unsafe_access(true); \
481   java_type x = *(volatile native_type*)p; \
482   t->set_doing_unsafe_access(false); \
483   return x; \
484 UNSAFE_END \
485  \
486 UNSAFE_ENTRY(void, Unsafe_SetNative##Type(JNIEnv *env, jobject unsafe, jlong add
487   UnsafeWrapper("Unsafe_SetNative"#Type); \
488   JavaThread* t = JavaThread::current(); \
489   t->set_doing_unsafe_access(true); \
490   void* p = addr_from_java(addr); \
491   *(volatile native_type*)p = x; \
492   t->set_doing_unsafe_access(false); \
493 UNSAFE_END \
494  \
495 // END DEFINE_GETSETNATIVE.

497 DEFINE_GETSETNATIVE(jbyte, Byte, signed char)
498 DEFINE_GETSETNATIVE(jshort, Short, signed short);
499 DEFINE_GETSETNATIVE(jchar, Char, unsigned short);
500 DEFINE_GETSETNATIVE(jint, Int, jint);
501 // no long -- handled specially
502 DEFINE_GETSETNATIVE(jfloat, Float, float);
503 DEFINE_GETSETNATIVE(jdouble, Double, double);

505 #undef DEFINE_GETSETNATIVE

507 UNSAFE_ENTRY(jlong, Unsafe_GetNativeLong(JNIEnv *env, jobject unsafe, jlong addr
508   UnsafeWrapper("Unsafe_GetNativeLong");
509   JavaThread* t = JavaThread::current();
510   // We do it this way to avoid problems with access to heap using 64
511   // bit loads, as jlong in heap could be not 64-bit aligned, and on
```

```
512   // some CPUs (SPARC) it leads to SIGBUS.
513   t->set_doing_unsafe_access(true);
514   void* p = addr_from_java(addr);
515   jlong x;
516   if (((intptr_t)p & 7) == 0) {
517     // jlong is aligned, do a volatile access
518     x = *(volatile jlong*)p;
519   } else {
520     jlong_accessor acc;
521     acc.words[0] = ((volatile jint*)p)[0];
522     acc.words[1] = ((volatile jint*)p)[1];
523     x = acc.long_value;
524   }
525   t->set_doing_unsafe_access(false);
526   return x;
527 UNSAFE_END

529 UNSAFE_ENTRY(void, Unsafe_SetNativeLong(JNIEnv *env, jobject unsafe, jlong addr,
530   UnsafeWrapper("Unsafe_SetNativeLong");
531   JavaThread* t = JavaThread::current();
532   // see comment for Unsafe_GetNativeLong
533   t->set_doing_unsafe_access(true);
534   void* p = addr_from_java(addr);
535   if (((intptr_t)p & 7) == 0) {
536     // jlong is aligned, do a volatile access
537     *(volatile jlong*)p = x;
538   } else {
539     jlong_accessor acc;
540     acc.long_value = x;
541     ((volatile jint*)p)[0] = acc.words[0];
542     ((volatile jint*)p)[1] = acc.words[1];
543   }
544   t->set_doing_unsafe_access(false);
545 UNSAFE_END


548 UNSAFE_ENTRY(jlong, Unsafe_GetNativeAddress(JNIEnv *env, jobject unsafe, jlong a
549   UnsafeWrapper("Unsafe_GetNativeAddress");
550   void* p = addr_from_java(addr);
551   return addr_to_java(*(void**)p);
552 UNSAFE_END

554 UNSAFE_ENTRY(void, Unsafe_SetNativeAddress(JNIEnv *env, jobject unsafe, jlong ad
555   UnsafeWrapper("Unsafe_SetNativeAddress");
556   void* p = addr_from_java(addr);
557   *(void**)p = addr_from_java(x);
558 UNSAFE_END


561 ////// Allocation requests

563 UNSAFE_ENTRY(jobject, Unsafe_AllocateInstance(JNIEnv *env, jobject unsafe, jclas
564   UnsafeWrapper("Unsafe_AllocateInstance");
565   {
566     ThreadToNativeFromVM ttnfv(thread);
567     return env->AllocObject(cls);
568   }
569 UNSAFE_END

571 UNSAFE_ENTRY(jlong, Unsafe_AllocateMemory(JNIEnv *env, jobject unsafe, jlong siz
572   UnsafeWrapper("Unsafe_AllocateMemory");
573   size_t sz = (size_t)size;
574   if (sz != (julong)size || size < 0) {
575     THROW_0(vmSymbols::java_lang_IllegalArgumentException());
576   }
577   if (sz == 0) {
```



```
578     return 0;
579   }
580   sz = round_to(sz, HeapWordSize);
581   void* x = os::malloc(sz);
582   if (x == NULL) {
583     THROW_0(vmSymbols::java_lang_OutOfMemoryError());
584   }
585   //Copy::fill_to_words((HeapWord*)x, sz / HeapWordSize);
586   return addr_to_java(x);
587 UNSAFE_END

589 UNSAFE_ENTRY(jlong, Unsafe_ReallocateMemory(JNIEnv *env, jobject unsafe, jlong a
590   UnsafeWrapper("Unsafe_ReallocateMemory");
591   void* p = addr_from_java(addr);
592   size_t sz = (size_t)size;
593   if (sz != (julong)size || size < 0) {
594     THROW_0(vmSymbols::java_lang_IllegalArgumentException());
595   }
596   if (sz == 0) {
597     os::free(p);
598     return 0;
599   }
600   sz = round_to(sz, HeapWordSize);
601   void* x = (p == NULL) ? os::malloc(sz) : os::realloc(p, sz);
602   if (x == NULL) {
603     THROW_0(vmSymbols::java_lang_OutOfMemoryError());
604   }
605   return addr_to_java(x);
606 UNSAFE_END

608 UNSAFE_ENTRY(void, Unsafe_FreeMemory(JNIEnv *env, jobject unsafe, jlong addr))
609   UnsafeWrapper("Unsafe_FreeMemory");
610   void* p = addr_from_java(addr);
611   if (p == NULL) {
612     return;
613   }
614   os::free(p);
615 UNSAFE_END

617 UNSAFE_ENTRY(void, Unsafe_SetMemory(JNIEnv *env, jobject unsafe, jlong addr, jlo
618   UnsafeWrapper("Unsafe_SetMemory");
619   size_t sz = (size_t)size;
620   if (sz != (julong)size || size < 0) {
621     THROW(vmSymbols::java_lang_IllegalArgumentException());
622   }
623   char* p = (char*) addr_from_java(addr);
624   Copy::fill_to_memory_atomic(p, sz, value);
625 UNSAFE_END

627 UNSAFE_ENTRY(void, Unsafe_SetMemory2(JNIEnv *env, jobject unsafe, jobject obj, j
628   UnsafeWrapper("Unsafe_SetMemory");
629   size_t sz = (size_t)size;
630   if (sz != (julong)size || size < 0) {
631     THROW(vmSymbols::java_lang_IllegalArgumentException());
632   }
633   oop base = JNIHandles::resolve(obj);
634   void* p = index_oop_from_field_offset_long(base, offset);
635   Copy::fill_to_memory_atomic(p, sz, value);
636 UNSAFE_END

638 UNSAFE_ENTRY(void, Unsafe_CopyMemory(JNIEnv *env, jobject unsafe, jlong srcAddr,
639   UnsafeWrapper("Unsafe_CopyMemory");
640   if (size == 0) {
641     return;
642   }
643   size_t sz = (size_t)size;
```

```
644   if (sz != (julong)size || size < 0) {
645     THROW(vmSymbols::java_lang_IllegalArgumentException());
646   }
647   void* src = addr_from_java(srcAddr);
648   void* dst = addr_from_java(dstAddr);
649   Copy::conjoint_memory_atomic(src, dst, sz);
650 UNSAFE_END

652 UNSAFE_ENTRY(void, Unsafe_CopyMemory2(JNIEnv *env, jobject unsafe, jobject srcOb
653   UnsafeWrapper("Unsafe_CopyMemory");
654   if (size == 0) {
655     return;
656   }
657   size_t sz = (size_t)size;
658   if (sz != (julong)size || size < 0) {
659     THROW(vmSymbols::java_lang_IllegalArgumentException());
660   }
661   oop srcp = JNIHandles::resolve(srcObj);
662   oop dstp = JNIHandles::resolve(dstObj);
663   if (dstp != NULL && !dstp->is_typeArray()) {
664     // NYI:  This works only for non-oop arrays at present.
665     // Generalizing it would be reasonable, but requires card marking.
666     // Also, autoboxing a Long from 0L in copyMemory(x,y, 0L,z, n) would be bad.
667     THROW(vmSymbols::java_lang_IllegalArgumentException());
668   }
669   void* src = index_oop_from_field_offset_long(srcp, srcOffset);
670   void* dst = index_oop_from_field_offset_long(dstp, dstOffset);
671   Copy::conjoint_memory_atomic(src, dst, sz);
672 UNSAFE_END


675 ////// Random queries

677 // See comment at file start about UNSAFE_LEAF
678 //UNSAFE_LEAF(jint, Unsafe_AddressSize())
679 UNSAFE_ENTRY(jint, Unsafe_AddressSize(JNIEnv *env, jobject unsafe))
680   UnsafeWrapper("Unsafe_AddressSize");
681   return sizeof(void*);
682 UNSAFE_END

684 // See comment at file start about UNSAFE_LEAF
685 //UNSAFE_LEAF(jint, Unsafe_PageSize())
686 UNSAFE_ENTRY(jint, Unsafe_PageSize(JNIEnv *env, jobject unsafe))
687   UnsafeWrapper("Unsafe_PageSize");
688   return os::vm_page_size();
689 UNSAFE_END

691 jint find_field_offset(jobject field, int must_be_static, TRAPS) {
692   if (field == NULL) {
693     THROW_0(vmSymbols::java_lang_NullPointerException());
694   }

696   oop reflected   = JNIHandles::resolve_non_null(field);
697   oop mirror      = java_lang_reflect_Field::clazz(reflected);
698   klassOop k      = java_lang_Class::as_klassOop(mirror);
699   int slot        = java_lang_reflect_Field::slot(reflected);
700   int modifiers   = java_lang_reflect_Field::modifiers(reflected);

702   if (must_be_static >= 0) {
703     int really_is_static = ((modifiers & JVM_ACC_STATIC) != 0);
704     if (must_be_static != really_is_static) {
705       THROW_0(vmSymbols::java_lang_IllegalArgumentException());
706     }
707   }

709   int offset = instanceKlass::cast(k)->field_offset(slot);
```



```
710   return field_offset_from_byte_offset(offset);
711 }
```

*unchanged_portion_omitted_*