```
**********************************************************
  108013 Thu Jun 14 10:53:37 2012
new/src/share/vm/classfile/vmSymbols.hpp
**********************************************************
```

```
/*
 * Copyright (c) 1997, 2012, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 *
 */

#ifndef SHARE_VM_CLASSFILE_VMSYMBOLS_HPP
#define SHARE_VM_CLASSFILE_VMSYMBOLS_HPP

#include "oops/symbol.hpp"
#include "memory/iterator.hpp"
#include "trace/traceMacros.hpp"

// The class vmSymbols is a name space for fast lookup of
// symbols commonly used in the VM.
//
// Sample usage:
//
//   Symbol* obj       = vmSymbols::java_lang_Object();


// Useful sub-macros exported by this header file:

#define VM_SYMBOL_ENUM_NAME(name)    name##_enum
#define VM_INTRINSIC_IGNORE(id, class, name, sig, flags) /*ignored*/
#define VM_SYMBOL_IGNORE(id, name)                       /*ignored*/
#define VM_ALIAS_IGNORE(id, id2)                         /*ignored*/


// Mapping function names to values. New entries should be added below.

#define VM_SYMBOLS_DO(template, do_alias)
  /* commonly used class names */
  template(java_lang_System,                      "java/lang/System")
  template(java_lang_Object,                      "java/lang/Object")
  template(java_lang_Class,                       "java/lang/Class")
  template(java_lang_String,                      "java/lang/String")
  template(java_lang_StringValue,                 "java/lang/StringValue")
  template(java_lang_StringCache,                 "java/lang/StringValue$Str
  template(java_lang_Thread,                      "java/lang/Thread")
  template(java_lang_ThreadGroup,                 "java/lang/ThreadGroup")
  template(java_lang_Cloneable,                   "java/lang/Cloneable")
  template(java_lang_Throwable,                   "java/lang/Throwable")
  template(java_lang_ClassLoader,                 "java/lang/ClassLoader")
```



```
  template(java_lang_ClassLoader_NativeLibrary,       "java/lang/ClassLoader\x02
  template(java_lang_ThreadDeath,                     "java/lang/ThreadDeath")
  template(java_lang_Boolean,                         "java/lang/Boolean")
  template(java_lang_Character,                       "java/lang/Character")
  template(java_lang_Character_CharacterCache,        "java/lang/Character$Chara
  template(java_lang_Float,                           "java/lang/Float")
  template(java_lang_Double,                          "java/lang/Double")
  template(java_lang_Byte,                            "java/lang/Byte")
  template(java_lang_Byte_Cache,                      "java/lang/Byte$ByteCache"
  template(java_lang_Short,                           "java/lang/Short")
  template(java_lang_Short_ShortCache,                "java/lang/Short$ShortCach
  template(java_lang_Integer,                         "java/lang/Integer")
  template(java_lang_Integer_IntegerCache,            "java/lang/Integer$Integer
  template(java_lang_Long,                            "java/lang/Long")
  template(java_lang_Long_LongCache,                  "java/lang/Long$LongCache"
  template(java_lang_Shutdown,                        "java/lang/Shutdown")
  template(java_lang_ref_Reference,                   "java/lang/ref/Reference")
  template(java_lang_ref_SoftReference,               "java/lang/ref/SoftReferen
  template(java_lang_ref_WeakReference,               "java/lang/ref/WeakReferen
  template(java_lang_ref_FinalReference,              "java/lang/ref/FinalRefere
  template(java_lang_ref_PhantomReference,            "java/lang/ref/PhantomRefe
  template(java_lang_ref_Finalizer,                   "java/lang/ref/Finalizer")
  template(java_lang_reflect_AccessibleObject,        "java/lang/reflect/Accessi
  template(java_lang_reflect_Method,                  "java/lang/reflect/Method"
  template(java_lang_reflect_Constructor,             "java/lang/reflect/Constru
  template(java_lang_reflect_Field,                   "java/lang/reflect/Field")
  template(java_lang_reflect_Array,                   "java/lang/reflect/Array")
  template(java_lang_StringBuffer,                    "java/lang/StringBuffer")
  template(java_lang_StringBuilder,                   "java/lang/StringBuilder")
  template(java_lang_CharSequence,                    "java/lang/CharSequence")
  template(java_security_AccessControlContext,        "java/security/AccessContr
  template(java_security_ProtectionDomain,            "java/security/ProtectionD
  template(java_io_OutputStream,                      "java/io/OutputStream")
  template(java_io_Reader,                            "java/io/Reader")
  template(java_io_BufferedReader,                    "java/io/BufferedReader")
  template(java_io_FileInputStream,                   "java/io/FileInputStream")
  template(java_io_ByteArrayInputStream,              "java/io/ByteArrayInputStr
  template(java_io_Serializable,                      "java/io/Serializable")
  template(java_util_Arrays,                          "java/util/Arrays")
  template(java_util_Properties,                      "java/util/Properties")
  template(java_util_Vector,                          "java/util/Vector")
  template(java_util_AbstractList,                    "java/util/AbstractList")
  template(java_util_Hashtable,                       "java/util/Hashtable")
  template(java_util_HashMap,                         "java/util/HashMap")
  template(java_lang_Compiler,                        "java/lang/Compiler")
  template(sun_misc_Signal,                           "sun/misc/Signal")
  template(java_lang_AssertionStatusDirectives,       "java/lang/AssertionStatus
  template(sun_jkernel_DownloadManager,               "sun/jkernel/DownloadManag
  template(getBootClassPathEntryForClass_name,        "getBootClassPathEntryForC
  template(sun_misc_PostVMInitHook,                   "sun/misc/PostVMInitHook")

  /* Java runtime version access */
  template(sun_misc_Version,                          "sun/misc/Version")
  template(java_runtime_name_name,                    "java_runtime_name")

#endif /* ! codereview */
  /* class file format tags */
  template(tag_source_file,                           "SourceFile")
  template(tag_inner_classes,                         "InnerClasses")
  template(tag_constant_value,                        "ConstantValue")
  template(tag_code,                                  "Code")
  template(tag_exceptions,                            "Exceptions")
  template(tag_line_number_table,                     "LineNumberTable")
  template(tag_local_variable_table,                  "LocalVariableTable")
  template(tag_local_variable_type_table,             "LocalVariableTypeTable")
  template(tag_stack_map_table,                       "StackMapTable")
```

```
129   template(tag_synthetic,                             "Synthetic")
130   template(tag_deprecated,                            "Deprecated")
131   template(tag_source_debug_extension,                "SourceDebugExtension")
132   template(tag_signature,                             "Signature")
133   template(tag_runtime_visible_annotations,           "RuntimeVisibleAnnotations
134   template(tag_runtime_invisible_annotations,         "RuntimeInvisibleAnnotatio
135   template(tag_runtime_visible_parameter_annotations, "RuntimeVisibleParameterAn
136   template(tag_runtime_invisible_parameter_annotations,"RuntimeInvisibleParamete
137   template(tag_annotation_default,                    "AnnotationDefault")
138   template(tag_enclosing_method,                      "EnclosingMethod")
139   template(tag_bootstrap_methods,                     "BootstrapMethods")
140
141   /* exception klasses: at least all exceptions thrown by the VM have entries he
142   template(java_lang_ArithmeticException,             "java/lang/ArithmeticExcep
143   template(java_lang_ArrayIndexOutOfBoundsException,  "java/lang/ArrayIndexOutOf
144   template(java_lang_ArrayStoreException,             "java/lang/ArrayStoreExcep
145   template(java_lang_ClassCastException,              "java/lang/ClassCastExcept
146   template(java_lang_ClassNotFoundException,          "java/lang/ClassNotFoundEx
147   template(java_lang_CloneNotSupportedException,      "java/lang/CloneNotSupport
148   template(java_lang_IllegalAccessException,          "java/lang/IllegalAccessEx
149   template(java_lang_IllegalArgumentException,        "java/lang/IllegalArgument
150   template(java_lang_IllegalStateException,           "java/lang/IllegalStateExc
151   template(java_lang_IllegalMonitorStateException,    "java/lang/IllegalMonitorS
152   template(java_lang_IllegalThreadStateException,     "java/lang/IllegalThreadSt
153   template(java_lang_IndexOutOfBoundsException,       "java/lang/IndexOutOfBound
154   template(java_lang_InstantiationException,          "java/lang/InstantiationEx
155   template(java_lang_InstantiationError,              "java/lang/InstantiationEr
156   template(java_lang_InterruptedException,            "java/lang/InterruptedExce
157   template(java_lang_BootstrapMethodError,            "java/lang/BootstrapMethod
158   template(java_lang_LinkageError,                    "java/lang/LinkageError")
159   template(java_lang_NegativeArraySizeException,      "java/lang/NegativeArraySi
160   template(java_lang_NoSuchFieldException,            "java/lang/NoSuchFieldExce
161   template(java_lang_NoSuchMethodException,           "java/lang/NoSuchMethodExc
162   template(java_lang_NullPointerException,            "java/lang/NullPointerExce
163   template(java_lang_StringIndexOutOfBoundsException, "java/lang/StringIndexOutO
164   template(java_lang_InvalidClassException,           "java/lang/InvalidClassExc
165   template(java_lang_reflect_InvocationTargetException, "java/lang/reflect/Invoc
166   template(java_lang_Exception,                       "java/lang/Exception")
167   template(java_lang_RuntimeException,                "java/lang/RuntimeExceptio
168   template(java_io_IOException,                       "java/io/IOException")
169   template(java_security_PrivilegedActionException,   "java/security/PrivilegedA
170
171   /* error klasses: at least all errors thrown by the VM have entries here */
172   template(java_lang_AbstractMethodError,             "java/lang/AbstractMethodE
173   template(java_lang_ClassCircularityError,           "java/lang/ClassCircularit
174   template(java_lang_ClassFormatError,                "java/lang/ClassFormatErro
175   template(java_lang_UnsupportedClassVersionError,    "java/lang/UnsupportedClas
176   template(java_lang_Error,                           "java/lang/Error")
177   template(java_lang_ExceptionInInitializerError,     "java/lang/ExceptionInInit
178   template(java_lang_IllegalAccessError,              "java/lang/IllegalAccessEr
179   template(java_lang_IncompatibleClassChangeError,    "java/lang/IncompatibleCla
180   template(java_lang_InternalError,                   "java/lang/InternalError")
181   template(java_lang_NoClassDefFoundError,            "java/lang/NoClassDefFound
182   template(java_lang_NoSuchFieldError,                "java/lang/NoSuchFieldErro
183   template(java_lang_NoSuchMethodError,               "java/lang/NoSuchMethodErr
184   template(java_lang_OutOfMemoryError,                "java/lang/OutOfMemoryErro
185   template(java_lang_UnsatisfiedLinkError,            "java/lang/UnsatisfiedLink
186   template(java_lang_VerifyError,                     "java/lang/VerifyError")
187   template(java_lang_SecurityException,               "java/lang/SecurityExcepti
188   template(java_lang_VirtualMachineError,             "java/lang/VirtualMachineE
189   template(java_lang_StackOverflowError,              "java/lang/StackOverflowEr
190   template(java_lang_StackTraceElement,               "java/lang/StackTraceEleme
191   template(java_util_concurrent_locks_AbstractOwnableSynchronizer,   "java/util/
192
193   /* class symbols needed by intrinsics */
194   VM_INTRINSICS_DO(VM_INTRINSIC_IGNORE, template, VM_SYMBOL_IGNORE, VM_SYMBOL_IG
```



```
195
196   /* Support for reflection based on dynamic bytecode generation (JDK 1.4 and ab
197
198   template(sun_reflect_FieldInfo,                     "sun/reflect/FieldInfo")
199   template(sun_reflect_MethodInfo,                    "sun/reflect/MethodInfo")
200   template(sun_reflect_MagicAccessorImpl,             "sun/reflect/MagicAccessor
201   template(sun_reflect_MethodAccessorImpl,            "sun/reflect/MethodAccesso
202   template(sun_reflect_ConstructorAccessorImpl,       "sun/reflect/ConstructorAc
203   template(sun_reflect_SerializationConstructorAccessorImpl, "sun/reflect/Serial
204   template(sun_reflect_DelegatingClassLoader,         "sun/reflect/DelegatingCla
205   template(sun_reflect_Reflection,                    "sun/reflect/Reflection")
206   template(checkedExceptions_name,                    "checkedExceptions")
207   template(clazz_name,                                "clazz")
208   template(exceptionTypes_name,                       "exceptionTypes")
209   template(modifiers_name,                            "modifiers")
210   template(newConstructor_name,                       "newConstructor")
211   template(newConstructor_signature,                  "(Lsun/reflect/MethodInfo;
212   template(newField_name,                             "newField")
213   template(newField_signature,                        "(Lsun/reflect/FieldInfo;)
214   template(newMethod_name,                            "newMethod")
215   template(newMethod_signature,                       "(Lsun/reflect/MethodInfo;
216   /* the following two names must be in order: */
217   template(invokeExact_name,                          "invokeExact")
218   template(invokeGeneric_name,                        "invokeGeneric")
219   template(invokeVarargs_name,                        "invokeVarargs")
220   template(star_name,                                 "*") /*not really a name*/
221   template(invoke_name,                               "invoke")
222   template(override_name,                             "override")
223   template(parameterTypes_name,                       "parameterTypes")
224   template(returnType_name,                           "returnType")
225   template(signature_name,                            "signature")
226   template(slot_name,                                 "slot")
227   template(selectAlternative_name,                    "selectAlternative")
228
229   /* Support for annotations (JDK 1.5 and above) */
230
231   template(annotations_name,                          "annotations")
232   template(parameter_annotations_name,                "parameterAnnotations")
233   template(annotation_default_name,                   "annotationDefault")
234   template(sun_reflect_ConstantPool,                  "sun/reflect/ConstantPool"
235   template(constantPoolOop_name,                      "constantPoolOop")
236   template(sun_reflect_UnsafeStaticFieldAccessorImpl, "sun/reflect/UnsafeStaticF
237   template(base_name,                                 "base")
238
239   /* Support for JSR 292 & invokedynamic (JDK 1.7 and above) */
240   template(java_lang_invoke_InvokeDynamic,            "java/lang/invoke/InvokeDy
241   template(java_lang_invoke_Linkage,                  "java/lang/invoke/Linkage"
242   template(java_lang_invoke_CallSite,                 "java/lang/invoke/CallSite
243   template(java_lang_invoke_ConstantCallSite,         "java/lang/invoke/Constant
244   template(java_lang_invoke_MutableCallSite,          "java/lang/invoke/MutableC
245   template(java_lang_invoke_VolatileCallSite,         "java/lang/invoke/Volatile
246   template(java_lang_invoke_MethodHandle,             "java/lang/invoke/MethodHa
247   template(java_lang_invoke_MethodType,               "java/lang/invoke/MethodTy
248   template(java_lang_invoke_WrongMethodTypeException, "java/lang/invoke/WrongMet
249   template(java_lang_invoke_MethodType_signature,     "Ljava/lang/invoke/MethodT
250   template(java_lang_invoke_MethodHandle_signature,   "Ljava/lang/invoke/MethodH
251   /* internal classes known only to the JVM: */
252   template(java_lang_invoke_MethodTypeForm,           "java/lang/invoke/MethodTy
253   template(java_lang_invoke_MethodTypeForm_signature, "Ljava/lang/invoke/MethodT
254   template(java_lang_invoke_MemberName,               "java/lang/invoke/MemberNa
255   template(java_lang_invoke_MethodHandleNatives,      "java/lang/invoke/MethodHa
256   template(java_lang_invoke_MethodHandleImpl,         "java/lang/invoke/MethodHa
257   template(java_lang_invoke_AdapterMethodHandle,      "java/lang/invoke/AdapterM
258   template(java_lang_invoke_BoundMethodHandle,        "java/lang/invoke/BoundMet
259   template(java_lang_invoke_DirectMethodHandle,       "java/lang/invoke/DirectMe
260   template(java_lang_invoke_CountingMethodHandle,     "java/lang/invoke/Counting
```

```
261   /* internal up-calls made only by the JVM, via class sun.invoke.MethodHandleNa
262   template(findMethodHandleType_name,                "findMethodHandleType")
263   template(findMethodHandleType_signature,       "(Ljava/lang/Class;[Ljava/lang/
264   template(notifyGenericMethodType_name,             "notifyGenericMethodType")
265   template(notifyGenericMethodType_signature,        "(Ljava/lang/invoke/Method
266   template(linkMethodHandleConstant_name,            "linkMethodHandleConstant"
267   template(linkMethodHandleConstant_signature, "(Ljava/lang/Class;ILjava/lang/Cl
268   template(makeDynamicCallSite_name,                 "makeDynamicCallSite")
269   template(makeDynamicCallSite_signature, "(Ljava/lang/invoke/MethodHandle;Ljava
270   template(setTargetNormal_name,                     "setTargetNormal")
271   template(setTargetVolatile_name,                   "setTargetVolatile")
272   template(setTarget_signature,                      "(Ljava/lang/invoke/Method
273   NOT_LP64(  do_alias(intptr_signature,              int_signature)  )
274   LP64_ONLY( do_alias(intptr_signature,              long_signature) )
275   template(selectAlternative_signature, "(ZLjava/lang/invoke/MethodHandle;Ljava/
276
277   /* common method and field names */
278   template(object_initializer_name,                  "<init>")
279   template(class_initializer_name,                   "<clinit>")
280   template(println_name,                             "println")
281   template(printStackTrace_name,                     "printStackTrace")
282   template(main_name,                                "main")
283   template(name_name,                                "name")
284   template(priority_name,                            "priority")
285   template(stillborn_name,                           "stillborn")
286   template(group_name,                               "group")
287   template(daemon_name,                              "daemon")
288   template(eetop_name,                               "eetop")
289   template(thread_status_name,                       "threadStatus")
290   template(run_method_name,                          "run")
291   template(exit_method_name,                         "exit")
292   template(add_method_name,                          "add")
293   template(remove_method_name,                       "remove")
294   template(parent_name,                              "parent")
295   template(threads_name,                             "threads")
296   template(groups_name,                              "groups")
297   template(maxPriority_name,                         "maxPriority")
298   template(destroyed_name,                           "destroyed")
299   template(vmAllowSuspension_name,                   "vmAllowSuspension")
300   template(nthreads_name,                            "nthreads")
301   template(ngroups_name,                             "ngroups")
302   template(shutdown_method_name,                     "shutdown")
303   template(finalize_method_name,                     "finalize")
304   template(reference_lock_name,                      "lock")
305   template(reference_discovered_name,                "discovered")
306   template(run_finalization_name,                    "runFinalization")
307   template(run_finalizers_on_exit_name,              "runFinalizersOnExit")
308   template(uncaughtException_name,                   "uncaughtException")
309   template(dispatchUncaughtException_name,           "dispatchUncaughtException
310   template(initializeSystemClass_name,               "initializeSystemClass")
311   template(loadClass_name,                           "loadClass")
312   template(loadClassInternal_name,                   "loadClassInternal")
313   template(get_name,                                 "get")
314   template(put_name,                                 "put")
315   template(type_name,                                "type")
316   template(findNative_name,                          "findNative")
317   template(deadChild_name,                           "deadChild")
318   template(addClass_name,                            "addClass")
319   template(getFromClass_name,                        "getFromClass")
320   template(dispatch_name,                            "dispatch")
321   template(getSystemClassLoader_name,                "getSystemClassLoader")
322   template(fillInStackTrace_name,                    "fillInStackTrace")
323   template(fillInStackTrace0_name,                   "fillInStackTrace0")
324   template(getCause_name,                            "getCause")
325   template(initCause_name,                           "initCause")
326   template(setProperty_name,                         "setProperty")
```



```
327   template(getProperty_name,                         "getProperty")
328   template(context_name,                             "context")
329   template(privilegedContext_name,                   "privilegedContext")
330   template(contextClassLoader_name,                  "contextClassLoader")
331   template(inheritedAccessControlContext_name,       "inheritedAccessControlCon
332   template(isPrivileged_name,                        "isPrivileged")
333   template(wait_name,                                "wait")
334   template(checkPackageAccess_name,                  "checkPackageAccess")
335   template(stackSize_name,                           "stackSize")
336   template(thread_id_name,                           "tid")
337   template(newInstance0_name,                        "newInstance0")
338   template(limit_name,                               "limit")
339   template(forName_name,                             "forName")
340   template(forName0_name,                            "forName0")
341   template(isJavaIdentifierStart_name,               "isJavaIdentifierStart")
342   template(isJavaIdentifierPart_name,                "isJavaIdentifierPart")
343   template(exclusive_owner_thread_name,              "exclusiveOwnerThread")
344   template(park_blocker_name,                        "parkBlocker")
345   template(park_event_name,                          "nativeParkEventPointer")
346   template(cache_field_name,                         "cache")
347   template(value_name,                               "value")
348   template(offset_name,                              "offset")
349   template(count_name,                               "count")
350   template(hash_name,                                "hash")
351   template(frontCacheEnabled_name,                   "frontCacheEnabled")
352   template(stringCacheEnabled_name,                  "stringCacheEnabled")
353   template(numberOfLeadingZeros_name,                "numberOfLeadingZeros")
354   template(numberOfTrailingZeros_name,               "numberOfTrailingZeros")
355   template(bitCount_name,                            "bitCount")
356   template(profile_name,                             "profile")
357   template(equals_name,                              "equals")
358   template(target_name,                              "target")
359   template(toString_name,                            "toString")
360   template(values_name,                              "values")
361   template(receiver_name,                            "receiver")
362   template(vmmethod_name,                            "vmmethod")
363   template(vmtarget_name,                            "vmtarget")
364   template(vmentry_name,                             "vmentry")
365   template(vmcount_name,                             "vmcount")
366   template(vmslots_name,                             "vmslots")
367   template(vmlayout_name,                            "vmlayout")
368   template(vmindex_name,                             "vmindex")
369   template(vmargslot_name,                           "vmargslot")
370   template(flags_name,                               "flags")
371   template(argument_name,                            "argument")
372   template(conversion_name,                          "conversion")
373   template(rtype_name,                               "rtype")
374   template(ptypes_name,                              "ptypes")
375   template(form_name,                                "form")
376   template(erasedType_name,                          "erasedType")
377   template(genericInvoker_name,                      "genericInvoker")
378   template(append_name,                              "append")
379   template(klass_name,                               "klass")
380   template(resolved_constructor_name,                "resolved_constructor")
381   template(array_klass_name,                         "array_klass")
382   template(oop_size_name,                            "oop_size")
383   template(static_oop_field_count_name,              "static_oop_field_count")
384
385   /* non-intrinsic name/signature pairs: */
386   template(register_method_name,                     "register")
387   do_alias(register_method_signature,         object_void_signature)
388
389   /* name symbols needed by intrinsics */
390   VM_INTRINSICS_DO(VM_INTRINSIC_IGNORE, VM_SYMBOL_IGNORE, template, VM_SYMBOL_IG
391
392   /* common signatures names */
```

```
393   template(void_method_signature,                 "()V")
394   template(void_boolean_signature,                "()Z")
395   template(void_byte_signature,                   "()B")
396   template(void_char_signature,                   "()C")
397   template(void_short_signature,                  "()S")
398   template(void_int_signature,                    "()I")
399   template(void_long_signature,                   "()J")
400   template(void_float_signature,                  "()F")
401   template(void_double_signature,                 "()D")
402   template(int_void_signature,                    "(I)V")
403   template(int_int_signature,                     "(I)I")
404   template(char_char_signature,                   "(C)C")
405   template(short_short_signature,                 "(S)S")
406   template(int_bool_signature,                    "(I)Z")
407   template(float_int_signature,                   "(F)I")
408   template(double_long_signature,                 "(D)J")
409   template(double_double_signature,               "(D)D")
410   template(int_float_signature,                   "(I)F")
411   template(long_int_signature,                    "(J)I")
412   template(long_long_signature,                   "(J)J")
413   template(long_double_signature,                 "(J)D")
414   template(byte_signature,                        "B")
415   template(char_signature,                        "C")
416   template(double_signature,                      "D")
417   template(float_signature,                       "F")
418   template(int_signature,                         "I")
419   template(long_signature,                        "J")
420   template(short_signature,                       "S")
421   template(bool_signature,                        "Z")
422   template(void_signature,                        "V")
423   template(byte_array_signature,                  "[B")
424   template(char_array_signature,                  "[C")
425   template(int_array_signature,                   "[I")
426   template(object_void_signature,                 "(Ljava/lang/Object;)V")
427   template(object_int_signature,                  "(Ljava/lang/Object;)I")
428   template(object_boolean_signature,              "(Ljava/lang/Object;)Z")
429   template(string_void_signature,                 "(Ljava/lang/String;)V")
430   template(string_int_signature,                  "(Ljava/lang/String;)I")
431   template(throwable_void_signature,              "(Ljava/lang/Throwable;)V"
432   template(void_throwable_signature,              "()Ljava/lang/Throwable;")
433   template(throwable_throwable_signature,         "(Ljava/lang/Throwable;)Lj
434   template(class_void_signature,                  "(Ljava/lang/Class;)V")
435   template(class_int_signature,                   "(Ljava/lang/Class;)I")
436   template(class_long_signature,                  "(Ljava/lang/Class;)J")
437   template(class_boolean_signature,               "(Ljava/lang/Class;)Z")
438   template(throwable_string_void_signature,       "(Ljava/lang/Throwable;Lja
439   template(string_array_void_signature,           "([Ljava/lang/String;)V")
440   template(string_array_string_array_void_signature,  "([Ljava/lang/String;[Ljav
441   template(thread_throwable_void_signature,        "(Ljava/lang/Thread;Ljava/
442   template(thread_void_signature,                 "(Ljava/lang/Thread;)V")
443   template(threadgroup_runnable_void_signature,   "(Ljava/lang/ThreadGroup;L
444   template(threadgroup_string_void_signature,     "(Ljava/lang/ThreadGroup;L
445   template(string_class_signature,                "(Ljava/lang/String;)Ljava
446   template(object_object_object_signature,        "(Ljava/lang/Object;Ljava/
447   template(string_string_string_signature,        "(Ljava/lang/String;Ljava/
448   template(string_string_signature,               "(Ljava/lang/String;)Ljava
449   template(classloader_string_long_signature,     "(Ljava/lang/ClassLoader;L
450   template(byte_array_void_signature,             "([B)V")
451   template(char_array_void_signature,             "([C)V")
452   template(int_int_void_signature,                "(II)V")
453   template(long_long_void_signature,              "(JJ)V")
454   template(void_classloader_signature,            "()Ljava/lang/ClassLoader;
455   template(void_object_signature,                 "()Ljava/lang/Object;")
456   template(void_class_signature,                  "()Ljava/lang/Class;")
457   template(void_string_signature,                 "()Ljava/lang/String;")
458   template(object_array_object_signature,         "([Ljava/lang/Object;)Ljav
```



```
459   template(object_object_array_object_signature,      "(Ljava/lang/Object;[Ljava
460   template(exception_void_signature,                  "(Ljava/lang/Exception;)V"
461   template(protectiondomain_signature,                "[Ljava/security/Protectio
462   template(accesscontrolcontext_signature,            "Ljava/security/AccessCont
463   template(class_protectiondomain_signature,          "(Ljava/lang/Class;Ljava/s
464   template(thread_signature,                          "Ljava/lang/Thread;")
465   template(thread_array_signature,                    "[Ljava/lang/Thread;")
466   template(threadgroup_signature,                     "Ljava/lang/ThreadGroup;")
467   template(threadgroup_array_signature,               "[Ljava/lang/ThreadGroup;"
468   template(class_array_signature,                     "[Ljava/lang/Class;")
469   template(classloader_signature,                     "Ljava/lang/ClassLoader;")
470   template(object_signature,                          "Ljava/lang/Object;")
471   template(class_signature,                           "Ljava/lang/Class;")
472   template(string_signature,                          "Ljava/lang/String;")
473   template(reference_signature,                       "Ljava/lang/ref/Reference;
474   template(concurrenthashmap_signature,               "Ljava/util/concurrent/Con
475   template(String_StringBuilder_signature,            "(Ljava/lang/String;)Ljava
476   template(int_StringBuilder_signature,               "(I)Ljava/lang/StringBuild
477   template(char_StringBuilder_signature,              "(C)Ljava/lang/StringBuild
478   template(String_StringBuffer_signature,             "(Ljava/lang/String;)Ljava
479   template(int_StringBuffer_signature,                "(I)Ljava/lang/StringBuffe
480   template(char_StringBuffer_signature,               "(C)Ljava/lang/StringBuffe
481   template(int_String_signature,                      "(I)Ljava/lang/String;")
482   /* signature symbols needed by intrinsics */
483   VM_INTRINSICS_DO(VM_INTRINSIC_IGNORE, VM_SYMBOL_IGNORE, VM_SYMBOL_IGNORE, temp
484
485   /* symbol aliases needed by intrinsics */
486   VM_INTRINSICS_DO(VM_INTRINSIC_IGNORE, VM_SYMBOL_IGNORE, VM_SYMBOL_IGNORE, VM_S
487
488   /* returned by the C1 compiler in case there's not enough memory to allocate a
489   template(dummy_symbol,                              "illegal symbol")
490
491   /* used by ClassFormatError when class name is not known yet */
492   template(unknown_class_name,                        "<Unknown>")
493
494   /* used to identify class loaders handling parallel class loading */
495   template(parallelCapable_name,                      "parallelLockMap")
496
497   /* JVM monitoring and management support */
498   template(java_lang_StackTraceElement_array,          "[Ljava/lang/StackTraceEl
499   template(java_lang_management_ThreadState,          "java/lang/management/Thr
500   template(java_lang_management_MemoryUsage,          "java/lang/management/Mem
501   template(java_lang_management_ThreadInfo,           "java/lang/management/Thr
502   template(sun_management_ManagementFactory,          "sun/management/Managemen
503   template(sun_management_Sensor,                     "sun/management/Sensor")
504   template(sun_management_Agent,                      "sun/management/Agent")
505   template(sun_management_GarbageCollectorImpl,       "sun/management/GarbageCo
506   template(getGcInfoBuilder_name,                     "getGcInfoBuilder")
507   template(getGcInfoBuilder_signature,                "()Lsun/management/GcInfo
508   template(com_sun_management_GcInfo,                 "com/sun/management/GcInf
509   template(com_sun_management_GcInfo_constructor_signature, "(Lsun/management/Gc
510   template(createGCNotification_name,                 "createGCNotification")
511   template(createGCNotification_signature,            "(JLjava/lang/String;Ljav
512   template(createMemoryPoolMBean_name,                "createMemoryPoolMBean")
513   template(createMemoryManagerMBean_name,             "createMemoryManagerMBean
514   template(createGarbageCollectorMBean_name,          "createGarbageCollectorMB
515   template(createMemoryPoolMBean_signature,           "(Ljava/lang/String;ZJJ)L
516   template(createMemoryManagerMBean_signature,        "(Ljava/lang/String;)Ljav
517   template(createGarbageCollectorMBean_signature,     "(Ljava/lang/String;Ljava
518   template(trigger_name,                              "trigger")
519   template(clear_name,                                "clear")
520   template(trigger_method_signature,                  "(ILjava/lang/management/
521   template(startAgent_name,                           "startAgent")
522   template(startRemoteAgent_name,                     "startRemoteManagementAge
523   template(startLocalAgent_name,                      "startLocalManagementAgen
524   template(stopRemoteAgent_name,                      "stopRemoteManagementAgen
```

```
  template(java_lang_management_ThreadInfo_constructor_signature, "(Ljava/lang/T
  template(java_lang_management_ThreadInfo_with_locks_constructor_signature, "(L
  template(long_long_long_long_void_signature,         "(JJJJ)V")

  template(java_lang_management_MemoryPoolMXBean,      "java/lang/management/Mem
  template(java_lang_management_MemoryManagerMXBean,   "java/lang/management/Mem
  template(java_lang_management_GarbageCollectorMXBean,"java/lang/management/Gar
  template(gcInfoBuilder_name,                         "gcInfoBuilder")
  template(createMemoryPool_name,                      "createMemoryPool")
  template(createMemoryManager_name,                   "createMemoryManager")
  template(createGarbageCollector_name,                "createGarbageCollector")
  template(createMemoryPool_signature,                 "(Ljava/lang/String;ZJJ)L
  template(createMemoryManager_signature,              "(Ljava/lang/String;)Ljav
  template(createGarbageCollector_signature,           "(Ljava/lang/String;Ljava
  template(addThreadDumpForMonitors_name,              "addThreadDumpForMonitors
  template(addThreadDumpForSynchronizers_name,         "addThreadDumpForSynchron
  template(addThreadDumpForMonitors_signature,         "(Ljava/lang/management/T
  template(addThreadDumpForSynchronizers_signature,    "(Ljava/lang/management/T

  /* JVMTI/java.lang.instrument support and VM Attach mechanism */
  template(sun_misc_VMSupport,                         "sun/misc/VMSupport")
  template(appendToClassPathForInstrumentation_name,   "appendToClassPathForInst
  do_alias(appendToClassPathForInstrumentation_signature, string_void_signature)
  template(serializePropertiesToByteArray_name,        "serializePropertiesToByt
  template(serializePropertiesToByteArray_signature,   "()[B")
  template(serializeAgentPropertiesToByteArray_name,   "serializeAgentProperties
  template(classRedefinedCount_name,                   "classRedefinedCount")

  /* trace signatures */
  TRACE_TEMPLATES(template)

  /*end*/

// Here are all the intrinsics known to the runtime and the CI.
// Each intrinsic consists of a public enum name (like _hashCode),
// followed by a specification of its klass, name, and signature:
//    template(<id>,  <klass>,  <name>, <sig>, <FCODE>)
//
// If you add an intrinsic here, you must also define its name
// and signature as members of the VM symbols.  The VM symbols for
// the intrinsic name and signature may be defined above.
//
// Because the VM_SYMBOLS_DO macro makes reference to VM_INTRINSICS_DO,
// you can also define an intrinsic's name and/or signature locally to the
// intrinsic, if this makes sense.  (It often does make sense.)
//
// For example:
//    do_intrinsic(_foo,  java_lang_Object,  foo_name, foo_signature, F_xx)
//     do_name(     foo_name, "foo")
//     do_signature(foo_signature, "()F")
// klass      = vmSymbols::java_lang_Object()
// name       = vmSymbols::foo_name()
// signature  = vmSymbols::foo_signature()
//
// The name and/or signature might be a "well known" symbol
// like "equal" or "()I", in which case there will be no local
// re-definition of the symbol.
//
// The do_class, do_name, and do_signature calls are all used for the
// same purpose:  Define yet another VM symbol.  They could all be merged
// into a common 'do_symbol' call, but it seems useful to record our
// intentions here about kinds of symbols (class vs. name vs. signature).
//
// The F_xx is one of the Flags enum; see below.
//
// for Emacs: (let ((c-backslash-column 120) (c-backslash-max-column 120)) (c-ba
```



```
#define VM_INTRINSICS_DO(do_intrinsic, do_class, do_name, do_signature, do_alias
  do_intrinsic(_hashCode,                 java_lang_Object,       hashCode_name,
   do_name(     hashCode_name,                                   "hashCode")
  do_intrinsic(_getClass,                 java_lang_Object,       getClass_name,
   do_name(     getClass_name,                                   "getClass")
  do_intrinsic(_clone,                    java_lang_Object,       clone_name, vo
   do_name(     clone_name,                                      "clone")

  /* Math & StrictMath intrinsics are defined in terms of just a few signatures:
  do_class(java_lang_Math,                "java/lang/Math")
  do_class(java_lang_StrictMath,          "java/lang/StrictMath")
  do_signature(double2_double_signature,  "(DD)D")
  do_signature(int2_int_signature,        "(II)I")

  /* here are the math names, all together: */
  do_name(abs_name,"abs")       do_name(sin_name,"sin")         do_name(cos_name
  do_name(tan_name,"tan")       do_name(atan2_name,"atan2")     do_name(sqrt_nam
  do_name(log_name,"log")       do_name(log10_name,"log10")     do_name(pow_name
  do_name(exp_name,"exp")       do_name(min_name,"min")         do_name(max_name

  do_intrinsic(_dabs,                     java_lang_Math,         abs_name,   do
  do_intrinsic(_dsin,                     java_lang_Math,         sin_name,   do
  do_intrinsic(_dcos,                     java_lang_Math,         cos_name,   do
  do_intrinsic(_dtan,                     java_lang_Math,         tan_name,   do
  do_intrinsic(_datan2,                   java_lang_Math,         atan2_name, do
  do_intrinsic(_dsqrt,                    java_lang_Math,         sqrt_name,  do
  do_intrinsic(_dlog,                     java_lang_Math,         log_name,   do
  do_intrinsic(_dlog10,                   java_lang_Math,         log10_name, do
  do_intrinsic(_dpow,                     java_lang_Math,         pow_name,   do
  do_intrinsic(_dexp,                     java_lang_Math,         exp_name,   do
  do_intrinsic(_min,                      java_lang_Math,         min_name,   in
  do_intrinsic(_max,                      java_lang_Math,         max_name,   in

  do_intrinsic(_floatToRawIntBits,        java_lang_Float,        floatToRawIntB
   do_name(     floatToRawIntBits_name,                          "floatToRawIntB
  do_intrinsic(_floatToIntBits,           java_lang_Float,        floatToIntBits
   do_name(     floatToIntBits_name,                             "floatToIntBits
  do_intrinsic(_intBitsToFloat,           java_lang_Float,        intBitsToFloat
   do_name(     intBitsToFloat_name,                             "intBitsToFloat
  do_intrinsic(_doubleToRawLongBits,      java_lang_Double,       doubleToRawLon
   do_name(     doubleToRawLongBits_name,                        "doubleToRawLon
  do_intrinsic(_doubleToLongBits,         java_lang_Double,       doubleToLongBi
   do_name(     doubleToLongBits_name,                           "doubleToLongBi
  do_intrinsic(_longBitsToDouble,         java_lang_Double,       longBitsToDoub
   do_name(     longBitsToDouble_name,                           "longBitsToDoub

  do_intrinsic(_numberOfLeadingZeros_i,   java_lang_Integer,      numberOfLeadin
  do_intrinsic(_numberOfLeadingZeros_l,   java_lang_Long,         numberOfLeadin

  do_intrinsic(_numberOfTrailingZeros_i,  java_lang_Integer,      numberOfTraili
  do_intrinsic(_numberOfTrailingZeros_l,  java_lang_Long,         numberOfTraili

  do_intrinsic(_bitCount_i,               java_lang_Integer,      bitCount_name,
  do_intrinsic(_bitCount_l,               java_lang_Long,         bitCount_name,

  do_intrinsic(_reverseBytes_i,           java_lang_Integer,      reverseBytes_n
   do_name(     reverseBytes_name,                               "reverseBytes")
  do_intrinsic(_reverseBytes_l,           java_lang_Long,         reverseBytes_n
    /*  (symbol reverseBytes_name defined above) */
  do_intrinsic(_reverseBytes_c,           java_lang_Character,    reverseBytes_n
    /*  (symbol reverseBytes_name defined above) */
  do_intrinsic(_reverseBytes_s,           java_lang_Short,        reverseBytes_n
    /*  (symbol reverseBytes_name defined above) */

  do_intrinsic(_identityHashCode,         java_lang_System,       identityHashCo
   do_name(     identityHashCode_name,                           "identityHashCo
```

```
  do_intrinsic(_currentTimeMillis,        java_lang_System,       currentTimeMil
   do_name(     currentTimeMillis_name,                           "currentTimeMil
  do_intrinsic(_nanoTime,                 java_lang_System,       nanoTime_name,
   do_name(     nanoTime_name,                                    "nanoTime")

  TRACE_INTRINSICS(do_intrinsic, do_class, do_name, do_signature, do_alias)

  do_intrinsic(_arraycopy,                java_lang_System,       arraycopy_name
   do_name(     arraycopy_name,                                   "arraycopy")
   do_signature(arraycopy_signature,                              "(Ljava/lang/Ob
  do_intrinsic(_isInterrupted,            java_lang_Thread,       isInterrupted_
   do_name(     isInterrupted_name,                               "isInterrupted"
   do_signature(isInterrupted_signature,                          "(Z)Z")
  do_intrinsic(_currentThread,            java_lang_Thread,       currentThread_
   do_name(     currentThread_name,                               "currentThread"
   do_signature(currentThread_signature,                          "()Ljava/lang/T

  /* reflective intrinsics, for java/lang/Class, etc. */
  do_intrinsic(_isAssignableFrom,         java_lang_Class,        isAssignableFr
   do_name(     isAssignableFrom_name,                            "isAssignableFr
  do_intrinsic(_isInstance,               java_lang_Class,        isInstance_nam
   do_name(     isInstance_name,                                  "isInstance")
  do_intrinsic(_getModifiers,             java_lang_Class,        getModifiers_n
   do_name(     getModifiers_name,                                "getModifiers")
  do_intrinsic(_isInterface,              java_lang_Class,        isInterface_na
   do_name(     isInterface_name,                                 "isInterface")
  do_intrinsic(_isArray,                  java_lang_Class,        isArray_name,
   do_name(     isArray_name,                                     "isArray")
  do_intrinsic(_isPrimitive,              java_lang_Class,        isPrimitive_na
   do_name(     isPrimitive_name,                                 "isPrimitive")
  do_intrinsic(_getSuperclass,            java_lang_Class,        getSuperclass_
   do_name(     getSuperclass_name,                               "getSuperclass"
  do_intrinsic(_getComponentType,         java_lang_Class,        getComponentTy
   do_name(     getComponentType_name,                            "getComponentTy

  do_intrinsic(_getClassAccessFlags,      sun_reflect_Reflection, getClassAccess
   do_name(     getClassAccessFlags_name,                         "getClassAccess
  do_intrinsic(_getLength,                java_lang_reflect_Array, getLength_nam
   do_name(     getLength_name,                                   "getLength")

  do_intrinsic(_getCallerClass,           sun_reflect_Reflection, getCallerClass
   do_name(     getCallerClass_name,                              "getCallerClass
   do_signature(getCallerClass_signature,                         "(I)Ljava/lang/

  do_intrinsic(_newArray,                 java_lang_reflect_Array, newArray_name
   do_name(     newArray_name,                                    "newArray")
   do_signature(newArray_signature,                               "(Ljava/lang/C

  do_intrinsic(_copyOf,                   java_util_Arrays,       copyOf_name, c
   do_name(     copyOf_name,                                      "copyOf")
   do_signature(copyOf_signature,            "([Ljava/lang/Object;ILjava/lang/C

  do_intrinsic(_copyOfRange,              java_util_Arrays,       copyOfRange_na
   do_name(     copyOfRange_name,                                 "copyOfRange")
   do_signature(copyOfRange_signature,       "([Ljava/lang/Object;IILjava/lang/

  do_intrinsic(_equalsC,                  java_util_Arrays,       equals_name,
   do_signature(equalsC_signature,                                "([C[C)Z")

  do_intrinsic(_compareTo,                java_lang_String,       compareTo_name
   do_name(     compareTo_name,                                   "compareTo")
  do_intrinsic(_indexOf,                  java_lang_String,       indexOf_name,
   do_name(     indexOf_name,                                     "indexOf")
  do_intrinsic(_equals,                   java_lang_String,       equals_name, o

```



```
  do_class(java_nio_Buffer,               "java/nio/Buffer")
  do_intrinsic(_checkIndex,               java_nio_Buffer,        checkIndex_nam
   do_name(     checkIndex_name,                                  "checkIndex")

  /* java/lang/ref/Reference */
  do_intrinsic(_Reference_get,            java_lang_ref_Reference, get_name,

  /* support for sun.misc.Unsafe */
  do_class(sun_misc_Unsafe,               "sun/misc/Unsafe")

  do_intrinsic(_allocateInstance,         sun_misc_Unsafe,        allocateInstan
   do_name(     allocateInstance_name,                            "allocateInstan
   do_signature(allocateInstance_signature,   "(Ljava/lang/Class;)Ljava/lang/Obj
  do_intrinsic(_copyMemory,               sun_misc_Unsafe,        copyMemory_nam
   do_name(     copyMemory_name,                                  "copyMemory")
   do_signature(copyMemory_signature,        "(Ljava/lang/Object;JLjava/lang/Ob
  do_intrinsic(_park,                     sun_misc_Unsafe,        park_name, par
   do_name(     park_name,                                        "park")
   do_signature(park_signature,                                   "(ZJ)V")
  do_intrinsic(_unpark,                   sun_misc_Unsafe,        unpark_name, u
   do_name(     unpark_name,                                      "unpark")
   do_alias(    unpark_signature,                                /*(LObject;)V*/

  /* unsafe memory references (there are a lot of them...) */
  do_signature(getObject_signature,       "(Ljava/lang/Object;J)Ljava/lang/Objec
  do_signature(putObject_signature,       "(Ljava/lang/Object;JLjava/lang/Object
  do_signature(getBoolean_signature,      "(Ljava/lang/Object;J)Z")
  do_signature(putBoolean_signature,      "(Ljava/lang/Object;JZ)V")
  do_signature(getByte_signature,         "(Ljava/lang/Object;J)B")
  do_signature(putByte_signature,         "(Ljava/lang/Object;JB)V")
  do_signature(getShort_signature,        "(Ljava/lang/Object;J)S")
  do_signature(putShort_signature,        "(Ljava/lang/Object;JS)V")
  do_signature(getChar_signature,         "(Ljava/lang/Object;J)C")
  do_signature(putChar_signature,         "(Ljava/lang/Object;JC)V")
  do_signature(getInt_signature,          "(Ljava/lang/Object;J)I")
  do_signature(putInt_signature,          "(Ljava/lang/Object;JI)V")
  do_signature(getLong_signature,         "(Ljava/lang/Object;J)J")
  do_signature(putLong_signature,         "(Ljava/lang/Object;JJ)V")
  do_signature(getFloat_signature,        "(Ljava/lang/Object;J)F")
  do_signature(putFloat_signature,        "(Ljava/lang/Object;JF)V")
  do_signature(getDouble_signature,       "(Ljava/lang/Object;J)D")
  do_signature(putDouble_signature,       "(Ljava/lang/Object;JD)V")

  do_name(getObject_name,"getObject")           do_name(putObject_name,"putObjec
  do_name(getBoolean_name,"getBoolean")         do_name(putBoolean_name,"putBool
  do_name(getByte_name,"getByte")               do_name(putByte_name,"putByte")
  do_name(getShort_name,"getShort")             do_name(putShort_name,"putShort"
  do_name(getChar_name,"getChar")               do_name(putChar_name,"putChar")
  do_name(getInt_name,"getInt")                 do_name(putInt_name,"putInt")
  do_name(getLong_name,"getLong")               do_name(putLong_name,"putLong")
  do_name(getFloat_name,"getFloat")             do_name(putFloat_name,"putFloat"
  do_name(getDouble_name,"getDouble")           do_name(putDouble_name,"putDoubl

  do_intrinsic(_getObject,                sun_misc_Unsafe,        getObject_name
  do_intrinsic(_getBoolean,               sun_misc_Unsafe,        getBoolean_nam
  do_intrinsic(_getByte,                  sun_misc_Unsafe,        getByte_name,
  do_intrinsic(_getShort,                 sun_misc_Unsafe,        getShort_name,
  do_intrinsic(_getChar,                  sun_misc_Unsafe,        getChar_name,
  do_intrinsic(_getInt,                   sun_misc_Unsafe,        getInt_name, g
  do_intrinsic(_getLong,                  sun_misc_Unsafe,        getLong_name,
  do_intrinsic(_getFloat,                 sun_misc_Unsafe,        getFloat_name,
  do_intrinsic(_getDouble,                sun_misc_Unsafe,        getDouble_name
  do_intrinsic(_putObject,                sun_misc_Unsafe,        putObject_name
  do_intrinsic(_putBoolean,               sun_misc_Unsafe,        putBoolean_nam
  do_intrinsic(_putByte,                  sun_misc_Unsafe,        putByte_name,
  do_intrinsic(_putShort,                 sun_misc_Unsafe,        putShort_name,
```

```
  do_intrinsic(_putChar,                  sun_misc_Unsafe,        putChar_name,
  do_intrinsic(_putInt,                   sun_misc_Unsafe,        putInt_name, p
  do_intrinsic(_putLong,                  sun_misc_Unsafe,        putLong_name,
  do_intrinsic(_putFloat,                 sun_misc_Unsafe,        putFloat_name,
  do_intrinsic(_putDouble,                sun_misc_Unsafe,        putDouble_name

  do_name(getObjectVolatile_name,"getObjectVolatile")   do_name(putObjectVolatil
  do_name(getBooleanVolatile_name,"getBooleanVolatile") do_name(putBooleanVolati
  do_name(getByteVolatile_name,"getByteVolatile")       do_name(putByteVolatile_
  do_name(getShortVolatile_name,"getShortVolatile")     do_name(putShortVolatile
  do_name(getCharVolatile_name,"getCharVolatile")       do_name(putCharVolatile_
  do_name(getIntVolatile_name,"getIntVolatile")         do_name(putIntVolatile_n
  do_name(getLongVolatile_name,"getLongVolatile")       do_name(putLongVolatile_
  do_name(getFloatVolatile_name,"getFloatVolatile")     do_name(putFloatVolatile
  do_name(getDoubleVolatile_name,"getDoubleVolatile")   do_name(putDoubleVolatil

  do_intrinsic(_getObjectVolatile,        sun_misc_Unsafe,        getObjectVolat
  do_intrinsic(_getBooleanVolatile,       sun_misc_Unsafe,        getBooleanVola
  do_intrinsic(_getByteVolatile,          sun_misc_Unsafe,        getByteVolatil
  do_intrinsic(_getShortVolatile,         sun_misc_Unsafe,        getShortVolati
  do_intrinsic(_getCharVolatile,          sun_misc_Unsafe,        getCharVolatil
  do_intrinsic(_getIntVolatile,           sun_misc_Unsafe,        getIntVolatile
  do_intrinsic(_getLongVolatile,          sun_misc_Unsafe,        getLongVolatil
  do_intrinsic(_getFloatVolatile,         sun_misc_Unsafe,        getFloatVolati
  do_intrinsic(_getDoubleVolatile,        sun_misc_Unsafe,        getDoubleVolat
  do_intrinsic(_putObjectVolatile,        sun_misc_Unsafe,        putObjectVolat
  do_intrinsic(_putBooleanVolatile,       sun_misc_Unsafe,        putBooleanVola
  do_intrinsic(_putByteVolatile,          sun_misc_Unsafe,        putByteVolatil
  do_intrinsic(_putShortVolatile,         sun_misc_Unsafe,        putShortVolati
  do_intrinsic(_putCharVolatile,          sun_misc_Unsafe,        putCharVolatil
  do_intrinsic(_putIntVolatile,           sun_misc_Unsafe,        putIntVolatile
  do_intrinsic(_putLongVolatile,          sun_misc_Unsafe,        putLongVolatil
  do_intrinsic(_putFloatVolatile,         sun_misc_Unsafe,        putFloatVolati
  do_intrinsic(_putDoubleVolatile,        sun_misc_Unsafe,        putDoubleVolat

  /* %%% these are redundant except perhaps for getAddress, but Unsafe has nativ
  do_signature(getByte_raw_signature,     "(J)B")
  do_signature(putByte_raw_signature,     "(JB)V")
  do_signature(getShort_raw_signature,    "(J)S")
  do_signature(putShort_raw_signature,    "(JS)V")
  do_signature(getChar_raw_signature,     "(J)C")
  do_signature(putChar_raw_signature,     "(JC)V")
  do_signature(putInt_raw_signature,      "(JI)V")
      do_alias(getLong_raw_signature,    /*(J)J*/ long_long_signature)
      do_alias(putLong_raw_signature,    /*(JJ)V*/ long_long_void_signature)
  do_signature(getFloat_raw_signature,    "(J)F")
  do_signature(putFloat_raw_signature,    "(JF)V")
      do_alias(getDouble_raw_signature,  /*(J)D*/ long_double_signature)
  do_signature(putDouble_raw_signature,   "(JD)V")
      do_alias(getAddress_raw_signature, /*(J)J*/ long_long_signature)
      do_alias(putAddress_raw_signature, /*(JJ)V*/ long_long_void_signature)

   do_name(    getAddress_name,           "getAddress")
   do_name(    putAddress_name,           "putAddress")

  do_intrinsic(_getByte_raw,              sun_misc_Unsafe,        getByte_name,
  do_intrinsic(_getShort_raw,             sun_misc_Unsafe,        getShort_name,
  do_intrinsic(_getChar_raw,              sun_misc_Unsafe,        getChar_name,
  do_intrinsic(_getInt_raw,               sun_misc_Unsafe,        getInt_name, l
  do_intrinsic(_getLong_raw,              sun_misc_Unsafe,        getLong_name,
  do_intrinsic(_getFloat_raw,             sun_misc_Unsafe,        getFloat_name,
  do_intrinsic(_getDouble_raw,            sun_misc_Unsafe,        getDouble_name
  do_intrinsic(_getAddress_raw,           sun_misc_Unsafe,        getAddress_nam
  do_intrinsic(_putByte_raw,              sun_misc_Unsafe,        putByte_name,
  do_intrinsic(_putShort_raw,             sun_misc_Unsafe,        putShort_name,
  do_intrinsic(_putChar_raw,              sun_misc_Unsafe,        putChar_name,
```



```
  do_intrinsic(_putInt_raw,               sun_misc_Unsafe,        putInt_name, p
  do_intrinsic(_putLong_raw,              sun_misc_Unsafe,        putLong_name,
  do_intrinsic(_putFloat_raw,             sun_misc_Unsafe,        putFloat_name,
  do_intrinsic(_putDouble_raw,            sun_misc_Unsafe,        putDouble_name
  do_intrinsic(_putAddress_raw,           sun_misc_Unsafe,        putAddress_nam

  do_intrinsic(_compareAndSwapObject,     sun_misc_Unsafe,        compareAndSwap
   do_name(     compareAndSwapObject_name,                       "compareAndSwap
   do_signature(compareAndSwapObject_signature,  "(Ljava/lang/Object;JLjava/lang
  do_intrinsic(_compareAndSwapLong,       sun_misc_Unsafe,        compareAndSwap
   do_name(     compareAndSwapLong_name,                         "compareAndSwap
   do_signature(compareAndSwapLong_signature,                    "(Ljava/lang/Ob
  do_intrinsic(_compareAndSwapInt,        sun_misc_Unsafe,        compareAndSwap
   do_name(     compareAndSwapInt_name,                          "compareAndSwap
   do_signature(compareAndSwapInt_signature,                     "(Ljava/lang/Ob
  do_intrinsic(_putOrderedObject,         sun_misc_Unsafe,        putOrderedObje
   do_name(     putOrderedObject_name,                           "putOrderedObje
   do_alias(    putOrderedObject_signature,                     /*(LObject;JLObj
  do_intrinsic(_putOrderedLong,           sun_misc_Unsafe,        putOrderedLong
   do_name(     putOrderedLong_name,                             "putOrderedLong
   do_alias(    putOrderedLong_signature,                       /*(Ljava/lang/Ob
  do_intrinsic(_putOrderedInt,            sun_misc_Unsafe,        putOrderedInt_
   do_name(     putOrderedInt_name,                              "putOrderedInt"
   do_alias(    putOrderedInt_signature,                        /*(Ljava/lang/Ob

  /* prefetch_signature is shared by all prefetch variants */
  do_signature( prefetch_signature,        "(Ljava/lang/Object;J)V")

  do_intrinsic(_prefetchRead,             sun_misc_Unsafe,        prefetchRead_n
   do_name(     prefetchRead_name,                               "prefetchRead")
  do_intrinsic(_prefetchWrite,            sun_misc_Unsafe,        prefetchWrite_
   do_name(     prefetchWrite_name,                              "prefetchWrite"
  do_intrinsic(_prefetchReadStatic,       sun_misc_Unsafe,        prefetchReadSt
   do_name(     prefetchReadStatic_name,                         "prefetchReadSt
  do_intrinsic(_prefetchWriteStatic,      sun_misc_Unsafe,        prefetchWriteS
   do_name(     prefetchWriteStatic_name,                        "prefetchWriteS
    /*== LAST_COMPILER_INLINE*/
    /*the compiler does have special inlining code for these; bytecode inline is

  do_intrinsic(_fillInStackTrace,         java_lang_Throwable, fillInStackTrace_

  do_intrinsic(_StringBuilder_void,   java_lang_StringBuilder, object_initialize
  do_intrinsic(_StringBuilder_int,    java_lang_StringBuilder, object_initialize
  do_intrinsic(_StringBuilder_String, java_lang_StringBuilder, object_initialize

  do_intrinsic(_StringBuilder_append_char,   java_lang_StringBuilder, append_nam
  do_intrinsic(_StringBuilder_append_int,    java_lang_StringBuilder, append_nam
  do_intrinsic(_StringBuilder_append_String, java_lang_StringBuilder, append_nam

  do_intrinsic(_StringBuilder_toString, java_lang_StringBuilder, toString_name,

  do_intrinsic(_StringBuffer_void,   java_lang_StringBuffer, object_initializer_
  do_intrinsic(_StringBuffer_int,    java_lang_StringBuffer, object_initializer_
  do_intrinsic(_StringBuffer_String, java_lang_StringBuffer, object_initializer_

  do_intrinsic(_StringBuffer_append_char,   java_lang_StringBuffer, append_name,
  do_intrinsic(_StringBuffer_append_int,    java_lang_StringBuffer, append_name,
  do_intrinsic(_StringBuffer_append_String, java_lang_StringBuffer, append_name,

  do_intrinsic(_StringBuffer_toString,  java_lang_StringBuffer, toString_name, v

  do_intrinsic(_Integer_toString,      java_lang_Integer, toString_name, int_Str

  do_intrinsic(_String_String, java_lang_String, object_initializer_name, string

  do_intrinsic(_Object_init,              java_lang_Object, object_initializer_n
```

```
  /*    (symbol object_initializer_name defined above) */

  do_intrinsic(_invoke,                   java_lang_reflect_Method, invoke_name,
  /*   (symbols invoke_name and invoke_signature defined above) */
  do_intrinsic(_checkSpreadArgument,      java_lang_invoke_MethodHandleNatives,
   do_name(    checkSpreadArgument_name,       "checkSpreadArgument")
   do_name(    checkSpreadArgument_signature,  "(Ljava/lang/Object;I)V")
  do_intrinsic(_invokeExact,              java_lang_invoke_MethodHandle, invokeE
  do_intrinsic(_invokeGeneric,            java_lang_invoke_MethodHandle, invokeG
  do_intrinsic(_invokeVarargs,            java_lang_invoke_MethodHandle, invokeV
  do_intrinsic(_invokeDynamic,            java_lang_invoke_InvokeDynamic, star_n

  do_intrinsic(_selectAlternative,        java_lang_invoke_MethodHandleImpl, sel

  /* unboxing methods: */
  do_intrinsic(_booleanValue,             java_lang_Boolean,      booleanValue_n
   do_name(     booleanValue_name,       "booleanValue")
  do_intrinsic(_byteValue,                java_lang_Byte,         byteValue_name
   do_name(     byteValue_name,          "byteValue")
  do_intrinsic(_charValue,                java_lang_Character,    charValue_name
   do_name(     charValue_name,          "charValue")
  do_intrinsic(_shortValue,               java_lang_Short,        shortValue_nam
   do_name(     shortValue_name,         "shortValue")
  do_intrinsic(_intValue,                 java_lang_Integer,      intValue_name,
   do_name(     intValue_name,           "intValue")
  do_intrinsic(_longValue,                java_lang_Long,         longValue_name
   do_name(     longValue_name,          "longValue")
  do_intrinsic(_floatValue,               java_lang_Float,        floatValue_nam
   do_name(     floatValue_name,         "floatValue")
  do_intrinsic(_doubleValue,              java_lang_Double,       doubleValue_na
   do_name(     doubleValue_name,        "doubleValue")

  /* boxing methods: */
   do_name(    valueOf_name,            "valueOf")
  do_intrinsic(_Boolean_valueOf,          java_lang_Boolean,      valueOf_name,
   do_name(     Boolean_valueOf_signature,                        "(Z)Ljava/lang/
  do_intrinsic(_Byte_valueOf,             java_lang_Byte,         valueOf_name,
   do_name(     Byte_valueOf_signature,                           "(B)Ljava/lang/
  do_intrinsic(_Character_valueOf,        java_lang_Character,    valueOf_name,
   do_name(     Character_valueOf_signature,                      "(C)Ljava/lang/
  do_intrinsic(_Short_valueOf,            java_lang_Short,        valueOf_name,
   do_name(     Short_valueOf_signature,                          "(S)Ljava/lang/
  do_intrinsic(_Integer_valueOf,          java_lang_Integer,      valueOf_name,
   do_name(     Integer_valueOf_signature,                        "(I)Ljava/lang/
  do_intrinsic(_Long_valueOf,             java_lang_Long,         valueOf_name,
   do_name(     Long_valueOf_signature,                           "(J)Ljava/lang/
  do_intrinsic(_Float_valueOf,            java_lang_Float,        valueOf_name,
   do_name(     Float_valueOf_signature,                          "(F)Ljava/lang/
  do_intrinsic(_Double_valueOf,           java_lang_Double,       valueOf_name,
   do_name(     Double_valueOf_signature,                         "(D)Ljava/lang/

    /*end*/



// Class vmSymbols

class vmSymbols: AllStatic {
  friend class vmIntrinsics;
  friend class VMStructs;
 public:
  // enum for figuring positions and size of array holding Symbol*s
  enum SID {
    NO_SID = 0,
```



```
    #define VM_SYMBOL_ENUM(name, string) VM_SYMBOL_ENUM_NAME(name),
    VM_SYMBOLS_DO(VM_SYMBOL_ENUM, VM_ALIAS_IGNORE)
    #undef VM_SYMBOL_ENUM

    SID_LIMIT,

    #define VM_ALIAS_ENUM(name, def) VM_SYMBOL_ENUM_NAME(name) = VM_SYMBOL_ENUM_
    VM_SYMBOLS_DO(VM_SYMBOL_IGNORE, VM_ALIAS_ENUM)
    #undef VM_ALIAS_ENUM

    FIRST_SID = NO_SID + 1
  };
  enum {
    log2_SID_LIMIT = 10         // checked by an assert at start-up
  };

 private:
  // The symbol array
  static Symbol* _symbols[];

  // Field signatures indexed by BasicType.
  static Symbol* _type_signatures[T_VOID+1];

 public:
  // Initialization
  static void initialize(TRAPS);
  // Accessing
  #define VM_SYMBOL_DECLARE(name, ignore)                 \
    static Symbol* name() {                               \
      return _symbols[VM_SYMBOL_ENUM_NAME(name)];         \
    }
  VM_SYMBOLS_DO(VM_SYMBOL_DECLARE, VM_SYMBOL_DECLARE)
  #undef VM_SYMBOL_DECLARE

  // Sharing support
  static void symbols_do(SymbolClosure* f);
  static void serialize(SerializeOopClosure* soc);

  static Symbol* type_signature(BasicType t) {
    assert((uint)t < T_VOID+1, "range check");
    assert(_type_signatures[t] != NULL, "domain check");
    return _type_signatures[t];
  }
  // inverse of type_signature; returns T_OBJECT if s is not recognized
  static BasicType signature_type(Symbol* s);

  static Symbol* symbol_at(SID id) {
    assert(id >= FIRST_SID && id < SID_LIMIT, "oob");
    assert(_symbols[id] != NULL, "init");
    return _symbols[id];
  }

  // Returns symbol's SID if one is assigned, else NO_SID.
  static SID find_sid(Symbol* symbol);
  static SID find_sid(const char* symbol_name);

#ifndef PRODUCT
  // No need for this in the product:
  static const char* name_for(SID sid);
#endif //PRODUCT
};

// VM Intrinsic ID's uniquely identify some very special methods
class vmIntrinsics: AllStatic {
  friend class vmSymbols;
  friend class ciObjectFactory;
```

```
1054  public:
1055   // Accessing
1056   enum ID {
1057     _none = 0,                      // not an intrinsic (default answer)

1059     #define VM_INTRINSIC_ENUM(id, klass, name, sig, flags)  id,
1060     VM_INTRINSICS_DO(VM_INTRINSIC_ENUM,
1061                      VM_SYMBOL_IGNORE, VM_SYMBOL_IGNORE, VM_SYMBOL_IGNORE, VM_AL
1062     #undef VM_INTRINSIC_ENUM

1064     ID_LIMIT,
1065     LAST_COMPILER_INLINE = _prefetchWriteStatic,
1066     FIRST_ID = _none + 1
1067   };

1069   enum Flags {
1070     // AccessFlags syndromes relevant to intrinsics.
1071     F_none = 0,
1072     F_R,                         // !static ?native !synchronized (R="regular")
1073     F_S,                         //  static ?native !synchronized
1074     F_Y,                         // !static ?native  synchronized
1075     F_RN,                        // !static  native !synchronized
1076     F_SN,                        //  static  native !synchronized
1077     F_RNY,                       // !static  native  synchronized

1079     FLAG_LIMIT
1080   };
1081   enum {
1082     log2_FLAG_LIMIT = 4          // checked by an assert at start-up
1083   };

1085 public:
1086   static ID ID_from(int raw_id) {
1087     assert(raw_id >= (int)_none && raw_id < (int)ID_LIMIT,
1088            "must be a valid intrinsic ID");
1089     return (ID)raw_id;
1090   }

1092   static const char* name_at(ID id);

1094 private:
1095   static ID find_id_impl(vmSymbols::SID holder,
1096                          vmSymbols::SID name,
1097                          vmSymbols::SID sig,
1098                          jshort flags);

1100 public:
1101   // Given a method's class, name, signature, and access flags, report its ID.
1102   static ID find_id(vmSymbols::SID holder,
1103                     vmSymbols::SID name,
1104                     vmSymbols::SID sig,
1105                     jshort flags) {
1106     ID id = find_id_impl(holder, name, sig, flags);
1107 #ifdef ASSERT
1108     // ID _none does not hold the following asserts.
1109     if (id == _none)  return id;
1110 #endif
1111     assert(    class_for(id) == holder, "correct id");
1112     assert(     name_for(id) == name,   "correct id");
1113     assert(signature_for(id) == sig,    "correct id");
1114     return id;
1115   }

1117   static void verify_method(ID actual_id, methodOop m) PRODUCT_RETURN;
```



```
1119   // Find out the symbols behind an intrinsic:
1120   static vmSymbols::SID     class_for(ID id);
1121   static vmSymbols::SID      name_for(ID id);
1122   static vmSymbols::SID signature_for(ID id);
1123   static Flags              flags_for(ID id);

1125   static const char* short_name_as_C_string(ID id, char* buf, int size);

1127   // Access to intrinsic methods:
1128   static methodOop method_for(ID id);

1130   // Wrapper object methods:
1131   static ID for_boxing(BasicType type);
1132   static ID for_unboxing(BasicType type);

1134   // Raw conversion:
1135   static ID for_raw_conversion(BasicType src, BasicType dest);
1136 };

1138 #endif // SHARE_VM_CLASSFILE_VMSYMBOLS_HPP
```

```
*********************************************************
   23360 Thu Jun 14 10:53:39 2012
new/src/share/vm/runtime/java.cpp
*********************************************************
______unchanged_portion_omitted_

 662 JDK_Version JDK_Version::_current;
 663 const char* JDK_Version::_runtime_name;
 664 #endif /* ! codereview */

 666 void JDK_Version::initialize() {
 667   jdk_version_info info;
 668   assert(!_current.is_valid(), "Don't initialize twice");

 670   void *lib_handle = os::native_java_library();
 671   jdk_version_info_fn_t func = CAST_TO_FN_PTR(jdk_version_info_fn_t,
 672      os::dll_lookup(lib_handle, "JDK_GetVersionInfo0"));

 674   if (func == NULL) {
 675     // JDK older than 1.6
 676     _current._partially_initialized = true;
 677   } else {
 678     (*func)(&info, sizeof(info));

 680     int major = JDK_VERSION_MAJOR(info.jdk_version);
 681     int minor = JDK_VERSION_MINOR(info.jdk_version);
 682     int micro = JDK_VERSION_MICRO(info.jdk_version);
 683     int build = JDK_VERSION_BUILD(info.jdk_version);
 684     if (major == 1 && minor > 4) {
 685       // We represent "1.5.0" as "5.0", but 1.4.2 as itself.
 686       major = minor;
 687       minor = micro;
 688       micro = 0;
 689     }
 690     _current = JDK_Version(major, minor, micro, info.update_version,
 691                            info.special_update_version, build,
 692                            info.thread_park_blocker == 1,
 693                            info.post_vm_init_hook_enabled == 1,
 694                            info.pending_list_uses_discovered_field == 1);
 695   }
 696 }

 698 void JDK_Version::fully_initialize(
 699     uint8_t major, uint8_t minor, uint8_t micro, uint8_t update) {
 700   // This is only called when current is less than 1.6 and we've gotten
 701   // far enough in the initialization to determine the exact version.
 702   assert(major < 6, "not needed for JDK version >= 6");
 703   assert(is_partially_initialized(), "must not initialize");
 704   if (major < 5) {
 705     // JDK verison sequence: 1.2.x, 1.3.x, 1.4.x, 5.0.x, 6.0.x, etc.
 706     micro = minor;
 707     minor = major;
 708     major = 1;
 709   }
 710   _current = JDK_Version(major, minor, micro, update);
 711 }

 713 void JDK_Version_init() {
 714   JDK_Version::initialize();
 715 }

 717 static int64_t encode_jdk_version(const JDK_Version& v) {
 718   return
 719     ((int64_t)v.major_version()         << (BitsPerByte * 5)) |
 720     ((int64_t)v.minor_version()         << (BitsPerByte * 4)) |
 721     ((int64_t)v.micro_version()         << (BitsPerByte * 3)) |
```



```
 722     ((int64_t)v.update_version()        << (BitsPerByte * 2)) |
 723     ((int64_t)v.special_update_version() << (BitsPerByte * 1)) |
 724     ((int64_t)v.build_number()          << (BitsPerByte * 0));
 725 }

 727 int JDK_Version::compare(const JDK_Version& other) const {
 728   assert(is_valid() && other.is_valid(), "Invalid version (uninitialized?)");
 729   if (!is_partially_initialized() && other.is_partially_initialized()) {
 730     return -(other.compare(*this)); // flip the comparators
 731   }
 732   assert(!other.is_partially_initialized(), "Not initialized yet");
 733   if (is_partially_initialized()) {
 734     assert(other.major_version() >= 6,
 735            "Invalid JDK version comparison during initialization");
 736     return -1;
 737   } else {
 738     uint64_t e = encode_jdk_version(*this);
 739     uint64_t o = encode_jdk_version(other);
 740     return (e > o) ? 1 : ((e == o) ? 0 : -1);
 741   }
 742 }

 744 void JDK_Version::to_string(char* buffer, size_t buflen) const {
 745   size_t index = 0;
 746   if (!is_valid()) {
 747     jio_snprintf(buffer, buflen, "%s", "(uninitialized)");
 748   } else if (is_partially_initialized()) {
 749     jio_snprintf(buffer, buflen, "%s", "(uninitialized) pre-1.6.0");
 750   } else {
 751     index += jio_snprintf(
 752         &buffer[index], buflen - index, "%d.%d", _major, _minor);
 753     if (_micro > 0) {
 754       index += jio_snprintf(&buffer[index], buflen - index, ".%d", _micro);
 755     }
 756     if (_update > 0) {
 757       index += jio_snprintf(&buffer[index], buflen - index, "_%02d", _update);
 758     }
 759     if (_special > 0) {
 760       index += jio_snprintf(&buffer[index], buflen - index, "%c", _special);
 761     }
 762     if (_build > 0) {
 763       index += jio_snprintf(&buffer[index], buflen - index, "-b%02d", _build);
 764     }
 765   }
 766 }
```

```
**********************************************************
    8478 Thu Jun 14 10:53:40 2012
new/src/share/vm/runtime/java.hpp
**********************************************************
```

```
/*
 * Copyright (c) 1997, 2011, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 *
 */

#ifndef SHARE_VM_RUNTIME_JAVA_HPP
#define SHARE_VM_RUNTIME_JAVA_HPP

#include "runtime/os.hpp"

// Register function to be called by before_exit
extern "C" { void register_on_exit_function(void (*func)(void)) ;}

// Execute code before all handles are released and thread is killed; prologue t
extern void before_exit(JavaThread * thread);

// Forced VM exit (i.e, internal error or JVM_Exit)
extern void vm_exit(int code);

// Wrapper for ::exit()
extern void vm_direct_exit(int code);

// Shutdown the VM but do not exit the process
extern void vm_shutdown();
// Shutdown the VM and abort the process
extern void vm_abort(bool dump_core=true);

// Trigger any necessary notification of the VM being shutdown
extern void notify_vm_shutdown();

// VM exit if error occurs during initialization of VM
extern void vm_exit_during_initialization(Handle exception);
extern void vm_exit_during_initialization(Symbol* exception_name, const char* me
extern void vm_exit_during_initialization(const char* error, const char* message
extern void vm_shutdown_during_initialization(const char* error, const char* mes

/**
 * Discovering the JDK_Version during initialization is tricky when the
 * running JDK is less than JDK6.  For JDK6 and greater, a "GetVersion"
 * function exists in libjava.so and we simply call it during the
 * 'initialize()' call to find the version.  For JDKs with version < 6, no
 * such call exists and we have to probe the JDK in order to determine
 * the exact version.  This probing cannot happen during late in
```



```
 * the VM initialization process so there's a period of time during
 * initialization when we don't know anything about the JDK version other than
 * that it less than version 6.  This is the "partially initialized" time,
 * when we can answer only certain version queries (such as, is the JDK
 * version greater than 5?  Answer: no).  Once the JDK probing occurs, we
 * know the version and are considered fully initialized.
 */
class JDK_Version VALUE_OBJ_CLASS_SPEC {
  friend class VMStructs;
  friend class Universe;
  friend void JDK_Version_init();
 private:

  static JDK_Version _current;
  static const char* _runtime_name;
#endif /* ! codereview */

  // In this class, we promote the minor version of release to be the
  // major version for releases >= 5 in anticipation of the JDK doing the
  // same thing.  For example, we represent "1.5.0" as major version 5 (we
  // drop the leading 1 and use 5 as the 'major').

  uint8_t _major;
  uint8_t _minor;
  uint8_t _micro;
  uint8_t _update;
  uint8_t _special;
  uint8_t _build;

  // If partially initialized, the above fields are invalid and we know
  // that we're less than major version 6.
  bool _partially_initialized;

  bool _thread_park_blocker;
  bool _pending_list_uses_discovered_field;
  bool _post_vm_init_hook_enabled;

  bool is_valid() const {
    return (_major != 0 || _partially_initialized);
  }

  // initializes or partially initializes the _current static field
  static void initialize();

  // Completes initialization for a pre-JDK6 version.
  static void fully_initialize(uint8_t major, uint8_t minor = 0,
                               uint8_t micro = 0, uint8_t update = 0);

 public:

  // Returns true if the the current version has only been partially initialized
  static bool is_partially_initialized() {
    return _current._partially_initialized;
  }

  JDK_Version() : _major(0), _minor(0), _micro(0), _update(0),
                  _special(0), _build(0), _partially_initialized(false),
                  _thread_park_blocker(false), _post_vm_init_hook_enabled(false)
                  _pending_list_uses_discovered_field(false) {}

  JDK_Version(uint8_t major, uint8_t minor = 0, uint8_t micro = 0,
              uint8_t update = 0, uint8_t special = 0, uint8_t build = 0,
              bool thread_park_blocker = false, bool post_vm_init_hook_enabled =
              bool pending_list_uses_discovered_field = false) :
      _major(major), _minor(minor), _micro(micro), _update(update),
      _special(special), _build(build), _partially_initialized(false),
```

```
129       _thread_park_blocker(thread_park_blocker),
130       _post_vm_init_hook_enabled(post_vm_init_hook_enabled),
131       _pending_list_uses_discovered_field(pending_list_uses_discovered_field) {}

133   // Returns the current running JDK version
134   static JDK_Version current() { return _current; }

136   // Factory methods for convenience
137   static JDK_Version jdk(uint8_t m) {
138     return JDK_Version(m);
139   }

141   static JDK_Version jdk_update(uint8_t major, uint8_t update_number) {
142     return JDK_Version(major, 0, 0, update_number);
143   }

145   uint8_t major_version() const          { return _major; }
146   uint8_t minor_version() const          { return _minor; }
147   uint8_t micro_version() const          { return _micro; }
148   uint8_t update_version() const         { return _update; }
149   uint8_t special_update_version() const { return _special; }
150   uint8_t build_number() const           { return _build; }

152   bool supports_thread_park_blocker() const {
153     return _thread_park_blocker;
154   }
155   bool post_vm_init_hook_enabled() const {
156     return _post_vm_init_hook_enabled;
157   }
158   // For compatibility wrt pre-4965777 JDK's
159   bool pending_list_uses_discovered_field() const {
160     return _pending_list_uses_discovered_field;
161   }

163   // Performs a full ordering comparison using all fields (update, build, etc.)
164   int compare(const JDK_Version& other) const;

166   /**
167    * Performs comparison using only the major version, returning negative
168    * if the major version of 'this' is less than the parameter, 0 if it is
169    * equal, and a positive value if it is greater.
170    */
171   int compare_major(int version) const {
172     if (_partially_initialized) {
173       if (version >= 6) {
174         return -1;
175       } else {
176         assert(false, "Can't make this comparison during init time");
177         return -1; // conservative
178       }
179     } else {
180       return major_version() - version;
181     }
182   }

184   void to_string(char* buffer, size_t buflen) const;

186   static const char* runtime_name() {
187     return _runtime_name;
188   }
189   static void set_runtime_name(const char* name) {
190     _runtime_name = name;
191   }

193 #endif /* ! codereview */
194   // Convenience methods for queries on the current major/minor version
```



```
195   static bool is_jdk12x_version() {
196     return current().compare_major(2) == 0;
197   }

199   static bool is_jdk13x_version() {
200     return current().compare_major(3) == 0;
201   }

203   static bool is_jdk14x_version() {
204     return current().compare_major(4) == 0;
205   }

207   static bool is_jdk15x_version() {
208     return current().compare_major(5) == 0;
209   }

211   static bool is_jdk16x_version() {
212     return current().compare_major(6) == 0;
213   }

215   static bool is_jdk17x_version() {
216     return current().compare_major(7) == 0;
217   }

219   static bool is_jdk18x_version() {
220     return current().compare_major(8) == 0;
221   }

223   static bool is_gte_jdk13x_version() {
224     return current().compare_major(3) >= 0;
225   }

227   static bool is_gte_jdk14x_version() {
228     return current().compare_major(4) >= 0;
229   }

231   static bool is_gte_jdk15x_version() {
232     return current().compare_major(5) >= 0;
233   }

235   static bool is_gte_jdk16x_version() {
236     return current().compare_major(6) >= 0;
237   }

239   static bool is_gte_jdk17x_version() {
240     return current().compare_major(7) >= 0;
241   }

243   static bool is_gte_jdk18x_version() {
244     return current().compare_major(8) >= 0;
245   }
246 };

248 #endif // SHARE_VM_RUNTIME_JAVA_HPP
```

```
*******************************************************
  163517 Thu Jun 14 10:53:42 2012
new/src/share/vm/runtime/thread.cpp
*******************************************************
______unchanged_portion_omitted_

 999 char java_runtime_name[128] = "";

1001 // extract the JRE name from sun.misc.Version.java_runtime_name
1002 static const char* get_java_runtime_name(TRAPS) {
1003   klassOop k = SystemDictionary::find(vmSymbols::sun_misc_Version(),
1004                                       Handle(), Handle(), CHECK_AND_CLEAR_NULL);
1005   fieldDescriptor fd;
1006   bool found = k != NULL &&
1007                instanceKlass::cast(k)->find_local_field(vmSymbols::java_runtime_
1008                                                         vmSymbols::string_signat
1009   if (found) {
1010     oop name_oop = k->java_mirror()->obj_field(fd.offset());
1011     if (name_oop == NULL)
1012       return NULL;
1013     const char* name = java_lang_String::as_utf8_string(name_oop,
1014                                                         java_runtime_name,
1015                                                         sizeof(java_runtime_name
1016     return name;
1017   } else {
1018     return NULL;
1019   }
1020 }

1022 #endif /* ! codereview */
1023 // General purpose hook into Java code, run once when the VM is initialized.
1024 // The Java library method itself may be changed independently from the VM.
1025 static void call_postVMInitHook(TRAPS) {
1026   klassOop k = SystemDictionary::PostVMInitHook_klass();
1027   instanceKlassHandle klass (THREAD, k);
1028   if (klass.not_null()) {
1029     JavaValue result(T_VOID);
1030     JavaCalls::call_static(&result, klass, vmSymbols::run_method_name(),
1031                                            vmSymbols::void_method_signature(),
1032                                            CHECK);
1033   }
1034 }

1036 static void reset_vm_info_property(TRAPS) {
1037   // the vm info string
1038   ResourceMark rm(THREAD);
1039   const char *vm_info = VM_Version::vm_info_string();

1041   // java.lang.System class
1042   klassOop k =  SystemDictionary::resolve_or_fail(vmSymbols::java_lang_System(),
1043   instanceKlassHandle klass (THREAD, k);

1045   // setProperty arguments
1046   Handle key_str    = java_lang_String::create_from_str("java.vm.info", CHECK);
1047   Handle value_str  = java_lang_String::create_from_str(vm_info, CHECK);

1049   // return value
1050   JavaValue r(T_OBJECT);

1052   // public static String setProperty(String key, String value);
1053   JavaCalls::call_static(&r,
1054                          klass,
1055                          vmSymbols::setProperty_name(),
1056                          vmSymbols::string_string_string_signature(),
1057                          key_str,
1058                          value_str,
```



```
1059                          CHECK);
1060 }


1063 void JavaThread::allocate_threadObj(Handle thread_group, char* thread_name, bool
1064   assert(thread_group.not_null(), "thread group should be specified");
1065   assert(threadObj() == NULL, "should only create Java thread object once");

1067   klassOop k = SystemDictionary::resolve_or_fail(vmSymbols::java_lang_Thread(),
1068   instanceKlassHandle klass (THREAD, k);
1069   instanceHandle thread_oop = klass->allocate_instance_handle(CHECK);

1071   java_lang_Thread::set_thread(thread_oop(), this);
1072   java_lang_Thread::set_priority(thread_oop(), NormPriority);
1073   set_threadObj(thread_oop());

1075   JavaValue result(T_VOID);
1076   if (thread_name != NULL) {
1077     Handle name = java_lang_String::create_from_str(thread_name, CHECK);
1078     // Thread gets assigned specified name and null target
1079     JavaCalls::call_special(&result,
1080                             thread_oop,
1081                             klass,
1082                             vmSymbols::object_initializer_name(),
1083                             vmSymbols::threadgroup_string_void_signature(),
1084                             thread_group, // Argument 1
1085                             name,         // Argument 2
1086                             THREAD);
1087   } else {
1088     // Thread gets assigned name "Thread-nnn" and null target
1089     // (java.lang.Thread doesn't have a constructor taking only a ThreadGroup ar
1090     JavaCalls::call_special(&result,
1091                             thread_oop,
1092                             klass,
1093                             vmSymbols::object_initializer_name(),
1094                             vmSymbols::threadgroup_runnable_void_signature(),
1095                             thread_group, // Argument 1
1096                             Handle(),     // Argument 2
1097                             THREAD);
1098   }


1101   if (daemon) {
1102       java_lang_Thread::set_daemon(thread_oop());
1103   }

1105   if (HAS_PENDING_EXCEPTION) {
1106     return;
1107   }

1109   KlassHandle group(this, SystemDictionary::ThreadGroup_klass());
1110   Handle threadObj(this, this->threadObj());

1112   JavaCalls::call_special(&result,
1113                           thread_group,
1114                           group,
1115                           vmSymbols::add_method_name(),
1116                           vmSymbols::thread_void_signature(),
1117                           threadObj,          // Arg 1
1118                           THREAD);


1121 }

1123 // NamedThread --  non-JavaThread subclasses with multiple
1124 // uniquely named instances should derive from this.
```

```
1125 NamedThread::NamedThread() : Thread() {
1126   _name = NULL;
1127   _processed_thread = NULL;
1128 }

1130 NamedThread::~NamedThread() {
1131   if (_name != NULL) {
1132     FREE_C_HEAP_ARRAY(char, _name);
1133     _name = NULL;
1134   }
1135 }

1137 void NamedThread::set_name(const char* format, ...) {
1138   guarantee(_name == NULL, "Only get to set name once.");
1139   _name = NEW_C_HEAP_ARRAY(char, max_name_len);
1140   guarantee(_name != NULL, "alloc failure");
1141   va_list ap;
1142   va_start(ap, format);
1143   jio_vsnprintf(_name, max_name_len, format, ap);
1144   va_end(ap);
1145 }

1147 // ======= WatcherThread ========

1149 // The watcher thread exists to simulate timer interrupts.  It should
1150 // be replaced by an abstraction over whatever native support for
1151 // timer interrupts exists on the platform.

1153 WatcherThread* WatcherThread::_watcher_thread   = NULL;
1154 volatile bool  WatcherThread::_should_terminate = false;

1156 WatcherThread::WatcherThread() : Thread() {
1157   assert(watcher_thread() == NULL, "we can only allocate one WatcherThread");
1158   if (os::create_thread(this, os::watcher_thread)) {
1159     _watcher_thread = this;

1161     // Set the watcher thread to the highest OS priority which should not be
1162     // used, unless a Java thread with priority java.lang.Thread.MAX_PRIORITY
1163     // is created. The only normal thread using this priority is the reference
1164     // handler thread, which runs for very short intervals only.
1165     // If the VMThread's priority is not lower than the WatcherThread profiling
1166     // will be inaccurate.
1167     os::set_priority(this, MaxPriority);
1168     if (!DisableStartThread) {
1169       os::start_thread(this);
1170     }
1171   }
1172 }

1174 void WatcherThread::run() {
1175   assert(this == watcher_thread(), "just checking");

1177   this->record_stack_base_and_size();
1178   this->initialize_thread_local_storage();
1179   this->set_active_handles(JNIHandleBlock::allocate_block());
1180   while(!_should_terminate) {
1181     assert(watcher_thread() == Thread::current(),  "thread consistency check");
1182     assert(watcher_thread() == this,  "thread consistency check");

1184     // Calculate how long it'll be until the next PeriodicTask work
1185     // should be done, and sleep that amount of time.
1186     size_t time_to_wait = PeriodicTask::time_to_wait();

1188     // we expect this to timeout - we only ever get unparked when
1189     // we should terminate
1190     {
```



```
1191       OSThreadWaitState osts(this->osthread(), false /* not Object.wait() */);

1193       jlong prev_time = os::javaTimeNanos();
1194       for (;;) {
1195         int res= _SleepEvent->park(time_to_wait);
1196         if (res == OS_TIMEOUT || _should_terminate)
1197           break;
1198         // spurious wakeup of some kind
1199         jlong now = os::javaTimeNanos();
1200         time_to_wait -= (now - prev_time) / 1000000;
1201         if (time_to_wait <= 0)
1202           break;
1203         prev_time = now;
1204       }
1205     }

1207     if (is_error_reported()) {
1208       // A fatal error has happened, the error handler(VMError::report_and_die)
1209       // should abort JVM after creating an error log file. However in some
1210       // rare cases, the error handler itself might deadlock. Here we try to
1211       // kill JVM if the fatal error handler fails to abort in 2 minutes.
1212       //
1213       // This code is in WatcherThread because WatcherThread wakes up
1214       // periodically so the fatal error handler doesn't need to do anything;
1215       // also because the WatcherThread is less likely to crash than other
1216       // threads.

1218       for (;;) {
1219         if (!ShowMessageBoxOnError
1220          && (OnError == NULL || OnError[0] == '\0')
1221          && Arguments::abort_hook() == NULL) {
1222             os::sleep(this, 2 * 60 * 1000, false);
1223             fdStream err(defaultStream::output_fd());
1224             err.print_raw_cr("# [ timer expired, abort... ]");
1225             // skip atexit/vm_exit/vm_abort hooks
1226             os::die();
1227         }

1229         // Wake up 5 seconds later, the fatal handler may reset OnError or
1230         // ShowMessageBoxOnError when it is ready to abort.
1231         os::sleep(this, 5 * 1000, false);
1232       }
1233     }

1235     PeriodicTask::real_time_tick(time_to_wait);

1237     // If we have no more tasks left due to dynamic disenrollment,
1238     // shut down the thread since we don't currently support dynamic enrollment
1239     if (PeriodicTask::num_tasks() == 0) {
1240       _should_terminate = true;
1241     }
1242   }

1244   // Signal that it is terminated
1245   {
1246     MutexLockerEx mu(Terminator_lock, Mutex::_no_safepoint_check_flag);
1247     _watcher_thread = NULL;
1248     Terminator_lock->notify();
1249   }

1251   // Thread destructor usually does this..
1252   ThreadLocalStorage::set_thread(NULL);
1253 }

1255 void WatcherThread::start() {
1256   if (watcher_thread() == NULL) {
```

```
1257     _should_terminate = false;
1258     // Create the single instance of WatcherThread
1259     new WatcherThread();
1260   }
1261 }

1263 void WatcherThread::stop() {
1264   // it is ok to take late safepoints here, if needed
1265   MutexLocker mu(Terminator_lock);
1266   _should_terminate = true;
1267   OrderAccess::fence();  // ensure WatcherThread sees update in main loop

1269   Thread* watcher = watcher_thread();
1270   if (watcher != NULL)
1271     watcher->_SleepEvent->unpark();

1273   while(watcher_thread() != NULL) {
1274     // This wait should make safepoint checks, wait without a timeout,
1275     // and wait as a suspend-equivalent condition.
1276     //
1277     // Note: If the FlatProfiler is running, then this thread is waiting
1278     // for the WatcherThread to terminate and the WatcherThread, via the
1279     // FlatProfiler task, is waiting for the external suspend request on
1280     // this thread to complete. wait_for_ext_suspend_completion() will
1281     // eventually timeout, but that takes time. Making this wait a
1282     // suspend-equivalent condition solves that timeout problem.
1283     //
1284     Terminator_lock->wait(!Mutex::_no_safepoint_check_flag, 0,
1285                           Mutex::_as_suspend_equivalent_flag);
1286   }
1287 }

1289 void WatcherThread::print_on(outputStream* st) const {
1290   st->print("\"%s\" ", name());
1291   Thread::print_on(st);
1292   st->cr();
1293 }

1295 // ======= JavaThread ========

1297 // A JavaThread is a normal Java thread

1299 void JavaThread::initialize() {
1300   // Initialize fields

1302   // Set the claimed par_id to -1 (ie not claiming any par_ids)
1303   set_claimed_par_id(-1);

1305   set_saved_exception_pc(NULL);
1306   set_threadObj(NULL);
1307   _anchor.clear();
1308   set_entry_point(NULL);
1309   set_jni_functions(jni_functions());
1310   set_callee_target(NULL);
1311   set_vm_result(NULL);
1312   set_vm_result_2(NULL);
1313   set_vframe_array_head(NULL);
1314   set_vframe_array_last(NULL);
1315   set_deferred_locals(NULL);
1316   set_deopt_mark(NULL);
1317   set_deopt_nmethod(NULL);
1318   clear_must_deopt_id();
1319   set_monitor_chunks(NULL);
1320   set_next(NULL);
1321   set_thread_state(_thread_new);
1322   _terminated = _not_terminated;
```



```
1323   _privileged_stack_top = NULL;
1324   _array_for_gc = NULL;
1325   _suspend_equivalent = false;
1326   _in_deopt_handler = 0;
1327   _doing_unsafe_access = false;
1328   _stack_guard_state = stack_guard_unused;
1329   _exception_oop = NULL;
1330   _exception_pc  = 0;
1331   _exception_handler_pc = 0;
1332   _is_method_handle_return = 0;
1333   _jvmti_thread_state= NULL;
1334   _should_post_on_exceptions_flag = JNI_FALSE;
1335   _jvmti_get_loaded_classes_closure = NULL;
1336   _interp_only_mode    = 0;
1337   _special_runtime_exit_condition = _no_async_condition;
1338   _pending_async_exception = NULL;
1339   _is_compiling = false;
1340   _thread_stat = NULL;
1341   _thread_stat = new ThreadStatistics();
1342   _blocked_on_compilation = false;
1343   _jni_active_critical = 0;
1344   _do_not_unlock_if_synchronized = false;
1345   _cached_monitor_info = NULL;
1346   _parker = Parker::Allocate(this) ;

1348 #ifndef PRODUCT
1349   _jmp_ring_index = 0;
1350   for (int ji = 0 ; ji < jump_ring_buffer_size ; ji++ ) {
1351     record_jump(NULL, NULL, NULL, 0);
1352   }
1353 #endif /* PRODUCT */

1355   set_thread_profiler(NULL);
1356   if (FlatProfiler::is_active()) {
1357     // This is where we would decide to either give each thread it's own profile
1358     // or use one global one from FlatProfiler,
1359     // or up to some count of the number of profiled threads, etc.
1360     ThreadProfiler* pp = new ThreadProfiler();
1361     pp->engage();
1362     set_thread_profiler(pp);
1363   }

1365   // Setup safepoint state info for this thread
1366   ThreadSafepointState::create(this);

1368   debug_only(_java_call_counter = 0);

1370   // JVMTI PopFrame support
1371   _popframe_condition = popframe_inactive;
1372   _popframe_preserved_args = NULL;
1373   _popframe_preserved_args_size = 0;

1375   pd_initialize();
1376 }

1378 #ifndef SERIALGC
1379 SATBMarkQueueSet JavaThread::_satb_mark_queue_set;
1380 DirtyCardQueueSet JavaThread::_dirty_card_queue_set;
1381 #endif // !SERIALGC

1383 JavaThread::JavaThread(bool is_attaching_via_jni) :
1384   Thread()
1385 #ifndef SERIALGC
1386   , _satb_mark_queue(&_satb_mark_queue_set),
1387   _dirty_card_queue(&_dirty_card_queue_set)
1388 #endif // !SERIALGC
```

```
1389 {
1390   initialize();
1391   if (is_attaching_via_jni) {
1392     _jni_attach_state = _attaching_via_jni;
1393   } else {
1394     _jni_attach_state = _not_attaching_via_jni;
1395   }
1396   assert(_deferred_card_mark.is_empty(), "Default MemRegion ctor");
1397 }

1399 bool JavaThread::reguard_stack(address cur_sp) {
1400   if (_stack_guard_state != stack_guard_yellow_disabled) {
1401     return true; // Stack already guarded or guard pages not needed.
1402   }

1404   if (register_stack_overflow()) {
1405     // For those architectures which have separate register and
1406     // memory stacks, we must check the register stack to see if
1407     // it has overflowed.
1408     return false;
1409   }

1411   // Java code never executes within the yellow zone: the latter is only
1412   // there to provoke an exception during stack banging.  If java code
1413   // is executing there, either StackShadowPages should be larger, or
1414   // some exception code in c1, c2 or the interpreter isn't unwinding
1415   // when it should.
1416   guarantee(cur_sp > stack_yellow_zone_base(), "not enough space to reguard - in

1418   enable_stack_yellow_zone();
1419   return true;
1420 }

1422 bool JavaThread::reguard_stack(void) {
1423   return reguard_stack(os::current_stack_pointer());
1424 }


1427 void JavaThread::block_if_vm_exited() {
1428   if (_terminated == _vm_exited) {
1429     // _vm_exited is set at safepoint, and Threads_lock is never released
1430     // we will block here forever
1431     Threads_lock->lock_without_safepoint_check();
1432     ShouldNotReachHere();
1433   }
1434 }


1437 // Remove this ifdef when C1 is ported to the compiler interface.
1438 static void compiler_thread_entry(JavaThread* thread, TRAPS);

1440 JavaThread::JavaThread(ThreadFunction entry_point, size_t stack_sz) :
1441   Thread()
1442 #ifndef SERIALGC
1443   , _satb_mark_queue(&_satb_mark_queue_set),
1444   _dirty_card_queue(&_dirty_card_queue_set)
1445 #endif // !SERIALGC
1446 {
1447   if (TraceThreadEvents) {
1448     tty->print_cr("creating thread %p", this);
1449   }
1450   initialize();
1451   _jni_attach_state = _not_attaching_via_jni;
1452   set_entry_point(entry_point);
1453   // Create the native thread itself.
1454   // %note runtime_23
```



```
1455   os::ThreadType thr_type = os::java_thread;
1456   thr_type = entry_point == &compiler_thread_entry ? os::compiler_thread :
1457                                                      os::java_thread;
1458   os::create_thread(this, thr_type, stack_sz);

1460   // The _osthread may be NULL here because we ran out of memory (too many threa
1461   // We need to throw and OutOfMemoryError - however we cannot do this here beca
1462   // may hold a lock and all locks must be unlocked before throwing the exceptio
1463   // the exception consists of creating the exception object & initializing it,
1464   // will leave the VM via a JavaCall and then all locks must be unlocked).
1465   //
1466   // The thread is still suspended when we reach here. Thread must be explicit s
1467   // by creator! Furthermore, the thread must also explicitly be added to the Th
1468   // by calling Threads:add. The reason why this is not done here, is because th
1469   // object must be fully initialized (take a look at JVM_Start)
1470 }

1472 JavaThread::~JavaThread() {
1473   if (TraceThreadEvents) {
1474       tty->print_cr("terminate thread %p", this);
1475   }

1477   // JSR166 -- return the parker to the free list
1478   Parker::Release(_parker);
1479   _parker = NULL ;

1481   // Free any remaining  previous UnrollBlock
1482   vframeArray* old_array = vframe_array_last();

1484   if (old_array != NULL) {
1485     Deoptimization::UnrollBlock* old_info = old_array->unroll_block();
1486     old_array->set_unroll_block(NULL);
1487     delete old_info;
1488     delete old_array;
1489   }

1491   GrowableArray<jvmtiDeferredLocalVariableSet*>* deferred = deferred_locals();
1492   if (deferred != NULL) {
1493     // This can only happen if thread is destroyed before deoptimization occurs.
1494     assert(deferred->length() != 0, "empty array!");
1495     do {
1496       jvmtiDeferredLocalVariableSet* dlv = deferred->at(0);
1497       deferred->remove_at(0);
1498       // individual jvmtiDeferredLocalVariableSet are CHeapObj's
1499       delete dlv;
1500     } while (deferred->length() != 0);
1501     delete deferred;
1502   }

1504   // All Java related clean up happens in exit
1505   ThreadSafepointState::destroy(this);
1506   if (_thread_profiler != NULL) delete _thread_profiler;
1507   if (_thread_stat != NULL) delete _thread_stat;
1508 }


1511 // The first routine called by a new Java thread
1512 void JavaThread::run() {
1513   // initialize thread-local alloc buffer related fields
1514   this->initialize_tlab();

1516   // used to test validitity of stack trace backs
1517   this->record_base_of_stack_pointer();

1519   // Record real stack base and size.
1520   this->record_stack_base_and_size();
```

```
1522   // Initialize thread local storage; set before calling MutexLocker
1523   this->initialize_thread_local_storage();

1525   this->create_stack_guard_pages();

1527   this->cache_global_variables();

1529   // Thread is now sufficient initialized to be handled by the safepoint code as
1530   // in the VM. Change thread state from _thread_new to _thread_in_vm
1531   ThreadStateTransition::transition_and_fence(this, _thread_new, _thread_in_vm);

1533   assert(JavaThread::current() == this, "sanity check");
1534   assert(!Thread::current()->owns_locks(), "sanity check");

1536   DTRACE_THREAD_PROBE(start, this);

1538   // This operation might block. We call that after all safepoint checks for a n
1539   // been completed.
1540   this->set_active_handles(JNIHandleBlock::allocate_block());

1542   if (JvmtiExport::should_post_thread_life()) {
1543     JvmtiExport::post_thread_start(this);
1544   }

1546   EVENT_BEGIN(TraceEventThreadStart, event);
1547   EVENT_COMMIT(event,
1548      EVENT_SET(event, javalangthread, java_lang_Thread::thread_id(this->threadOb

1550   // We call another function to do the rest so we are sure that the stack addre
1551   // from there will be lower than the stack base just computed
1552   thread_main_inner();

1554   // Note, thread is no longer valid at this point!
1555 }


1558 void JavaThread::thread_main_inner() {
1559   assert(JavaThread::current() == this, "sanity check");
1560   assert(this->threadObj() != NULL, "just checking");

1562   // Execute thread entry point unless this thread has a pending exception
1563   // or has been stopped before starting.
1564   // Note: Due to JVM_StopThread we can have pending exceptions already!
1565   if (!this->has_pending_exception() &&
1566       !java_lang_Thread::is_stillborn(this->threadObj())) {
1567     {
1568       ResourceMark rm(this);
1569       this->set_native_thread_name(this->get_thread_name());
1570     }
1571     HandleMark hm(this);
1572     this->entry_point()(this, this);
1573   }

1575   DTRACE_THREAD_PROBE(stop, this);

1577   this->exit(false);
1578   delete this;
1579 }


1582 static void ensure_join(JavaThread* thread) {
1583   // We do not need to grap the Threads_lock, since we are operating on ourself.
1584   Handle threadObj(thread, thread->threadObj());
1585   assert(threadObj.not_null(), "java thread object must exist");
1586   ObjectLocker lock(threadObj, thread);
```



```
1587   // Ignore pending exception (ThreadDeath), since we are exiting anyway
1588   thread->clear_pending_exception();
1589   // Thread is exiting. So set thread_status field in  java.lang.Thread class to
1590   java_lang_Thread::set_thread_status(threadObj(), java_lang_Thread::TERMINATED)
1591   // Clear the native thread instance - this makes isAlive return false and allo
1592   // to complete once we've done the notify_all below
1593   java_lang_Thread::set_thread(threadObj(), NULL);
1594   lock.notify_all(thread);
1595   // Ignore pending exception (ThreadDeath), since we are exiting anyway
1596   thread->clear_pending_exception();
1597 }


1600 // For any new cleanup additions, please check to see if they need to be applied
1601 // cleanup_failed_attach_current_thread as well.
1602 void JavaThread::exit(bool destroy_vm, ExitType exit_type) {
1603   assert(this == JavaThread::current(),  "thread consistency check");
1604   if (!InitializeJavaLangSystem) return;

1606   HandleMark hm(this);
1607   Handle uncaught_exception(this, this->pending_exception());
1608   this->clear_pending_exception();
1609   Handle threadObj(this, this->threadObj());
1610   assert(threadObj.not_null(), "Java thread object should be created");

1612   if (get_thread_profiler() != NULL) {
1613     get_thread_profiler()->disengage();
1614     ResourceMark rm;
1615     get_thread_profiler()->print(get_thread_name());
1616   }


1619   // FIXIT: This code should be moved into else part, when reliable 1.2/1.3 chec
1620   {
1621     EXCEPTION_MARK;

1623     CLEAR_PENDING_EXCEPTION;
1624   }
1625   // FIXIT: The is_null check is only so it works better on JDK1.2 VM's. This
1626   // has to be fixed by a runtime query method
1627   if (!destroy_vm || JDK_Version::is_jdk12x_version()) {
1628     // JSR-166: change call from from ThreadGroup.uncaughtException to
1629     // java.lang.Thread.dispatchUncaughtException
1630     if (uncaught_exception.not_null()) {
1631       Handle group(this, java_lang_Thread::threadGroup(threadObj()));
1632       {
1633         EXCEPTION_MARK;
1634         // Check if the method Thread.dispatchUncaughtException() exists. If so
1635         // call it.  Otherwise we have an older library without the JSR-166 chan
1636         // so call ThreadGroup.uncaughtException()
1637         KlassHandle recvrKlass(THREAD, threadObj->klass());
1638         CallInfo callinfo;
1639         KlassHandle thread_klass(THREAD, SystemDictionary::Thread_klass());
1640         LinkResolver::resolve_virtual_call(callinfo, threadObj, recvrKlass, thre
1641                                            vmSymbols::dispatchUncaughtException_
1642                                            vmSymbols::throwable_void_signature()
1643                                            KlassHandle(), false, false, THREAD);
1644         CLEAR_PENDING_EXCEPTION;
1645         methodHandle method = callinfo.selected_method();
1646         if (method.not_null()) {
1647           JavaValue result(T_VOID);
1648           JavaCalls::call_virtual(&result,
1649                                   threadObj, thread_klass,
1650                                   vmSymbols::dispatchUncaughtException_name(),
1651                                   vmSymbols::throwable_void_signature(),
1652                                   uncaught_exception,
```

```
1653                                 THREAD);
1654         } else {
1655           KlassHandle thread_group(THREAD, SystemDictionary::ThreadGroup_klass()
1656           JavaValue result(T_VOID);
1657           JavaCalls::call_virtual(&result,
1658                                   group, thread_group,
1659                                   vmSymbols::uncaughtException_name(),
1660                                   vmSymbols::thread_throwable_void_signature(),
1661                                   threadObj,          // Arg 1
1662                                   uncaught_exception,  // Arg 2
1663                                   THREAD);
1664         }
1665         if (HAS_PENDING_EXCEPTION) {
1666           ResourceMark rm(this);
1667           jio_fprintf(defaultStream::error_stream(),
1668                 "\nException: %s thrown from the UncaughtExceptionHandler"
1669                 " in thread \"%s\"\n",
1670                 Klass::cast(pending_exception()->klass())->external_name(),
1671                 get_thread_name());
1672           CLEAR_PENDING_EXCEPTION;
1673         }
1674       }
1675     }

1677     // Called before the java thread exit since we want to read info
1678     // from java_lang_Thread object
1679     EVENT_BEGIN(TraceEventThreadEnd, event);
1680     EVENT_COMMIT(event,
1681         EVENT_SET(event, javalangthread, java_lang_Thread::thread_id(this->threa

1683     // Call after last event on thread
1684     EVENT_THREAD_EXIT(this);

1686     // Call Thread.exit(). We try 3 times in case we got another Thread.stop dur
1687     // the execution of the method. If that is not enough, then we don't really
1688     // is deprecated anyhow.
1689     { int count = 3;
1690       while (java_lang_Thread::threadGroup(threadObj()) != NULL && (count-- > 0)
1691         EXCEPTION_MARK;
1692         JavaValue result(T_VOID);
1693         KlassHandle thread_klass(THREAD, SystemDictionary::Thread_klass());
1694         JavaCalls::call_virtual(&result,
1695                                 threadObj, thread_klass,
1696                                 vmSymbols::exit_method_name(),
1697                                 vmSymbols::void_method_signature(),
1698                                 THREAD);
1699         CLEAR_PENDING_EXCEPTION;
1700       }
1701     }

1703     // notify JVMTI
1704     if (JvmtiExport::should_post_thread_life()) {
1705       JvmtiExport::post_thread_end(this);
1706     }

1708     // We have notified the agents that we are exiting, before we go on,
1709     // we must check for a pending external suspend request and honor it
1710     // in order to not surprise the thread that made the suspend request.
1711     while (true) {
1712       {
1713         MutexLockerEx ml(SR_lock(), Mutex::_no_safepoint_check_flag);
1714         if (!is_external_suspend()) {
1715           set_terminated(_thread_exiting);
1716           ThreadService::current_thread_exiting(this);
1717           break;
1718         }
```



```
1719         // Implied else:
1720         // Things get a little tricky here. We have a pending external
1721         // suspend request, but we are holding the SR_lock so we
1722         // can't just self-suspend. So we temporarily drop the lock
1723         // and then self-suspend.
1724       }

1726       ThreadBlockInVM tbivm(this);
1727       java_suspend_self();

1729       // We're done with this suspend request, but we have to loop around
1730       // and check again. Eventually we will get SR_lock without a pending
1731       // external suspend request and will be able to mark ourselves as
1732       // exiting.
1733     }
1734     // no more external suspends are allowed at this point
1735   } else {
1736     // before_exit() has already posted JVMTI THREAD_END events
1737   }

1739   // Notify waiters on thread object. This has to be done after exit() is called
1740   // on the thread (if the thread is the last thread in a daemon ThreadGroup the
1741   // group should have the destroyed bit set before waiters are notified).
1742   ensure_join(this);
1743   assert(!this->has_pending_exception(), "ensure_join should have cleared");

1745   // 6282335 JNI DetachCurrentThread spec states that all Java monitors
1746   // held by this thread must be released.  A detach operation must only
1747   // get here if there are no Java frames on the stack.  Therefore, any
1748   // owned monitors at this point MUST be JNI-acquired monitors which are
1749   // pre-inflated and in the monitor cache.
1750   //
1751   // ensure_join() ignores IllegalThreadStateExceptions, and so does this.
1752   if (exit_type == jni_detach && JNIDetachReleasesMonitors) {
1753     assert(!this->has_last_Java_frame(), "detaching with Java frames?");
1754     ObjectSynchronizer::release_monitors_owned_by_thread(this);
1755     assert(!this->has_pending_exception(), "release_monitors should have cleared
1756   }

1758   // These things needs to be done while we are still a Java Thread. Make sure t
1759   // is in a consistent state, in case GC happens
1760   assert(_privileged_stack_top == NULL, "must be NULL when we get here");

1762   if (active_handles() != NULL) {
1763     JNIHandleBlock* block = active_handles();
1764     set_active_handles(NULL);
1765     JNIHandleBlock::release_block(block);
1766   }

1768   if (free_handle_block() != NULL) {
1769     JNIHandleBlock* block = free_handle_block();
1770     set_free_handle_block(NULL);
1771     JNIHandleBlock::release_block(block);
1772   }

1774   // These have to be removed while this is still a valid thread.
1775   remove_stack_guard_pages();

1777   if (UseTLAB) {
1778     tlab().make_parsable(true);  // retire TLAB
1779   }

1781   if (JvmtiEnv::environments_might_exist()) {
1782     JvmtiExport::cleanup_thread(this);
1783   }
```

```
#ifndef SERIALGC
  // We must flush G1-related buffers before removing a thread from
  // the list of active threads.
  if (UseG1GC) {
    flush_barrier_queues();
  }
#endif

  // Remove from list of active threads list, and notify VM thread if we are the
  Threads::remove(this);
}

#ifndef SERIALGC
// Flush G1-related queues.
void JavaThread::flush_barrier_queues() {
  satb_mark_queue().flush();
  dirty_card_queue().flush();
}

void JavaThread::initialize_queues() {
  assert(!SafepointSynchronize::is_at_safepoint(),
         "we should not be at a safepoint");

  ObjPtrQueue& satb_queue = satb_mark_queue();
  SATBMarkQueueSet& satb_queue_set = satb_mark_queue_set();
  // The SATB queue should have been constructed with its active
  // field set to false.
  assert(!satb_queue.is_active(), "SATB queue should not be active");
  assert(satb_queue.is_empty(), "SATB queue should be empty");
  // If we are creating the thread during a marking cycle, we should
  // set the active field of the SATB queue to true.
  if (satb_queue_set.is_active()) {
    satb_queue.set_active(true);
  }

  DirtyCardQueue& dirty_queue = dirty_card_queue();
  // The dirty card queue should have been constructed with its
  // active field set to true.
  assert(dirty_queue.is_active(), "dirty card queue should be active");
}
#endif // !SERIALGC

void JavaThread::cleanup_failed_attach_current_thread() {
  if (get_thread_profiler() != NULL) {
    get_thread_profiler()->disengage();
    ResourceMark rm;
    get_thread_profiler()->print(get_thread_name());
  }

  if (active_handles() != NULL) {
    JNIHandleBlock* block = active_handles();
    set_active_handles(NULL);
    JNIHandleBlock::release_block(block);
  }

  if (free_handle_block() != NULL) {
    JNIHandleBlock* block = free_handle_block();
    set_free_handle_block(NULL);
    JNIHandleBlock::release_block(block);
  }

  // These have to be removed while this is still a valid thread.
  remove_stack_guard_pages();

  if (UseTLAB) {
    tlab().make_parsable(true);  // retire TLAB, if any
```



```
  }

#ifndef SERIALGC
  if (UseG1GC) {
    flush_barrier_queues();
  }
#endif

  Threads::remove(this);
  delete this;
}




JavaThread* JavaThread::active() {
  Thread* thread = ThreadLocalStorage::thread();
  assert(thread != NULL, "just checking");
  if (thread->is_Java_thread()) {
    return (JavaThread*) thread;
  } else {
    assert(thread->is_VM_thread(), "this must be a vm thread");
    VM_Operation* op = ((VMThread*) thread)->vm_operation();
    JavaThread *ret=op == NULL ? NULL : (JavaThread *)op->calling_thread();
    assert(ret->is_Java_thread(), "must be a Java thread");
    return ret;
  }
}

bool JavaThread::is_lock_owned(address adr) const {
  if (Thread::is_lock_owned(adr)) return true;

  for (MonitorChunk* chunk = monitor_chunks(); chunk != NULL; chunk = chunk->nex
    if (chunk->contains(adr)) return true;
  }

  return false;
}


void JavaThread::add_monitor_chunk(MonitorChunk* chunk) {
  chunk->set_next(monitor_chunks());
  set_monitor_chunks(chunk);
}

void JavaThread::remove_monitor_chunk(MonitorChunk* chunk) {
  guarantee(monitor_chunks() != NULL, "must be non empty");
  if (monitor_chunks() == chunk) {
    set_monitor_chunks(chunk->next());
  } else {
    MonitorChunk* prev = monitor_chunks();
    while (prev->next() != chunk) prev = prev->next();
    prev->set_next(chunk->next());
  }
}

// JVM support.

// Note: this function shouldn't block if it's called in
// _thread_in_native_trans state (such as from
// check_special_condition_for_native_trans()).
void JavaThread::check_and_handle_async_exceptions(bool check_unsafe_error) {

  if (has_last_Java_frame() && has_async_condition()) {
    // If we are at a polling page safepoint (not a poll return)
    // then we must defer async exception because live registers
```

```
1917     // will be clobbered by the exception path. Poll return is
1918     // ok because the call we a returning from already collides
1919     // with exception handling registers and so there is no issue.
1920     // (The exception handling path kills call result registers but
1921     //  this is ok since the exception kills the result anyway).

1923     if (is_at_poll_safepoint()) {
1924       // if the code we are returning to has deoptimized we must defer
1925       // the exception otherwise live registers get clobbered on the
1926       // exception path before deoptimization is able to retrieve them.
1927       //
1928       RegisterMap map(this, false);
1929       frame caller_fr = last_frame().sender(&map);
1930       assert(caller_fr.is_compiled_frame(), "what?");
1931       if (caller_fr.is_deoptimized_frame()) {
1932         if (TraceExceptions) {
1933           ResourceMark rm;
1934           tty->print_cr("deferred async exception at compiled safepoint");
1935         }
1936         return;
1937       }
1938     }
1939   }

1941   JavaThread::AsyncRequests condition = clear_special_runtime_exit_condition();
1942   if (condition == _no_async_condition) {
1943     // Conditions have changed since has_special_runtime_exit_condition()
1944     // was called:
1945     // - if we were here only because of an external suspend request,
1946     //   then that was taken care of above (or cancelled) so we are done
1947     // - if we were here because of another async request, then it has
1948     //   been cleared between the has_special_runtime_exit_condition()
1949     //   and now so again we are done
1950     return;
1951   }

1953   // Check for pending async. exception
1954   if (_pending_async_exception != NULL) {
1955     // Only overwrite an already pending exception, if it is not a threadDeath.
1956     if (!has_pending_exception() || !pending_exception()->is_a(SystemDictionary:

1958       // We cannot call Exceptions::_throw(...) here because we cannot block
1959       set_pending_exception(_pending_async_exception, __FILE__, __LINE__);

1961       if (TraceExceptions) {
1962         ResourceMark rm;
1963         tty->print("Async. exception installed at runtime exit (" INTPTR_FORMAT
1964         if (has_last_Java_frame() ) {
1965           frame f = last_frame();
1966           tty->print(" (pc: " INTPTR_FORMAT " sp: " INTPTR_FORMAT " )", f.pc(),
1967         }
1968         tty->print_cr(" of type: %s", instanceKlass::cast(_pending_async_excepti
1969       }
1970       _pending_async_exception = NULL;
1971       clear_has_async_exception();
1972     }
1973   }

1975   if (check_unsafe_error &&
1976       condition == _async_unsafe_access_error && !has_pending_exception()) {
1977     condition = _no_async_condition;  // done
1978     switch (thread_state()) {
1979     case _thread_in_vm:
1980       {
1981         JavaThread* THREAD = this;
1982         THROW_MSG(vmSymbols::java_lang_InternalError(), "a fault occurred in an
```



```
1983       }
1984     case _thread_in_native:
1985       {
1986         ThreadInVMfromNative tiv(this);
1987         JavaThread* THREAD = this;
1988         THROW_MSG(vmSymbols::java_lang_InternalError(), "a fault occurred in an
1989       }
1990     case _thread_in_Java:
1991       {
1992         ThreadInVMfromJava tiv(this);
1993         JavaThread* THREAD = this;
1994         THROW_MSG(vmSymbols::java_lang_InternalError(), "a fault occurred in a r
1995       }
1996     default:
1997       ShouldNotReachHere();
1998     }
1999   }

2001   assert(condition == _no_async_condition || has_pending_exception() ||
2002          (!check_unsafe_error && condition == _async_unsafe_access_error),
2003          "must have handled the async condition, if no exception");
2004 }

2006 void JavaThread::handle_special_runtime_exit_condition(bool check_asyncs) {
2007   //
2008   // Check for pending external suspend. Internal suspend requests do
2009   // not use handle_special_runtime_exit_condition().
2010   // If JNIEnv proxies are allowed, don't self-suspend if the target
2011   // thread is not the current thread. In older versions of jdbx, jdbx
2012   // threads could call into the VM with another thread's JNIEnv so we
2013   // can be here operating on behalf of a suspended thread (4432884).
2014   bool do_self_suspend = is_external_suspend_with_lock();
2015   if (do_self_suspend && (!AllowJNIEnvProxy || this == JavaThread::current())) {
2016     //
2017     // Because thread is external suspended the safepoint code will count
2018     // thread as at a safepoint. This can be odd because we can be here
2019     // as _thread_in_Java which would normally transition to _thread_blocked
2020     // at a safepoint. We would like to mark the thread as _thread_blocked
2021     // before calling java_suspend_self like all other callers of it but
2022     // we must then observe proper safepoint protocol. (We can't leave
2023     // _thread_blocked with a safepoint in progress). However we can be
2024     // here as _thread_in_native_trans so we can't use a normal transition
2025     // constructor/destructor pair because they assert on that type of
2026     // transition. We could do something like:
2027     //
2028     // JavaThreadState state = thread_state();
2029     // set_thread_state(_thread_in_vm);
2030     // {
2031     //   ThreadBlockInVM tbivm(this);
2032     //   java_suspend_self()
2033     // }
2034     // set_thread_state(_thread_in_vm_trans);
2035     // if (safepoint) block;
2036     // set_thread_state(state);
2037     //
2038     // but that is pretty messy. Instead we just go with the way the
2039     // code has worked before and note that this is the only path to
2040     // java_suspend_self that doesn't put the thread in _thread_blocked
2041     // mode.

2043     frame_anchor()->make_walkable(this);
2044     java_suspend_self();

2046     // We might be here for reasons in addition to the self-suspend request
2047     // so check for other async requests.
2048   }
```

```
2050   if (check_asyncs) {
2051     check_and_handle_async_exceptions();
2052   }
2053 }

2055 void JavaThread::send_thread_stop(oop java_throwable)  {
2056   assert(Thread::current()->is_VM_thread(), "should be in the vm thread");
2057   assert(Threads_lock->is_locked(), "Threads_lock should be locked by safepoint
2058   assert(SafepointSynchronize::is_at_safepoint(), "all threads are stopped");

2060   // Do not throw asynchronous exceptions against the compiler thread
2061   // (the compiler thread should not be a Java thread -- fix in 1.4.2)
2062   if (is_Compiler_thread()) return;

2064   {
2065     // Actually throw the Throwable against the target Thread - however
2066     // only if there is no thread death exception installed already.
2067     if (_pending_async_exception == NULL || !_pending_async_exception->is_a(Syst
2068       // If the topmost frame is a runtime stub, then we are calling into
2069       // OptoRuntime from compiled code. Some runtime stubs (new, monitor_exit..
2070       // must deoptimize the caller before continuing, as the compiled  exceptio
2071       // may not be valid
2072       if (has_last_Java_frame()) {
2073         frame f = last_frame();
2074         if (f.is_runtime_frame() || f.is_safepoint_blob_frame()) {
2075           // BiasedLocking needs an updated RegisterMap for the revoke monitors
2076           RegisterMap reg_map(this, UseBiasedLocking);
2077           frame compiled_frame = f.sender(&reg_map);
2078           if (compiled_frame.can_be_deoptimized()) {
2079             Deoptimization::deoptimize(this, compiled_frame, &reg_map);
2080           }
2081         }
2082       }

2084       // Set async. pending exception in thread.
2085       set_pending_async_exception(java_throwable);

2087       if (TraceExceptions) {
2088        ResourceMark rm;
2089        tty->print_cr("Pending Async. exception installed of type: %s", instanceK
2090       }
2091       // for AbortVMOnException flag
2092       NOT_PRODUCT(Exceptions::debug_check_abort(instanceKlass::cast(_pending_asy
2093     }
2094   }


2097   // Interrupt thread so it will wake up from a potential wait()
2098   Thread::interrupt(this);
2099 }

2101 // External suspension mechanism.
2102 //
2103 // Tell the VM to suspend a thread when ever it knows that it does not hold on
2104 // to any VM_locks and it is at a transition
2105 // Self-suspension will happen on the transition out of the vm.
2106 // Catch "this" coming in from JNIEnv pointers when the thread has been freed
2107 //
2108 // Guarantees on return:
2109 //   + Target thread will not execute any new bytecode (that's why we need to
2110 //     force a safepoint)
2111 //   + Target thread will not enter any new monitors
2112 //
2113 void JavaThread::java_suspend() {
2114   { MutexLocker mu(Threads_lock);
```



```
2115     if (!Threads::includes(this) || is_exiting() || this->threadObj() == NULL) {
2116       return;
2117     }
2118   }

2120   { MutexLockerEx ml(SR_lock(), Mutex::_no_safepoint_check_flag);
2121     if (!is_external_suspend()) {
2122       // a racing resume has cancelled us; bail out now
2123       return;
2124     }

2126     // suspend is done
2127     uint32_t debug_bits = 0;
2128     // Warning: is_ext_suspend_completed() may temporarily drop the
2129     // SR_lock to allow the thread to reach a stable thread state if
2130     // it is currently in a transient thread state.
2131     if (is_ext_suspend_completed(false /* !called_by_wait */,
2132                                  SuspendRetryDelay, &debug_bits) ) {
2133       return;
2134     }
2135   }

2137   VM_ForceSafepoint vm_suspend;
2138   VMThread::execute(&vm_suspend);
2139 }

2141 // Part II of external suspension.
2142 // A JavaThread self suspends when it detects a pending external suspend
2143 // request. This is usually on transitions. It is also done in places
2144 // where continuing to the next transition would surprise the caller,
2145 // e.g., monitor entry.
2146 //
2147 // Returns the number of times that the thread self-suspended.
2148 //
2149 // Note: DO NOT call java_suspend_self() when you just want to block current
2150 //       thread. java_suspend_self() is the second stage of cooperative
2151 //       suspension for external suspend requests and should only be used
2152 //       to complete an external suspend request.
2153 //
2154 int JavaThread::java_suspend_self() {
2155   int ret = 0;

2157   // we are in the process of exiting so don't suspend
2158   if (is_exiting()) {
2159     clear_external_suspend();
2160     return ret;
2161   }

2163   assert(_anchor.walkable() ||
2164     (is_Java_thread() && !((JavaThread*)this)->has_last_Java_frame()),
2165     "must have walkable stack");

2167   MutexLockerEx ml(SR_lock(), Mutex::_no_safepoint_check_flag);

2169   assert(!this->is_ext_suspended(),
2170     "a thread trying to self-suspend should not already be suspended");

2172   if (this->is_suspend_equivalent()) {
2173     // If we are self-suspending as a result of the lifting of a
2174     // suspend equivalent condition, then the suspend_equivalent
2175     // flag is not cleared until we set the ext_suspended flag so
2176     // that wait_for_ext_suspend_completion() returns consistent
2177     // results.
2178     this->clear_suspend_equivalent();
2179   }
```

```
2181   // A racing resume may have cancelled us before we grabbed SR_lock
2182   // above. Or another external suspend request could be waiting for us
2183   // by the time we return from SR_lock()->wait(). The thread
2184   // that requested the suspension may already be trying to walk our
2185   // stack and if we return now, we can change the stack out from under
2186   // it. This would be a "bad thing (TM)" and cause the stack walker
2187   // to crash. We stay self-suspended until there are no more pending
2188   // external suspend requests.
2189   while (is_external_suspend()) {
2190     ret++;
2191     this->set_ext_suspended();

2193     // _ext_suspended flag is cleared by java_resume()
2194     while (is_ext_suspended()) {
2195       this->SR_lock()->wait(Mutex::_no_safepoint_check_flag);
2196     }
2197   }

2199   return ret;
2200 }

2202 #ifdef ASSERT
2203 // verify the JavaThread has not yet been published in the Threads::list, and
2204 // hence doesn't need protection from concurrent access at this stage
2205 void JavaThread::verify_not_published() {
2206   if (!Threads_lock->owned_by_self()) {
2207    MutexLockerEx ml(Threads_lock,  Mutex::_no_safepoint_check_flag);
2208    assert( !Threads::includes(this),
2209            "java thread shouldn't have been published yet!");
2210   }
2211   else {
2212    assert( !Threads::includes(this),
2213            "java thread shouldn't have been published yet!");
2214   }
2215 }
2216 #endif

2218 // Slow path when the native==>VM/Java barriers detect a safepoint is in
2219 // progress or when _suspend_flags is non-zero.
2220 // Current thread needs to self-suspend if there is a suspend request and/or
2221 // block if a safepoint is in progress.
2222 // Async exception ISN'T checked.
2223 // Note only the ThreadInVMfromNative transition can call this function
2224 // directly and when thread state is _thread_in_native_trans
2225 void JavaThread::check_safepoint_and_suspend_for_native_trans(JavaThread *thread
2226   assert(thread->thread_state() == _thread_in_native_trans, "wrong state");

2228   JavaThread *curJT = JavaThread::current();
2229   bool do_self_suspend = thread->is_external_suspend();

2231   assert(!curJT->has_last_Java_frame() || curJT->frame_anchor()->walkable(), "Un

2233   // If JNIEnv proxies are allowed, don't self-suspend if the target
2234   // thread is not the current thread. In older versions of jdbx, jdbx
2235   // threads could call into the VM with another thread's JNIEnv so we
2236   // can be here operating on behalf of a suspended thread (4432884).
2237   if (do_self_suspend && (!AllowJNIEnvProxy || curJT == thread)) {
2238     JavaThreadState state = thread->thread_state();

2240     // We mark this thread_blocked state as a suspend-equivalent so
2241     // that a caller to is_ext_suspend_completed() won't be confused.
2242     // The suspend-equivalent state is cleared by java_suspend_self().
2243     thread->set_suspend_equivalent();

2245     // If the safepoint code sees the _thread_in_native_trans state, it will
2246     // wait until the thread changes to other thread state. There is no
```



```
2247     // guarantee on how soon we can obtain the SR_lock and complete the
2248     // self-suspend request. It would be a bad idea to let safepoint wait for
2249     // too long. Temporarily change the state to _thread_blocked to
2250     // let the VM thread know that this thread is ready for GC. The problem
2251     // of changing thread state is that safepoint could happen just after
2252     // java_suspend_self() returns after being resumed, and VM thread will
2253     // see the _thread_blocked state. We must check for safepoint
2254     // after restoring the state and make sure we won't leave while a safepoint
2255     // is in progress.
2256     thread->set_thread_state(_thread_blocked);
2257     thread->java_suspend_self();
2258     thread->set_thread_state(state);
2259     // Make sure new state is seen by VM thread
2260     if (os::is_MP()) {
2261       if (UseMembar) {
2262         // Force a fence between the write above and read below
2263         OrderAccess::fence();
2264       } else {
2265         // Must use this rather than serialization page in particular on Windows
2266         InterfaceSupport::serialize_memory(thread);
2267       }
2268     }
2269   }

2271   if (SafepointSynchronize::do_call_back()) {
2272     // If we are safepointing, then block the caller which may not be
2273     // the same as the target thread (see above).
2274     SafepointSynchronize::block(curJT);
2275   }

2277   if (thread->is_deopt_suspend()) {
2278     thread->clear_deopt_suspend();
2279     RegisterMap map(thread, false);
2280     frame f = thread->last_frame();
2281     while ( f.id() != thread->must_deopt_id() && ! f.is_first_frame()) {
2282       f = f.sender(&map);
2283     }
2284     if (f.id() == thread->must_deopt_id()) {
2285       thread->clear_must_deopt_id();
2286       f.deoptimize(thread);
2287     } else {
2288       fatal("missed deoptimization!");
2289     }
2290   }
2291 }

2293 // Slow path when the native==>VM/Java barriers detect a safepoint is in
2294 // progress or when _suspend_flags is non-zero.
2295 // Current thread needs to self-suspend if there is a suspend request and/or
2296 // block if a safepoint is in progress.
2297 // Also check for pending async exception (not including unsafe access error).
2298 // Note only the native==>VM/Java barriers can call this function and when
2299 // thread state is _thread_in_native_trans.
2300 void JavaThread::check_special_condition_for_native_trans(JavaThread *thread) {
2301   check_safepoint_and_suspend_for_native_trans(thread);

2303   if (thread->has_async_exception()) {
2304     // We are in _thread_in_native_trans state, don't handle unsafe
2305     // access error since that may block.
2306     thread->check_and_handle_async_exceptions(false);
2307   }
2308 }

2310 // This is a variant of the normal
2311 // check_special_condition_for_native_trans with slightly different
2312 // semantics for use by critical native wrappers.  It does all the
```

```
2313 // normal checks but also performs the transition back into
2314 // thread_in_Java state.  This is required so that critical natives
2315 // can potentially block and perform a GC if they are the last thread
2316 // exiting the GC_locker.
2317 void JavaThread::check_special_condition_for_native_trans_and_transition(JavaThr
2318   check_special_condition_for_native_trans(thread);

2320   // Finish the transition
2321   thread->set_thread_state(_thread_in_Java);

2323   if (thread->do_critical_native_unlock()) {
2324     ThreadInVMfromJavaNoAsyncException tiv(thread);
2325     GC_locker::unlock_critical(thread);
2326     thread->clear_critical_native_unlock();
2327   }
2328 }

2330 // We need to guarantee the Threads_lock here, since resumes are not
2331 // allowed during safepoint synchronization
2332 // Can only resume from an external suspension
2333 void JavaThread::java_resume() {
2334   assert_locked_or_safepoint(Threads_lock);

2336   // Sanity check: thread is gone, has started exiting or the thread
2337   // was not externally suspended.
2338   if (!Threads::includes(this) || is_exiting() || !is_external_suspend()) {
2339     return;
2340   }

2342   MutexLockerEx ml(SR_lock(), Mutex::_no_safepoint_check_flag);

2344   clear_external_suspend();

2346   if (is_ext_suspended()) {
2347     clear_ext_suspended();
2348     SR_lock()->notify_all();
2349   }
2350 }

2352 void JavaThread::create_stack_guard_pages() {
2353   if (! os::uses_stack_guard_pages() || _stack_guard_state != stack_guard_unused
2354   address low_addr = stack_base() - stack_size();
2355   size_t len = (StackYellowPages + StackRedPages) * os::vm_page_size();

2357   int allocate = os::allocate_stack_guard_pages();
2358   // warning("Guarding at " PTR_FORMAT " for len " SIZE_FORMAT "\n", low_addr, l

2360   if (allocate && !os::create_stack_guard_pages((char *) low_addr, len)) {
2361     warning("Attempt to allocate stack guard pages failed.");
2362     return;
2363   }

2365   if (os::guard_memory((char *) low_addr, len)) {
2366     _stack_guard_state = stack_guard_enabled;
2367   } else {
2368     warning("Attempt to protect stack guard pages failed.");
2369     if (os::uncommit_memory((char *) low_addr, len)) {
2370       warning("Attempt to deallocate stack guard pages failed.");
2371     }
2372   }
2373 }

2375 void JavaThread::remove_stack_guard_pages() {
2376   if (_stack_guard_state == stack_guard_unused) return;
2377   address low_addr = stack_base() - stack_size();
2378   size_t len = (StackYellowPages + StackRedPages) * os::vm_page_size();
```



```
2380   if (os::allocate_stack_guard_pages()) {
2381     if (os::remove_stack_guard_pages((char *) low_addr, len)) {
2382       _stack_guard_state = stack_guard_unused;
2383     } else {
2384       warning("Attempt to deallocate stack guard pages failed.");
2385     }
2386   } else {
2387     if (_stack_guard_state == stack_guard_unused) return;
2388     if (os::unguard_memory((char *) low_addr, len)) {
2389       _stack_guard_state = stack_guard_unused;
2390     } else {
2391         warning("Attempt to unprotect stack guard pages failed.");
2392     }
2393   }
2394 }

2396 void JavaThread::enable_stack_yellow_zone() {
2397   assert(_stack_guard_state != stack_guard_unused, "must be using guard pages.")
2398   assert(_stack_guard_state != stack_guard_enabled, "already enabled");

2400   // The base notation is from the stacks point of view, growing downward.
2401   // We need to adjust it to work correctly with guard_memory()
2402   address base = stack_yellow_zone_base() - stack_yellow_zone_size();

2404   guarantee(base < stack_base(),"Error calculating stack yellow zone");
2405   guarantee(base < os::current_stack_pointer(),"Error calculating stack yellow z

2407   if (os::guard_memory((char *) base, stack_yellow_zone_size())) {
2408     _stack_guard_state = stack_guard_enabled;
2409   } else {
2410     warning("Attempt to guard stack yellow zone failed.");
2411   }
2412   enable_register_stack_guard();
2413 }

2415 void JavaThread::disable_stack_yellow_zone() {
2416   assert(_stack_guard_state != stack_guard_unused, "must be using guard pages.")
2417   assert(_stack_guard_state != stack_guard_yellow_disabled, "already disabled");

2419   // Simply return if called for a thread that does not use guard pages.
2420   if (_stack_guard_state == stack_guard_unused) return;

2422   // The base notation is from the stacks point of view, growing downward.
2423   // We need to adjust it to work correctly with guard_memory()
2424   address base = stack_yellow_zone_base() - stack_yellow_zone_size();

2426   if (os::unguard_memory((char *)base, stack_yellow_zone_size())) {
2427     _stack_guard_state = stack_guard_yellow_disabled;
2428   } else {
2429     warning("Attempt to unguard stack yellow zone failed.");
2430   }
2431   disable_register_stack_guard();
2432 }

2434 void JavaThread::enable_stack_red_zone() {
2435   // The base notation is from the stacks point of view, growing downward.
2436   // We need to adjust it to work correctly with guard_memory()
2437   assert(_stack_guard_state != stack_guard_unused, "must be using guard pages.")
2438   address base = stack_red_zone_base() - stack_red_zone_size();

2440   guarantee(base < stack_base(),"Error calculating stack red zone");
2441   guarantee(base < os::current_stack_pointer(),"Error calculating stack red zone

2443   if(!os::guard_memory((char *) base, stack_red_zone_size())) {
2444     warning("Attempt to guard stack red zone failed.");
```

```
2445   }
2446 }

2448 void JavaThread::disable_stack_red_zone() {
2449   // The base notation is from the stacks point of view, growing downward.
2450   // We need to adjust it to work correctly with guard_memory()
2451   assert(_stack_guard_state != stack_guard_unused, "must be using guard pages.")
2452   address base = stack_red_zone_base() - stack_red_zone_size();
2453   if (!os::unguard_memory((char *)base, stack_red_zone_size())) {
2454     warning("Attempt to unguard stack red zone failed.");
2455   }
2456 }

2458 void JavaThread::frames_do(void f(frame*, const RegisterMap* map)) {
2459   // ignore is there is no stack
2460   if (!has_last_Java_frame()) return;
2461   // traverse the stack frames. Starts from top frame.
2462   for(StackFrameStream fst(this); !fst.is_done(); fst.next()) {
2463     frame* fr = fst.current();
2464     f(fr, fst.register_map());
2465   }
2466 }


2469 #ifndef PRODUCT
2470 // Deoptimization
2471 // Function for testing deoptimization
2472 void JavaThread::deoptimize() {
2473   // BiasedLocking needs an updated RegisterMap for the revoke monitors pass
2474   StackFrameStream fst(this, UseBiasedLocking);
2475   bool deopt = false;           // Dump stack only if a deopt actually happens.
2476   bool only_at = strlen(DeoptimizeOnlyAt) > 0;
2477   // Iterate over all frames in the thread and deoptimize
2478   for(; !fst.is_done(); fst.next()) {
2479     if(fst.current()->can_be_deoptimized()) {

2481       if (only_at) {
2482         // Deoptimize only at particular bcis.  DeoptimizeOnlyAt
2483         // consists of comma or carriage return separated numbers so
2484         // search for the current bci in that string.
2485         address pc = fst.current()->pc();
2486         nmethod* nm =  (nmethod*) fst.current()->cb();
2487         ScopeDesc* sd = nm->scope_desc_at( pc);
2488         char buffer[8];
2489         jio_snprintf(buffer, sizeof(buffer), "%d", sd->bci());
2490         size_t len = strlen(buffer);
2491         const char * found = strstr(DeoptimizeOnlyAt, buffer);
2492         while (found != NULL) {
2493           if ((found[len] == ',' || found[len] == '\n' || found[len] == '\0') &&
2494               (found == DeoptimizeOnlyAt || found[-1] == ',' || found[-1] == '\n
2495             // Check that the bci found is bracketed by terminators.
2496             break;
2497           }
2498           found = strstr(found + 1, buffer);
2499         }
2500         if (!found) {
2501           continue;
2502         }
2503       }

2505       if (DebugDeoptimization && !deopt) {
2506         deopt = true; // One-time only print before deopt
2507         tty->print_cr("[BEFORE Deoptimization]");
2508         trace_frames();
2509         trace_stack();
2510       }
```



```
2511       Deoptimization::deoptimize(this, *fst.current(), fst.register_map());
2512     }
2513   }

2515   if (DebugDeoptimization && deopt) {
2516     tty->print_cr("[AFTER Deoptimization]");
2517     trace_frames();
2518   }
2519 }


2522 // Make zombies
2523 void JavaThread::make_zombies() {
2524   for(StackFrameStream fst(this); !fst.is_done(); fst.next()) {
2525     if (fst.current()->can_be_deoptimized()) {
2526       // it is a Java nmethod
2527       nmethod* nm = CodeCache::find_nmethod(fst.current()->pc());
2528       nm->make_not_entrant();
2529     }
2530   }
2531 }
2532 #endif // PRODUCT


2535 void JavaThread::deoptimized_wrt_marked_nmethods() {
2536   if (!has_last_Java_frame()) return;
2537   // BiasedLocking needs an updated RegisterMap for the revoke monitors pass
2538   StackFrameStream fst(this, UseBiasedLocking);
2539   for(; !fst.is_done(); fst.next()) {
2540     if (fst.current()->should_be_deoptimized()) {
2541       Deoptimization::deoptimize(this, *fst.current(), fst.register_map());
2542     }
2543   }
2544 }


2547 // GC support
2548 static void frame_gc_epilogue(frame* f, const RegisterMap* map) { f->gc_epilogue

2550 void JavaThread::gc_epilogue() {
2551   frames_do(frame_gc_epilogue);
2552 }


2555 static void frame_gc_prologue(frame* f, const RegisterMap* map) { f->gc_prologue

2557 void JavaThread::gc_prologue() {
2558   frames_do(frame_gc_prologue);
2559 }

2561 // If the caller is a NamedThread, then remember, in the current scope,
2562 // the given JavaThread in its _processed_thread field.
2563 class RememberProcessedThread: public StackObj {
2564   NamedThread* _cur_thr;
2565 public:
2566   RememberProcessedThread(JavaThread* jthr) {
2567     Thread* thread = Thread::current();
2568     if (thread->is_Named_thread()) {
2569       _cur_thr = (NamedThread *)thread;
2570       _cur_thr->set_processed_thread(jthr);
2571     } else {
2572       _cur_thr = NULL;
2573     }
2574   }

2576   ~RememberProcessedThread() {
```

```
2577     if (_cur_thr) {
2578       _cur_thr->set_processed_thread(NULL);
2579     }
2580   }
2581 };

2583 void JavaThread::oops_do(OopClosure* f, CodeBlobClosure* cf) {
2584   // Verify that the deferred card marks have been flushed.
2585   assert(deferred_card_mark().is_empty(), "Should be empty during GC");

2587   // The ThreadProfiler oops_do is done from FlatProfiler::oops_do
2588   // since there may be more than one thread using each ThreadProfiler.

2590   // Traverse the GCHandles
2591   Thread::oops_do(f, cf);

2593   assert( (!has_last_Java_frame() && java_call_counter() == 0) ||
2594           (has_last_Java_frame() && java_call_counter() > 0), "wrong java_sp inf

2596   if (has_last_Java_frame()) {
2597     // Record JavaThread to GC thread
2598     RememberProcessedThread rpt(this);

2600     // Traverse the privileged stack
2601     if (_privileged_stack_top != NULL) {
2602       _privileged_stack_top->oops_do(f);
2603     }

2605     // traverse the registered growable array
2606     if (_array_for_gc != NULL) {
2607       for (int index = 0; index < _array_for_gc->length(); index++) {
2608         f->do_oop(_array_for_gc->adr_at(index));
2609       }
2610     }

2612     // Traverse the monitor chunks
2613     for (MonitorChunk* chunk = monitor_chunks(); chunk != NULL; chunk = chunk->n
2614       chunk->oops_do(f);
2615     }

2617     // Traverse the execution stack
2618     for(StackFrameStream fst(this); !fst.is_done(); fst.next()) {
2619       fst.current()->oops_do(f, cf, fst.register_map());
2620     }
2621   }

2623   // callee_target is never live across a gc point so NULL it here should
2624   // it still contain a methdOop.

2626   set_callee_target(NULL);

2628   assert(vframe_array_head() == NULL, "deopt in progress at a safepoint!");
2629   // If we have deferred set_locals there might be oops waiting to be
2630   // written
2631   GrowableArray<jvmtiDeferredLocalVariableSet*>* list = deferred_locals();
2632   if (list != NULL) {
2633     for (int i = 0; i < list->length(); i++) {
2634       list->at(i)->oops_do(f);
2635     }
2636   }

2638   // Traverse instance variables at the end since the GC may be moving things
2639   // around using this function
2640   f->do_oop((oop*) &_threadObj);
2641   f->do_oop((oop*) &_vm_result);
2642   f->do_oop((oop*) &_vm_result_2);
```



```
2643   f->do_oop((oop*) &_exception_oop);
2644   f->do_oop((oop*) &_pending_async_exception);

2646   if (jvmti_thread_state() != NULL) {
2647     jvmti_thread_state()->oops_do(f);
2648   }
2649 }

2651 void JavaThread::nmethods_do(CodeBlobClosure* cf) {
2652   Thread::nmethods_do(cf);  // (super method is a no-op)

2654   assert( (!has_last_Java_frame() && java_call_counter() == 0) ||
2655           (has_last_Java_frame() && java_call_counter() > 0), "wrong java_sp inf

2657   if (has_last_Java_frame()) {
2658     // Traverse the execution stack
2659     for(StackFrameStream fst(this); !fst.is_done(); fst.next()) {
2660       fst.current()->nmethods_do(cf);
2661     }
2662   }
2663 }

2665 // Printing
2666 const char* _get_thread_state_name(JavaThreadState _thread_state) {
2667   switch (_thread_state) {
2668   case _thread_uninitialized:     return "_thread_uninitialized";
2669   case _thread_new:               return "_thread_new";
2670   case _thread_new_trans:         return "_thread_new_trans";
2671   case _thread_in_native:         return "_thread_in_native";
2672   case _thread_in_native_trans:   return "_thread_in_native_trans";
2673   case _thread_in_vm:             return "_thread_in_vm";
2674   case _thread_in_vm_trans:       return "_thread_in_vm_trans";
2675   case _thread_in_Java:           return "_thread_in_Java";
2676   case _thread_in_Java_trans:     return "_thread_in_Java_trans";
2677   case _thread_blocked:           return "_thread_blocked";
2678   case _thread_blocked_trans:     return "_thread_blocked_trans";
2679   default:                        return "unknown thread state";
2680   }
2681 }

2683 #ifndef PRODUCT
2684 void JavaThread::print_thread_state_on(outputStream *st) const {
2685   st->print_cr("   JavaThread state: %s", _get_thread_state_name(_thread_state))
2686 };
2687 void JavaThread::print_thread_state() const {
2688   print_thread_state_on(tty);
2689 };
2690 #endif // PRODUCT

2692 // Called by Threads::print() for VM_PrintThreads operation
2693 void JavaThread::print_on(outputStream *st) const {
2694   st->print("\"%s\" ", get_thread_name());
2695   oop thread_oop = threadObj();
2696   if (thread_oop != NULL && java_lang_Thread::is_daemon(thread_oop))  st->print(
2697   Thread::print_on(st);
2698   // print guess for valid stack memory region (assume 4K pages); helps lock deb
2699   st->print_cr("[" INTPTR_FORMAT "]", (intptr_t)last_Java_sp() & ~right_n_bits(1
2700   if (thread_oop != NULL && JDK_Version::is_gte_jdk15x_version()) {
2701     st->print_cr("   java.lang.Thread.State: %s", java_lang_Thread::thread_statu
2702   }
2703 #ifndef PRODUCT
2704   print_thread_state_on(st);
2705   _safepoint_state->print_on(st);
2706 #endif // PRODUCT
2707 }
```

```
2709 // Called by fatal error handler. The difference between this and
2710 // JavaThread::print() is that we can't grab lock or allocate memory.
2711 void JavaThread::print_on_error(outputStream* st, char *buf, int buflen) const {
2712   st->print("JavaThread \"%s\"",  get_thread_name_string(buf, buflen));
2713   oop thread_obj = threadObj();
2714   if (thread_obj != NULL) {
2715     if (java_lang_Thread::is_daemon(thread_obj)) st->print(" daemon");
2716   }
2717   st->print(" [");
2718   st->print("%s", _get_thread_state_name(_thread_state));
2719   if (osthread()) {
2720     st->print(", id=%d", osthread()->thread_id());
2721   }
2722   st->print(", stack(" PTR_FORMAT "," PTR_FORMAT ")",
2723             _stack_base - _stack_size, _stack_base);
2724   st->print("]");
2725   return;
2726 }

2728 // Verification

2730 static void frame_verify(frame* f, const RegisterMap *map) { f->verify(map); }

2732 void JavaThread::verify() {
2733   // Verify oops in the thread.
2734   oops_do(&VerifyOopClosure::verify_oop, NULL);

2736   // Verify the stack frames.
2737   frames_do(frame_verify);
2738 }

2740 // CR 6300358 (sub-CR 2137150)
2741 // Most callers of this method assume that it can't return NULL but a
2742 // thread may not have a name whilst it is in the process of attaching to
2743 // the VM - see CR 6412693, and there are places where a JavaThread can be
2744 // seen prior to having it's threadObj set (eg JNI attaching threads and
2745 // if vm exit occurs during initialization). These cases can all be accounted
2746 // for such that this method never returns NULL.
2747 const char* JavaThread::get_thread_name() const {
2748 #ifdef ASSERT
2749   // early safepoints can hit while current thread does not yet have TLS
2750   if (!SafepointSynchronize::is_at_safepoint()) {
2751     Thread *cur = Thread::current();
2752     if (!(cur->is_Java_thread() && cur == this)) {
2753       // Current JavaThreads are allowed to get their own name without
2754       // the Threads_lock.
2755       assert_locked_or_safepoint(Threads_lock);
2756     }
2757   }
2758 #endif // ASSERT
2759     return get_thread_name_string();
2760 }

2762 // Returns a non-NULL representation of this thread's name, or a suitable
2763 // descriptive string if there is no set name
2764 const char* JavaThread::get_thread_name_string(char* buf, int buflen) const {
2765   const char* name_str;
2766   oop thread_obj = threadObj();
2767   if (thread_obj != NULL) {
2768     typeArrayOop name = java_lang_Thread::name(thread_obj);
2769     if (name != NULL) {
2770       if (buf == NULL) {
2771         name_str = UNICODE::as_utf8((jchar*) name->base(T_CHAR), name->length())
2772       }
2773       else {
2774         name_str = UNICODE::as_utf8((jchar*) name->base(T_CHAR), name->length(),
```



```
2775       }
2776     }
2777     else if (is_attaching_via_jni()) { // workaround for 6412693 - see 6404306
2778       name_str = "<no-name - thread is attaching>";
2779     }
2780     else {
2781       name_str = Thread::name();
2782     }
2783   }
2784   else {
2785     name_str = Thread::name();
2786   }
2787   assert(name_str != NULL, "unexpected NULL thread name");
2788   return name_str;
2789 }


2792 const char* JavaThread::get_threadgroup_name() const {
2793   debug_only(if (JavaThread::current() != this) assert_locked_or_safepoint(Threa
2794   oop thread_obj = threadObj();
2795   if (thread_obj != NULL) {
2796     oop thread_group = java_lang_Thread::threadGroup(thread_obj);
2797     if (thread_group != NULL) {
2798       typeArrayOop name = java_lang_ThreadGroup::name(thread_group);
2799       // ThreadGroup.name can be null
2800       if (name != NULL) {
2801         const char* str = UNICODE::as_utf8((jchar*) name->base(T_CHAR), name->le
2802         return str;
2803       }
2804     }
2805   }
2806   return NULL;
2807 }

2809 const char* JavaThread::get_parent_name() const {
2810   debug_only(if (JavaThread::current() != this) assert_locked_or_safepoint(Threa
2811   oop thread_obj = threadObj();
2812   if (thread_obj != NULL) {
2813     oop thread_group = java_lang_Thread::threadGroup(thread_obj);
2814     if (thread_group != NULL) {
2815       oop parent = java_lang_ThreadGroup::parent(thread_group);
2816       if (parent != NULL) {
2817         typeArrayOop name = java_lang_ThreadGroup::name(parent);
2818         // ThreadGroup.name can be null
2819         if (name != NULL) {
2820           const char* str = UNICODE::as_utf8((jchar*) name->base(T_CHAR), name->
2821           return str;
2822         }
2823       }
2824     }
2825   }
2826   return NULL;
2827 }

2829 ThreadPriority JavaThread::java_priority() const {
2830   oop thr_oop = threadObj();
2831   if (thr_oop == NULL) return NormPriority; // Bootstrapping
2832   ThreadPriority priority = java_lang_Thread::priority(thr_oop);
2833   assert(MinPriority <= priority && priority <= MaxPriority, "sanity check");
2834   return priority;
2835 }

2837 void JavaThread::prepare(jobject jni_thread, ThreadPriority prio) {

2839   assert(Threads_lock->owner() == Thread::current(), "must have threads lock");
2840   // Link Java Thread object <-> C++ Thread
```

```
2842   // Get the C++ thread object (an oop) from the JNI handle (a jthread)
2843   // and put it into a new Handle.  The Handle "thread_oop" can then
2844   // be used to pass the C++ thread object to other methods.

2846   // Set the Java level thread object (jthread) field of the
2847   // new thread (a JavaThread *) to C++ thread object using the
2848   // "thread_oop" handle.

2850   // Set the thread field (a JavaThread *) of the
2851   // oop representing the java_lang_Thread to the new thread (a JavaThread *).

2853   Handle thread_oop(Thread::current(),
2854                     JNIHandles::resolve_non_null(jni_thread));
2855   assert(instanceKlass::cast(thread_oop->klass())->is_linked(),
2856     "must be initialized");
2857   set_threadObj(thread_oop());
2858   java_lang_Thread::set_thread(thread_oop(), this);

2860   if (prio == NoPriority) {
2861     prio = java_lang_Thread::priority(thread_oop());
2862     assert(prio != NoPriority, "A valid priority should be present");
2863   }

2865   // Push the Java priority down to the native thread; needs Threads_lock
2866   Thread::set_priority(this, prio);

2868   // Add the new thread to the Threads list and set it in motion.
2869   // We must have threads lock in order to call Threads::add.
2870   // It is crucial that we do not block before the thread is
2871   // added to the Threads list for if a GC happens, then the java_thread oop
2872   // will not be visited by GC.
2873   Threads::add(this);
2874 }

2876 oop JavaThread::current_park_blocker() {
2877   // Support for JSR-166 locks
2878   oop thread_oop = threadObj();
2879   if (thread_oop != NULL &&
2880       JDK_Version::current().supports_thread_park_blocker()) {
2881     return java_lang_Thread::park_blocker(thread_oop);
2882   }
2883   return NULL;
2884 }


2887 void JavaThread::print_stack_on(outputStream* st) {
2888   if (!has_last_Java_frame()) return;
2889   ResourceMark rm;
2890   HandleMark   hm;

2892   RegisterMap reg_map(this);
2893   vframe* start_vf = last_java_vframe(&reg_map);
2894   int count = 0;
2895   for (vframe* f = start_vf; f; f = f->sender() ) {
2896     if (f->is_java_frame()) {
2897       javaVFrame* jvf = javaVFrame::cast(f);
2898       java_lang_Throwable::print_stack_element(st, jvf->method(), jvf->bci());

2900       // Print out lock information
2901       if (JavaMonitorsInStackTrace) {
2902         jvf->print_lock_info_on(st, count);
2903       }
2904     } else {
2905       // Ignore non-Java frames
2906     }
```



```
2908     // Bail-out case for too deep stacks
2909     count++;
2910     if (MaxJavaStackTraceDepth == count) return;
2911   }
2912 }


2915 // JVMTI PopFrame support
2916 void JavaThread::popframe_preserve_args(ByteSize size_in_bytes, void* start) {
2917   assert(_popframe_preserved_args == NULL, "should not wipe out old PopFrame pre
2918   if (in_bytes(size_in_bytes) != 0) {
2919     _popframe_preserved_args = NEW_C_HEAP_ARRAY(char, in_bytes(size_in_bytes));
2920     _popframe_preserved_args_size = in_bytes(size_in_bytes);
2921     Copy::conjoint_jbytes(start, _popframe_preserved_args, _popframe_preserved_a
2922   }
2923 }

2925 void* JavaThread::popframe_preserved_args() {
2926   return _popframe_preserved_args;
2927 }

2929 ByteSize JavaThread::popframe_preserved_args_size() {
2930   return in_ByteSize(_popframe_preserved_args_size);
2931 }

2933 WordSize JavaThread::popframe_preserved_args_size_in_words() {
2934   int sz = in_bytes(popframe_preserved_args_size());
2935   assert(sz % wordSize == 0, "argument size must be multiple of wordSize");
2936   return in_WordSize(sz / wordSize);
2937 }

2939 void JavaThread::popframe_free_preserved_args() {
2940   assert(_popframe_preserved_args != NULL, "should not free PopFrame preserved a
2941   FREE_C_HEAP_ARRAY(char, (char*) _popframe_preserved_args);
2942   _popframe_preserved_args = NULL;
2943   _popframe_preserved_args_size = 0;
2944 }

2946 #ifndef PRODUCT

2948 void JavaThread::trace_frames() {
2949   tty->print_cr("[Describe stack]");
2950   int frame_no = 1;
2951   for(StackFrameStream fst(this); !fst.is_done(); fst.next()) {
2952     tty->print("  %d. ", frame_no++);
2953     fst.current()->print_value_on(tty,this);
2954     tty->cr();
2955   }
2956 }

2958 class PrintAndVerifyOopClosure: public OopClosure {
2959  protected:
2960   template <class T> inline void do_oop_work(T* p) {
2961     oop obj = oopDesc::load_decode_heap_oop(p);
2962     if (obj == NULL) return;
2963     tty->print(INTPTR_FORMAT ": ", p);
2964     if (obj->is_oop_or_null()) {
2965       if (obj->is_objArray()) {
2966         tty->print_cr("valid objArray: " INTPTR_FORMAT, (oopDesc*) obj);
2967       } else {
2968         obj->print();
2969       }
2970     } else {
2971       tty->print_cr("invalid oop: " INTPTR_FORMAT, (oopDesc*) obj);
2972     }
```

```
2973     tty->cr();
2974   }
2975  public:
2976   virtual void do_oop(oop* p) { do_oop_work(p); }
2977   virtual void do_oop(narrowOop* p)  { do_oop_work(p); }
2978 };


2981 static void oops_print(frame* f, const RegisterMap *map) {
2982   PrintAndVerifyOopClosure print;
2983   f->print_value();
2984   f->oops_do(&print, NULL, (RegisterMap*)map);
2985 }

2987 // Print our all the locations that contain oops and whether they are
2988 // valid or not.  This useful when trying to find the oldest frame
2989 // where an oop has gone bad since the frame walk is from youngest to
2990 // oldest.
2991 void JavaThread::trace_oops() {
2992   tty->print_cr("[Trace oops]");
2993   frames_do(oops_print);
2994 }


2997 #ifdef ASSERT
2998 // Print or validate the layout of stack frames
2999 void JavaThread::print_frame_layout(int depth, bool validate_only) {
3000   ResourceMark rm;
3001   PRESERVE_EXCEPTION_MARK;
3002   FrameValues values;
3003   int frame_no = 0;
3004   for(StackFrameStream fst(this, false); !fst.is_done(); fst.next()) {
3005     fst.current()->describe(values, ++frame_no);
3006     if (depth == frame_no) break;
3007   }
3008   if (validate_only) {
3009     values.validate();
3010   } else {
3011     tty->print_cr("[Describe stack layout]");
3012     values.print(this);
3013   }
3014 }
3015 #endif

3017 void JavaThread::trace_stack_from(vframe* start_vf) {
3018   ResourceMark rm;
3019   int vframe_no = 1;
3020   for (vframe* f = start_vf; f; f = f->sender() ) {
3021     if (f->is_java_frame()) {
3022       javaVFrame::cast(f)->print_activation(vframe_no++);
3023     } else {
3024       f->print();
3025     }
3026     if (vframe_no > StackPrintLimit) {
3027       tty->print_cr("...<more frames>...");
3028       return;
3029     }
3030   }
3031 }


3034 void JavaThread::trace_stack() {
3035   if (!has_last_Java_frame()) return;
3036   ResourceMark rm;
3037   HandleMark   hm;
3038   RegisterMap reg_map(this);
```



```
3039   trace_stack_from(last_java_vframe(&reg_map));
3040 }


3043 #endif // PRODUCT


3046 javaVFrame* JavaThread::last_java_vframe(RegisterMap *reg_map) {
3047   assert(reg_map != NULL, "a map must be given");
3048   frame f = last_frame();
3049   for (vframe* vf = vframe::new_vframe(&f, reg_map, this); vf; vf = vf->sender()
3050     if (vf->is_java_frame()) return javaVFrame::cast(vf);
3051   }
3052   return NULL;
3053 }


3056 klassOop JavaThread::security_get_caller_class(int depth) {
3057   vframeStream vfst(this);
3058   vfst.security_get_caller_frame(depth);
3059   if (!vfst.at_end()) {
3060     return vfst.method()->method_holder();
3061   }
3062   return NULL;
3063 }

3065 static void compiler_thread_entry(JavaThread* thread, TRAPS) {
3066   assert(thread->is_Compiler_thread(), "must be compiler thread");
3067   CompileBroker::compiler_thread_loop();
3068 }

3070 // Create a CompilerThread
3071 CompilerThread::CompilerThread(CompileQueue* queue, CompilerCounters* counters)
3072 : JavaThread(&compiler_thread_entry) {
3073   _env   = NULL;
3074   _log   = NULL;
3075   _task  = NULL;
3076   _queue = queue;
3077   _counters = counters;
3078   _buffer_blob = NULL;
3079   _scanned_nmethod = NULL;

3081 #ifndef PRODUCT
3082   _ideal_graph_printer = NULL;
3083 #endif
3084 }

3086 void CompilerThread::oops_do(OopClosure* f, CodeBlobClosure* cf) {
3087   JavaThread::oops_do(f, cf);
3088   if (_scanned_nmethod != NULL && cf != NULL) {
3089     // Safepoints can occur when the sweeper is scanning an nmethod so
3090     // process it here to make sure it isn't unloaded in the middle of
3091     // a scan.
3092     cf->do_code_blob(_scanned_nmethod);
3093   }
3094 }

3096 // ======= Threads ========

3098 // The Threads class links together all active threads, and provides
3099 // operations over all threads.  It is protected by its own Mutex
3100 // lock, which is also used in other contexts to protect thread
3101 // operations from having the thread being operated on from exiting
3102 // and going away unexpectedly (e.g., safepoint synchronization)

3104 JavaThread* Threads::_thread_list = NULL;
```

```
3105 int         Threads::_number_of_threads = 0;
3106 int         Threads::_number_of_non_daemon_threads = 0;
3107 int         Threads::_return_code = 0;
3108 size_t      JavaThread::_stack_size_at_create = 0;

3110 // All JavaThreads
3111 #define ALL_JAVA_THREADS(X) for (JavaThread* X = _thread_list; X; X = X->next())

3113 void os_stream();

3115 // All JavaThreads + all non-JavaThreads (i.e., every thread in the system)
3116 void Threads::threads_do(ThreadClosure* tc) {
3117   assert_locked_or_safepoint(Threads_lock);
3118   // ALL_JAVA_THREADS iterates through all JavaThreads
3119   ALL_JAVA_THREADS(p) {
3120     tc->do_thread(p);
3121   }
3122   // Someday we could have a table or list of all non-JavaThreads.
3123   // For now, just manually iterate through them.
3124   tc->do_thread(VMThread::vm_thread());
3125   Universe::heap()->gc_threads_do(tc);
3126   WatcherThread *wt = WatcherThread::watcher_thread();
3127   // Strictly speaking, the following NULL check isn't sufficient to make sure
3128   // the data for WatcherThread is still valid upon being examined. However,
3129   // considering that WatchThread terminates when the VM is on the way to
3130   // exit at safepoint, the chance of the above is extremely small. The right
3131   // way to prevent termination of WatcherThread would be to acquire
3132   // Terminator_lock, but we can't do that without violating the lock rank
3133   // checking in some cases.
3134   if (wt != NULL)
3135     tc->do_thread(wt);

3137   // If CompilerThreads ever become non-JavaThreads, add them here
3138 }

3140 jint Threads::create_vm(JavaVMInitArgs* args, bool* canTryAgain) {

3142   extern void JDK_Version_init();

3144   // Check version
3145   if (!is_supported_jni_version(args->version)) return JNI_EVERSION;

3147   // Initialize the output stream module
3148   ostream_init();

3150   // Process java launcher properties.
3151   Arguments::process_sun_java_launcher_properties(args);

3153   // Initialize the os module before using TLS
3154   os::init();

3156   // Initialize system properties.
3157   Arguments::init_system_properties();

3159   // So that JDK version can be used as a discrimintor when parsing arguments
3160   JDK_Version_init();

3162   // Update/Initialize System properties after JDK version number is known
3163   Arguments::init_version_specific_system_properties();

3165   // Parse arguments
3166   jint parse_result = Arguments::parse(args);
3167   if (parse_result != JNI_OK) return parse_result;

3169   if (PauseAtStartup) {
3170     os::pause();
```



```
3171   }

3173 #ifndef USDT2
3174   HS_DTRACE_PROBE(hotspot, vm__init__begin);
3175 #else /* USDT2 */
3176   HOTSPOT_VM_INIT_BEGIN();
3177 #endif /* USDT2 */

3179   // Record VM creation timing statistics
3180   TraceVmCreationTime create_vm_timer;
3181   create_vm_timer.start();

3183   // Timing (must come after argument parsing)
3184   TraceTime timer("Create VM", TraceStartupTime);

3186   // Initialize the os module after parsing the args
3187   jint os_init_2_result = os::init_2();
3188   if (os_init_2_result != JNI_OK) return os_init_2_result;

3190   // Initialize output stream logging
3191   ostream_init_log();

3193   // Convert -Xrun to -agentlib: if there is no JVM_OnLoad
3194   // Must be before create_vm_init_agents()
3195   if (Arguments::init_libraries_at_startup()) {
3196     convert_vm_init_libraries_to_agents();
3197   }

3199   // Launch -agentlib/-agentpath and converted -Xrun agents
3200   if (Arguments::init_agents_at_startup()) {
3201     create_vm_init_agents();
3202   }

3204   // Initialize Threads state
3205   _thread_list = NULL;
3206   _number_of_threads = 0;
3207   _number_of_non_daemon_threads = 0;

3209   // Initialize TLS
3210   ThreadLocalStorage::init();

3212   // Initialize global data structures and create system classes in heap
3213   vm_init_globals();

3215   // Attach the main thread to this os thread
3216   JavaThread* main_thread = new JavaThread();
3217   main_thread->set_thread_state(_thread_in_vm);
3218   // must do this before set_active_handles and initialize_thread_local_storage
3219   // Note: on solaris initialize_thread_local_storage() will (indirectly)
3220   // change the stack size recorded here to one based on the java thread
3221   // stacksize. This adjusted size is what is used to figure the placement
3222   // of the guard pages.
3223   main_thread->record_stack_base_and_size();
3224   main_thread->initialize_thread_local_storage();

3226   main_thread->set_active_handles(JNIHandleBlock::allocate_block());

3228   if (!main_thread->set_as_starting_thread()) {
3229     vm_shutdown_during_initialization(
3230       "Failed necessary internal allocation. Out of swap space");
3231     delete main_thread;
3232     *canTryAgain = false; // don't let caller call JNI_CreateJavaVM again
3233     return JNI_ENOMEM;
3234   }

3236   // Enable guard page *after* os::create_main_thread(), otherwise it would
```

```
3237   // crash Linux VM, see notes in os_linux.cpp.
3238   main_thread->create_stack_guard_pages();
3239
3240   // Initialize Java-Level synchronization subsystem
3241   ObjectMonitor::Initialize() ;
3242
3243   // Initialize global modules
3244   jint status = init_globals();
3245   if (status != JNI_OK) {
3246     delete main_thread;
3247     *canTryAgain = false; // don't let caller call JNI_CreateJavaVM again
3248     return status;
3249   }
3250
3251   // Should be done after the heap is fully created
3252   main_thread->cache_global_variables();
3253
3254   HandleMark hm;
3255
3256   { MutexLocker mu(Threads_lock);
3257     Threads::add(main_thread);
3258   }
3259
3260   // Any JVMTI raw monitors entered in onload will transition into
3261   // real raw monitor. VM is setup enough here for raw monitor enter.
3262   JvmtiExport::transition_pending_onload_raw_monitors();
3263
3264   if (VerifyBeforeGC &&
3265       Universe::heap()->total_collections() >= VerifyGCStartAt) {
3266     Universe::heap()->prepare_for_verify();
3267     Universe::verify();   // make sure we're starting with a clean slate
3268   }
3269
3270   // Create the VMThread
3271   { TraceTime timer("Start VMThread", TraceStartupTime);
3272     VMThread::create();
3273     Thread* vmthread = VMThread::vm_thread();
3274
3275     if (!os::create_thread(vmthread, os::vm_thread))
3276       vm_exit_during_initialization("Cannot create VM thread. Out of system reso
3277
3278     // Wait for the VM thread to become ready, and VMThread::run to initialize
3279     // Monitors can have spurious returns, must always check another state flag
3280     {
3281       MutexLocker ml(Notify_lock);
3282       os::start_thread(vmthread);
3283       while (vmthread->active_handles() == NULL) {
3284         Notify_lock->wait();
3285       }
3286     }
3287   }
3288
3289   assert (Universe::is_fully_initialized(), "not initialized");
3290   EXCEPTION_MARK;
3291
3292   // At this point, the Universe is initialized, but we have not executed
3293   // any byte code.  Now is a good time (the only time) to dump out the
3294   // internal state of the JVM for sharing.
3295
3296   if (DumpSharedSpaces) {
3297     Universe::heap()->preload_and_dump(CHECK_0);
3298     ShouldNotReachHere();
3299   }
3300
3301   // Always call even when there are not JVMTI environments yet, since environme
3302   // may be attached late and JVMTI must track phases of VM execution
```



```
3303   JvmtiExport::enter_start_phase();
3304
3305   // Notify JVMTI agents that VM has started (JNI is up) - nop if no agents.
3306   JvmtiExport::post_vm_start();
3307
3308   {
3309     TraceTime timer("Initialize java.lang classes", TraceStartupTime);
3310
3311     if (EagerXrunInit && Arguments::init_libraries_at_startup()) {
3312       create_vm_init_libraries();
3313     }
3314
3315     if (InitializeJavaLangString) {
3316       initialize_class(vmSymbols::java_lang_String(), CHECK_0);
3317     } else {
3318       warning("java.lang.String not initialized");
3319     }
3320
3321     if (AggressiveOpts) {
3322       {
3323         // Forcibly initialize java/util/HashMap and mutate the private
3324         // static final "frontCacheEnabled" field before we start creating insta
3325 #ifdef ASSERT
3326         klassOop tmp_k = SystemDictionary::find(vmSymbols::java_util_HashMap(),
3327         assert(tmp_k == NULL, "java/util/HashMap should not be loaded yet");
3328 #endif
3329         klassOop k_o = SystemDictionary::resolve_or_null(vmSymbols::java_util_Ha
3330         KlassHandle k = KlassHandle(THREAD, k_o);
3331         guarantee(k.not_null(), "Must find java/util/HashMap");
3332         instanceKlassHandle ik = instanceKlassHandle(THREAD, k());
3333         ik->initialize(CHECK_0);
3334         fieldDescriptor fd;
3335         // Possible we might not find this field; if so, don't break
3336         if (ik->find_local_field(vmSymbols::frontCacheEnabled_name(), vmSymbols:
3337           k()->java_mirror()->bool_field_put(fd.offset(), true);
3338         }
3339       }
3340
3341       if (UseStringCache) {
3342         // Forcibly initialize java/lang/StringValue and mutate the private
3343         // static final "stringCacheEnabled" field before we start creating inst
3344         klassOop k_o = SystemDictionary::resolve_or_null(vmSymbols::java_lang_St
3345         // Possible that StringValue isn't present: if so, silently don't break
3346         if (k_o != NULL) {
3347           KlassHandle k = KlassHandle(THREAD, k_o);
3348           instanceKlassHandle ik = instanceKlassHandle(THREAD, k());
3349           ik->initialize(CHECK_0);
3350           fieldDescriptor fd;
3351           // Possible we might not find this field: if so, silently don't break
3352           if (ik->find_local_field(vmSymbols::stringCacheEnabled_name(), vmSymbo
3353             k()->java_mirror()->bool_field_put(fd.offset(), true);
3354           }
3355         }
3356       }
3357     }
3358
3359     // Initialize java_lang.System (needed before creating the thread)
3360     if (InitializeJavaLangSystem) {
3361       initialize_class(vmSymbols::java_lang_System(), CHECK_0);
3362       initialize_class(vmSymbols::java_lang_ThreadGroup(), CHECK_0);
3363       Handle thread_group = create_initial_thread_group(CHECK_0);
3364       Universe::set_main_thread_group(thread_group());
3365       initialize_class(vmSymbols::java_lang_Thread(), CHECK_0);
3366       oop thread_object = create_initial_thread(thread_group, main_thread, CHECK
3367       main_thread->set_threadObj(thread_object);
3368       // Set thread status to running since main thread has
```

```
3369       // been started and running.
3370       java_lang_Thread::set_thread_status(thread_object,
3371                                           java_lang_Thread::RUNNABLE);

3373       // The VM preresolve methods to these classes. Make sure that get initiali
3374       initialize_class(vmSymbols::java_lang_reflect_Method(), CHECK_0);
3375       initialize_class(vmSymbols::java_lang_ref_Finalizer(),  CHECK_0);
3376       // The VM creates & returns objects of this class. Make sure it's initiali
3377       initialize_class(vmSymbols::java_lang_Class(), CHECK_0);
3378       call_initializeSystemClass(CHECK_0);

3380       // get the Java runtime name after java.lang.System is initialized
3381       JDK_Version::set_runtime_name(get_java_runtime_name(THREAD));
3382 #endif /* ! codereview */
3383     } else {
3384       warning("java.lang.System not initialized");
3385     }

3387     // an instance of OutOfMemory exception has been allocated earlier
3388     if (InitializeJavaLangExceptionsErrors) {
3389       initialize_class(vmSymbols::java_lang_OutOfMemoryError(), CHECK_0);
3390       initialize_class(vmSymbols::java_lang_NullPointerException(), CHECK_0);
3391       initialize_class(vmSymbols::java_lang_ClassCastException(), CHECK_0);
3392       initialize_class(vmSymbols::java_lang_ArrayStoreException(), CHECK_0);
3393       initialize_class(vmSymbols::java_lang_ArithmeticException(), CHECK_0);
3394       initialize_class(vmSymbols::java_lang_StackOverflowError(), CHECK_0);
3395       initialize_class(vmSymbols::java_lang_IllegalMonitorStateException(), CHEC
3396       initialize_class(vmSymbols::java_lang_IllegalArgumentException(), CHECK_0)
3397     } else {
3398       warning("java.lang.OutOfMemoryError has not been initialized");
3399       warning("java.lang.NullPointerException has not been initialized");
3400       warning("java.lang.ClassCastException has not been initialized");
3401       warning("java.lang.ArrayStoreException has not been initialized");
3402       warning("java.lang.ArithmeticException has not been initialized");
3403       warning("java.lang.StackOverflowError has not been initialized");
3404       warning("java.lang.IllegalArgumentException has not been initialized");
3405     }
3406   }

3408   // See        : bugid 4211085.
3409   // Background : the static initializer of java.lang.Compiler tries to read
3410   //              property"java.compiler" and read & write property "java.vm.inf
3411   //              When a security manager is installed through the command line
3412   //              option "-Djava.security.manager", the above properties are not
3413   //              readable and the static initializer for java.lang.Compiler fai
3414   //              resulting in a NoClassDefFoundError.  This can happen in any
3415   //              user code which calls methods in java.lang.Compiler.
3416   // Hack :       the hack is to pre-load and initialize this class, so that onl
3417   //              system domains are on the stack when the properties are read.
3418   //              Currently even the AWT code has calls to methods in java.lang.
3419   //              On the classic VM, java.lang.Compiler is loaded very early to
3420   // Future Fix : the best fix is to grant everyone permissions to read "java.co
3421   //              read and write"java.vm.info" in the default policy file. See b
3422   //              Once that is done, we should remove this hack.
3423   initialize_class(vmSymbols::java_lang_Compiler(), CHECK_0);

3425   // More hackery - the static initializer of java.lang.Compiler adds the string
3426   // the java.vm.info property if no jit gets loaded through java.lang.Compiler
3427   // compiler does not get loaded through java.lang.Compiler).  "java -version"
3428   // hotspot vm says "nojit" all the time which is confusing.  So, we reset it h
3429   // This should also be taken out as soon as 4211383 gets fixed.
3430   reset_vm_info_property(CHECK_0);

3432   quicken_jni_functions();

3434   // Must be run after init_ft which initializes ft_enabled
```



```
3435   if (TRACE_INITIALIZE() != JNI_OK) {
3436     vm_exit_during_initialization("Failed to initialize tracing backend");
3437   }

3439   // Set flag that basic initialization has completed. Used by exceptions and va
3440   // debug stuff, that does not work until all basic classes have been initializ
3441   set_init_completed();

3443 #ifndef USDT2
3444   HS_DTRACE_PROBE(hotspot, vm__init__end);
3445 #else /* USDT2 */
3446   HOTSPOT_VM_INIT_END();
3447 #endif /* USDT2 */

3449   // record VM initialization completion time
3450   Management::record_vm_init_completed();

3452   // Compute system loader. Note that this has to occur after set_init_completed
3453   // valid exceptions may be thrown in the process.
3454   // Note that we do not use CHECK_0 here since we are inside an EXCEPTION_MARK
3455   // set_init_completed has just been called, causing exceptions not to be short
3456   // anymore. We call vm_exit_during_initialization directly instead.
3457   SystemDictionary::compute_java_system_loader(THREAD);
3458   if (HAS_PENDING_EXCEPTION) {
3459     vm_exit_during_initialization(Handle(THREAD, PENDING_EXCEPTION));
3460   }

3462 #ifndef SERIALGC
3463   // Support for ConcurrentMarkSweep. This should be cleaned up
3464   // and better encapsulated. The ugly nested if test would go away
3465   // once things are properly refactored. XXX YSR
3466   if (UseConcMarkSweepGC || UseG1GC) {
3467     if (UseConcMarkSweepGC) {
3468       ConcurrentMarkSweepThread::makeSurrogateLockerThread(THREAD);
3469     } else {
3470       ConcurrentMarkThread::makeSurrogateLockerThread(THREAD);
3471     }
3472     if (HAS_PENDING_EXCEPTION) {
3473       vm_exit_during_initialization(Handle(THREAD, PENDING_EXCEPTION));
3474     }
3475   }
3476 #endif // SERIALGC

3478   // Always call even when there are not JVMTI environments yet, since environme
3479   // may be attached late and JVMTI must track phases of VM execution
3480   JvmtiExport::enter_live_phase();

3482   // Signal Dispatcher needs to be started before VMInit event is posted
3483   os::signal_init();

3485   // Start Attach Listener if +StartAttachListener or it can't be started lazily
3486   if (!DisableAttachMechanism) {
3487     if (StartAttachListener || AttachListener::init_at_startup()) {
3488       AttachListener::init();
3489     }
3490   }

3492   // Launch -Xrun agents
3493   // Must be done in the JVMTI live phase so that for backward compatibility the
3494   // back-end can launch with -Xdebug -Xrunjdwp.
3495   if (!EagerXrunInit && Arguments::init_libraries_at_startup()) {
3496     create_vm_init_libraries();
3497   }

3499   // Notify JVMTI agents that VM initialization is complete - nop if no agents.
3500   JvmtiExport::post_vm_initialized();
```

```
3502   if (!TRACE_START()) {
3503     vm_exit_during_initialization(Handle(THREAD, PENDING_EXCEPTION));
3504   }

3506   if (CleanChunkPoolAsync) {
3507     Chunk::start_chunk_pool_cleaner_task();
3508   }

3510   // initialize compiler(s)
3511   CompileBroker::compilation_init();

3513   Management::initialize(THREAD);
3514   if (HAS_PENDING_EXCEPTION) {
3515     // management agent fails to start possibly due to
3516     // configuration problem and is responsible for printing
3517     // stack trace if appropriate. Simply exit VM.
3518     vm_exit(1);
3519   }

3521   if (Arguments::has_profile())       FlatProfiler::engage(main_thread, true);
3522   if (Arguments::has_alloc_profile()) AllocationProfiler::engage();
3523   if (MemProfiling)                   MemProfiler::engage();
3524   StatSampler::engage();
3525   if (CheckJNICalls)                  JniPeriodicChecker::engage();

3527   BiasedLocking::init();

3529   if (JDK_Version::current().post_vm_init_hook_enabled()) {
3530     call_postVMInitHook(THREAD);
3531     // The Java side of PostVMInitHook.run must deal with all
3532     // exceptions and provide means of diagnosis.
3533     if (HAS_PENDING_EXCEPTION) {
3534       CLEAR_PENDING_EXCEPTION;
3535     }
3536   }

3538   // Start up the WatcherThread if there are any periodic tasks
3539   // NOTE:  All PeriodicTasks should be registered by now. If they
3540   //   aren't, late joiners might appear to start slowly (we might
3541   //   take a while to process their first tick).
3542   if (PeriodicTask::num_tasks() > 0) {
3543     WatcherThread::start();
3544   }

3546   // Give os specific code one last chance to start
3547   os::init_3();

3549   create_vm_timer.end();
3550   return JNI_OK;
3551 }

3553 // type for the Agent_OnLoad and JVM_OnLoad entry points
3554 extern "C" {
3555   typedef jint (JNICALL *OnLoadEntry_t)(JavaVM *, char *, void *);
3556 }
3557 // Find a command line agent library and return its entry point for
3558 //         -agentlib:  -agentpath:   -Xrun
3559 // num_symbol_entries must be passed-in since only the caller knows the number o
3560 static OnLoadEntry_t lookup_on_load(AgentLibrary* agent, const char *on_load_sym
3561   OnLoadEntry_t on_load_entry = NULL;
3562   void *library = agent->os_lib();  // check if we have looked it up before

3564   if (library == NULL) {
3565     char buffer[JVM_MAXPATHLEN];
3566     char ebuf[1024];
```



```
3567     const char *name = agent->name();
3568     const char *msg = "Could not find agent library ";

3570     if (agent->is_absolute_path()) {
3571       library = os::dll_load(name, ebuf, sizeof ebuf);
3572       if (library == NULL) {
3573         const char *sub_msg = " in absolute path, with error: ";
3574         size_t len = strlen(msg) + strlen(name) + strlen(sub_msg) + strlen(ebuf)
3575         char *buf = NEW_C_HEAP_ARRAY(char, len);
3576         jio_snprintf(buf, len, "%s%s%s%s", msg, name, sub_msg, ebuf);
3577         // If we can't find the agent, exit.
3578         vm_exit_during_initialization(buf, NULL);
3579         FREE_C_HEAP_ARRAY(char, buf);
3580       }
3581     } else {
3582       // Try to load the agent from the standard dll directory
3583       os::dll_build_name(buffer, sizeof(buffer), Arguments::get_dll_dir(), name)
3584       library = os::dll_load(buffer, ebuf, sizeof ebuf);
3585 #ifdef KERNEL
3586       // Download instrument dll
3587       if (library == NULL && strcmp(name, "instrument") == 0) {
3588         char *props = Arguments::get_kernel_properties();
3589         char *home  = Arguments::get_java_home();
3590         const char *fmt   = "%s/bin/java %s -Dkernel.background.download=false"
3591                             " sun.jkernel.DownloadManager -download client_jvm";
3592         size_t length = strlen(props) + strlen(home) + strlen(fmt) + 1;
3593         char *cmd = NEW_C_HEAP_ARRAY(char, length);
3594         jio_snprintf(cmd, length, fmt, home, props);
3595         int status = os::fork_and_exec(cmd);
3596         FreeHeap(props);
3597         if (status == -1) {
3598           warning(cmd);
3599           vm_exit_during_initialization("fork_and_exec failed: %s",
3600                                         strerror(errno));
3601         }
3602         FREE_C_HEAP_ARRAY(char, cmd);
3603         // when this comes back the instrument.dll should be where it belongs.
3604         library = os::dll_load(buffer, ebuf, sizeof ebuf);
3605       }
3606 #endif // KERNEL
3607       if (library == NULL) { // Try the local directory
3608         char ns[1] = {0};
3609         os::dll_build_name(buffer, sizeof(buffer), ns, name);
3610         library = os::dll_load(buffer, ebuf, sizeof ebuf);
3611         if (library == NULL) {
3612           const char *sub_msg = " on the library path, with error: ";
3613           size_t len = strlen(msg) + strlen(name) + strlen(sub_msg) + strlen(ebu
3614           char *buf = NEW_C_HEAP_ARRAY(char, len);
3615           jio_snprintf(buf, len, "%s%s%s%s", msg, name, sub_msg, ebuf);
3616           // If we can't find the agent, exit.
3617           vm_exit_during_initialization(buf, NULL);
3618           FREE_C_HEAP_ARRAY(char, buf);
3619         }
3620       }
3621     }
3622     agent->set_os_lib(library);
3623   }

3625   // Find the OnLoad function.
3626   for (size_t symbol_index = 0; symbol_index < num_symbol_entries; symbol_index+
3627     on_load_entry = CAST_TO_FN_PTR(OnLoadEntry_t, os::dll_lookup(library, on_loa
3628     if (on_load_entry != NULL) break;
3629   }
3630   return on_load_entry;
3631 }
```

```
3633 // Find the JVM_OnLoad entry point
3634 static OnLoadEntry_t lookup_jvm_on_load(AgentLibrary* agent) {
3635   const char *on_load_symbols[] = JVM_ONLOAD_SYMBOLS;
3636   return lookup_on_load(agent, on_load_symbols, sizeof(on_load_symbols) / sizeof
3637 }

3639 // Find the Agent_OnLoad entry point
3640 static OnLoadEntry_t lookup_agent_on_load(AgentLibrary* agent) {
3641   const char *on_load_symbols[] = AGENT_ONLOAD_SYMBOLS;
3642   return lookup_on_load(agent, on_load_symbols, sizeof(on_load_symbols) / sizeof
3643 }

3645 // For backwards compatibility with -Xrun
3646 // Convert libraries with no JVM_OnLoad, but which have Agent_OnLoad to be
3647 // treated like -agentpath:
3648 // Must be called before agent libraries are created
3649 void Threads::convert_vm_init_libraries_to_agents() {
3650   AgentLibrary* agent;
3651   AgentLibrary* next;

3653   for (agent = Arguments::libraries(); agent != NULL; agent = next) {
3654     next = agent->next();  // cache the next agent now as this agent may get mov
3655     OnLoadEntry_t on_load_entry = lookup_jvm_on_load(agent);

3657     // If there is an JVM_OnLoad function it will get called later,
3658     // otherwise see if there is an Agent_OnLoad
3659     if (on_load_entry == NULL) {
3660       on_load_entry = lookup_agent_on_load(agent);
3661       if (on_load_entry != NULL) {
3662         // switch it to the agent list -- so that Agent_OnLoad will be called,
3663         // JVM_OnLoad won't be attempted and Agent_OnUnload will
3664         Arguments::convert_library_to_agent(agent);
3665       } else {
3666         vm_exit_during_initialization("Could not find JVM_OnLoad or Agent_OnLoad
3667       }
3668     }
3669   }
3670 }

3672 // Create agents for -agentlib:  -agentpath:  and converted -Xrun
3673 // Invokes Agent_OnLoad
3674 // Called very early -- before JavaThreads exist
3675 void Threads::create_vm_init_agents() {
3676   extern struct JavaVM_ main_vm;
3677   AgentLibrary* agent;

3679   JvmtiExport::enter_onload_phase();
3680   for (agent = Arguments::agents(); agent != NULL; agent = agent->next()) {
3681     OnLoadEntry_t  on_load_entry = lookup_agent_on_load(agent);

3683     if (on_load_entry != NULL) {
3684       // Invoke the Agent_OnLoad function
3685       jint err = (*on_load_entry)(&main_vm, agent->options(), NULL);
3686       if (err != JNI_OK) {
3687         vm_exit_during_initialization("agent library failed to init", agent->nam
3688       }
3689     } else {
3690       vm_exit_during_initialization("Could not find Agent_OnLoad function in the
3691     }
3692   }
3693   JvmtiExport::enter_primordial_phase();
3694 }

3696 extern "C" {
3697   typedef void (JNICALL *Agent_OnUnload_t)(JavaVM *);
3698 }
```



```
3700 void Threads::shutdown_vm_agents() {
3701   // Send any Agent_OnUnload notifications
3702   const char *on_unload_symbols[] = AGENT_ONUNLOAD_SYMBOLS;
3703   extern struct JavaVM_ main_vm;
3704   for (AgentLibrary* agent = Arguments::agents(); agent != NULL; agent = agent->

3706     // Find the Agent_OnUnload function.
3707     for (uint symbol_index = 0; symbol_index < ARRAY_SIZE(on_unload_symbols); sy
3708       Agent_OnUnload_t unload_entry = CAST_TO_FN_PTR(Agent_OnUnload_t,
3709               os::dll_lookup(agent->os_lib(), on_unload_symbols[symbol_index]))

3711       // Invoke the Agent_OnUnload function
3712       if (unload_entry != NULL) {
3713         JavaThread* thread = JavaThread::current();
3714         ThreadToNativeFromVM ttn(thread);
3715         HandleMark hm(thread);
3716         (*unload_entry)(&main_vm);
3717         break;
3718       }
3719     }
3720   }
3721 }

3723 // Called for after the VM is initialized for -Xrun libraries which have not bee
3724 // Invokes JVM_OnLoad
3725 void Threads::create_vm_init_libraries() {
3726   extern struct JavaVM_ main_vm;
3727   AgentLibrary* agent;

3729   for (agent = Arguments::libraries(); agent != NULL; agent = agent->next()) {
3730     OnLoadEntry_t on_load_entry = lookup_jvm_on_load(agent);

3732     if (on_load_entry != NULL) {
3733       // Invoke the JVM_OnLoad function
3734       JavaThread* thread = JavaThread::current();
3735       ThreadToNativeFromVM ttn(thread);
3736       HandleMark hm(thread);
3737       jint err = (*on_load_entry)(&main_vm, agent->options(), NULL);
3738       if (err != JNI_OK) {
3739         vm_exit_during_initialization("-Xrun library failed to init", agent->nam
3740       }
3741     } else {
3742       vm_exit_during_initialization("Could not find JVM_OnLoad function in -Xrun
3743     }
3744   }
3745 }

3747 // Last thread running calls java.lang.Shutdown.shutdown()
3748 void JavaThread::invoke_shutdown_hooks() {
3749   HandleMark hm(this);

3751   // We could get here with a pending exception, if so clear it now.
3752   if (this->has_pending_exception()) {
3753     this->clear_pending_exception();
3754   }

3756   EXCEPTION_MARK;
3757   klassOop k =
3758     SystemDictionary::resolve_or_null(vmSymbols::java_lang_Shutdown(),
3759                                       THREAD);
3760   if (k != NULL) {
3761     // SystemDictionary::resolve_or_null will return null if there was
3762     // an exception.  If we cannot load the Shutdown class, just don't
3763     // call Shutdown.shutdown() at all.  This will mean the shutdown hooks
3764     // and finalizers (if runFinalizersOnExit is set) won't be run.
```

```
3765     // Note that if a shutdown hook was registered or runFinalizersOnExit
3766     // was called, the Shutdown class would have already been loaded
3767     // (Runtime.addShutdownHook and runFinalizersOnExit will load it).
3768     instanceKlassHandle shutdown_klass (THREAD, k);
3769     JavaValue result(T_VOID);
3770     JavaCalls::call_static(&result,
3771                            shutdown_klass,
3772                            vmSymbols::shutdown_method_name(),
3773                            vmSymbols::void_method_signature(),
3774                            THREAD);
3775   }
3776   CLEAR_PENDING_EXCEPTION;
3777 }

3779 // Threads::destroy_vm() is normally called from jni_DestroyJavaVM() when
3780 // the program falls off the end of main(). Another VM exit path is through
3781 // vm_exit() when the program calls System.exit() to return a value or when
3782 // there is a serious error in VM. The two shutdown paths are not exactly
3783 // the same, but they share Shutdown.shutdown() at Java level and before_exit()
3784 // and VM_Exit op at VM level.
3785 //
3786 // Shutdown sequence:
3787 //   + Wait until we are the last non-daemon thread to execute
3788 //     <-- every thing is still working at this moment -->
3789 //   + Call java.lang.Shutdown.shutdown(), which will invoke Java level
3790 //        shutdown hooks, run finalizers if finalization-on-exit
3791 //   + Call before_exit(), prepare for VM exit
3792 //      > run VM level shutdown hooks (they are registered through JVM_OnExit(),
3793 //        currently the only user of this mechanism is File.deleteOnExit())
3794 //      > stop flat profiler, StatSampler, watcher thread, CMS threads,
3795 //        post thread end and vm death events to JVMTI,
3796 //        stop signal thread
3797 //   + Call JavaThread::exit(), it will:
3798 //      > release JNI handle blocks, remove stack guard pages
3799 //      > remove this thread from Threads list
3800 //     <-- no more Java code from this thread after this point -->
3801 //   + Stop VM thread, it will bring the remaining VM to a safepoint and stop
3802 //     the compiler threads at safepoint
3803 //     <-- do not use anything that could get blocked by Safepoint -->
3804 //   + Disable tracing at JNI/JVM barriers
3805 //   + Set _vm_exited flag for threads that are still running native code
3806 //   + Delete this thread
3807 //   + Call exit_globals()
3808 //      > deletes tty
3809 //      > deletes PerfMemory resources
3810 //   + Return to caller

3812 bool Threads::destroy_vm() {
3813   JavaThread* thread = JavaThread::current();

3815   // Wait until we are the last non-daemon thread to execute
3816   { MutexLocker nu(Threads_lock);
3817     while (Threads::number_of_non_daemon_threads() > 1 )
3818       // This wait should make safepoint checks, wait without a timeout,
3819       // and wait as a suspend-equivalent condition.
3820       //
3821       // Note: If the FlatProfiler is running and this thread is waiting
3822       // for another non-daemon thread to finish, then the FlatProfiler
3823       // is waiting for the external suspend request on this thread to
3824       // complete. wait_for_ext_suspend_completion() will eventually
3825       // timeout, but that takes time. Making this wait a suspend-
3826       // equivalent condition solves that timeout problem.
3827       //
3828       Threads_lock->wait(!Mutex::_no_safepoint_check_flag, 0,
3829                          Mutex::_as_suspend_equivalent_flag);
3830   }
```



```
3832   // Hang forever on exit if we are reporting an error.
3833   if (ShowMessageBoxOnError && is_error_reported()) {
3834     os::infinite_sleep();
3835   }
3836   os::wait_for_keypress_at_exit();

3838   if (JDK_Version::is_jdk12x_version()) {
3839     // We are the last thread running, so check if finalizers should be run.
3840     // For 1.3 or later this is done in thread->invoke_shutdown_hooks()
3841     HandleMark rm(thread);
3842     Universe::run_finalizers_on_exit();
3843   } else {
3844     // run Java level shutdown hooks
3845     thread->invoke_shutdown_hooks();
3846   }

3848   before_exit(thread);

3850   thread->exit(true);

3852   // Stop VM thread.
3853   {
3854     // 4945125 The vm thread comes to a safepoint during exit.
3855     // GC vm_operations can get caught at the safepoint, and the
3856     // heap is unparseable if they are caught. Grab the Heap_lock
3857     // to prevent this. The GC vm_operations will not be able to
3858     // queue until after the vm thread is dead.
3859     // After this point, we'll never emerge out of the safepoint before
3860     // the VM exits, so concurrent GC threads do not need to be explicitly
3861     // stopped; they remain inactive until the process exits.
3862     // Note: some concurrent G1 threads may be running during a safepoint,
3863     // but these will not be accessing the heap, just some G1-specific side
3864     // data structures that are not accessed by any other threads but them
3865     // after this point in a terminal safepoint.

3867     MutexLocker ml(Heap_lock);

3869     VMThread::wait_for_vm_thread_exit();
3870     assert(SafepointSynchronize::is_at_safepoint(), "VM thread should exit at Sa
3871     VMThread::destroy();
3872   }

3874   // clean up ideal graph printers
3875 #if defined(COMPILER2) && !defined(PRODUCT)
3876   IdealGraphPrinter::clean_up();
3877 #endif

3879   // Now, all Java threads are gone except daemon threads. Daemon threads
3880   // running Java code or in VM are stopped by the Safepoint. However,
3881   // daemon threads executing native code are still running.  But they
3882   // will be stopped at native=>Java/VM barriers. Note that we can't
3883   // simply kill or suspend them, as it is inherently deadlock-prone.

3885 #ifndef PRODUCT
3886   // disable function tracing at JNI/JVM barriers
3887   TraceJNICalls = false;
3888   TraceJVMCalls = false;
3889   TraceRuntimeCalls = false;
3890 #endif

3892   VM_Exit::set_vm_exited();

3894   notify_vm_shutdown();

3896   delete thread;
```

```
3898   // exit_globals() will delete tty
3899   exit_globals();

3901   return true;
3902 }


3905 jboolean Threads::is_supported_jni_version_including_1_1(jint version) {
3906   if (version == JNI_VERSION_1_1) return JNI_TRUE;
3907   return is_supported_jni_version(version);
3908 }


3911 jboolean Threads::is_supported_jni_version(jint version) {
3912   if (version == JNI_VERSION_1_2) return JNI_TRUE;
3913   if (version == JNI_VERSION_1_4) return JNI_TRUE;
3914   if (version == JNI_VERSION_1_6) return JNI_TRUE;
3915   return JNI_FALSE;
3916 }


3919 void Threads::add(JavaThread* p, bool force_daemon) {
3920   // The threads lock must be owned at this point
3921   assert_locked_or_safepoint(Threads_lock);

3923   // See the comment for this method in thread.hpp for its purpose and
3924   // why it is called here.
3925   p->initialize_queues();
3926   p->set_next(_thread_list);
3927   _thread_list = p;
3928   _number_of_threads++;
3929   oop threadObj = p->threadObj();
3930   bool daemon = true;
3931   // Bootstrapping problem: threadObj can be null for initial
3932   // JavaThread (or for threads attached via JNI)
3933   if ((!force_daemon) && (threadObj == NULL || !java_lang_Thread::is_daemon(thre
3934     _number_of_non_daemon_threads++;
3935     daemon = false;
3936   }

3938   ThreadService::add_thread(p, daemon);

3940   // Possible GC point.
3941   Events::log(p, "Thread added: " INTPTR_FORMAT, p);
3942 }

3944 void Threads::remove(JavaThread* p) {
3945   // Extra scope needed for Thread_lock, so we can check
3946   // that we do not remove thread without safepoint code notice
3947   { MutexLocker ml(Threads_lock);

3949     assert(includes(p), "p must be present");

3951     JavaThread* current = _thread_list;
3952     JavaThread* prev    = NULL;

3954     while (current != p) {
3955       prev    = current;
3956       current = current->next();
3957     }

3959     if (prev) {
3960       prev->set_next(current->next());
3961     } else {
3962       _thread_list = p->next();
```



```
3963     }
3964     _number_of_threads--;
3965     oop threadObj = p->threadObj();
3966     bool daemon = true;
3967     if (threadObj == NULL || !java_lang_Thread::is_daemon(threadObj)) {
3968       _number_of_non_daemon_threads--;
3969       daemon = false;

3971       // Only one thread left, do a notify on the Threads_lock so a thread waiti
3972       // on destroy_vm will wake up.
3973       if (number_of_non_daemon_threads() == 1)
3974         Threads_lock->notify_all();
3975     }
3976     ThreadService::remove_thread(p, daemon);

3978     // Make sure that safepoint code disregard this thread. This is needed since
3979     // the thread might mess around with locks after this point. This can cause
3980     // to do callbacks into the safepoint code. However, the safepoint code is n
3981     // of this thread since it is removed from the queue.
3982     p->set_terminated_value();
3983   } // unlock Threads_lock

3985   // Since Events::log uses a lock, we grab it outside the Threads_lock
3986   Events::log(p, "Thread exited: " INTPTR_FORMAT, p);
3987 }

3989 // Threads_lock must be held when this is called (or must be called during a saf
3990 bool Threads::includes(JavaThread* p) {
3991   assert(Threads_lock->is_locked(), "sanity check");
3992   ALL_JAVA_THREADS(q) {
3993     if (q == p ) {
3994       return true;
3995     }
3996   }
3997   return false;
3998 }

4000 // Operations on the Threads list for GC.  These are not explicitly locked,
4001 // but the garbage collector must provide a safe context for them to run.
4002 // In particular, these things should never be called when the Threads_lock
4003 // is held by some other thread. (Note: the Safepoint abstraction also
4004 // uses the Threads_lock to gurantee this property. It also makes sure that
4005 // all threads gets blocked when exiting or starting).

4007 void Threads::oops_do(OopClosure* f, CodeBlobClosure* cf) {
4008   ALL_JAVA_THREADS(p) {
4009     p->oops_do(f, cf);
4010   }
4011   VMThread::vm_thread()->oops_do(f, cf);
4012 }

4014 void Threads::possibly_parallel_oops_do(OopClosure* f, CodeBlobClosure* cf) {
4015   // Introduce a mechanism allowing parallel threads to claim threads as
4016   // root groups.  Overhead should be small enough to use all the time,
4017   // even in sequential code.
4018   SharedHeap* sh = SharedHeap::heap();
4019   // Cannot yet substitute active_workers for n_par_threads
4020   // because of G1CollectedHeap::verify() use of
4021   // SharedHeap::process_strong_roots().  n_par_threads == 0 will
4022   // turn off parallelism in process_strong_roots while active_workers
4023   // is being used for parallelism elsewhere.
4024   bool is_par = sh->n_par_threads() > 0;
4025   assert(!is_par ||
4026          (SharedHeap::heap()->n_par_threads() ==
4027           SharedHeap::heap()->workers()->active_workers()), "Mismatch");
4028   int cp = SharedHeap::heap()->strong_roots_parity();
```

```
4029   ALL_JAVA_THREADS(p) {
4030     if (p->claim_oops_do(is_par, cp)) {
4031       p->oops_do(f, cf);
4032     }
4033   }
4034   VMThread* vmt = VMThread::vm_thread();
4035   if (vmt->claim_oops_do(is_par, cp)) {
4036     vmt->oops_do(f, cf);
4037   }
4038 }

4040 #ifndef SERIALGC
4041 // Used by ParallelScavenge
4042 void Threads::create_thread_roots_tasks(GCTaskQueue* q) {
4043   ALL_JAVA_THREADS(p) {
4044     q->enqueue(new ThreadRootsTask(p));
4045   }
4046   q->enqueue(new ThreadRootsTask(VMThread::vm_thread()));
4047 }

4049 // Used by Parallel Old
4050 void Threads::create_thread_roots_marking_tasks(GCTaskQueue* q) {
4051   ALL_JAVA_THREADS(p) {
4052     q->enqueue(new ThreadRootsMarkingTask(p));
4053   }
4054   q->enqueue(new ThreadRootsMarkingTask(VMThread::vm_thread()));
4055 }
4056 #endif // SERIALGC

4058 void Threads::nmethods_do(CodeBlobClosure* cf) {
4059   ALL_JAVA_THREADS(p) {
4060     p->nmethods_do(cf);
4061   }
4062   VMThread::vm_thread()->nmethods_do(cf);
4063 }

4065 void Threads::gc_epilogue() {
4066   ALL_JAVA_THREADS(p) {
4067     p->gc_epilogue();
4068   }
4069 }

4071 void Threads::gc_prologue() {
4072   ALL_JAVA_THREADS(p) {
4073     p->gc_prologue();
4074   }
4075 }

4077 void Threads::deoptimized_wrt_marked_nmethods() {
4078   ALL_JAVA_THREADS(p) {
4079     p->deoptimized_wrt_marked_nmethods();
4080   }
4081 }


4084 // Get count Java threads that are waiting to enter the specified monitor.
4085 GrowableArray<JavaThread*>* Threads::get_pending_threads(int count,
4086   address monitor, bool doLock) {
4087   assert(doLock || SafepointSynchronize::is_at_safepoint(),
4088     "must grab Threads_lock or be at safepoint");
4089   GrowableArray<JavaThread*>* result = new GrowableArray<JavaThread*>(count);

4091   int i = 0;
4092   {
4093     MutexLockerEx ml(doLock ? Threads_lock : NULL);
4094     ALL_JAVA_THREADS(p) {
```



```
4095       if (p->is_Compiler_thread()) continue;

4097       address pending = (address)p->current_pending_monitor();
4098       if (pending == monitor) {             // found a match
4099         if (i < count) result->append(p);   // save the first count matches
4100         i++;
4101       }
4102     }
4103   }
4104   return result;
4105 }


4108 JavaThread *Threads::owning_thread_from_monitor_owner(address owner, bool doLock
4109   assert(doLock ||
4110          Threads_lock->owned_by_self() ||
4111          SafepointSynchronize::is_at_safepoint(),
4112          "must grab Threads_lock or be at safepoint");

4114   // NULL owner means not locked so we can skip the search
4115   if (owner == NULL) return NULL;

4117   {
4118     MutexLockerEx ml(doLock ? Threads_lock : NULL);
4119     ALL_JAVA_THREADS(p) {
4120       // first, see if owner is the address of a Java thread
4121       if (owner == (address)p) return p;
4122     }
4123   }
4124   assert(UseHeavyMonitors == false, "Did not find owning Java thread with UseHea
4125   if (UseHeavyMonitors) return NULL;

4127   //
4128   // If we didn't find a matching Java thread and we didn't force use of
4129   // heavyweight monitors, then the owner is the stack address of the
4130   // Lock Word in the owning Java thread's stack.
4131   //
4132   JavaThread* the_owner = NULL;
4133   {
4134     MutexLockerEx ml(doLock ? Threads_lock : NULL);
4135     ALL_JAVA_THREADS(q) {
4136       if (q->is_lock_owned(owner)) {
4137         the_owner = q;
4138         break;
4139       }
4140     }
4141   }
4142   assert(the_owner != NULL, "Did not find owning Java thread for lock word addre
4143   return the_owner;
4144 }

4146 // Threads::print_on() is called at safepoint by VM_PrintThreads operation.
4147 void Threads::print_on(outputStream* st, bool print_stacks, bool internal_format
4148   char buf[32];
4149   st->print_cr(os::local_time_string(buf, sizeof(buf)));

4151   st->print_cr("Full thread dump %s (%s %s):",
4152                Abstract_VM_Version::vm_name(),
4153                Abstract_VM_Version::vm_release(),
4154                Abstract_VM_Version::vm_info_string()
4155               );
4156   st->cr();

4158 #ifndef SERIALGC
4159   // Dump concurrent locks
4160   ConcurrentLocksDump concurrent_locks;
```

```
4161   if (print_concurrent_locks) {
4162     concurrent_locks.dump_at_safepoint();
4163   }
4164 #endif // SERIALGC

4166   ALL_JAVA_THREADS(p) {
4167     ResourceMark rm;
4168     p->print_on(st);
4169     if (print_stacks) {
4170       if (internal_format) {
4171         p->trace_stack();
4172       } else {
4173         p->print_stack_on(st);
4174       }
4175     }
4176     st->cr();
4177 #ifndef SERIALGC
4178     if (print_concurrent_locks) {
4179       concurrent_locks.print_locks_on(p, st);
4180     }
4181 #endif // SERIALGC
4182   }

4184   VMThread::vm_thread()->print_on(st);
4185   st->cr();
4186   Universe::heap()->print_gc_threads_on(st);
4187   WatcherThread* wt = WatcherThread::watcher_thread();
4188   if (wt != NULL) wt->print_on(st);
4189   st->cr();
4190   CompileBroker::print_compiler_threads_on(st);
4191   st->flush();
4192 }

4194 // Threads::print_on_error() is called by fatal error handler. It's possible
4195 // that VM is not at safepoint and/or current thread is inside signal handler.
4196 // Don't print stack trace, as the stack may not be walkable. Don't allocate
4197 // memory (even in resource area), it might deadlock the error handler.
4198 void Threads::print_on_error(outputStream* st, Thread* current, char* buf, int b
4199   bool found_current = false;
4200   st->print_cr("Java Threads: ( => current thread )");
4201   ALL_JAVA_THREADS(thread) {
4202     bool is_current = (current == thread);
4203     found_current = found_current || is_current;

4205     st->print("%s", is_current ? "=>" : "  ");

4207     st->print(PTR_FORMAT, thread);
4208     st->print(" ");
4209     thread->print_on_error(st, buf, buflen);
4210     st->cr();
4211   }
4212   st->cr();

4214   st->print_cr("Other Threads:");
4215   if (VMThread::vm_thread()) {
4216     bool is_current = (current == VMThread::vm_thread());
4217     found_current = found_current || is_current;
4218     st->print("%s", current == VMThread::vm_thread() ? "=>" : "  ");

4220     st->print(PTR_FORMAT, VMThread::vm_thread());
4221     st->print(" ");
4222     VMThread::vm_thread()->print_on_error(st, buf, buflen);
4223     st->cr();
4224   }
4225   WatcherThread* wt = WatcherThread::watcher_thread();
4226   if (wt != NULL) {
```



```
4227     bool is_current = (current == wt);
4228     found_current = found_current || is_current;
4229     st->print("%s", is_current ? "=>" : "  ");

4231     st->print(PTR_FORMAT, wt);
4232     st->print(" ");
4233     wt->print_on_error(st, buf, buflen);
4234     st->cr();
4235   }
4236   if (!found_current) {
4237     st->cr();
4238     st->print("=>" PTR_FORMAT " (exited) ", current);
4239     current->print_on_error(st, buf, buflen);
4240     st->cr();
4241   }
4242 }

4244 // Internal SpinLock and Mutex
4245 // Based on ParkEvent

4247 // Ad-hoc mutual exclusion primitives: SpinLock and Mux
4248 //
4249 // We employ SpinLocks _only for low-contention, fixed-length
4250 // short-duration critical sections where we're concerned
4251 // about native mutex_t or HotSpot Mutex:: latency.
4252 // The mux construct provides a spin-then-block mutual exclusion
4253 // mechanism.
4254 //
4255 // Testing has shown that contention on the ListLock guarding gFreeList
4256 // is common.  If we implement ListLock as a simple SpinLock it's common
4257 // for the JVM to devolve to yielding with little progress.  This is true
4258 // despite the fact that the critical sections protected by ListLock are
4259 // extremely short.
4260 //
4261 // TODO-FIXME: ListLock should be of type SpinLock.
4262 // We should make this a 1st-class type, integrated into the lock
4263 // hierarchy as leaf-locks.  Critically, the SpinLock structure
4264 // should have sufficient padding to avoid false-sharing and excessive
4265 // cache-coherency traffic.


4268 typedef volatile int SpinLockT ;

4270 void Thread::SpinAcquire (volatile int * adr, const char * LockName) {
4271   if (Atomic::cmpxchg (1, adr, 0) == 0) {
4272      return ;   // normal fast-path return
4273   }

4275   // Slow-path : We've encountered contention -- Spin/Yield/Block strategy.
4276   TEVENT (SpinAcquire - ctx) ;
4277   int ctr = 0 ;
4278   int Yields = 0 ;
4279   for (;;) {
4280      while (*adr != 0) {
4281         ++ctr ;
4282         if ((ctr & 0xFFF) == 0 || !os::is_MP()) {
4283            if (Yields > 5) {
4284              // Consider using a simple NakedSleep() instead.
4285              // Then SpinAcquire could be called by non-JVM threads
4286              Thread::current()->_ParkEvent->park(1) ;
4287            } else {
4288              os::NakedYield() ;
4289              ++Yields ;
4290            }
4291         } else {
4292            SpinPause() ;
```

```
4293         }
4294       }
4295       if (Atomic::cmpxchg (1, adr, 0) == 0) return ;
4296     }
4297 }

4299 void Thread::SpinRelease (volatile int * adr) {
4300   assert (*adr != 0, "invariant") ;
4301   OrderAccess::fence() ;      // guarantee at least release consistency.
4302   // Roach-motel semantics.
4303   // It's safe if subsequent LDs and STs float "up" into the critical section,
4304   // but prior LDs and STs within the critical section can't be allowed
4305   // to reorder or float past the ST that releases the lock.
4306   *adr = 0 ;
4307 }

4309 // muxAcquire and muxRelease:
4310 //
4311 // *  muxAcquire and muxRelease support a single-word lock-word construct.
4312 //    The LSB of the word is set IFF the lock is held.
4313 //    The remainder of the word points to the head of a singly-linked list
4314 //    of threads blocked on the lock.
4315 //
4316 // *  The current implementation of muxAcquire-muxRelease uses its own
4317 //    dedicated Thread._MuxEvent instance.  If we're interested in
4318 //    minimizing the peak number of extant ParkEvent instances then
4319 //    we could eliminate _MuxEvent and "borrow" _ParkEvent as long
4320 //    as certain invariants were satisfied.  Specifically, care would need
4321 //    to be taken with regards to consuming unpark() "permits".
4322 //    A safe rule of thumb is that a thread would never call muxAcquire()
4323 //    if it's enqueued (cxq, EntryList, WaitList, etc) and will subsequently
4324 //    park().  Otherwise the _ParkEvent park() operation in muxAcquire() could
4325 //    consume an unpark() permit intended for monitorenter, for instance.
4326 //    One way around this would be to widen the restricted-range semaphore
4327 //    implemented in park().  Another alternative would be to provide
4328 //    multiple instances of the PlatformEvent() for each thread.  One
4329 //    instance would be dedicated to muxAcquire-muxRelease, for instance.
4330 //
4331 // *  Usage:
4332 //    -- Only as leaf locks
4333 //    -- for short-term locking only as muxAcquire does not perform
4334 //       thread state transitions.
4335 //
4336 // Alternatives:
4337 // *  We could implement muxAcquire and muxRelease with MCS or CLH locks
4338 //    but with parking or spin-then-park instead of pure spinning.
4339 // *  Use Taura-Oyama-Yonenzawa locks.
4340 // *  It's possible to construct a 1-0 lock if we encode the lockword as
4341 //    (List,LockByte).  Acquire will CAS the full lockword while Release
4342 //    will STB 0 into the LockByte.  The 1-0 scheme admits stranding, so
4343 //    acquiring threads use timers (ParkTimed) to detect and recover from
4344 //    the stranding window.  Thread/Node structures must be aligned on 256-byte
4345 //    boundaries by using placement-new.
4346 // *  Augment MCS with advisory back-link fields maintained with CAS().
4347 //    Pictorially:  LockWord -> T1 <-> T2 <-> T3 <-> ... <-> Tn <-> Owner.
4348 //    The validity of the backlinks must be ratified before we trust the value.
4349 //    If the backlinks are invalid the exiting thread must back-track through th
4350 //    the forward links, which are always trustworthy.
4351 // *  Add a successor indication.  The LockWord is currently encoded as
4352 //    (List, LOCKBIT:1).  We could also add a SUCCBIT or an explicit _succ varia
4353 //    to provide the usual futile-wakeup optimization.
4354 //    See RTStt for details.
4355 // *  Consider schedctl.sc_nopreempt to cover the critical section.
4356 //
```



```
4359 typedef volatile intptr_t MutexT ;      // Mux Lock-word
4360 enum MuxBits { LOCKBIT = 1 } ;

4362 void Thread::muxAcquire (volatile intptr_t * Lock, const char * LockName) {
4363   intptr_t w = Atomic::cmpxchg_ptr (LOCKBIT, Lock, 0) ;
4364   if (w == 0) return ;
4365   if ((w & LOCKBIT) == 0 && Atomic::cmpxchg_ptr (w|LOCKBIT, Lock, w) == w) {
4366     return ;
4367   }

4369   TEVENT (muxAcquire - Contention) ;
4370   ParkEvent * const Self = Thread::current()->_MuxEvent ;
4371   assert ((intptr_t(Self) & LOCKBIT) == 0, "invariant") ;
4372   for (;;) {
4373     int its = (os::is_MP() ? 100 : 0) + 1 ;

4375     // Optional spin phase: spin-then-park strategy
4376     while (--its >= 0) {
4377       w = *Lock ;
4378       if ((w & LOCKBIT) == 0 && Atomic::cmpxchg_ptr (w|LOCKBIT, Lock, w) == w)
4379         return ;
4380       }
4381     }

4383     Self->reset() ;
4384     Self->OnList = intptr_t(Lock) ;
4385     // The following fence() isn't _strictly necessary as the subsequent
4386     // CAS() both serializes execution and ratifies the fetched *Lock value.
4387     OrderAccess::fence();
4388     for (;;) {
4389        w = *Lock ;
4390        if ((w & LOCKBIT) == 0) {
4391            if (Atomic::cmpxchg_ptr (w|LOCKBIT, Lock, w) == w) {
4392                Self->OnList = 0 ;   // hygiene - allows stronger asserts
4393                return ;
4394            }
4395            continue ;      // Interference -- *Lock changed -- Just retry
4396        }
4397        assert (w & LOCKBIT, "invariant") ;
4398        Self->ListNext = (ParkEvent *) (w & ~LOCKBIT );
4399        if (Atomic::cmpxchg_ptr (intptr_t(Self)|LOCKBIT, Lock, w) == w) break ;
4400     }

4402     while (Self->OnList != 0) {
4403       Self->park() ;
4404     }
4405   }
4406 }

4408 void Thread::muxAcquireW (volatile intptr_t * Lock, ParkEvent * ev) {
4409   intptr_t w = Atomic::cmpxchg_ptr (LOCKBIT, Lock, 0) ;
4410   if (w == 0) return ;
4411   if ((w & LOCKBIT) == 0 && Atomic::cmpxchg_ptr (w|LOCKBIT, Lock, w) == w) {
4412     return ;
4413   }

4415   TEVENT (muxAcquire - Contention) ;
4416   ParkEvent * ReleaseAfter = NULL ;
4417   if (ev == NULL) {
4418     ev = ReleaseAfter = ParkEvent::Allocate (NULL) ;
4419   }
4420   assert ((intptr_t(ev) & LOCKBIT) == 0, "invariant") ;
4421   for (;;) {
4422     guarantee (ev->OnList == 0, "invariant") ;
4423     int its = (os::is_MP() ? 100 : 0) + 1 ;
```

```
4425     // Optional spin phase: spin-then-park strategy
4426     while (--its >= 0) {
4427       w = *Lock ;
4428       if ((w & LOCKBIT) == 0 && Atomic::cmpxchg_ptr (w|LOCKBIT, Lock, w) == w) {
4429         if (ReleaseAfter != NULL) {
4430           ParkEvent::Release (ReleaseAfter) ;
4431         }
4432         return ;
4433       }
4434     }
4435
4436     ev->reset() ;
4437     ev->OnList = intptr_t(Lock) ;
4438     // The following fence() isn't _strictly necessary as the subsequent
4439     // CAS() both serializes execution and ratifies the fetched *Lock value.
4440     OrderAccess::fence();
4441     for (;;) {
4442       w = *Lock ;
4443       if ((w & LOCKBIT) == 0) {
4444         if (Atomic::cmpxchg_ptr (w|LOCKBIT, Lock, w) == w) {
4445           ev->OnList = 0 ;
4446           // We call ::Release while holding the outer lock, thus
4447           // artificially lengthening the critical section.
4448           // Consider deferring the ::Release() until the subsequent unlock(),
4449           // after we've dropped the outer lock.
4450           if (ReleaseAfter != NULL) {
4451             ParkEvent::Release (ReleaseAfter) ;
4452           }
4453           return ;
4454         }
4455         continue ;      // Interference -- *Lock changed -- Just retry
4456       }
4457       assert (w & LOCKBIT, "invariant") ;
4458       ev->ListNext = (ParkEvent *) (w & ~LOCKBIT );
4459       if (Atomic::cmpxchg_ptr (intptr_t(ev)|LOCKBIT, Lock, w) == w) break ;
4460     }
4461
4462     while (ev->OnList != 0) {
4463       ev->park() ;
4464     }
4465   }
4466 }
4467
4468 // Release() must extract a successor from the list and then wake that thread.
4469 // It can "pop" the front of the list or use a detach-modify-reattach (DMR) sche
4470 // similar to that used by ParkEvent::Allocate() and ::Release().  DMR-based
4471 // Release() would :
4472 // (A) CAS() or swap() null to *Lock, releasing the lock and detaching the list.
4473 // (B) Extract a successor from the private list "in-hand"
4474 // (C) attempt to CAS() the residual back into *Lock over null.
4475 //     If there were any newly arrived threads and the CAS() would fail.
4476 //     In that case Release() would detach the RATs, re-merge the list in-hand
4477 //     with the RATs and repeat as needed.  Alternately, Release() might
4478 //     detach and extract a successor, but then pass the residual list to the wa
4479 //     The wakee would be responsible for reattaching and remerging before it
4480 //     competed for the lock.
4481 //
4482 // Both "pop" and DMR are immune from ABA corruption -- there can be
4483 // multiple concurrent pushers, but only one popper or detacher.
4484 // This implementation pops from the head of the list.  This is unfair,
4485 // but tends to provide excellent throughput as hot threads remain hot.
4486 // (We wake recently run threads first).
4487
4488 void Thread::muxRelease (volatile intptr_t * Lock)  {
4489   for (;;) {
4490     const intptr_t w = Atomic::cmpxchg_ptr (0, Lock, LOCKBIT) ;
```



```
4491     assert (w & LOCKBIT, "invariant") ;
4492     if (w == LOCKBIT) return ;
4493     ParkEvent * List = (ParkEvent *) (w & ~LOCKBIT) ;
4494     assert (List != NULL, "invariant") ;
4495     assert (List->OnList == intptr_t(Lock), "invariant") ;
4496     ParkEvent * nxt = List->ListNext ;
4497
4498     // The following CAS() releases the lock and pops the head element.
4499     if (Atomic::cmpxchg_ptr (intptr_t(nxt), Lock, w) != w) {
4500       continue ;
4501     }
4502     List->OnList = 0 ;
4503     OrderAccess::fence() ;
4504     List->unpark () ;
4505     return ;
4506   }
4507 }
4508
4509
4510 void Threads::verify() {
4511   ALL_JAVA_THREADS(p) {
4512     p->verify();
4513   }
4514   VMThread* thread = VMThread::vm_thread();
4515   if (thread != NULL) thread->verify();
4516 }
```

****
33531 Thu Jun 14 10:53:43 2012
new/src/share/vm/utilities/vmError.cpp
****
_____unchanged_portion_omitted_

```
 303 // This is the main function to report a fatal error. Only one thread can
 304 // call this function, so we don't need to worry about MT-safety. But it's
 305 // possible that the error handler itself may crash or die on an internal
 306 // error, for example, when the stack/heap is badly damaged. We must be
 307 // able to handle recursive errors that happen inside error handler.
 308 //
 309 // Error reporting is done in several steps. If a crash or internal error
 310 // occurred when reporting an error, the nested signal/exception handler
 311 // can skip steps that are already (or partially) done. Error reporting will
 312 // continue from the next step. This allows us to retrieve and print
 313 // information that may be unsafe to get after a fatal error. If it happens,
 314 // you may find nested report_and_die() frames when you look at the stack
 315 // in a debugger.
 316 //
 317 // In general, a hang in error handler is much worse than a crash or internal
 318 // error, as it's harder to recover from a hang. Deadlock can happen if we
 319 // try to grab a lock that is already owned by current thread, or if the
 320 // owner is blocked forever (e.g. in os::infinite_sleep()). If possible, the
 321 // error handler and all the functions it called should avoid grabbing any
 322 // lock. An important thing to notice is that memory allocation needs a lock.
 323 //
 324 // We should avoid using large stack allocated buffers. Many errors happen
 325 // when stack space is already low. Making things even worse is that there
 326 // could be nested report_and_die() calls on stack (see above). Only one
 327 // thread can report error, so large buffers are statically allocated in data
 328 // segment.

 330 void VMError::report(outputStream* st) {
 331 # define BEGIN if (_current_step == 0) { _current_step = 1;
 332 # define STEP(n, s) } if (_current_step < n) { _current_step = n; _current_step_
 333 # define END }

 335   // don't allocate large buffer on stack
 336   static char buf[O_BUFLEN];

 338   BEGIN

 340   STEP(10, "(printing fatal error message)")

 342     st->print_cr("#");
 343     if (should_report_bug(_id)) {
 344       st->print_cr("# A fatal error has been detected by the Java Runtime Enviro
 345     } else {
 346       st->print_cr("# There is insufficient memory for the Java "
 347                    "Runtime Environment to continue.");
 348     }

 350   STEP(15, "(printing type of error)")

 352      switch(_id) {
 353        case oom_error:
 354          if (_size) {
 355            st->print("# Native memory allocation (malloc) failed to allocate ");
 356            jio_snprintf(buf, sizeof(buf), SIZE_FORMAT, _size);
 357            st->print(buf);
 358            st->print(" bytes");
 359            if (_message != NULL) {
 360              st->print(" for ");
 361              st->print(_message);
 362            }
```



```
 363            st->cr();
 364          } else {
 365            if (_message != NULL)
 366              st->print("# ");
 367              st->print_cr(_message);
 368          }
 369          // In error file give some solutions
 370          if (_verbose) {
 371            st->print_cr("# Possible reasons:");
 372            st->print_cr("#   The system is out of physical RAM or swap space");
 373            st->print_cr("#   In 32 bit mode, the process size limit was hit");
 374            st->print_cr("# Possible solutions:");
 375            st->print_cr("#   Reduce memory load on the system");
 376            st->print_cr("#   Increase physical memory or swap space");
 377            st->print_cr("#   Check if swap backing store is full");
 378            st->print_cr("#   Use 64 bit Java on a 64 bit OS");
 379            st->print_cr("#   Decrease Java heap size (-Xmx/-Xms)");
 380            st->print_cr("#   Decrease number of Java threads");
 381            st->print_cr("#   Decrease Java thread stack sizes (-Xss)");
 382            st->print_cr("#   Set larger code cache with -XX:ReservedCodeCacheSiz
 383            st->print_cr("# This output file may be truncated or incomplete.");
 384          } else {
 385            return;  // that's enough for the screen
 386          }
 387          break;
 388        case internal_error:
 389        default:
 390          break;
 391      }

 393   STEP(20, "(printing exception/signal name)")

 395      st->print_cr("#");
 396      st->print("#  ");
 397      // Is it an OS exception/signal?
 398      if (os::exception_name(_id, buf, sizeof(buf))) {
 399        st->print("%s", buf);
 400        st->print(" (0x%x)", _id);                // signal number
 401        st->print(" at pc=" PTR_FORMAT, _pc);
 402      } else {
 403        if (should_report_bug(_id)) {
 404          st->print("Internal Error");
 405        } else {
 406          st->print("Out of Memory Error");
 407        }
 408        if (_filename != NULL && _lineno > 0) {
 409 #ifdef PRODUCT
 410          // In product mode chop off pathname?
 411          char separator = os::file_separator()[0];
 412          const char *p = strrchr(_filename, separator);
 413          const char *file = p ? p+1 : _filename;
 414 #else
 415          const char *file = _filename;
 416 #endif
 417          size_t len = strlen(file);
 418          size_t buflen = sizeof(buf);

 420          strncpy(buf, file, buflen);
 421          if (len + 10 < buflen) {
 422            sprintf(buf + len, ":%d", _lineno);
 423          }
 424          st->print(" (%s)", buf);
 425        } else {
 426          st->print(" (0x%x)", _id);
 427        }
 428      }
```

```
430   STEP(30, "(printing current thread and pid)")

432      // process id, thread id
433      st->print(", pid=%d", os::current_process_id());
434      st->print(", tid=" UINTX_FORMAT, os::current_thread_id());
435      st->cr();

437   STEP(40, "(printing error message)")

439      if (should_report_bug(_id)) {  // already printed the message.
440        // error message
441        if (_detail_msg) {
442          st->print_cr("#  %s: %s", _message ? _message : "Error", _detail_msg);
443        } else if (_message) {
444          st->print_cr("#  Error: %s", _message);
445        }
446     }

448   STEP(50, "(printing Java version string)")

450      // VM version
451      st->print_cr("#");
452      JDK_Version::current().to_string(buf, sizeof(buf));
453      const char* runtime_name = JDK_Version::runtime_name() != NULL ?
454                                 JDK_Version::runtime_name() : "";
455      st->print_cr("# JRE version: %s (%s)", runtime_name, buf);
456      st->print_cr("# Java VM    : %s (%s %s %s %s)",
453      st->print_cr("# JRE version: %s", buf);
454      st->print_cr("# Java VM: %s (%s %s %s %s)",
457                   Abstract_VM_Version::vm_name(),
458                   Abstract_VM_Version::vm_release(),
459                   Abstract_VM_Version::vm_info_string(),
460                   Abstract_VM_Version::vm_platform_string(),
461                   UseCompressedOops ? "compressed oops" : ""
462                 );

464   STEP(60, "(printing problematic frame)")

466      // Print current frame if we have a context (i.e. it's a crash)
467      if (_context) {
468        st->print_cr("# Problematic frame:");
469        st->print("# ");
470        frame fr = os::fetch_frame_from_context(_context);
471        fr.print_on_error(st, buf, sizeof(buf));
472        st->cr();
473        st->print_cr("#");
474      }
475   STEP(63, "(printing core file information)")
476     st->print("# ");
477     if (coredump_status) {
478       st->print("Core dump written. Default location: %s", coredump_message);
479     } else {
480       st->print("Failed to write core dump. %s", coredump_message);
481     }
482     st->print_cr("");
483     st->print_cr("#");

485   STEP(65, "(printing bug submit message)")

487      if (should_report_bug(_id) && _verbose) {
488        print_bug_submit_message(st, _thread);
489      }

491   STEP(70, "(printing thread)" )
```



```
493      if (_verbose) {
494        st->cr();
495        st->print_cr("---------------  T H R E A D  ---------------");
496        st->cr();
497      }

499   STEP(80, "(printing current thread)" )

501      // current thread
502      if (_verbose) {
503        if (_thread) {
504          st->print("Current thread (" PTR_FORMAT "):  ", _thread);
505          _thread->print_on_error(st, buf, sizeof(buf));
506          st->cr();
507        } else {
508          st->print_cr("Current thread is native thread");
509        }
510        st->cr();
511      }

513   STEP(90, "(printing siginfo)" )

515      // signal no, signal code, address that caused the fault
516      if (_verbose && _siginfo) {
517        os::print_siginfo(st, _siginfo);
518        st->cr();
519      }

521   STEP(100, "(printing registers, top of stack, instructions near pc)")

523      // registers, top of stack, instructions near pc
524      if (_verbose && _context) {
525        os::print_context(st, _context);
526        st->cr();
527      }

529   STEP(105, "(printing register info)")

531      // decode register contents if possible
532      if (_verbose && _context && Universe::is_fully_initialized()) {
533        os::print_register_info(st, _context);
534        st->cr();
535      }

537   STEP(110, "(printing stack bounds)" )

539      if (_verbose) {
540        st->print("Stack: ");

542        address stack_top;
543        size_t stack_size;

545        if (_thread) {
546           stack_top = _thread->stack_base();
547           stack_size = _thread->stack_size();
548        } else {
549           stack_top = os::current_stack_base();
550           stack_size = os::current_stack_size();
551        }

553        address stack_bottom = stack_top - stack_size;
554        st->print("[" PTR_FORMAT "," PTR_FORMAT "]", stack_bottom, stack_top);

556        frame fr = _context ? os::fetch_frame_from_context(_context)
557                            : os::current_frame();
```

```
559       if (fr.sp()) {
560         st->print(",  sp=" PTR_FORMAT, fr.sp());
561         size_t free_stack_size = pointer_delta(fr.sp(), stack_bottom, 1024);
562         st->print(",  free space=" SIZE_FORMAT "k", free_stack_size);
563       }
564
565       st->cr();
566     }
567
568   STEP(120, "(printing native stack)" )
569
570     if (_verbose) {
571       frame fr = _context ? os::fetch_frame_from_context(_context)
572                           : os::current_frame();
573
574       // see if it's a valid frame
575       if (fr.pc()) {
576          st->print_cr("Native frames: (J=compiled Java code, j=interpreted, Vv=
577
578
579          int count = 0;
580          while (count++ < StackPrintLimit) {
581             fr.print_on_error(st, buf, sizeof(buf));
582             st->cr();
583             if (os::is_first_C_frame(&fr)) break;
584             fr = os::get_sender_for_C_frame(&fr);
585          }
586
587          if (count > StackPrintLimit) {
588             st->print_cr("...<more frames>...");
589          }
590
591          st->cr();
592       }
593     }
594
595   STEP(130, "(printing Java stack)" )
596
597     if (_verbose && _thread && _thread->is_Java_thread()) {
598       print_stack_trace(st, (JavaThread*)_thread, buf, sizeof(buf));
599     }
600
601   STEP(135, "(printing target Java thread stack)" )
602
603     // printing Java thread stack trace if it is involved in GC crash
604     if (_verbose && _thread && (_thread->is_Named_thread())) {
605       JavaThread*  jt = ((NamedThread *)_thread)->processed_thread();
606       if (jt != NULL) {
607         st->print_cr("JavaThread " PTR_FORMAT " (nid = " UINTX_FORMAT ") was be
608         print_stack_trace(st, jt, buf, sizeof(buf), true);
609       }
610     }
611
612   STEP(140, "(printing VM operation)" )
613
614     if (_verbose && _thread && _thread->is_VM_thread()) {
615        VMThread* t = (VMThread*)_thread;
616        VM_Operation* op = t->vm_operation();
617        if (op) {
618          op->print_on_error(st);
619          st->cr();
620          st->cr();
621        }
622     }
623
624   STEP(150, "(printing current compile task)" )
```



```
625
626     if (_verbose && _thread && _thread->is_Compiler_thread()) {
627        CompilerThread* t = (CompilerThread*)_thread;
628        if (t->task()) {
629           st->cr();
630           st->print_cr("Current CompileTask:");
631           t->task()->print_line_on_error(st, buf, sizeof(buf));
632           st->cr();
633        }
634     }
635
636   STEP(160, "(printing process)" )
637
638     if (_verbose) {
639       st->cr();
640       st->print_cr("---------------  P R O C E S S  ---------------");
641       st->cr();
642     }
643
644   STEP(170, "(printing all threads)" )
645
646     // all threads
647     if (_verbose && _thread) {
648       Threads::print_on_error(st, _thread, buf, sizeof(buf));
649       st->cr();
650     }
651
652   STEP(175, "(printing VM state)" )
653
654     if (_verbose) {
655       // Safepoint state
656       st->print("VM state:");
657
658       if (SafepointSynchronize::is_synchronizing()) st->print("synchronizing");
659       else if (SafepointSynchronize::is_at_safepoint()) st->print("at safepoint
660       else st->print("not at safepoint");
661
662       // Also see if error occurred during initialization or shutdown
663       if (!Universe::is_fully_initialized()) {
664         st->print(" (not fully initialized)");
665       } else if (VM_Exit::vm_exited()) {
666         st->print(" (shutting down)");
667       } else {
668         st->print(" (normal execution)");
669       }
670       st->cr();
671       st->cr();
672     }
673
674   STEP(180, "(printing owned locks on error)" )
675
676     // mutexes/monitors that currently have an owner
677     if (_verbose) {
678       print_owned_locks_on_error(st);
679       st->cr();
680     }
681
682   STEP(190, "(printing heap information)" )
683
684     if (_verbose && Universe::is_fully_initialized()) {
685       // Print heap information before vm abort. As we'd like as much
686       // information as possible in the report we ask for the
687       // extended (i.e., more detailed) version.
688       Universe::print_on(st, true /* extended */);
689       st->cr();
```

```
691         Universe::heap()->barrier_set()->print_on(st);
692         st->cr();
693
694         st->print_cr("Polling page: " INTPTR_FORMAT, os::get_polling_page());
695         st->cr();
696       }
697
698     STEP(195, "(printing code cache information)" )
699
700       if (_verbose && Universe::is_fully_initialized()) {
701         // print code cache information before vm abort
702         CodeCache::print_bounds(st);
703         st->cr();
704       }
705
706     STEP(200, "(printing ring buffers)" )
707
708       if (_verbose) {
709         Events::print_all(st);
710         st->cr();
711       }
712
713     STEP(205, "(printing dynamic libraries)" )
714
715       if (_verbose) {
716         // dynamic libraries, or memory map
717         os::print_dll_info(st);
718         st->cr();
719       }
720
721     STEP(210, "(printing VM options)" )
722
723       if (_verbose) {
724         // VM options
725         Arguments::print_on(st);
726         st->cr();
727       }
728
729     STEP(215, "(printing warning if internal testing API used)" )
730
731       if (WhiteBox::used()) {
732         st->print_cr("Unsupported internal testing APIs have been used.");
733         st->cr();
734       }
735
736     STEP(220, "(printing environment variables)" )
737
738       if (_verbose) {
739         os::print_environment_variables(st, env_list, buf, sizeof(buf));
740         st->cr();
741       }
742
743     STEP(225, "(printing signal handlers)" )
744
745       if (_verbose) {
746         os::print_signal_handlers(st, buf, sizeof(buf));
747         st->cr();
748       }
749
750     STEP(230, "" )
751
752       if (_verbose) {
753         st->cr();
754         st->print_cr("---------------  S Y S T E M  ---------------");
755         st->cr();
756       }
```



```
757
758     STEP(240, "(printing OS information)" )
759
760       if (_verbose) {
761         os::print_os_info(st);
762         st->cr();
763       }
764
765     STEP(250, "(printing CPU info)" )
766       if (_verbose) {
767         os::print_cpu_info(st);
768         st->cr();
769       }
770
771     STEP(260, "(printing memory info)" )
772
773       if (_verbose) {
774         os::print_memory_info(st);
775         st->cr();
776       }
777
778     STEP(270, "(printing internal vm info)" )
779
780       if (_verbose) {
781         st->print_cr("vm_info: %s", Abstract_VM_Version::internal_vm_info_string(
782         st->cr();
783       }
784
785     STEP(280, "(printing date and time)" )
786
787       if (_verbose) {
788         os::print_date_and_time(st);
789         st->cr();
790       }
791
792     END
793
794 # undef BEGIN
795 # undef STEP
796 # undef END
797 }
```

_____***unchanged_portion_omitted_***