SMR and JavaThread Lifecycle

This wiki is about Safe Memory Reclamation (SMR) and how the Hazard Pointer version of that technique can be applied to the JavaThread lifecycle.

THIS IS A WORK IN PROGRESS!

See "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects" for the IEEE paper.

Here is the bug we're using to track this work: JDK-8167108 inconsistent handling of SR_lock can lead to crashes

Here is the current webrevs:

open repo: http://cr.openjdk.java.net/~dcubed/8167108-webrev/jdk10-04-full

Table of Contents

- Terminology
- Section 1 - Typical Mostly Lock-Free List
    - Delete-Part-1
    - Delete-Part-2
    - Delete-Part-3
    - Walk&Read
    - Overlapping Delete-Part-2 and Walk-&-Read
    - Overlapping Delete-Part-3 and Walk-&-Read
    - Conclusions About Mostly Lock-Free List
- Section 2 - Lock-Free List with SMR
    - Overlapping Delete-Part-2 and Walk-&-Read
    - Overlapping Delete-Part-3 and Walk-&-Read
    - Conclusions About Lock-Free List with SMR
- Section 3 - JavaThread Examples
    - jobject Parameter Converted to JavaThread
    - jthread Parameter Converted to JavaThread
- Section 4 - 3 Critical Operations in New JavaThread Code
    - Registering Interest In and Validating a JavaThread Reference
    - Safely Deleting a JavaThread Reference
    - Is It Safe to Delete This JavaThread Reference
    - For Completeness, Unregistering Interest In a JavaThread Reference
- Section 5 - How Do You Use This Stuff?
    - ThreadsListHandle Helper Object
    - cv_internal_thread_to_JavaThread() service function/method
    - cv_external_thread_to_JavaThread() service function/method
    - cv_oop_to_JavaThread() service function/method
- Section 6 - Examples Uses
    - jobject Parameter Converted to JavaThread Redux
    - jthread Parameter Converted to JavaThread Redux
    - Walking a ThreadsList
- Section 7 - What the Rest of This Code Does
- Section 8 - Testing That's Been Done So Far
- Section 9 - How Do You Debug This Code?
    - -Xlog Support
    - Thread Dump Support
    - hs_err_pid Support
- Section 10 - Gory Details
    - Behind the Threads::_smr_java_thread_list Abstraction
    - Behind the Thread::_threads_hazard_ptr Abstraction
    - Threads::acquire_stable_list_fast_path() is Lock Free
        - Acquire-Add or Acquire-Remove Race
        - Acquire-Scan Race
    - Fast JavaThreads List
    - Memory Management With SMR
    - Double-Check Locking is Evil, But...
    - ThreadsListSetter - A Special Helper
        - ThreadsListSetter Example That Always Sets
        - ThreadsListSetter Example That Sometimes Sets
    - NestedThreadsList - Another Special Helper
        - Nesting Should Be Rare
        - Nesting Is Mostly Hidden
    - Lock Free Algorithms Are Difficult to Get Right!
        - The Crashing Thread
        - The JVM_Interrupt Thread
        - The Threads::smr_free_list() Thread

- Analysis Round 1
- Analysis Round 2
- Section 11 - Performance Testing Ideas
  - nsk/monitoring/stress/thread/cmon001 Might Provide Interesting Data
  - Revisit Threads::is_a_protected_JavaThread()
- Section 12 - Remaining Work

Terminology

"Threads list"	Refers to the list of JavaThreads in the Threads class; from the Threads class point of view, there is only one list of JavaThreads.
"ThreadsList"	Refers to the new fast JavaThreads list class; there can be many active ThreadsList objects in the system, each of which contains a specific list of JavaThreads. However, there is only one ThreadsList that matches the current list of JavaThreads in the Threads class.
_smr_java_thread_list	A field in the Threads class that points to the ThreadsList that matches the current list of JavaThreads in the Threads class; the other active ThreadsList objects in the system represent different lists of protected JavaThreads. The same JavaThread may appear on more than one ThreadsList due to normal Threads::add() and Threads::delete() calls.
stable hazard ptr	A hazard ptr that has been successfully published at one point in time. A stable hazard pointer does not have a tag. An untagged hazard ptr that has never been published is not considered a stable hazard ptr.
tagged hazard ptr	A hazard ptr with a tag that marks the hazard ptr as not being a stable hazard ptr.
unverified hazard ptr	A hazard ptr with a tag that marks the hazard ptr as not being a stable hazard ptr or hazard ptr that has just been untagged, but has not yet been published.

It is important to understand exactly what a stable hazard ptr is so this next table tries to drive that point home:

_smr_java_thread_list	not a stable hazard ptr because it is not used to protect its JavaThreads
local variable copy of _smr_java_thread_list	not a stable hazard ptr because it has not been published
tagged hazard ptr	not a stable hazard ptr because it is tagged
untagged hazard ptr in a local variable	not a stable hazard ptr because it has not been published
untagged hazard ptr in _threads_hazard_ptr	a stable hazard ptr because it has been published

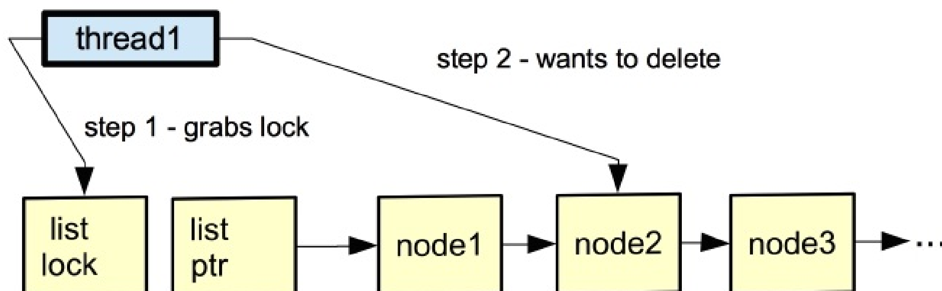
Section 1 - Typical Mostly Lock-Free List

Let me be upfront about this: I hate the concept of lock-free data structures. When you have lock-free anything, you have to spend a non-trivial amount of time and energy to analyze and reason about the algorithms that are used and whether they are safe or not.

Let's start off with a few diagrams to show example operations of a typical, mostly lock-free list:

Delete-Part-1

In "Fig. 1a - Delete-Part-1", we assume a lot of basic knowlege about lists and just dive right in:



- "list lock" is the lock that is held for adding and deleting nodes
- "list ptr" points to the first node in the list
- "node[1-3...]" is our list of nodes
- "thread1" is executing the code that wants to delete "node2" from the list

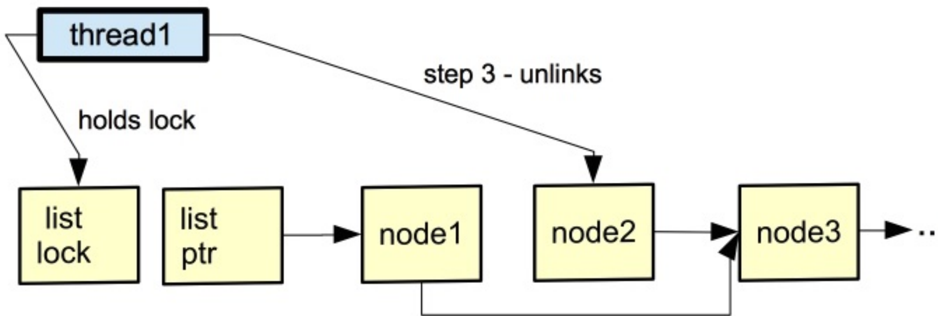
Pretty typical stuff:

- 1) grab the lock
- 2) find the node to be deleted

Note: We could find the node to be deleted before grabbing the lock, but then we would have to revalidate our assumptions after grabbing the lock. Let's not go there...

Delete-Part-2

In "Fig. 1b - Delete-Part-2", we show the housekeeping:



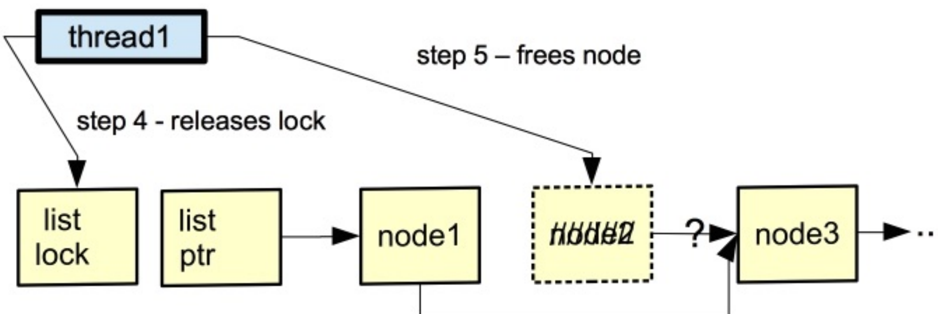
Still pretty typical stuff:

- "thread1" holds the "list lock"
- 3) "node2" is unlinked

Note: "node2" still points to "node3".

Delete-Part-3

In "Fig. 1c - Delete-Part-3", we show the cleanup:



Mostly pretty typical stuff:

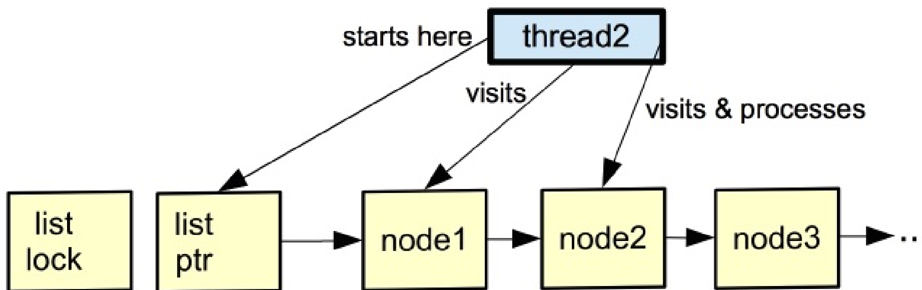
- 4) "thread1" releases the "list lock"
- 5) "node2" is freed

Note: "node2" may still point to "node3" or the pointer might have been overwritten with a debugging pattern.

Note: If "thread1" held "list lock" over the "free" operation, then contention for the malloc lock might artificially extend the hold time on the "list lock".

Walk&Read

In "Fig. 2 - Walk&Read", we show where the "mostly" comes from in the phrase "mostly lock-free list":



- "thread2" uses "list ptr" to start walking the list
- "thread2" visits "node1", decides it's not interesting and moves on
- "thread2" visits "node2", likes it and does some arbitrary processing

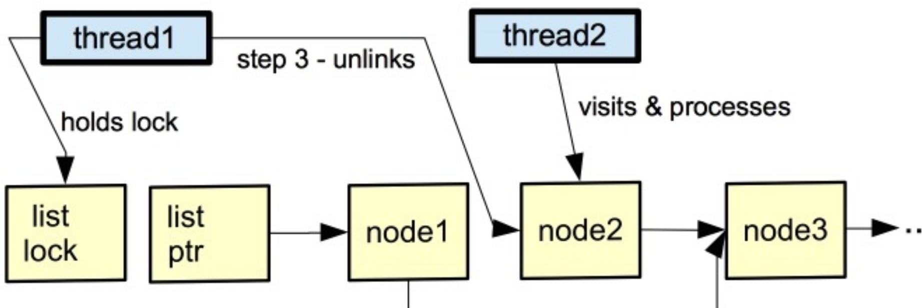
Note: In this world, "add" and "delete" operations are fairly rare and "walk&read" operations are frequent. "thread2" does not grab the "list lock" because there can be multiple parallel readers, hence "mostly lock-free list". These parallel readers use load-acquire operations as needed.

Note: Although not shown in these diagrams, assume that the "add node" code does the right release-store operations to make sure that a node being added is properly flushed before being made accessible via the list.

Note: In the delete diagrams, "node2" is shown as having its next field no longer pointing at "node3". That was only done for drawing space; in reality, the field doesn't need to be cleared since the node is about to be freed.

Overlapping Delete-Part-2 and Walk-&-Read

In "Fig. 3a - Overlapping Delete-Part-2 and Walk-&-Read", we show a benign overlap between Delete-Part-2 and Walk-&-Read":

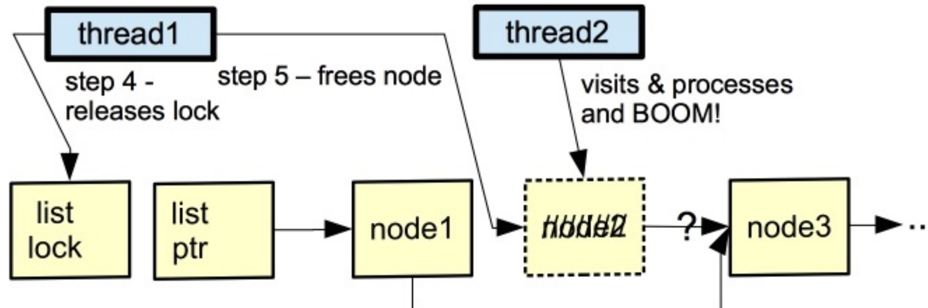


- 3) "thread1" unlinks "node2" at the same time as
- "thread2" visits and processes "node2"

The unlink done by "thread1" is not destructive to "thread2" which had just followed "node1"'s next pointer to "node2". Even if "thread2" was actually interested in "node3", it would be able to follow "node2"'s next pointer to "node3".

Overlapping Delete-Part-3 and Walk-&-Read

In "Fig. 3b - Overlapping Delete-Part-3 and Walk-&-Read", we show a problematic overlap between Delete-Part-3 and Walk-&-Read":



- 4) "thread1" releases the "list lock"
- 5) "thread1" frees "node2" at the same time as
- "thread2" visits and processes "node2"

The free done by "thread1" is potentially destructive to "thread2" which had just followed "node1"'s next pointer to "node2". Even if "thread2" was actually interested in "node3", it is possible that accessing "node2"'s next field could crash "thread2". Even worse, the memory occupied by "node2" could be immediately reused by something else and the new value in "node2"'s next field could send "thread2" off into the weeds.

Note: In a system that supports ADI (expand this), the dereference of anything in "node2" by "thread2" after "thread1" has freed "node2" will result in an ADI trap.

Conclusions About Mostly Lock-Free List

So the "Typical Mostly Lock-Free List" has some races and risks of crashes, but, unfortunately, they won't happen that often or they will result in silent data corruption without a crash.

Wait, that must be a typo! You said "unfortunately". Yes, I did and I meant it. If the "Typical Mostly Lock-Free List" crashed more often, people would realize that it's a bad idea and stop using it.

Section 2 - Lock-Free List with SMR

The Safe Memory Reclamation (SMR) technique is specifically designed to make lock-free algorithms safe. Not "more safe". Not "mostly safe". Not "narrowing the race window" safe. I mean safe. S-A-F-E. Of course, like any algorithm, it has to be applied correctly.

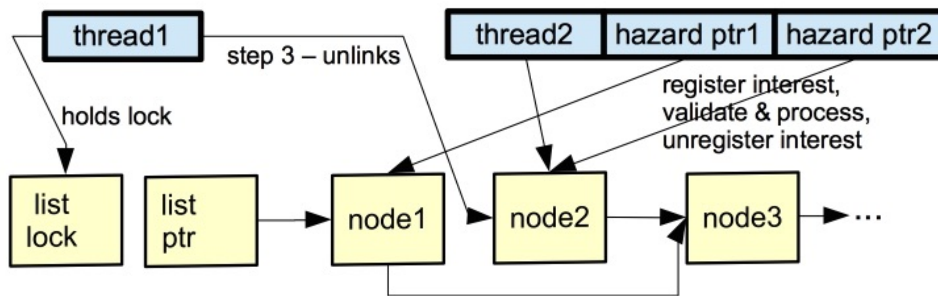
Note: Of course, even when initially applied correctly, someone will come along with a way to make it faster and that will open a subtle window where it's no longer safe. Sigh...

Overlapping Delete-Part-2 and Walk-&-Read

In "Fig. 4a - Overlapping Delete-Part-2 and Walk-&-Read",

Lock-Free List With SMR

Fig. 4a - Overlapping Delete-Part-2 and Walk-&-Read



- In order to safely visit "node2", "thread2" has to do this dance with "node1":
  - The "node1" dance is not labeled in the diagram because our focus is on "node2".
  - register interest in "node1" (using "hazard ptr1")
  - validate reference to "node1" (via "list ptr")
  - use "node1"'s next field to visit "node2"
- 3) "thread1" unlinks "node2" at the same time as
- "thread2" visits "node2" and does this dance:
  - register interest in "node2" (using "hazard ptr2")
  - validate reference to "node2"
  - processes "node2"
  - unregister interest in "node2"

"Fig. 3a" and "Fig. 4a" are almost identical. In "Fig. 3a", we had to analyze and reason about why the parallel operation was safe for that scenario. If you're like most folks, you have some doubts about the safety of the implied edge cases in "Fig. 3a". That's OK because the scenario is racy and we know it.

In "Fig. 4a", the "hazard ptr" boxes are new and the list of what "thread2" does is longer. Let's take a closer look at that list:

- register interest in "node2" (using "hazard ptr2")

  This step is where the "hazard ptr2" field is updated to refer to "node2". The update is done with a proper release-store operation which is the first critical piece to the SMR algorithm.

- validate reference to "node2"

  This step is where "thread2" validates that "node2" is still valid and, by valid, I mean that "node1"'s next field still refers to "node2". This validity check is the second critical piece to the SMR algorithm. This step's use of "node1"'s next field is protected by "hazard ptr1".

- processes "node2"

  By first registering interest in "node2" and then validating that "node2" is still reachable, "thread2" can safely process "node2" without any races with a delete operation.

- unregister interest in "node2"

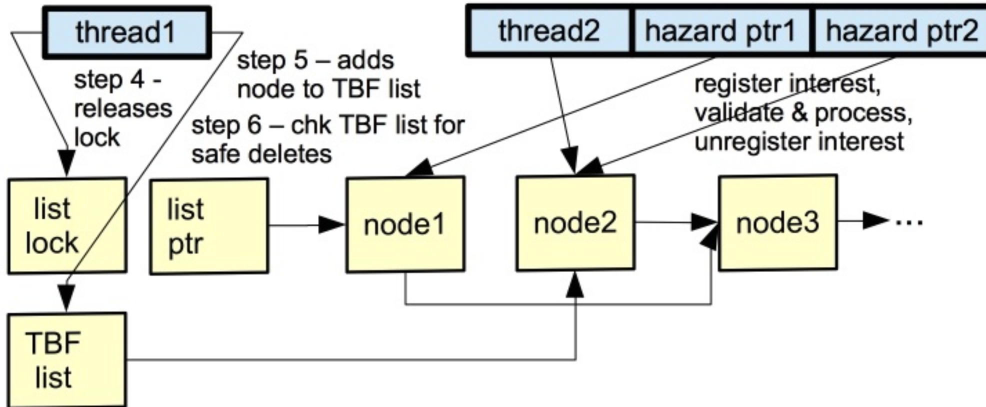
  This step is where the "hazard ptr2" field is cleared.

  Note: If the traversal algorithm is going to continue on to "node3", then we would leave "hazard ptr2" alone while we did the work to safely visit "node3" (using "hazard ptr1" to protect "node3").

At this point, you should have this tickle of a thought in the back of your mind that something is missing... and you would be right! Like "Fig. 3a", "Fig. 4a" doesn't show a destructive race. We've simply used this diagram to introduce you to the idea of "register interest" and "validate reference".

Overlapping Delete-Part-3 and Walk-&-Read

In "Fig. 4b - Overlapping Delete-Part-3 and Walk-&-Read",



In "Fig. 4b", we have three steps:

4) "thread1" releases the "list lock"

5) "thread1" adds "node2" to the TBF (To-Be-Freed) list

6) "thread1" checks the TBF list for safe deletes

and the overlap (repeated from "Fig. 4a"):

- In order to safely visit "node2", "thread2" has to do this dance with "node1":
  - The "node1" dance is not labeled in the diagram because our focus is on "node2".
  - register interest in "node1" (using "hazard ptr1")
  - validate reference to "node1" (via "list ptr")
  - use "node1"'s next field to visit "node2"
- "thread2" visits "node2" and does this dance:
  - register interest in "node2" (using "hazard ptr2")
  - validate reference to "node2"
  - processes "node2"
  - unregister interest in "node2"

    Note: If the traversal algorithm is going to continue on to "node3", then we would leave "hazard ptr2" alone while we did the work to safely visit "node3" (using "hazard ptr1" to protect "node3").

In "Fig. 3b", we had two steps and the overlap:

4) "thread1" releases the "list lock"

5) "thread1" frees "node2" at the same time as

- "thread2" visits and processes "node2"

Step 4 in both figures is the same and uninteresting. With SMR, the previously dangerous Step 5 is turned into two safe steps:

5) "thread1" adds "node2" to the TBF (To-Be-Freed) list

Instead of always deleting "node2", "thread1" always puts it on the TBF list for later processing.

6) "thread1" checks the TBF list for safe deletes

"thread1" walks the entire TBF list, determines if each node is "safe to delete", and deletes it if safe.

This "safe to delete" step is the third critical piece to the SMR algorithm. Simply put: a node is "safe to delete" if there are no "hazard ptr" references to the node.

Conclusions About Lock-Free List with SMR

Right now, your brain should be screaming that all we did is move the race from one location to another! Fair enough... since we haven't explained why the three critical operations means that there is no race.

Here are "thread2"'s two critical operations:

- Register interest in "node2" (using "hazard ptr2").
- Validate "node2".

Here is "thread1"'s single critical operation:

- Is "node2" "safe to delete"?

Both "thread1" and "thread2" can be executing their critical operations concurrently without using locks and we can have different overlapping scenarios:

> Scenario 1) "thread2" registers interest in "node2" before "thread1" starts the "safe to delete" check.
>
> > Result: "thread1" decides that "node2" isn't safe to delete (because of the "hazard ptr2") and skips it; "node2" will be checked again by another execution of Step 6 by "thread1" or by some other thread.
>
> Scenario 2) "thread1" starts the "safe to delete" check before "thread2" registers interest in "node2".
>
> > Result: "thread1" has determined that "node2" is safe to delete (because no "hazard ptr" was found) and frees it; "thread2" has registered interest in "node2" and then tries to validate "node2". "thread2" sees that "node1"'s "next" field no longer refers to "node2" so it realizes that "node2" is not valid and skips it.

It is VERY important to note that when "thread2" validates "node2" it is not accessing "node2" at all; "thread2" is comparing "node1"'s "next" field to "node2"'s address which is safe. "node1"'s "next" field was updated in Step 3 which happens before Step 6 which does the free operation.

Since validation occurs after registering interest, there is no race.

Section 3 - JavaThread Examples

This section contains examples of existing code where JavaThreads are used and assumed to be safe, but in fact contain a race between applying an operation on a target thread, and the target thread terminating and its memory and other resources being freed. Other similar API's avoid this race by one of:

1. Acquiring the Threads_lock for the duration of the operation (which prevents the thread from terminating); or
2. At the Java-level synchronizing on the java.lang.Thread object (e.g., calling from synchronized methods of class java.lang.Thread) which also prevents a thread from terminating due to the Java-level exit protocol that a thread must follow. Or
3. Always operating on the current thread.

But the lock-based solutions are not always possible due to the possibility of deadlock being introduced, as well as performance latencies.

jobject Parameter Converted to JavaThread

In this example, a jobject referring to a java.lang.Thread object is passed from Java code to a JVM call that implements java.lang.Thread.suspend(). We trust the Java class library code that is calling JVM_SuspendThread() to pass a proper reference to a java.lang.Thread object so we don't do a lot of sanity checking that the jobject actually refers to what we expect.

The interesting parts of the code:

- resolve the jthread parameter into a java_thread oop
- fetch the JavaThread* that is cached in the java.lang.Thread object
- if receiver is not NULL, then we assume it is a valid JavaThread* and proceed:
  - grab the SR_lock (Suspend/Resume lock)
  - check for nested suspend requests
  - check for the is_exiting() flag
  - set the external suspend flag
  - drop the SR_lock
  - call java_suspend() to finish the work

```
open/src/hotspot/share/prims/jvm.cpp:

JVM_ENTRY(void, JVM_SuspendThread(JNIEnv* env, jobject jthread))
  JVMWrapper("JVM_SuspendThread");
  oop java_thread = JNIHandles::resolve_non_null(jthread);
  JavaThread* receiver = java_lang_Thread::thread(java_thread);

  if (receiver != NULL) {
    // thread has run and has not exited (still on threads list)

    {
      MutexLockerEx ml(receiver->SR_lock(), Mutex::_no_safepoint_check_flag);
      if (receiver->is_external_suspend()) {
        // Don't allow nested external suspend requests. We can't return
        // an error from this interface so just ignore the problem.
        return;
      }
      if (receiver->is_exiting()) { // thread is in the process of exiting
        return;
      }
      receiver->set_external_suspend();
    }

    // java_suspend() will catch threads in the process of exiting
    // and will ignore them.
    receiver->java_suspend();

    // It would be nice to have the following assertion in all the
    // time, but it is possible for a racing resume request to have
    // resumed this thread right after we suspended it. Temporarily
    // enable this assertion if you are chasing a different kind of
    // bug.
    //
    // assert(java_lang_Thread::thread(receiver->threadObj()) == NULL ||
    //   receiver->is_being_ext_suspended(), "thread is not suspended");
  }
JVM_END
```

Interesting comments above:

- // thread has run and has not exited (still on threads list)
  - How do we know that the thread is still on the threads list?
- // java_suspend() will catch threads in the process of exiting
  // and will ignore them.
  - How do we "catch" the exiting threads?

The above code races with the "target" JavaThread which might be trying to exit:

- java_lang_Thread::thread(java_thread) returns the cached JavaThread*
- that cached JavaThread* is cleared as part of the JavaThread's exit path so we have a chance of realizing the JavaThread is exiting with the "if (receiver != NULL) {" check
- the "if (receiver->is_exiting()) {" check is another way we can see if the JavaThread is exiting, but we did two other JavaThread operations on the way to that check
  - we call receiver->SR_lock() even though the exiting JavaThread might have deleted the SR_lock
  - we call receiver->is_external_suspend() even though the exiting JavaThread's memory might be freed
- the SR_lock'ed block is targeted at the JavaThread::exit() dance we do when a JavaThread is marked as terminating:
  - if the suspend request comes in before the JavaThread can mark itself as 'terminating', then the JavaThread honors the suspend request
  - if the suspend request comes in after the JavaThread has marked itself as terminating, then the suspend request is ignored (for this API)
  - of course, even if we recognized that the JavaThread is exiting, we still have to drop the SR_lock and that unlock operation might fail if the exiting JavaThread deleted the SR_lock

jthread Parameter Converted to JavaThread

In this example, a jthread referring to a java.lang.Thread object is passed from native code to JVM/TI SuspendThread(). Since this is an external API, we make more sanity checks before we trust that we actually have a java.lang.Thread object.

The not very interesting parts of the code:

- JVM/TI can be configured out of the system
- common JVM/TI entry boilerplate code

The more interesting parts of the code:

- SuspendThread() requires a capability
- a JavaThread can suspend itself by passing NULL
- more sanity checks than JVM interfaces:
  - resolve_external_guard() versus resolve_non_null()
  - check that oop is-a java.lang.Thread class
- fetch the JavaThread* that is cached in the java.lang.Thread object
- also JVM/TI SuspendThread returns error codes
- the SR_lock using parts of the code are in JvmtiEnv::SuspendThread()

`build/solaris-x86_64-normal-server-release/hotspot/variant-server/gensrc/jvmtifiles/jvmtiEnter.cpp:`

```
static jvmtiError JNICALL
jvmti_SuspendThread(jvmtiEnv* env,
            jthread thread) {

#if !INCLUDE_JVMTI
  return JVMTI_ERROR_NOT_AVAILABLE;
#else
  if(!JvmtiEnv::is_vm_live()) {
    return JVMTI_ERROR_WRONG_PHASE;
  }
  Thread* this_thread = Thread::current_or_null();
  if (this_thread == NULL || !this_thread->is_Java_thread()) {
    return JVMTI_ERROR_UNATTACHED_THREAD;
  }
  JavaThread* current_thread = (JavaThread*)this_thread;
  ThreadInVMfromNative __tiv(current_thread);
  VM_ENTRY_BASE(jvmtiError, jvmti_SuspendThread , current_thread)
  debug_only(VMNativeEntryWrapper __vew;)
  CautiouslyPreserveExceptionMark __em(this_thread);
  JvmtiEnv* jvmti_env = JvmtiEnv::JvmtiEnv_from_jvmti_env(env);
  if (!jvmti_env->is_valid()) {
    return JVMTI_ERROR_INVALID_ENVIRONMENT;
  }

  if (jvmti_env->get_capabilities()->can_suspend == 0) {
    return JVMTI_ERROR_MUST_POSSESS_CAPABILITY;
  }
  jvmtiError err;
  JavaThread* java_thread;
  if (thread == NULL) {
    java_thread = current_thread;
  } else {
    oop thread_oop = JNIHandles::resolve_external_guard(thread);

    if (thread_oop == NULL) {
      return JVMTI_ERROR_INVALID_THREAD;
    }
    if (!thread_oop->is_a(SystemDictionary::Thread_klass())) {
      return JVMTI_ERROR_INVALID_THREAD;
    }
    java_thread = java_lang_Thread::thread(thread_oop);
    if (java_thread == NULL) {
      return JVMTI_ERROR_THREAD_NOT_ALIVE;
    }
  }
  err = jvmti_env->SuspendThread(java_thread);
  return err;
#endif // INCLUDE_JVMTI
}
```

The above code races with the "target" JavaThread which might be trying to exit:

- java_lang_Thread::thread(java_thread) returns the cached JavaThread*
- that cached JavaThread* is cleared as part of the JavaThread's exit path so we have a chance of realizing the JavaThread is exiting with the "if (java_thread == NULL) {" check
- all of the rest of the sanity checks are in JvmtiEnv::SuspendThread() and have the same races as JVM_SuspendThread()

Section 4 - 3 Critical Operations in New JavaThread Code

Here's a reminder of the three critical operations necessary for safe lock-free list operations:

The data reader/writer thread has two critical operations (this is "thread2" in the generic example):

- Register interest in the object (this is "node2" in the generic example).
- Validate object reference (this is "node2" in the generic example).

The object deleting thread has a single critical operation (this is "thread1" in the generic example):

- Is the object "safe to delete" (this is "node2" in the generic example)?

A JavaThread is the object we mention above. The data reader/writer thread is another Thread that wants to read data from the JavaThread or write data to the JavaThread. The object deleting thread is actually the JavaThread itself. As part of the JavaThread exit process, a JavaThread will delete itself (usually).

Registering Interest In and Validating a JavaThread Reference

Threads::acquire_stable_list() registers interest in JavaThread references and validates those same references.

The interesting parts of the fast path version of the function:

- The Thread::set_threads_hazard_ptr() call registers the calling thread's interesting in the current _smr_java_thread_list by storing the current value (plus a tag) in the calling Thread's _threads_hazard_ptr field.
- The second get_smr_java_thread_list() call validates that _smr_java_thread_list did not change while we were updating _threads_hazard_ptr:
    - This call races with Threads::add() or Threads::delete()!
    - If the current _smr_java_thread_list does not match the one in 'threads', then we lost the race and we simply loop around and try again.
    - If we won the race, then we try to officially publish the hazard pointer.
- The Thread::cmpxchg_threads_hazard_ptr() call tries to remove the tag from the _threads_hazard_ptr field:
    - This call races with the scanning thread (see Threads::is_a_protected_JavaThread())!
    - If this thread wins the race, then the validated hazard pointer is published.
    - If this thread loses the race, then we simply loop around and try again.
- At the point where we return 'threads', we have successfully published a hazard pointer to a ThreadsList and we are now protecting the JavaThreads on that ThreadsList.

Note: Once a ThreadsList value has been published via a _threads_hazard_ptr field, it can be seen by Threads::is_a_protected_JavaThread(); see the "Is It Safe to Delete This JavaThread Reference" subsection below.

```
open/src/hotspot/share/runtime/thread.cpp:

ThreadsList *Threads::acquire_stable_list_fast_path(Thread *self) {
  assert(self != NULL, "sanity check");
assert(self->get_threads_hazard_ptr() == NULL, "sanity check");
assert(self->get_nested_threads_hazard_ptr() == NULL,
"cannot have a nested hazard ptr with a NULL regular hazard ptr");

ThreadsList* threads;

// Stable recording of a hazard ptr for SMR. This code does not use
// locks so its use of the _smr_java_thread_list & _threads_hazard_ptr
// fields is racy relative to code that uses those fields with locks.
// OrderAccess and Atomic functions are used to deal with those races.
//
while (true) {
threads = get_smr_java_thread_list();

// Publish a tagged hazard ptr to denote that the hazard ptr is not
// yet verified as being stable. Due to the fence after the hazard
// ptr write, it will be sequentially consistent w.r.t. the
// sequentially consistent writes of the ThreadsList, even on
    // non-multiple copy atomic machines where stores can be observed
    // in different order from different observer threads.
    ThreadsList* unverified_threads = Thread::tag_hazard_ptr(threads);
self->set_threads_hazard_ptr(unverified_threads);

    // If _smr_java_thread_list has changed, we have lost a race with
    // Threads::add() or Threads::remove() and have to try again.
    if (get_smr_java_thread_list() != threads) {
      continue;
    }

    // We try to remove the tag which will verify the hazard ptr as
    // being stable. This exchange can race with a scanning thread
    // which might invalidate the tagged hazard ptr to keep it from
    // being followed to access JavaThread ptrs. If we lose the race,
    // we simply retry. If we win the race, then the stable hazard
    // ptr is officially published.
    if (self->cmpxchg_threads_hazard_ptr(threads, unverified_threads) == unverified_threads) {
      break;
    }
  }

  // A stable hazard ptr has been published letting other threads know
  // that the ThreadsList and the JavaThreads reachable from this list
  // are protected and hence they should not be deleted until everyone
  // agrees it is safe to do so.

  return threads;
}
```

Wait a minute! The above code registers interest in and validates a *list* of JavaThreads and not just one JavaThread. This subsection title says "JavaThread reference" and that's singular! Why do we do that?

Good question! Here are the reasons we use SMR on a list of JavaThreads:

- Several JavaThread operations can operate on one or more JavaThreads so we associate a hazard pointer with a ThreadsList instead of having a hazard pointer for each JavaThread.
- The linked list traversal of JavaThreads (following *next* pointers) is slower than traversing an array; some of the newer GC algorithms need a faster traversal of a list of JavaThreads.

Note: We're planning to entirely replace the linked list traversal used by the Threads class with the fast array traversal in the ThreadsList class; right now we support both because we have not converted all the old uses to the new style.

Note: See the "Threads::acquire_stable_list_fast_path() is Lock Free" subsection in the "Gory Details" section below for... you guessed it ...the gory details...

Safely Deleting a JavaThread Reference

Threads::smr_delete() is the function used to safely delete a JavaThread. Calls to "delete thread" are generally replaced in this project with a call to "thread->smr_delete()" which is a wrapper used to call Threads::smr_delete() when the JavaThread is managed using SMR. If a JavaThread

has not yet been added to the Threads list, then "thread->smr_delete()" will directly call "delete thread" since SMR is not needed in that case.

When Threads::add() is used to add a JavaThread to the Threads list, a new _smr_java_thread_list is allocated, populated with the current Threads list, and the new JavaThread is appended. When Threads::remove() is used to delete a JavaThread from the Threads list, a new _smr_java_thread_list is allocated and populated with the current Threads list minus the to-be-deleted JavaThread. In both add() and remove() cases, the old ThreadsList is managed with SMR and is only freed once there is no interest in that particular ThreadsList.

A JavaThread can appear on more than one ThreadsList due to normal adds and deletes of other JavaThreads so Threads::smr_delete() has to make sure that the to-be-deleted JavaThread is not protected by any hazard pointers (ThreadsLists).

The interesting parts of the function:

- Need to grab the Threads_lock in order to call is_a_protected_JavaThread(); see the next subsection.

  Note: We've tried several times to do this check without holding the Threads_lock, but we haven't found a safe algorithm.

- We off-load the wait-notify traffic to the Threads::smr_delete_lock() in order to reduce traffic on the Threads_lock.
- If the JavaThread is not protected, we do the lock housekeeping and break out to delete the JavaThread.

  Note: Yes, a JavaThread deletes itself. At various times over the years we've tried to change that without success. We also tried to change it during this project without success.

- If the JavaThread is protected, we have to wait for a release_stable_list() call which is how a JavaThread unregisters its interest in a ThreadsList.
- Notice that we've added logging from the beginning; we have also enabled logging in the stress tests that we're writing for the feature and have some rudimentary reporting on when SMR comes into play to protect a JavaThread from deletion.
- Note: The current version of the Threads::smr_delete() function has statistics and timer code that is enabled via the 'EnableThreadSMRStatistics' option; this code is elided from this presentation for clarity.

```
open/src/hotspot/share/runtime/thread.cpp:

void Threads::smr_delete(JavaThread *thread) {
  assert(!Threads_lock->owned_by_self(), "sanity");

  bool has_logged_once = false;
  while (true) {
    {
      // No safepoint check because this JavaThread is not on the
      // Threads list.
      MutexLockerEx ml(Threads_lock, Mutex::_no_safepoint_check_flag);
      // Cannot use a MonitorLockerEx helper here because we have
      // to drop the Threads_lock first if we wait.
      Threads::smr_delete_lock()->lock_without_safepoint_check();
      // Set the smr_delete_notify flag after we grab smr_delete_lock
      // and before we scan hazard pointers because we're doing
      // double-check locking in release_stable_list().
      Threads::set_smr_delete_notify();

      if (!is_a_protected_JavaThread(thread)) {
        // This is the common case.
        Threads::clear_smr_delete_notify();
        Threads::smr_delete_lock()->unlock();
        break;
      }
      if (!has_logged_once) {
        has_logged_once = true;
        log_debug(os, thread, smr)("Threads::smr_delete: thread=" INTPTR_FORMAT " is not deleted.",
p2i(thread));
        if (log_is_enabled(Debug, os, thread)) {
          ScanHazardPtrPrintMatchingThreadsClosure scan_cl(thread);
          Threads::threads_do(&scan_cl);
        }
      }
    } // We have to drop the Threads_lock to wait or delete the thread

    // Wait for a release_stable_list() call before we check again. No
    // safepoint check, no timeout, and not as suspend equivalent flag
    // because this JavaThread is not on the Threads list.
    Threads::smr_delete_lock()->wait(Mutex::_no_safepoint_check_flag, 0,
                                     !Mutex::_as_suspend_equivalent_flag);

    Threads::clear_smr_delete_notify();
    Threads::smr_delete_lock()->unlock();
    // Retry the whole scenario.
  }

  delete thread;
  log_debug(os, thread, smr)("Threads::smr_delete: thread=" INTPTR_FORMAT " is deleted.", p2i(thread));
}
```

Is It Safe to Delete This JavaThread Reference

Threads::is_a_protected_JavaThread() is the function used to determine if a JavaThread is protected by a hazard pointer.

The interesting parts of the function:

- We create a ThreadScanHashtable (which will eventually hold all the unique JavaThreads that are protected by a hazard ptr).
- We create a ScanHazardPtrGatherProtectedThreadsClosure which will gather (in two steps) all the unique JavaThreads that are protected by a hazard ptr:
  - Scan the current Threads list (via _smr_java_thread_list) and visit all the non-NULL hazard ptrs (ThreadsList references).
  - Use AddThreadHazardPointerThreadClosure to gather all the unique JavaThreads mentioned in each ThreadsList that is visited.
- If the specified JavaThread is found in ThreadScanHashtable when ScanHazardPtrGatherProtectedThreadsClosure is done, then the target JavaThread is protected.

open/src/hotspot/share/runtime/thread.cpp:

```
bool Threads::is_a_protected_JavaThread(JavaThread *thread) {
  assert_locked_or_safepoint(Threads_lock);

  // Hash table size should be first power of two higher than twice
  // the length of the Threads list.
  int hash_table_size = MIN2(_number_of_threads, 32) << 1;
  hash_table_size--;
  hash_table_size |= hash_table_size >> 1;
  hash_table_size |= hash_table_size >> 2;
  hash_table_size |= hash_table_size >> 4;
  hash_table_size |= hash_table_size >> 8;
  hash_table_size |= hash_table_size >> 16;
  hash_table_size++;

  // Gather a hash table of the JavaThreads indirectly referenced by
  // hazard pointers.
  ThreadScanHashtable *scan_table = new ThreadScanHashtable(hash_table_size);
  ScanHazardPtrGatherProtectedThreadsClosure scan_cl(scan_table);
  Threads::threads_do(&scan_cl);

  bool thread_is_protected = false;
  if (scan_table->get_entry((void*)thread) != NULL) {
    thread_is_protected = true;
  }
  delete scan_table;
  return thread_is_protected;
}
```

For Completeness, Unregistering Interest In a JavaThread Reference

Threads::release_stable_list() unregisters interest in JavaThread references; we don't consider it a critical operation, but it is important to see how it works.

The interesting parts of the fast path version of the function:

- The Thread::set_threads_hazard_ptr() call unregisters the calling thread's interesting in the ThreadsList by storing NULL in the calling Thread's _threads_hazard_ptr field.

  Note: The ThreadsList the calling thread is no longer interested in may not be the current _smr_java_thread_list.

- We use Threads::smr_delete_lock() instead of the Threads_lock to reduce the traffic on the Threads_lock.
- We use double-check locking via Threads::smr_delete_notify() and Threads::smr_delete_lock() in Threads::release_stable_list_wake_up() to reduce the number of lock operations.

open/src/hotspot/share/runtime/thread.cpp:

```
void Threads::release_stable_list_fast_path(Thread *self) {
  assert(self != NULL, "sanity check");
assert(self->get_threads_hazard_ptr() != NULL, "sanity check");
assert(self->get_nested_threads_hazard_ptr() == NULL,
         "cannot have a nested hazard ptr when releasing a regular hazard ptr");

  // After releasing the hazard pointer, other threads may go ahead and
  // free up some memory temporarily used by a ThreadsList snapshot.
  self->set_threads_hazard_ptr(NULL);

  // We use double-check locking to reduce traffic on the system
  // wide smr_delete_lock.
  if (Threads::smr_delete_notify()) {
    // An exiting thread might be waiting in smr_delete(); we need to
    // check with smr_delete_lock to be sure.
release_stable_list_wake_up((char *) "regular hazard ptr");
  }
}

// Wake up portion of the release stable ThreadsList protocol;
// uses the smr_delete_lock().
//
void Threads::release_stable_list_wake_up(char *log_str) {
  assert(log_str != NULL, "sanity check");

  // Note: smr_delete_lock is held in smr_delete() for the entire
  // hazard ptr search so that we do not lose this notify() if
  // the exiting thread has to wait. That code path also holds
  // Threads_lock (which was grabbed before smr_delete_lock) so that
  // threads_do() can be called. This means the system can't start a
  // safepoint which means this thread can't take too long to get to
  // a safepoint because of being blocked on smr_delete_lock.
  //
  MonitorLockerEx ml(Threads::smr_delete_lock(), Monitor::_no_safepoint_check_flag);
  if (Threads::smr_delete_notify()) {
    // Notify any exiting JavaThreads that are waiting in smr_delete()
    // that we've released a ThreadsList.
    ml.notify_all();
    log_debug(os, thread, smr)("tid=" UINTX_FORMAT ": Threads::release_stable_list notified %s",
os::current_thread_id(), log_str);
  }
}
```

Section 5 - How Do You Use This Stuff?

In keeping with HotSpot style, this project uses both helper objects and service functions/methods to make this facility easier to use.

ThreadsListHandle Helper Object

ThreadsListHandle is the helper object for acquiring a ThreadsList and protecting all of the JavaThreads on that ThreadsList.

The constructor is very simple and is mostly a wrapper around a Threads::acquire_stable_list() call:

open/src/hotspot/share/runtime/thread.cpp:

```
ThreadsListHandle::ThreadsListHandle(Thread *self) : _list(Threads::acquire_stable_list(self)), _self(self)
{
  assert(self == Thread::current(), "sanity check");
}
```

Note: The statistics code guarded by the 'EnableThreadSMRStatistics' option is elided from the above code.

The destructor is even simpler and is just a wrapper around a Threads::release_stable_list() call:

open/src/hotspot/share/runtime/thread.cpp:

```
ThreadsListHandle::~ThreadsListHandle() {
  Threads::release_stable_list(_self);
}
```

Note: The statistics code guarded by the 'EnableThreadSMRStatistics' option is elided from the above code.

Like most helper objects, a ThreadsListHandle is easy to declare and use:

```
ThreadsListHandle tlh(thread);
...
some_func(tlh.list());
```

Interesting parts of the above example:

- The 'thread' parameter is optional and, if omitted, the current Thread is used.

  Note: If you have the appropriate Thread value available, then please use it since Thread::current() calls are not cheap on some platforms.

- The JavaThreads on the ThreadsList in the ThreadsListHandle are protected until the handle goes out of scope.

cv_internal_thread_to_JavaThread() service function/method

In order to reduce duplicate boiler plate code, we have provided the cv_internal_thread_to_JavaThread() function to convert internal Thread references into safe JavaThread pointers.

Interesting parts of the function:

- same checks as the JVM_SuspendThread() example (with more comments)
- verify the derived JavaThread* against the specified ThreadsList which is entirely new

```
open/src/hotspot/share/runtime/thread.cpp:

bool ThreadsListHandle::cv_internal_thread_to_JavaThread(jobject jthread,
                                                         JavaThread ** jt_pp,
                                                         oop * thread_oop_p) {
  assert(this->t_list != NULL, "must have a ThreadsList");
  assert(jt_pp != NULL, "must have a return JavaThread pointer");
  // thread_oop_p is optional so no assert()

  // The JVM_* interfaces don't allow a NULL thread parameter; JVM/TI
  // allows a NULL thread parameter to signify "current thread" which
  // allows us to avoid calling cv_external_thread_to_JavaThread().
  // The JVM_* interfaces have no such leeway.

  oop thread_oop = JNIHandles::resolve_non_null(jthread);
  // Looks like an oop at this point.
  if (thread_oop_p != NULL) {
    // Return the oop to the caller; the caller may still want
    // the oop even if this function returns false.
    *thread_oop_p = thread_oop;
  }

  JavaThread *java_thread = java_lang_Thread::thread(thread_oop);
  if (java_thread == NULL) {
    // The java.lang.Thread does not contain a JavaThread * so it has
    // not yet run or it has died.
    return false;
  }
  // Looks like a live JavaThread at this point.

  if (java_thread != JavaThread::current()) {
    // jthread is not for the current JavaThread so have to verify
    // the JavaThread * against the ThreadsList.
    if (EnableThreadSMRExtraValidityChecks && !includes(java_thread)) {
      // Not on the JavaThreads list so it is not alive.
      return false;
    }
  }

  // Return a live JavaThread that is "protected" by the
  // ThreadsListHandle in the caller.
  *jt_pp = java_thread;
  return true;
}
```

cv_external_thread_to_JavaThread() service function/method

In order to reduce duplicate boiler plate code, we have provided the cv_external_thread_to_JavaThread() function to convert external Thread references into safe JavaThread pointers.

Interesting parts of the function:

- same checks as the JVM/TI SuspendThread() example (with more comments)
- verify the derived JavaThread* against the specified ThreadsList which is entirely new
- similar calling convention as cv_internal_thread_to_JavaThread():
  - JVM/TI error codes instead of true/false

```
open/src/hotspot/share/prims/jvmtiExport.cpp:

jvmtiError
JvmtiExport::cv_external_thread_to_JavaThread(ThreadsList * t_list,
                                              jthread thread,
                                              JavaThread ** jt_pp,
oop * thread_oop_p) {
  assert(t_list != NULL, "must have a ThreadsList");
  assert(jt_pp != NULL, "must have a return JavaThread pointer");
// thread_oop_p is optional so no assert()

  oop thread_oop = JNIHandles::resolve_external_guard(thread);
  if (thread_oop == NULL) {
    // NULL jthread, GC'ed jthread or a bad JNI handle.
    return JVMTI_ERROR_INVALID_THREAD;
  }
  // Looks like an oop at this point.

  if (!thread_oop->is_a(SystemDictionary::Thread_klass())) {
    // The oop is not a java.lang.Thread.
    return JVMTI_ERROR_INVALID_THREAD;
  }
  // Looks like a java.lang.Thread oop at this point.

if (thread_oop_p != NULL) {
// Return the oop to the caller; the caller may still want
// the oop even if this function returns an error.
*thread_oop_p = thread_oop;
}

JavaThread * java_thread = java_lang_Thread::thread(thread_oop);
  if (java_thread == NULL) {
    // The java.lang.Thread does not contain a JavaThread * so it has
    // not yet run or it has died.
    return JVMTI_ERROR_THREAD_NOT_ALIVE;
  }
  // Looks like a live JavaThread at this point.

// We do not check the EnableThreadSMRExtraValidityChecks option
// for this includes() call because JVM/TI's spec is tighter.
  if (!t_list->includes(java_thread)) {
    // Not on the JavaThreads list so it is not alive.
    return JVMTI_ERROR_THREAD_NOT_ALIVE;
  }

  // Return a live JavaThread that is "protected" by the
  // ThreadsListHandle in the caller.
  *jt_pp = java_thread;

  return JVMTI_ERROR_NONE;
}
```

cv_oop_to_JavaThread() service function/method

In order to reduce duplicate boiler plate code, we have provided the cv_oop_to_JavaThread() function to convert oops into safe JavaThread pointers. Yes, there are some JVM/TI APIs where we are passed jthreads and some where we are passed oops... go figure!

Interesting parts of the function:

- Subset of the checks in cv_external_thread_to_JavaThread() because we're starting with an oop.

- Not able to refactor cv_external_thread_to_JavaThread() to call cv_oop_to_JavaThread() without duplicating the "thread_oop->is_a(SystemDictionary::Thread_klass())" check.

```
open/src/hotspot/share/prims/jvmtiExport.cpp:

jvmtiError
JvmtiExport::cv_oop_to_JavaThread(ThreadsList * t_list, oop thread_oop,
JavaThread ** jt_pp) {
assert(t_list != NULL, "must have a ThreadsList");
assert(thread_oop != NULL, "must have an oop");
assert(jt_pp != NULL, "must have a return JavaThread pointer");

if (!thread_oop->is_a(SystemDictionary::Thread_klass())) {
// The oop is not a java.lang.Thread.
return JVMTI_ERROR_INVALID_THREAD;
}
// Looks like a java.lang.Thread oop at this point.

JavaThread * java_thread = java_lang_Thread::thread(thread_oop);
if (java_thread == NULL) {
// The java.lang.Thread does not contain a JavaThread * so it has
// not yet run or it has died.
return JVMTI_ERROR_THREAD_NOT_ALIVE;
}
// Looks like a live JavaThread at this point.

// We do not check the EnableThreadSMRExtraValidityChecks option
// for this includes() call because JVM/TI's spec is tighter.
if (!t_list->includes(java_thread)) {
// Not on the JavaThreads list so it is not alive.
return JVMTI_ERROR_THREAD_NOT_ALIVE;
}

// Return a live JavaThread that is "protected" by the
// ThreadsListHandle in the caller.
*jt_pp = java_thread;

return JVMTI_ERROR_NONE;
}
```

Section 6 - Examples Uses

jobject Parameter Converted to JavaThread Redux

So let's see the new version of JVM_SuspendThread().

Interesting points about the function:

- The ThreadsList is protected for the entire function until the tlh is destroyed.
- The racy uses in the SR_lock block are no longer racy.
- Yes, we still need the is_exiting() check in order to properly honor (or igore) the final suspend request.

```
open/src/hotspot/share/prims/jvm.cpp:

JVM_ENTRY(void, JVM_SuspendThread(JNIEnv* env, jobject jthread))
JVMWrapper("JVM_SuspendThread");

  ThreadsListHandle tlh(thread);
  JavaThread* receiver = NULL;
  bool is_alive = tlh.cv_internal_thread_to_JavaThread(jthread, &receiver, NULL);
  if (is_alive) {
    // jthread refers to a live JavaThread.
```

```
    {
      MutexLockerEx ml(receiver->SR_lock(), Mutex::_no_safepoint_check_flag);
      if (receiver->is_external_suspend()) {
        // Don't allow nested external suspend requests. We can't return
        // an error from this interface so just ignore the problem.
        return;
      }
      if (receiver->is_exiting()) { // thread is in the process of exiting
        return;
      }
     receiver->set_external_suspend();
    }

    // java_suspend() will catch threads in the process of exiting
    // and will ignore them.
    receiver->java_suspend();

    // It would be nice to have the following assertion in all the
    // time, but it is possible for a racing resume request to have
    // resumed this thread right after we suspended it. Temporarily
    // enable this assertion if you are chasing a different kind of
    // bug.
    //
    // assert(java_lang_Thread::thread(receiver->threadObj()) == NULL ||
    //   receiver->is_being_ext_suspended(), "thread is not suspended");
  }
JVM_END
```

jthread Parameter Converted to JavaThread Redux

So let's see the new version of JVM/TI SuspendThread().

Interesting points about the function:

- The ThreadsList is protected for the entire function until the tlh is destroyed.
- The racy uses in the SR_lock block in JvmtiEnv::SuspendThread() are no longer racy.
- Yes, we still need the is_exiting() check in JvmtiEnv::SuspendThread() in order to properly honor (or igore) the final suspend request.

```
build/solaris-x86_64-normal-server-release/hotspot/variant-server/gensrc/jvmtifiles/jvmtiEnter.cpp

static jvmtiError JNICALL
jvmti_SuspendThread(jvmtiEnv* env,
            jthread thread) {

#if !INCLUDE_JVMTI
  return JVMTI_ERROR_NOT_AVAILABLE;
#else
  if(!JvmtiEnv::is_vm_live()) {
    return JVMTI_ERROR_WRONG_PHASE;
  }
  Thread* this_thread = Thread::current_or_null();
  if (this_thread == NULL || !this_thread->is_Java_thread()) {
    return JVMTI_ERROR_UNATTACHED_THREAD;
  }
  JavaThread* current_thread = (JavaThread*)this_thread;
  ThreadInVMfromNative __tiv(current_thread);
  VM_ENTRY_BASE(jvmtiError, jvmti_SuspendThread , current_thread)
  debug_only(VMNativeEntryWrapper __vew;)
  CautiouslyPreserveExceptionMark __em(this_thread);
  JvmtiEnv* jvmti_env = JvmtiEnv::JvmtiEnv_from_jvmti_env(env);
  if (!jvmti_env->is_valid()) {
    return JVMTI_ERROR_INVALID_ENVIRONMENT;
  }

  if (jvmti_env->get_capabilities()->can_suspend == 0) {
    return JVMTI_ERROR_MUST_POSSESS_CAPABILITY;
  }
  jvmtiError err;

  JavaThread* java_thread = NULL;
  ThreadsListHandle tlh(this_thread);
```

```
  if (thread == NULL) {
    java_thread = current_thread;
  } else {

    err = JvmtiExport::cv_external_thread_to_JavaThread(tlh.list(), thread, &java_thread, NULL);
    if (err != JVMTI_ERROR_NONE) {
      return err;
    }

}
  err = jvmti_env->SuspendThread(java_thread);
  return err;
#endif // INCLUDE_JVMTI
}
```

Walking a ThreadsList

So let's take a look at walking a list of JavaThreads. Here's the new code:

```
open/src/hotspot/share/services/threadService.cpp:

DeadlockCycle* ThreadService::find_deadlocks_at_safepoint(ThreadsList * t_list, bool concurrent_locks) {

<snip>

  JavaThreadIterator jti(t_list);
  for (JavaThread* jt = jti.first(); jt != NULL; jt = jti.next()) {

p->set_depth_first_number(-1);
}
```

And here's the original for-loop line:

```
for (JavaThread* p = Threads::first(); p != NULL; p = p->next()) {
```

Interesting points about the new code:

- We added a ThreadsList parameter to the function.
- We use the new JavaThreadIterator instead of walking the current list of JavaThreads in the Threads class.
- We're only going to walk the JavaThreads that are protected by a hazard ptr.
- The new way of walking a list of JavaThreads does not require holding the Threads_lock.

Section 7 - What the Rest of This Code Does

This section briefly describes other classes and functions/methods added by this project.

File	Class or Function/ Method	Description
open/src/hotspot/share/runtime/thread.cpp	JavaThread::smr_delete()	Calls Threads::smr_delete() on the JavaThread if it has appeared on a ThreadsList. Otherwise calls "delete this". We added this wrapper and the on_thread_list flag in order to not have to write rationale comments in all the places where calling "delete this" is safe to do.
open/src/hotspot/share/runtime/thread.cpp	prefetch_and_load_ptr()	Helper function for quickly loading the ThreadsList array; used by the new DO_JAVA_THREADS macro.
open/src/hotspot/share/runtime/thread.cpp	DO_JAVA_THREADS(LIST, X)	Macro for applying "X" to every element in the ThreadsList "LIST".

open/src/hotspot/share/runtime/thread.cpp	class ThreadScanEntry, class ThreadScanHashtable, class AddThreadHazardPointerThreadClosure, class ScanHazardPtrGatherProtectedThreadsClosure, class ScanHazardPtrGatherThreadsListClosure	Support classes for using closures to gather hazard pointers and JavaThread pointers into hash tables for fast access.
open/src/hotspot/share/runtime/thread.cpp	class ScanHazardPtrPrintMatchingThreadsClosure	Support class for using a closure to log JavaThreads that have a hazard pointer that is protecting a specific JavaThread; this support class is used to debug whether SMR is incorrectly keeping a JavaThread from exiting.
open/src/hotspot/share/runtime/thread.cpp	Threads::smr_free_list()	Function for safely freeing a specified ThreadsList. The specified ThreadsList may not get deleted during this call if it is still in-use (referenced by a hazard pointer). Other ThreadsLists in the to-be-deleted chain may get deleted by this call if they are no longer in-use.
open/src/hotspot/share/runtime/thread.cpp	ThreadsList::remove_thread(), ThreadsList::add_thread(), ThreadsList::find_JavaThread_from_java_tid(), ThreadsList::includes()	Functions for managing ThreadsLists and the JavaThreads on those ThreadsLists.
open/src/hotspot/share/runtime/thread.hpp	class ThreadsListIterator	Helper class for walking a ThreadsList.
open/src/hotspot/share/runtime/thread.hpp	class ThreadsListSetter	Helper class to set the hazard pointer in the thread that creates the helper object; the creating thread optionally delegates setting its ThreadsList to the VMThread. See the "ThreadsListSetter - A Special Helper" subsection in the "Gory Details" section.
open/src/hotspot/share/runtime/thread.hpp	class NestedThreadsList	Helper class to manage the optional linked list of nested ThreadsLists for a thread; very rarely used by JVM/TI functions in event handlers. See the "NestedThreadsList - Another Special Helper" subsection in the "Gory Details" section.

Section 8 - Testing That's Been Done So Far

We are using a variety of development platforms on this project:

- Dan uses Solaris X64 for builds (product, fastdebug, and slowdebug) and test runs (product, fastdebug, and slowdebug).
- Erik O. uses Linux X64 and SPARC T7-4 with 1024 strands for builds and test runs.
- Robbin uses Linux X64 for builds and test runs.

Dan runs the following existing test suites regularly on this project's bits:

- VM/NSK JVM/TI test suite
- VM/NSK jdwp test suite
- VM/NSK JDI test suite
- VM/NSK monitoring test suite
- jdk/test/:jdk_svc JTREG tests
- hotspot/test/:hotspot_jprt, hotspot/test/:hotspot_runtime_tier[23] JTREG tests

Erik O. runs the following existing test suites regularly on this project's bits:

- GC Test Suite, tonga and jtreg hotspot_all

Robbin runs the following existing test suites regularly on this project's bits:

- All tonga except largepages.testlist and old jtreg group hotspot_all.

Dan has added the following new VM/NSK JVM/TI stress tests:

- nsk/jvmti/GetObjectMonitorUsage/objmonusage006
- nsk/jvmti/InterruptThread/intrpthrd003
- nsk/jvmti/PopFrame/popframe011
- nsk/jvmti/SuspendThread/suspendthrd003

Dan has added the following new VM/NSK monitoring stress tests:

- nsk/monitoring/ThreadInfo/isSuspended/issuspended002
- nsk/monitoring/ThreadMXBean/findMonitorDeadlockedThreads/find006

Dan has added the following new hotspot JTREG stress tests:

- test/runtime/Thread/CountStackFramesAtExit.java
- test/runtime/Thread/InterruptAtExit.java
- test/runtime/Thread/IsInterruptedAtExit.java
- test/runtime/Thread/ResumeAtExit.java
- test/runtime/Thread/SetNameAtExit.java
- test/runtime/Thread/SetPriorityAtExit.java
- test/runtime/Thread/StopAtExit.java
- test/runtime/Thread/SuspendAtExit.java

Dan has added the following new hotspot JTREG functional tests:

- test/runtime/ErrorHandling/NestedThreadsListHandleInErrorHandlingTest.java
- test/runtime/ErrorHandling/ThreadsListHandleInErrorHandlingTest.java
- test/runtime/Thread/TestThreadDumpSMRInfo.java

Section 9 - How Do You Debug This Code?

This section lists tips and tricks for debugging the code in this project.

-Xlog Support

Logging for this project has been added using the "thread" and "smr" tags. Most of the logging for this project is at the "debug" level; some summary logging is at the "info" level.

Here are two examples for logging with a simple hello world program:

```
$ cat Hello.java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}

$ java -Xlog:thread+smr -XX:CICompilerCount=2 Hello
Hello world!
[0.250s][info][thread,smr] Threads class SMR info:
[0.250s][info][thread,smr] _smr_java_thread_list=0x0000000000ab2e80, length=9, elements={
[0.250s][info][thread,smr] 0x00000000008a9800, 0x00000000008be000, 0x00000000008d2800, 0x00000000008d6800,
[0.250s][info][thread,smr] 0x00000000008d8000, 0x00000000008da800, 0x0000000000977000, 0x0000000000ab6000,
[0.250s][info][thread,smr] 0x0000000000433000
[0.250s][info][thread,smr] }
[0.250s][info][thread,smr] _smr_java_thread_list_alloc_cnt=12, _smr_java_thread_list_free_cnt=11,
_smr_java_thread_list_max=9, _smr_nested_thread_list_max=0
[0.250s][info][thread,smr] _smr_tlh_cnt=14, _smr_tlh_times=0, avg_smr_tlh_time=0.00, _smr_tlh_time_max=0
[0.250s][info][thread,smr] _smr_deleted_thread_cnt=1, _smr_deleted_thread_times=0,
avg_smr_deleted_thread_time=0.00, _smr_deleted_thread_time_max=0
[0.250s][info][thread,smr] _smr_delete_lock_wait_cnt=0, _smr_delete_lock_wait_max=0
[0.250s][info][thread,smr] _smr_to_delete_list_cnt=0, _smr_to_delete_list_max=1

$ java -Xlog:thread+smr=debug -XX:CICompilerCount=2 Hello
[0.120s][debug][thread,smr] tid=2: Threads::add: new ThreadsList=0x000000000073f660
[0.120s][debug][thread,smr] tid=2: Threads::smr_free_list: threads=0x00000000004172b0 is freed.
[0.133s][debug][thread,smr] tid=2: Threads::add: new ThreadsList=0x00000000008a2b50
[0.133s][debug][thread,smr] tid=2: Threads::smr_free_list: threads=0x000000000073f660 is freed.
[0.142s][debug][thread,smr] tid=2: Threads::add: new ThreadsList=0x00000000008a2d90
[0.142s][debug][thread,smr] tid=2: Threads::smr_free_list: threads=0x00000000008a2b50 is freed.
[0.173s][debug][thread,smr] tid=2: Threads::add: new ThreadsList=0x00000000008a3330
[0.173s][debug][thread,smr] tid=2: Threads::smr_free_list: threads=0x00000000008a2d90 is freed.
[0.173s][debug][thread,smr] tid=2: Threads::add: new ThreadsList=0x00000000008a3390
[0.173s][debug][thread,smr] tid=2: Threads::smr_free_list: threads=0x00000000008a3330 is freed.
[0.173s][debug][thread,smr] tid=2: Threads::add: new ThreadsList=0x00000000008a33f0
[0.173s][debug][thread,smr] tid=2: Threads::smr_free_list: threads=0x00000000008a3390 is freed.
[0.174s][debug][thread,smr] tid=2: Threads::add: new ThreadsList=0x00000000008a3450
[0.174s][debug][thread,smr] tid=2: Threads::smr_free_list: threads=0x00000000008a33f0 is freed.
[0.196s][debug][thread,smr] tid=2: Threads::add: new ThreadsList=0x0000000000969050
[0.196s][debug][thread,smr] tid=2: Threads::smr_free_list: threads=0x00000000008a3450 is freed.
[0.265s][debug][thread,smr] tid=2: Threads::add: new ThreadsList=0x0000000000aaa7e0
[0.265s][debug][thread,smr] tid=2: Threads::smr_free_list: threads=0x0000000000969050 is freed.
Hello world!
[0.286s][debug][thread,smr] tid=2: Threads::remove: new ThreadsList=0x0000000000aaab70
[0.286s][debug][thread,smr] tid=2: Threads::smr_free_list: threads=0x0000000000aaa7e0 is freed.
[0.286s][debug][thread,smr] tid=2: Threads::smr_delete: thread=0x0000000000433000 is deleted.
[0.286s][debug][thread,smr] tid=2: Threads::add: new ThreadsList=0x0000000000aaa7e0
[0.286s][debug][thread,smr] tid=2: Threads::smr_free_list: threads=0x0000000000aaab70 is freed.
[0.289s][info ][thread,smr] Threads class SMR info:
[0.289s][info ][thread,smr] _smr_java_thread_list=0x0000000000aaa7e0, length=9, elements={
[0.289s][info ][thread,smr] 0x00000000008a9800, 0x00000000008be000, 0x00000000008d2800, 0x00000000008d6800,
[0.289s][info ][thread,smr] 0x00000000008d8000, 0x00000000008da800, 0x0000000000997000, 0x0000000000aae000,
[0.289s][info ][thread,smr] 0x0000000000433000
[0.289s][info ][thread,smr] }
[0.289s][info ][thread,smr] _smr_java_thread_list_alloc_cnt=12, _smr_java_thread_list_free_cnt=11,
_smr_java_thread_list_max=9, _smr_nested_thread_list_max=0
[0.289s][info ][thread,smr] _smr_tlh_cnt=14, _smr_tlh_times=0, avg_smr_tlh_time=0.00, _smr_tlh_time_max=0
[0.289s][info ][thread,smr] _smr_deleted_thread_cnt=1, _smr_deleted_thread_times=0,
avg_smr_deleted_thread_time=0.00, _smr_deleted_thread_time_max=0
[0.289s][info ][thread,smr] _smr_delete_lock_wait_cnt=0, _smr_delete_lock_wait_max=0
[0.289s][info ][thread,smr] _smr_to_delete_list_cnt=0, _smr_to_delete_list_max=1
[0.289s][debug][thread,smr] tid=2: Threads::remove: new ThreadsList=0x0000000000aaad50
[0.289s][debug][thread,smr] tid=2: Threads::smr_free_list: threads=0x0000000000aaa7e0 is freed.
[0.289s][debug][thread,smr] tid=60: Threads::remove: new ThreadsList=0x0000000000aaa7e0
[0.289s][debug][thread,smr] tid=60: Threads::smr_free_list: threads=0x0000000000aaad50 is freed.
[0.289s][debug][thread,smr] tid=60: Threads::smr_delete: thread=0x00000000008d2800 is deleted.
```

It never ceases to amaze me how many Threads a simple Java program spins up; the '-XX:CICompilerCount=2' option is used to reduce the Compiler Thread count to 2 otherwise there would be many more lines.

Thread Dump Support

The CTRL-Break/SIGQUIT Thread dump output has been modified by this project.

Here's an example Thread dump with a simple sleeper program:

```
$ cat Sleeper.java
public class Sleeper {
    public static void main(String[] args) {
        System.out.println("Hello from Sleeper!");
        try {
            Thread.sleep(60 * 1000);
        } catch (InterruptedException ie) {
        }
    }
}

$ java -XX:+UseSerialGC -XX:CICompilerCount=2 Sleeper
Hello from Sleeper!
^\2017-09-15 11:13:24
Full thread dump Java HotSpot(TM) 64-Bit Server VM (10-internal+0-adhoc.dcubed.SMRprototype10 mixed mode):
```

Threads class SMR info:
_smr_java_thread_list=0x00000000007af360, length=9, elements={
0x0000000000431800, 0x00000000005a8800, 0x00000000005bd000, 0x00000000005d2000,
0x00000000005d3800, 0x00000000005d5800, 0x00000000005d7800, 0x0000000000672000,
0x00000000007b5000
}
_smr_java_thread_list_alloc_cnt=10, _smr_java_thread_list_free_cnt=9, _smr_java_thread_list_max=9,
_smr_nested_thread_list_max=0
_smr_tlh_cnt=18, _smr_tlh_times=0, avg_smr_tlh_time=0.00, _smr_tlh_time_max=0
_smr_delete_lock_wait_cnt=0, _smr_delete_lock_wait_max=0
_smr_to_delete_list_cnt=0, _smr_to_delete_list_max=1

```
"main" #1 prio=5 os_prio=64 tid=0x0000000000431800 nid=0x2 waiting on condition [0xffff80ffbf18e000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(java.base@9-internal/Native Method)
        at Sleeper.main(Sleeper.java:5)

"Reference Handler" #2 daemon prio=10 os_prio=64 tid=0x00000000005a8800 nid=0x4 waiting on condition
[0xffff80ffb7dfe000]
   java.lang.Thread.State: RUNNABLE
        at java.lang.ref.Reference.waitForReferencePendingList(java.base@9-internal/Native Method)
        at java.lang.ref.Reference.processPendingReferences(java.base@9-internal/Reference.java:174)
        at java.lang.ref.Reference.access$000(java.base@9-internal/Reference.java:44)
        at java.lang.ref.Reference$ReferenceHandler.run(java.base@9-internal/Reference.java:138)

"Finalizer" #3 daemon prio=8 os_prio=64 tid=0x00000000005bd000 nid=0x5 in Object.wait()
[0xffff80ffbecbe000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(java.base@9-internal/Native Method)
        - waiting on <0x00000003c120d070> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(java.base@9-internal/ReferenceQueue.java:151)
        - waiting to re-lock in wait() <0x00000003c120d070> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(java.base@9-internal/ReferenceQueue.java:172)
        at java.lang.ref.Finalizer$FinalizerThread.run(java.base@9-internal/Finalizer.java:216)

"Signal Dispatcher" #4 daemon prio=9 os_prio=64 tid=0x00000000005d2000 nid=0x6 waiting on condition
[0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" #5 daemon prio=9 os_prio=64 tid=0x00000000005d3800 nid=0x7 waiting on condition
[0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
   No compile task

"C1 CompilerThread1" #6 daemon prio=9 os_prio=64 tid=0x00000000005d5800 nid=0x8 waiting on condition
[0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
   No compile task

"Sweeper thread" #7 daemon prio=9 os_prio=64 tid=0x00000000005d7800 nid=0x9 runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
```

```
"Common-Cleaner" #8 daemon prio=8 os_prio=64 tid=0x0000000000672000 nid=0xa in Object.wait()
[0xffff80ffbd9ae000]
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(java.base@9-internal/Native Method)
        - waiting on <0x00000003c123b058> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(java.base@9-internal/ReferenceQueue.java:151)
        - waiting to re-lock in wait() <0x00000003c123b058> (a java.lang.ref.ReferenceQueue$Lock)
        at jdk.internal.ref.CleanerImpl.run(java.base@9-internal/CleanerImpl.java:148)
        at java.lang.Thread.run(java.base@9-internal/Thread.java:844)
        at jdk.internal.misc.InnocuousThread.run(java.base@9-internal/InnocuousThread.java:122)

"Service Thread" #9 daemon prio=9 os_prio=64 tid=0x00000000007b5000 nid=0xb runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"VM Thread" os_prio=64 tid=0x000000000059d800 nid=0x3 runnable

"VM Periodic Task Thread" os_prio=64 tid=0x00000000007b7800 nid=0xc waiting on condition

JNI global references: 6

Heap
def new generation   total 314560K, used 22369K [0x00000003c1200000, 0x00000003d6750000,
0x0000000516150000)
  eden space 279616K,   8% used [0x00000003c1200000, 0x00000003c27d8658, 0x00000003d2310000)
  from space 34944K,   0% used [0x00000003d2310000, 0x00000003d2310000, 0x00000003d4530000)
  to   space 34944K,   0% used [0x00000003d4530000, 0x00000003d4530000, 0x00000003d6750000)
tenured generation   total 699072K, used 0K [0x0000000516150000, 0x0000000540c00000, 0x00000007c0000000)
   the space 699072K,   0% used [0x0000000516150000, 0x0000000516150000, 0x0000000516150200,
0x0000000540c00000)
Metaspace       used 3976K, capacity 4486K, committed 4864K, reserved 1056768K
```

```
  class space    used 353K, capacity 386K, committed 512K, reserved 1048576K
```

```
^C
```

The '-XX:+UseSerialGC' and '-XX:CICompilerCount=2' options are used to reduce the Thread count for the example. The new output added by this project is shown in **bold** above:

> _smr_java_thread_list - shows that we have 9 Threads in the current ThreadsList; the elements are JavaThread ptrs.
>
> _smr_java_thread_list_alloc_cnt - shows that we've allocated 10 ThreadsLists up to this point.
>
> _smr_java_thread_list_free_cnt - shows we've freed 9 ThreadsLists up to this point; since the alloc and free counts only differ by one, there are no ThreadsLists on the to-be-deleted list.
>
> _smr_java_thread_list_max - shows that the most Threads on a ThreadsList has been 9.

If a Thread happens to have an active hazard ptr or an active hazard ptr and nested hazard ptr(s), then the Thread specific output would also show that info.

hs_err_pid Support

The hs_err_pid output has been modified by this project.

Here's an example way to generate an hs_err_pid file with an active ThreadsListHandle:

```
$ java -XX:+UseSerialGC -XX:CICompilerCount=2 -XX:ErrorHandlerTest=16 -XX:-CreateCoredumpOnCrash
# To suppress the following error report, specify this argument
# after -XX: or in .hotspotrc:  SuppressErrorAt=/debug.cpp:451
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  Internal Error
(/work/shared/bug_hunt/8167108/SMR_prototype_10/open/src/hotspot/share/utilities/debug.cpp:451), pid=24306,
tid=2
#  fatal error: Force crash with an active ThreadsListHandle.
#
# JRE version: Java(TM) SE Runtime Environment (10.0) (fastdebug build
9-internal+0-adhoc.dcubed.SMRprototype10)
# Java VM: Java HotSpot(TM) 64-Bit Server VM (fastdebug 10-internal+0-adhoc.dcubed.SMRprototype10, mixed
mode, tiered, compressed oops, serial gc, solaris-amd64)
# CreateCoredumpOnCrash turned off, no core file dumped
#
# An error report file with more information is saved as:
# /work/shared/bug_hunt/8167108/SMR_prototype_10/hs_err_pid24306.log
#
# If you would like to submit a bug report, please visit:
#   http://bugreport.java.com/bugreport/crash.jsp
#
```

Here's just the interesting parts of hs_err_pid24306.log:

```
<snip>
---------------  T H R E A D  ---------------


Current thread (0x000000000043c000):  JavaThread "main" [_thread_in_vm, id=2,
stack(0xffff80ffbf07f000,0xffff80ffbf17f000)] _threads_hazard_ptr=0x0000000000a6ea60

Stack: [0xffff80ffbf07f000,0xffff80ffbf17f000],  sp=0xffff80ffbf17e840,  free space=1022k
<snip>

---------------  P R O C E S S  ---------------

Threads class SMR info:
_smr_java_thread_list=0x0000000000a6ea60, length=9, elements={
0x000000000043c000, 0x00000000007ba800, 0x00000000007d3000, 0x00000000007f2800,
0x00000000007f5000, 0x00000000007f8000, 0x00000000007fa800, 0x00000000008b9000,
0x0000000000a6d000
}
_smr_java_thread_list_alloc_cnt=10, _smr_java_thread_list_free_cnt=9, _smr_java_thread_list_max=9,
_smr_nested_thread_list_max=0
_smr_tlh_cnt=13, _smr_tlh_times=0, avg_smr_tlh_time=0.00, _smr_tlh_time_max=0
_smr_delete_lock_wait_cnt=0, _smr_delete_lock_wait_max=0
_smr_to_delete_list_cnt=0, _smr_to_delete_list_max=1
```

```
Java Threads: ( => current thread )

=>0x000000000043c000 JavaThread "main" [_thread_in_vm, id=2, stack(0xffff80ffbf07f000,0xffff80ffbf17f000)]
_threads_hazard_ptr=0x0000000000a6ea60

  0x00000000007ba800 JavaThread "Reference Handler" daemon [_thread_blocked, id=4,
stack(0xffff80ffbed2e000,0xffff80ffbee2e000)]
<snip>
```

Here's an example way to generate an hs_err_pid file with an active nested ThreadsListHandle:

```
$ java -XX:+UseSerialGC -XX:CICompilerCount=2 -XX:ErrorHandlerTest=17 -XX:-CreateCoredumpOnCrash
# To suppress the following error report, specify this argument
# after -XX: or in .hotspotrc:  SuppressErrorAt=/debug.cpp:457
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  Internal Error
(/work/shared/bug_hunt/8167108/SMR_prototype_10/open/src/hotspot/share/utilities/debug.cpp:457), pid=2165,
tid=2
#  fatal error: Force crash with a nested ThreadsListHandle.
#
# JRE version: Java(TM) SE Runtime Environment (10.0) (fastdebug build
10-internal+0-adhoc.dcubed.SMRprototype10)
# Java VM: Java HotSpot(TM) 64-Bit Server VM (fastdebug 10-internal+0-adhoc.dcubed.SMRprototype10, mixed
mode, tiered, compressed oops, serial gc, solaris-amd64)
# CreateCoredumpOnCrash turned off, no core file dumped
#
# An error report file with more information is saved as:
# /work/shared/bug_hunt/8167108/SMR_prototype_10/hs_err_pid2165.log
#
# If you would like to submit a bug report, please visit:
#   http://bugreport.java.com/bugreport/crash.jsp
#
```

Here's just the interesting parts of hs_err_pid2165.log:

```
<snip>
---------------  T H R E A D  ---------------

Current thread (0x000000000043c000):  JavaThread "main" [_thread_in_vm, id=2,
stack(0xffff80ffbf08f000,0xffff80ffbf18f000)] _threads_hazard_ptr=0x0000000000a6e2d0 ,
_nested_threads_hazard_ptr_cnt=1, _nested_threads_hazard_ptrs=0x0000000000a6e2d0

Stack: [0xffff80ffbf08f000,0xffff80ffbf18f000],  sp=0xffff80ffbf18e840,  free space=1022k
<snip>

---------------  P R O C E S S  ---------------


Threads class SMR info:
_smr_java_thread_list=0x0000000000a6e2d0, length=9, elements={
0x000000000043c000, 0x00000000007ba800, 0x00000000007d3000, 0x00000000007f2800,
0x00000000007f5000, 0x00000000007f7800, 0x00000000007fa800, 0x00000000008b9800,
0x0000000000a6c800
}
_smr_java_thread_list_alloc_cnt=10, _smr_java_thread_list_free_cnt=9, _smr_java_thread_list_max=9,
_smr_nested_thread_list_max=1
_smr_tlh_cnt=13, _smr_tlh_times=0, avg_smr_tlh_time=0.00, _smr_tlh_time_max=0
_smr_delete_lock_wait_cnt=0, _smr_delete_lock_wait_max=0
_smr_to_delete_list_cnt=0, _smr_to_delete_list_max=1

Java Threads: ( => current thread )

=>0x000000000043c000 JavaThread "main" [_thread_in_vm, id=2, stack(0xffff80ffbf08f000,0xffff80ffbf18f000)]
_threads_hazard_ptr=0x0000000000a6e2d0 , _nested_threads_hazard_ptr_cnt=1,
_nested_threads_hazard_ptrs=0x0000000000a6e2d0

  0x00000000007ba800 JavaThread "Reference Handler" daemon [_thread_blocked, id=4,
stack(0xffff80ffbed3e000,0xffff80ffbee3e000)]
<snip>
```

The output added to hs_err_pid is identical to the output added to the Thread dump.

Section 10 - Gory Details

Behind the Threads::_smr_java_thread_list Abstraction

Here's the real code that manages the Threads::_smr_java_thread_list field:

```
  open/src/hotspot/share/runtime/thread.hpp:

static ThreadsList* volatile _smr_java_thread_list;
  static ThreadsList*          get_smr_java_thread_list() {
    return (ThreadsList*)OrderAccess::load_ptr_acquire((void* volatile*)&_smr_java_thread_list);
  }
  static ThreadsList*          xchg_smr_java_thread_list(ThreadsList* new_list) {
    return (ThreadsList*)Atomic::xchg_ptr((void*)new_list, (volatile void*)&_smr_java_thread_list);
  }
```

The Threads::_smr_java_thread_list field is marked volatile in order to tell the compiler to not do behind the scenes "optimizations" with it. This abstraction is pretty simple, just a getter and a setter.

Behind the Thread::_threads_hazard_ptr Abstraction

Here's the real code that manages the Thread::_threads_hazard_ptr field:

```
  open/src/hotspot/share/runtime/thread.hpp:

ThreadsList* volatile _threads_hazard_ptr;
  ThreadsList*          cmpxchg_threads_hazard_ptr(ThreadsList* exchange_value, ThreadsList* compare_value)
{
    return (ThreadsList*)Atomic::cmpxchg_ptr(exchange_value, (void* volatile*)&_threads_hazard_ptr,
compare_value);
  }
  ThreadsList*          get_threads_hazard_ptr() {
    return (ThreadsList*)OrderAccess::load_ptr_acquire((void* volatile*)&_threads_hazard_ptr);
  }
  void                  set_threads_hazard_ptr(ThreadsList* new_list) {
    OrderAccess::release_store_ptr_fence((void* volatile*)&_threads_hazard_ptr, (void*)new_list);
  }
  static bool           is_hazard_ptr_tagged(ThreadsList* list) {
    return (intptr_t(list) & intptr_t(1)) == intptr_t(1);
  }
  static ThreadsList*   tag_hazard_ptr(ThreadsList* list) {
    return (ThreadsList*)(intptr_t(list) | intptr_t(1));
  }
  static ThreadsList*   untag_hazard_ptr(ThreadsList* list) {
    return (ThreadsList*)(intptr_t(list) & ~intptr_t(1));
  }
```

The Thread::_threads_hazard_ptr field is marked volatile in order to tell the compiler to not do behind the scenes "optimizations" with it. This abstraction is more complicated than the one for Threads::_smr_java_thread_list. In addition to a getter and setter, we have a compare-and-exchange function and some functions to deal with tagging and untagging. If you have read "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects", then our use of tagging may seem a little out of place for an SMR implementation. The reason we have added a tag to our hazard ptr that protects the JavaThreads on a ThreadsList is because we sometimes need to know when the hazard ptr is stable so that we can follow the JavaThreads in the ThreadsList:

- ScanHazardPtrGatherProtectedThreadsClosure recognizes the tagged hazard pointer and attempts to clear it because it cannot be safely followed.
- ScanHazardPtrPrintMatchingThreadsClosure recognizes the tagged hazard pointer and can safely ignore it.
- ScanHazardPtrGatherThreadsListClosure recognizes the tagged hazard pointer and can safely use an untagged version of it.

Threads::acquire_stable_list_fast_path() is Lock Free

Threads::acquire_stable_list_fast_path() is lock free and therefore racy with other code that uses the same fields. acquire_stable_list_fast_path() is even more special in that it has two different styles of races with other code.

Acquire-Add or Acquire-Remove Race

The first race is between acquire_stable_list_fast_path() and Threads::add() or Threads::remove(); the racy code is almost the same so this only counts as one style of race. We refer to this race as the "acquire-add" or "acquire-remove" race.

Interesting points about the race:

- The code highlighted in red is the critical part.
- T1's call to get_smr_java_thread_list() may finish before or after T2's or T3's call to xchg_smr_java_thread_list().
- If T1 wins the race, then it can consider the _smr_java_thread_list value it has in the 'threads' variable to be "stable" and T1 can proceed to the "acquire-scan" race. T2's or T3's updates to the _smr_java_thread_list value do not need to be retried; they simply happen after T1's ld_acquire().
- If T2 or T3 wins the race, then T1 loops around and tries again; since 'threads' is not stable it cannot be published as a hazard pointer.

Notes:

- When T1 wins the race the fact that T1 has observed the current _smr_java_thread_list value as matching the 'threads' local variable is critical to the safety of this algorithm because it means that both T2 and T3 must also be able to observe the hazard ptr value that was stored in T1's _threads_hazard_ptr field. This makes the T2's and T3's subsequent calls to smr_free_list() safe.
- When T2 or T3 loses the race, T2 and T3 don't need to retry because they are simply setting the current _smr_java_thread_list value and the previous value gets added to the to-be-deleted list, i.e., the previous value is not lost.
- When T2 publishes a new _smr_java_thread_list value with a JavaThread added, that JavaThread will NOT be on the ThreadsList that T1 is using so that JavaThread would not be considered valid for T1's operation.
- When T3 publishes a new _smr_java_thread_list value with a JavaThread removed, that JavaThread will still be on the ThreadsList that T1 is using so that JavaThread will still be considered valid for T1's operation. That's the whole point of SMR!

T1: acquire_stable_list_fast_path()	T2: Threads::add()	T3: Threads::r
`ThreadsList* threads;`	`assert_locked_or_safepoint(Threads_lock);`	`MutexLocker`
`while (true) {`	`:`	`:`
`threads = get_smr_java_thread_list();`	`:`	`:`
`ThreadsList* unverified_threads = Thread::tag_hazard_ptr(threads);`	`:`	`:`
`self->set_threads_hazard_ptr(unverified_threads);`	`ThreadsList *new_list = ThreadsList::add_thread( get_smr_java_thread_list(), p);`	`ThreadsList ThreadsList get_smr_jav`
`if (get_smr_java_thread_list() != threads) {`	`ThreadsList *old_list = xchg_smr_java_thread_list(new_list);`	`ThreadsList xchg_smr_ja`
`continue;`	`smr_free_list(old_list);`	`smr_free_li`
`}`		
`if (self->cmpxchg_threads_hazard_ptr(threads, unverified_threads) == unverified_threads) {`		
`break;`		
`}`		
`}`		
`return threads;`		

Acquire-Scan Race

The second race is between acquire_stable_list_fast_path() and ScanHazardPtrGatherProtectedThreadsClosure::do_thread(). We refer to this race as the "acquire-scan" race.

Interesting points about the race:

- The code highlighted in red is the critical part.
- T4's call to cmpxchg_threads_hazard_ptr() may finish before or after T5's call to cmpxchg_threads_hazard_ptr().
- If T4 wins the race, then it has successfully published the hazard ptr and T5 will loop around and retry in order to include the stable hazard ptr in the scan.
- If T5 wins the race, then it has successfully NULL'ed out the tagged hazard ptr and can safely continue with its scan; T4 will loop around and retry.

Notes:

- The retries in both of these functions are critical to the algorithm's safety.
- acquire_stable_list_fast_path() cannot publish a hazard ptr that has been seen as tagged by the concurrent scan.
    - acquire_stable_list_fast_path() would be claiming the ThreadsList contents as protected.
    - The scan could be claiming the ThreadsList as unused which could result in the ThreadsList being deleted.
    - See "Analysis Round 2" below.
- Scan cannot skip following a hazard ptr that was just published for the same reasons.

T4: acquire_stable_list_fast_path()	**T5: ScanHazardPtrGatherProtectedThreadsClosure:: do_thread()**
	`assert_locked_or_safepoint(Threads_lock);`
`ThreadsList* threads;`	`if (thread == NULL) return;`
`while (true) {`	`ThreadsList *current_list = NULL;`
`threads = get_smr_java_thread_list();`	`while (true) {`
`ThreadsList* unverified_threads = Thread::tag_hazard_ptr(threads);`	`current_list = thread->get_threads_hazard_ptr();`
`self->set_threads_hazard_ptr(unverified_threads);`	`if (current_list == NULL) return;`
`if (get_smr_java_thread_list() != threads) {`	`if (!Thread::is_hazard_ptr_tagged(current_list)) {`
`continue;`	`break;`
`}`	`}`
`if (self->cmpxchg_threads_hazard_ptr(threads, unverified_threads) == unverified_threads) {`	`if (thread->cmpxchg_threads_hazard_ptr( NULL, current_list) == current_list) {`
`break;`	`return;`
`}`	`}`
`}`	`AddThreadHazardPointerThreadClosure add_cl(_table);`
`return threads;`	`current_list->threads_do(&add_cl);`

Fast JavaThreads List

The Threads class manages the current list of JavaThreads as a linked list (via the JavaThread::_next field, no_ JavaThread::_prev field). Walking the JavaThreads list via the linked list algorithm requires holding the Threads_lock. Some of the closures used by the SMR algorithms require walking an unchanging JavaThreads list so the Threads_lock is also held during these walks. Some of our advanced GC investigations also need to walk JavaThreads lists.

The ThreadsList class was added by this project for two primary reasons: 1) To provide a way to easily protect 1 -> N JavaThreads where N is dynamically determined at runtime, and 2) To provide a faster means of walking a JavaThreads list in order to reduce Threads_lock hold time.

There are two pieces to the "faster" implementation, 1) this inline function:

```
open/src/hotspot/share/runtime/thread.cpp:

static inline void *prefetch_and_load_ptr(void **addr, intx prefetch_interval) {
  Prefetch::read((void*)addr, prefetch_interval);
  return *addr;
}
```

and 2) this macro:

```
// Possibly the ugliest for loop the world has seen. C++ does not allow
// multiple types in the declaration section of the for loop. In this case
// we are only dealing with pointers and hence can cast them. It looks ugly
// but macros are ugly and therefore it's fine to make things absurdly ugly.
#define DO_JAVA_THREADS(LIST,
X)                                                                                            \
    for (JavaThread *MACRO_scan_interval =
(JavaThread*)(uintptr_t)PrefetchScanIntervalInBytes,                           \
             *MACRO_list =
(JavaThread*)(LIST),                                                                           \
             **MACRO_end = ((JavaThread**)((ThreadsList*)MACRO_list)->threads()) +
((ThreadsList*)MACRO_list)->length(),  \
             **MACRO_current_p =
(JavaThread**)((ThreadsList*)MACRO_list)->threads(),                                \
             *X = (JavaThread*)prefetch_and_load_ptr((void**)MACRO_current_p,
(intx)MACRO_scan_interval);                \
         MACRO_current_p !=
MACRO_end;                                                                                  \

MACRO_current_p++,
      \
             X = (JavaThread*)prefetch_and_load_ptr((void**)MACRO_current_p, (intx)MACRO_scan_interval))
```

For those of you that are very familiar with the code in threads.cpp, this was the old JavaThreads list macro:

```
#define ALL_JAVA_THREADS(X) for (JavaThread* X = _thread_list; X; X = X->next())
```

That macro is now restated in terms of the faster DO_JAVA_THREADS macro:

```
#define ALL_JAVA_THREADS(X) DO_JAVA_THREADS(get_smr_java_thread_list(), X)
```

The ALL_JAVA_THREADS macro is still used to walk the current list of JavaThreads (in threads.cpp). The DO_JAVA_THREADS macro is used to walk any ThreadsList, including Threads::_smr_java_thread_list (in threads.cpp).

Note: As ThreadsListHandles are applied throughout the system, we're converting existing JavaThreads list walkers (in thread.cpp) from using ALL_JAVA_THREADS (which requires holding the Threads_lock) to DO_JAVA_THREADS which does not require holding the Threads_lock. It may be possible for this project to completely obsolete the need for the Threads class linked list walking support. See the "Walking a ThreadsList" subsection for an example of how we're converting JavaThread list walkers outside of thread.cpp.

Memory Management With SMR

We've talked very little about the memory management aspects of the new SMR mechanism. Most of the mentions-in-passing are in the "Gory Details" section because memory management should be a gory detail. You have to get memory management right, but that is not what is critical to using SMR. The critical parts of SMR are discussed above and are pretty much written like we have an unchanging JavaThreads list that we sometimes have to protect in order to do some operation and then we unprotect it when we are done. In reality, JavaThreads are trivial to create and the simplest of programs can be multi-threaded without even realizing it.

Threads::add()	Add a JavaThread to current list of JavaThreads managed by the Threads class. This function is called with the Threads_lock held (or at a safepoint). **New with SMR**: this function creates a new ThreadsList that includes the current ThreadsList plus the specified JavaThread ptr, carefully swaps in the new ThreadsList and calls Threads::smr_free_list() on the now previous ThreadsList.
Threads::remove()	Remove a JavaThread from the current list of JavaThreads managed by the Threads class. This function grabs the Threads_lock. **New with SMR**: this function creates a new ThreadsList that includes the current ThreadsList minus the specified JavaThread ptr, carefully swaps in the new ThreadsList and calls Threads::smr_free_list() on the now previous ThreadsList.

Threads::smr_free_list()	In the pre-SMR system, the Threads class was used to manage *the current list* of JavaThreads. Yes, "*the current list*", one list, just one list, as in you always know what JavaThreads you have because there is one true list of JavaThreads in the system. Don't worry, SMR doesn't change the fact that we have one list that is *the current list* of JavaThreads.
	
	New with SMR: However (yes, the other shoe has to drop sometime), this project has found it useful to have copies of the current list of JavaThreads from various points in time. Since we can have more than one ThreadsList, we have to manage the memory occupied by those ThreadsLists and we have to do it safely. Enter Threads::smr_free_list()... this function is called by Threads::add() and Threads::delete() to get rid of the now previous ThreadsList; this function is called with the Threads_lock held (or at a safepoint). However, we can't just blindly "free" the ThreadsList because some other part of the SMR mechanism might be using that ThreadsList to protect one or more JavaThreads. The ThreadsList that is passed to Threads::smr_free_list() is always prepended to the _smr_to_delete_list. Next we walk the entire chain of ThreadsLists referred to by _smr_to_delete_list and we use closures to determine if that ThreadsList is not being referred to by a hazard ptr. Any ThreadsList that is not being used as a hazard ptr is freed.
	
	Notes:
	
	• A call to Threads::smr_free_list() may not free anything at all.
	• A call to Threads::smr_free_list() may free one or more ThreadsLists that are not the same as the ThreadsList specified in the current call.
	• What happens if an acquire_stable_list_fast_path() is racing with the closure that is determining if a ThreadsList is being used as a hazard ptr?
	    • This race is only interesting if the acquire_stable_list_fast_path() happens in a JavaThread that has already been processed by the closure.
	    • The Threads_lock is held by the caller of Threads::smr_free_list(), so the acquire_stable_list_fast_path() has to be dealing with the current _smr_java_threads_list value which is a ThreadsList that cannot be on the _smr_to_delete_list so there is no danger of the just acquired hazard ptr referring to a ThreadsList that is about to be freed.

Threads::_smr_to_delete_list	**New with SMR**: A linked list of ThreadsLists that need to be deleted when it is safe to do so. Threads::add() and Threads::remove() both add "previous" ThreadsLists to this list. A ThreadsList on this list is only kept alive as long as there is a hazard ptr that refers to the ThreadsList. Hazard ptrs are generally managed with the ThreadsListHandle helper object which is stack allocated so the "hold times" on a hazard ptr are short. Notes: • This is the risky part of SMR with respect to memory management; we have to get this right since memory leaks of this size would be catastrophic. • We have included logging code specifically targeted at reporting which JavaThreads have hazard ptrs (ThreadsLists) that are keeping an exiting JavaThread protected and unable to exit. • We have found some existing tests to be tremendously stressful on the new SMR Threads::_smr_to_delete_list and the associated closures; there are opportunities to measure and improve performance here.
JavaThread::smr_delete()	**New with SMR**: This is a wrapper function that is called to delete a JavaThread and generally replaces uses of "delete this" and "delete thread". This function calls Threads::smr_delete() on the JavaThread if it has appeared on a ThreadsList. Otherwise calls "delete this". We added this wrapper and the on_thread_list flag in order to not have to write rationale comments in all the places where calling "delete this" is safe to do.
Threads::smr_delete()	See the "Safely Deleting a JavaThread Reference" and "Is It Safe to Delete This JavaThread Reference" subsections above for a complete discussion of this function.

Double-Check Locking is Evil, But...

Double-check locking is a special form of a lock-free algorithm. It's hard to do it correctly and it requires quite a bit of analysis and reasoning. Unlike lock-free data structures (which I already admitted I hate), double-check locking fools the reader/reviewer into believing that a lock is grabbed only when needed, and when the lock isn't grabbed, then everything is safe and happy! In general, I think double-check locking is evil; not take-over-the-world evil, but just a little evil... Unfortunately, sometimes you have to put up with a little bit of evil...

In the SMR mechanism we use double-check locking in Threads::release_stable_list() in order to avoid lock-notify_all-unlock traffic on the Threads::smr_delete_lock(). Yes, the whole reason we created the Threads::smr_delete_lock() was to off load lock-operations traffic off of the Threads_lock. That doesn't mean that we want the Threads::smr_delete_lock() to have unnecessary lock-operations traffic.

Interesting points about the double-check locking race:

- The code highlighted in red is the critical part.
- T6 has the only hazard ptr that is protecting the JavaThread T7 is trying to delete (the only interesting case).
- If T6 manages to clear its hazard ptr before T7's scan in is_a_protected_JavaThread() reaches T6:
    - T6 will not been seen as protecting the target JavaThread ('thread') and T7 can clear the flag, unlock Threads::set_smr_delete_notify() and break out of the loop in order to delete 'thread'.

    - T7's Threads::set_smr_delete_notify() call published the flag-is-set value before doing the scan so T6 has to see the flag-is-set value after it has called set_threads_hazard_ptr(NULL).
    - T6 will block on Threads::smr_delete_lock()
    - When T6 manages to enter the lock, it will see the flag-is-not-set value that T7 set before T7 unlocked Threads::smr_delete_lock(), and T6 will skip the notify_all() call.
    - In this scenario, T7 deleted the 'thread' and T6 didn't have to do any notify_all operations.
- If T6 manages to clear its hazard ptr after T7's scan in is_a_protected_JavaThread() reached T6:
    - T6 will be seen as protecting the target JavaThread ('thread') and T7 will leave the flag-is-set value alone and call Threads::smr_delete_lock()->wait().
    - T6 will block on Threads::smr_delete_lock() until T7 calls wait().
    - When T6 manages to enter the lock, it will see the flag-is-set value and do the notify_all() call before unlocking Threads::smr_delete_lock().
    - T7 will return from the wait() call, clear the flag, unlock Threads::smr_delete_lock(), and loop around to retry the whole dance.
    - In this scenario, T6 did the notify_all operation only because we knew that T7 was waiting for it.

T6: Threads::release_stable_list()	T7: Threads::smr_delete()
	`while (true) {`
	`{`
	`MutexLockerEx ml(Threads_lock,...)`
	`grab Threads::smr_delete_lock()`
	`Threads::set_smr_delete_notify()`
`set_threads_hazard_ptr(NULL)`	`if (!is_a_protected_JavaThread(thread)) {`
`if (Threads::smr_delete_notify()) {`	`Threads::clear_smr_delete_notify()`
`MonitorLockerEx ml(Threads::smr_delete_lock(),...)`	`Threads::smr_delete_lock()->unlock()`
`: <block on enter>`	`break`
`:`	`}`
`:`	`} // drop the Threads_lock`
`:`	`Threads::smr_delete_lock()->wait(...)`
`: <enter lock>`	`: <waiting>`
`if (Threads::smr_delete_notify()) {`	`:`
`ml.notify_all()`	`: <notified>`
`}`	`: <block on reenter>`
`}  // drop Threads::smr_delete_lock()`	`:`
	`: <reentered lock>`
	`Threads::clear_smr_delete_notify()`
	`Threads::smr_delete_lock()->unlock()`
	`// Retry the whole scenario.`

	} // end while (true)

So the red marked code above identifies the critical parts of double-check locking race, but that doesn't explain why the double-check locking algorithm works in this case. The key is this line of code and its location in the smr_delete() algorithm:

```
Threads::set_smr_delete_notify()
```

which is unabstracted into this code:

```
void Threads::set_smr_delete_notify() {
  Atomic::inc(&_smr_delete_notify);
}
```

The atomic increment of the _smr_delete_notify field happens after T7 has grabbed the Threads::smr_delete_lock() and happens before T7 does the scan that races with T6's call to set_threads_hazard_ptr(NULL). That positioning guarantees that when T6 finishes its call to set_threads_hazard_ptr(NULL), it will see Threads::smr_delete_notify() return true and T6 will block trying to lock Threads::smr_delete_lock(), i.e., no lost notify_all() operation when we're in the double-check lock race window.

If T6's call to set_threads_hazard_ptr(NULL) happens any earlier on T7's code path, then T6 is not protecting the 'thread', T7 will not call wait() and T6 will not need to call notify_all().

If T6's call to set_threads_hazard_ptr(NULL) happens any later on T7's code path, then the second scenario comes into play, T7 will call wait() and block and T6 will call notify_all().

ThreadsListSetter - A Special Helper

In a perfect world, the ThreadsListHandle helper object would be enough to cover all the cases where SMR is needed. However, we're adding SMR to an existing body of code and that means there are corner cases where our nicely designed ThreadsListHandle helper object doesn't work. We've added a special helper class called ThreadsListSetter to solve the corner case that involves a Thread optionally delegating to another Thread an operation where Threads need to be protected. Here's the general use case:

- The requesting Thread (T1) creates the ThreadsListSetter helper object: 'setter'.
- T1 passes the ThreadsListSetter helper object to the delegate Thread (T2) and waits for T2 to do the work.
- If T2 can perform the operation where Threads need to be protected:
  - T2 calls 'setter->set()' to acquire a stable ThreadsList and store it in T1's hazard ptr field.
  - T2 performs the operation where Threads need to be protected, e.g., gathers a list of Threads.
  - T2 returns control to T1.
- T1 checks the results of the delegated operation:
  - If T2 did the work, sometimes T1 does additional work on results returned from T2.
  - If T2 didn't do the work, sometimes T1 will do the work itself after acquiring its own stable ThreadsList.
  - In either case, when T1 processes the data, the Threads are protected.
- When T1 is done, the ThreadsListSetter destructor releases the stable ThreadsList as needed.

ThreadsListSetter Example That Always Sets

Here are some example snippets of code to illustrate the usage when the setter is always used.

Notice the helper, _setter, is included in the VM operation that contains the field that needs protecting:

```
open/src/hotspot/share/runtime/vm_operations.hpp:

class VM_FindDeadlocks: public VM_Operation {
 private:
bool                _concurrent_locks;
  DeadlockCycle*     _deadlocks;
  outputStream*      _out;
  ThreadsListSetter _setter; // Helper to set hazard ptr in the originating thread
                             // which protects the JavaThreads in _deadlocks.
```

The VM operation's doit() function has a minor modification:

```
open/src/hotspot/share/runtime/vm_operations.cpp:

void VM_FindDeadlocks::doit() {

  // Update the hazard ptr in the originating thread to the current
  // list of threads. This VM operation needs the current list of
  // threads for proper deadlock detection and those are the
  // JavaThreads we need to be protected when we return info to the
  // originating thread.
```

```
  _setter.set();

   _deadlocks = ThreadService::find_deadlocks_at_safepoint(_setter.list(), _concurrent_locks);
```

The VM operation's call site has no changes! Because the ThreadsListSetter helper is embedded in VM_FindDeadlocks, the JavaThread ptrs that are in DeadlockCycle objects are protected until we're done with the VM_FindDeadlocks object:

```
open/src/hotspot/share/services/management.cpp:

static Handle find_deadlocks(bool object_monitors_only, TRAPS) {
  ResourceMark rm(THREAD);

  VM_FindDeadlocks op(!object_monitors_only /* also check concurrent locks? */);
  VMThread::execute(&op);

  DeadlockCycle* deadlocks = op.result();
  if (deadlocks == NULL) {
    // no deadlock found and return
    return Handle();
  }

  int num_threads = 0;
  DeadlockCycle* cycle;
  for (cycle = deadlocks; cycle != NULL; cycle = cycle->next()) {
    num_threads += cycle->num_threads();
  }

  objArrayOop r = oopFactory::new_objArray(SystemDictionary::Thread_klass(), num_threads, CHECK_NH);
  objArrayHandle threads_ah(THREAD, r);

  int index = 0;
  for (cycle = deadlocks; cycle != NULL; cycle = cycle->next()) {
    GrowableArray<JavaThread*>* deadlock_threads = cycle->threads();
    int len = deadlock_threads->length();
    for (int i = 0; i < len; i++) {
      threads_ah->obj_at_put(index, deadlock_threads->at(i)->threadObj());
      index++;
    }
  }
  return threads_ah;
}
```

ThreadsListSetter Example That Sometimes Sets

Here are some example snippets of code to illustrate the usage when the setter is sometimes used.

Again notice the helper, _setter, is included in the class that contains the field that needs protecting:

```
open/src/hotspot/share/services/threadService.hpp

class ThreadDumpResult : public StackObj {
private:
  int                    _num_threads;
  int                    _num_snapshots;
  ThreadSnapshot*        _snapshots;
  ThreadSnapshot*        _last;
  ThreadDumpResult*      _next;
  ThreadsListSetter      _setter;  // Helper to set hazard ptr in the originating thread
                                   // which protects the JavaThreads in _snapshots.
```

However, in this case, we also need a little more infrastructure since the _setter is optionally used:

```
  ThreadSnapshot*        snapshots()                       { return _snapshots; }

  void                   set_t_list()                      { _setter.set(); }
  ThreadsList*           t_list()                          { return _setter.list(); }
  bool                   t_list_has_been_set()             { return _setter.target_needs_release(); }

  void                   oops_do(OopClosure* f);
```

Here's an example of optional usage at the call site:

open/src/hotspot/share/services/management.cpp:

```
JVM_ENTRY(jint, jmm_GetThreadInfo(JNIEnv *env, jlongArray ids, jint maxDepth, jobjectArray infoArray))
<snip>
ThreadDumpResult dump_result(num_threads);

  if (maxDepth == 0) {

    // No stack trace to dump so we do not need to stop the world.
    // Since we never do the VM op here we must set the threads list.
    dump_result.set_t_list();

    for (int i = 0; i < num_threads; i++) {
      jlong tid = ids_ah->long_at(i);

      JavaThread* jt = dump_result.t_list()->find_JavaThread_from_java_tid(tid);

      ThreadSnapshot* ts;
      if (jt == NULL) {
        // if the thread does not exist or now it is terminated,
        // create dummy snapshot
        ts = new ThreadSnapshot();
      } else {

        ts = new ThreadSnapshot(dump_result.t_list(), jt);

      }
      dump_result.add_thread_snapshot(ts);
    }
  } else {
    // obtain thread dump with the specific list of threads with stack trace
    do_thread_dump(&dump_result,
                   ids_ah,
                   num_threads,
                   maxDepth,
                   false, /* no locked monitor */
                   false, /* no locked synchronizers */
                   CHECK_0);
  }

  int num_snapshots = dump_result.num_snapshots();
  assert(num_snapshots == num_threads, "Must match the number of thread snapshots");

  assert(num_snapshots == 0 || dump_result.t_list_has_been_set(), "ThreadsList must have been set if we
have a snapshot");

  int index = 0;
  for (ThreadSnapshot* ts = dump_result.snapshots(); ts != NULL; index++, ts = ts->next()) {
    // For each thread, create an java/lang/management/ThreadInfo object
    // and fill with the thread information

    if (ts->threadObj() == NULL) {
     // if the thread does not exist or now it is terminated, set threadinfo to NULL
      infoArray_h->obj_at_put(index, NULL);
      continue;
    }

    // Create java.lang.management.ThreadInfo object
    instanceOop info_obj = Management::create_thread_info_instance(ts, CHECK_0);
    infoArray_h->obj_at_put(index, info_obj);
  }
  return 0;
JVM_END
```

In the above example:

1. In the "if (maxDepth == 0)" code path, the JavaThread is able to do the work without a VM operation so it does its own ThreadsListSetter.set() via dump_result.set_t_list(), or
2. in the do_thread_dump() code path, the JavaThread creates a VM_ThreadDump operation and waits for the VMThread to execute it. If the VMThread executes the VM_ThreadDump operation, then it will call ThreadsListSetter.set() via dump_result.set_t_list().
3. In any case, the ThreadsListSetter that is embedded in dump_result will protect the JavaThread ptrs that are in ThreadSnapshot objects until we're done with the dump_result object.

Please note that we very carefully handle the case there are no ThreadsSnapshot objects present. In particular, it's possible in the do_thread_dump() code path for ThreadsListSetter.set() to not be called because there are no matching threads and thus no ThreadSnapshot objects to return.

NestedThreadsList - Another Special Helper

Events add another special corner case to our ThreadsListHandle helper object dominated world view. We've added another special helper class called NestedThreadsList to solve the corner case that involves an event handler calling an API that acquires a new stable ThreadsList when the calling Thread already has a stable ThreadsList in place. Here's the specific case that we ran into during the development phase:

1. The JVM/TI ClassFileLoadHook event handler is enabled.
2. A Thread (T1) is doing an M&M DumpAllThreads() call:
    a. T1 creates a ThreadsListHandle to protect the Thread snapshot.
    b. The M&M infrastructure for doing the Thread dump causes a class to be loaded.
    c. The ClassFileLoadHook event handler is invoked:
        i. The event handler calls JVM/TI SetThreadLocalStorage() which creates a ThreadsListHandle to protect the target Thread.

The more general case is this:

- A JVM/TI event can be enabled that causes an event handler to be invoked sometime after a ThreadsListHandle is created; this first ThreadsListHandle refers to a specific ThreadsList we'll call TL-1 (step 2-a in the specific example).
- Event handlers have few restrictions and because JVM/TI has quite a few APIs that take Thread parameters, those Threads have to be protected for the execution time of the specific JVM/TI API; this second ThreadsListHandle refers to a specific ThreadsList we'll call TL-2 (step 2-c-i in the specific example).
- The points at which TL-1 and TL-2 are stably acquired are asynchronous to each other which means TL-1 and TL-2 can have different relationships:
    - TL-1 == TL-2; exactly the same ThreadsList; no Threads added or removed between the two points.
    - contents(TL-1) == contents(TL-2); different ThreadsLists, but they contain the same entries; one or more Threads (TM...N) was added and TM...N were removed between the two points.
    - contents(TL-1) != contents(TL-2)
        - TL-1 contains one or more entries not in TL-2; one or more Threads (TM...N) were removed between the two points.
        - TL-2 contains one or more entries not in TL-1; one or more Threads (TM...N) were added between the two points.
        - TL-1 and TL-2 contain one or more entries that are not in the other ThreadsList; one or more Threads (TM...N) were removed between the two points; one or more Threads (TO...P) were added between the two points.

The key point to take away from the verbosity of the general case is that the two ThreadsLists (TL-1 and TL-2) need to be independent of each other because they may need to protect different sets of Threads. In other words, between step 2-a and step 2-c-i in the specific example, an arbitrary numbers of Threads can be added or removed from the system.

Nesting Should Be Rare

The use of NestedThreadsLists is expected to be rare for the following reasons:

- You have to be executing code that needs the initial ThreadsListHandle.
- The code you are executing after the ThreadsListHandle is created has to do something that is interesting enough to generate an event.
- You have to have an event handler in place and enabled for the event.
- The event handler has to call code that also needs a ThreadsListHandle.

Because we expect NestedThreadsList usage to be rare, we have used much simpler Threads_lock based techniques for managing the NestedThreadsLists. For example, out of the many thousands of tests run for this project, the need for nesting was only detected in one test.

Nesting Is Mostly Hidden

We have intentionally hidden the use of NestedThreadsLists as much as possible:

- ThreadsListHandle remains the preferred means of acquiring a stable ThreadsList.
- There is no NestedThreadsListHandle nor do we expect to need one.
- ThreadsListSetter does not support use of nesting intentionally:
    - A ThreadsListSetter is created in close proximity to the delegation object, e.g., a VM_Operation.
    - The Thread creating the delegation object typically passes flow control over to the delegation object quickly after creation and waits for the delegated operation to finish.
    - There is little chance (and little code to analyze) for the Thread that created the delegation object to do something interesting enough for an event to be generated before it blocks on waiting for the delegated operation to finish.
    - Once control flow is passed to the delegate Thread (usually the VMThread), any need for a stable ThreadsList is performed via the ThreadsListSetter object.

As mentioned before, the ThreadsListHandle constructor is very simple and is mostly a wrapper around a Threads::acquire_stable_list() call. Most of the time Threads::acquire_stable_list() calls Threads::acquire_stable_list_fast_path() and we execute all the crazy optimized algorithms described in detail above. When the need for nesting is detected by Threads::acquire_stable_list(), it calls Threads::acquire_stable_list_nested_path() which grabs the Threads_lock and inserts a NestedThreadsList object at the beginning of the list tracking NestedThreadsList objects for the calling Thread. Because we expect nesting to be rare, we haven't tried to create crazy optimized algorithms for managing the NestedThreadsList linked list.

As mentioned before, the ThreadsListHandle destructor is even simpler and is just a wrapper around a Threads::release_stable_list() call. Again, most of the time Threads::release_stable_list() calls Threads::release_stable_list_fast_path() and we execute all the crazy optimized algorithms described in detail above. When the presence of nesting is detected by Threads::release_stable_list(), it calls Threads::release_stable_list_nested_path() which grabs the Threads_lock and removes a NestedThreadsList object from the beginning of the list tracking NestedThreadsList objects for the calling Thread. Again, we expect nesting to be rare so we haven't tried heroic measures for managing the NestedThreadsList linked list.

Lock Free Algorithms Are Difficult to Get Right!

An intrpthrd003 run crashed. This subsection is included in this wiki as an attempt to illustrate the dangers of lock-free algorithms.

Note: The code quoted in this subsection of the wiki shows the version with the bug (for obvious reasons).

Here's a snippet of the hs_err_pid file:

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0xffff80ff45b35925, pid=19418, tid=122
#
# JRE version: Java(TM) SE Runtime Environment (9.0) (slowdebug build
9-internal+0-adhoc.dcubed.SMRprototype)
# Java VM: Java HotSpot(TM) 64-Bit Server VM (slowdebug 9-internal+0-adhoc.dcubed.SMRprototype, mixed mode,
tiered, compressed oops, g1 gc, solaris-amd64)
# Problematic frame:
# V  [libjvm.so+0x2335925]  void
ThreadsList::threads_do<AddThreadHazardPointerThreadClosure>(__type_0*)const+0x65
#
# Core dump will be written. Default location:
/work/shared/bug_hunt/8167108/SMR_prototype/build/solaris-x86_64-normal-server-slowdebug/test-results/hotsp
ot/tonga/nsk.jvmti_new/dcubed.SunOS.amd64/intrpthrd003/core or core.19418
#
# If you would like to submit a bug report, please visit:
#   http://bugreport.java.com/bugreport/crash.jsp
#

---------------  S U M M A R Y ------------

Command Line: -Xlog:os+thread+smr=debug -agentlib:intrpthrd003 nsk.jvmti.InterruptThread.intrpthrd003

Host: ddaugher-twvpn-dhcp-10-159-209-58, x86 64 bit 2400 MHz, 32 cores, 63G, Oracle Solaris 11.2 X86
Time: Fri May 26 14:32:22 2017 MDT elapsed time: 2 seconds (0d 0h 0m 2s)

---------------  T H R E A D  ---------------

Current thread (0x000000000168b000):  JavaThread "Thread-42" [_thread_in_vm, id=122,
stack(0xffff80ff730eb000,0xffff80ff731eb000)]

Stack: [0xffff80ff730eb000,0xffff80ff731eb000],  sp=0xffff80ff731ea8c0,  free space=1022k
Native frames: (J=compiled Java code, A=aot compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [libjvm.so+0x2335925]  void
ThreadsList::threads_do<AddThreadHazardPointerThreadClosure>(__type_0*)const+0x65
V  [libjvm.so+0x2335290]  void ScanHazardPtrGatherProtectedThreadsClosure::do_thread(Thread*)+0x80
V  [libjvm.so+0x232a1dd]  void Threads::threads_do(ThreadClosure*)+0xbd
V  [libjvm.so+0x232d313]  bool Threads::is_a_protected_JavaThread(JavaThread*)+0xf3
V  [libjvm.so+0x232d433]  void Threads::smr_delete(JavaThread*)+0xd3
V  [libjvm.so+0x231d5fb]  void JavaThread::smr_delete()+0x2b
V  [libjvm.so+0x23238f5]  void JavaThread::thread_main_inner()+0x2d5
V  [libjvm.so+0x23235f1]  void JavaThread::run()+0x2f1
V  [libjvm.so+0x20e89fa]  thread_native_entry+0x27a
C  [libc.so.1+0x125221]  _thrp_setup+0xa5
C  [libc.so.1+0x1254c0]  _lwp_start+0x0
```

The Crashing Thread

The crashing thread ("Thread-42" aka t@122) is in the smr_delete() call after it has finished running its Java code so this JavaThread is exiting and Threads::smr_delete()'ing itself.

The crashing thread is running this code:

```
src/share/vm/runtime/thread.inline.hpp:

inline void ThreadsList::threads_do(T *cl) const {
  const intx scan_interval = PrefetchScanIntervalInBytes;
  JavaThread *const *const end = _threads + _length;
  for (JavaThread *const *current_p = _threads; current_p != end; current_p++) {
    Prefetch::read((void*)current_p, scan_interval);
    JavaThread *const current = *current_p;  // SIGSEGV!
    threads_do_dispatch(cl, current);
  }
}
```

The current_p that we dereference caused a SIGSEGV and that came from the _threads field.

Here's 'this' in ThreadsList::threads_do():

```
(dbx) print this
this = 0x1688ec0

(dbx) print *this
*this = {
    _length        = 0
    _threads_index = 4059165169U
    _next_list     = (nil)
    _threads       = 0xabababababababab
}
```

so the current ThreadsList has been deleted (0xabababababababab is a slowdebug freed memory overwrite pattern).

Here's the code for ScanHazardPtrGatherProtectedThreadsClosure::do_thread():

```
  virtual void do_thread(Thread *thread) {
    assert_locked_or_safepoint(Threads_lock);

    if (thread == NULL) return;
    // We use load_ptr_acquire() here because we're going to look
    // inside the ThreadsList if we get one and we want a consistent
    // view of that memory.
    ThreadsList *current_list = (ThreadsList*)OrderAccess::load_ptr_acquire((void*
volatile*)&thread->_threads_hazard_ptr);
    if (current_list == NULL) return;

   // The current JavaThread has a hazard pointer (ThreadsList reference)
    // which might be _fast_java_thread_list or it might be an older
    // ThreadsList that has been removed but not freed. In either case,
    // the hazard pointer is protecting all the JavaThreads on that
   // ThreadsList.
   AddThreadHazardPointerThreadClosure add_cl(_table);
    current_list->threads_do(&add_cl);
  }
```

The local current_list matches the ThreadsList we dumped above:

```
(dbx) print thread
thread = 0x43e800
(dbx) print *thread
*thread = {
    _real_malloc_address                    = 0x43e5b0
    _threads_hazard_ptr                     = (nil)
    _SR_lock                                = 0x43fd10
    _suspend_flags                          = 0
    _num_nested_signal                      = 0
<snip>
}
```

so we fetched _threads_hazard_ptr into current_list and _threads_hazard_ptr has since been zero'ed out.

The JVM_Interrupt Thread

Here's the stack for the JavaThread we were examining with ScanHazardPtrGatherProtectedThreadsClosure::do_thread():

```
(dbx) thread t@2
t@2 (l@2) stopped in ___lwp_cond_wait at 0xffff80ffbf51e82a
0xffff80ffbf51e82a: ___lwp_cond_wait+0x000a:    jae      ___lwp_cond_wait+0x18  [ 0xffff80ffbf51e838, .+0xe
]
(dbx) where
current thread: t@2
  [1] ___lwp_cond_wait(0x440250, 0x440238, 0x0, 0x0, 0xffff80ffbf4efed8, 0x440200), at 0xffff80ffbf51e82a
  [2] _lwp_cond_wait(), at 0xffff80ffbf4efeec
  [3] os::Solaris::cond_wait(cv = 0x440250, mx = 0x440238), line 219 in "os_solaris.hpp"
  [4] os::PlatformEvent::park(this = 0x440200), line 5192 in "os_solaris.cpp"
  [5] ParkCommon(ev = 0x440200, timo = 0), line 411 in "mutex.cpp"
  [6] Monitor::ILock(this = 0x41d790, Self = 0x43e800), line 478 in "mutex.cpp"
  [7] Monitor::lock_without_safepoint_check(this = 0x41d790, Self = 0x43e800), line 957 in "mutex.cpp"
  [8] Monitor::lock_without_safepoint_check(this = 0x41d790), line 963 in "mutex.cpp"
  [9] MutexLockerEx::MutexLockerEx(this = 0xffff80ffbf12e570, mutex = 0x41d790, no_safepoint_check = true),
line 210 in "mutexLocker.hpp"
  [10] MonitorLockerEx::MonitorLockerEx(this = 0xffff80ffbf12e570, monitor = 0x41d790, no_safepoint_check =
true), line 232 in "mutexLocker.hpp"
=>[11] Threads::release_stable_list(self = 0x43e800), line 3607 in "thread.cpp"
  [12] ThreadsListHandle::~ThreadsListHandle(this = 0xffff80ffbf12e628), line 3635 in "thread.cpp"
  [13] JVM_Interrupt(env = 0x43ea78, jthread = 0xffff80ffbf12e710), line 3172 in "jvm.cpp"
  [14] 0xffff80ffa044262f(), at 0xffff80ffa044262f
<snip>
```

The above JavaThread is calling JVM_Interrupt() which is the code path we're stress testing...

Here's the Threads::release_stable_list() code:

```
void Threads::release_stable_list(Thread *self) {
  assert(self != NULL, "sanity check");

  // After releasing the hazard pointer, other threads may go ahead and free up some memory
  // temporarily used by a ThreadsList snapshot.
  OrderAccess::release_store_ptr_fence((void* volatile*)&self->_threads_hazard_ptr, NULL);

  // We use double-check locking to reduce traffic on the system
  // wide smr_delete_lock.
  if (Threads::smr_delete_notify()) {
    // An exiting thread might be waiting in smr_delete(); we need to
    // check with smr_delete_lock to be sure.
    //
    // Note: smr_delete_lock is held in smr_delete() for the entire
    // hazard pointer search so that we do not lose this notify() if
    // the exiting thread has to wait. That code path also holds
    // Threads_lock (which was grabbed before smr_delete_lock) so that
    // threads_do() can be called. This means the system can't start a
    // safepoint which means this thread can't take too long to get to
    // a safepoint because of being blocked on smr_delete_lock.
    //
    MonitorLockerEx ml(Threads::smr_delete_lock(), Monitor::_no_safepoint_check_flag);
    if (Threads::smr_delete_notify()) {
      // Notify any exiting JavaThreads that are waiting in smr_delete()
      // that we've released a ThreadsList.
      ml.notify_all();
      log_trace(os, thread, smr)("Threads::release_stable_list notified");
    }
  }
}
```

Our JVM_Interrupt thread is blocked on the smr_delete_lock and has just cleared its hazard pointer:

```
OrderAccess::release_store_ptr_fence((void* volatile*)&self->_threads_hazard_ptr, NULL);
```

so that explains why we see a NULL _threads_hazard_ptr field after ScanHazardPtrGatherProtectedThreadsClosure::do_thread() acquired a non-NULL _threads_hazard_ptr into current_list.

The Threads::smr_free_list() Thread

Here are the last few lines before the crash in doit.log:

```
INFO: thread #9: N_LATE_CALLS==1000 value is large enough to cause a Thread.interrupt() call after thread exit.
[2.000s][debug][os,thread,smr] Threads::add: new ThreadsList=0x0000000001688ec0
[2.000s][debug][os,thread,smr] Threads::smr_free_list: threads=0x000000000166c940 is freed.
[2.001s][debug][os,thread,smr] Threads::remove: new ThreadsList=0x000000000166c940
[2.001s][debug][os,thread,smr] Threads::smr_free_list: threads=0x0000000001688ec0 is freed.
```

so the ThreadsList being examined by the crashing Thread (0x1688ec0) was freed via Threads::smr_free_list() while we were examining it in ScanHazardPtrGatherProtectedThreadsClosure::do_thread(). The log doesn't tell us which thread called Threads::smr_free_list() but it was likely originated from a Threads::remove() call (previous log line). It is possible that the Threads::smr_free_list() call was made by the crashing thread itself when it removed itself from the threads list.

Analysis Round 1

Dan has a theory to explain how the Crashing Thread got a bad ThreadsList.

The JVM_Interrupt Thread sets its own _threads_hazard_ptr field to NULL asynchronously relative to the other threads in this dance.

Key points (for JVM_Interrupt Thread):

- It doesn't need any locks for that part of its code path...
- This must be the last reference to this particular ThreadsList in order for the Threads::smr_free_list() Thread to be able to free it.
- The JVM_Interrupt Thread's clearing of its _threads_hazard_ptr field has to be running in parallel with the Crashing Thread in order for the Crashing Thread to be able to acquire the non-NULL _threads_hazard_ptr value into the current_list local variable.

The JVM_Interrupt Thread called Threads::acquire_stable_list() (now called acquire_stable_list_fast_path()) to originally set its own _threads_hazard_ptr field to current ThreadsList and this is also done asynchronously.

Key points (again for JVM_Interrupt Thread):

- The acquire_stable_list() (now called acquire_stable_list_fast_path()) had to be done before the Threads::smr_free_list() Thread xchg_ptr()'ed the new ThreadsList for the to-be-deleted ThreadsList.

The Threads::smr_free_list() Thread is not nailed down to a specific thread because that action/event happened in the past so we only have a log message and no stack trace or calling thread identity.

Key points (for smr_free_list() Thread):

- assert_locked_or_safepoint(Threads_lock) was true when smr_free_list() was called.
- The specified ThreadsList was pre-pended to the to-be-freed list.
- ScanHazardPtrGatherThreadsListClosure was used to gather the current hazard ptrs (ThreadsList references) into scan_table.
- The current hazard pointers are not gathered via ld_ptr_acquire() which allows NULL to be seen.

  Here's the comment and code:

```
// We do not use load_ptr_acquire() here because we're not looking
// inside the ThreadsList; we only want the hazard pointer for
// comparison purposes.
ThreadsList *threads = thread->_threads_hazard_ptr;
```

- The to-be-freed list is walked and all ThreadsLists that are not mentioned in scan_table (in smr_free_list()) are freed.

The Crashing Thread acquires a non-NULL _threads_hazard_ptr field from the JVM_Interrupt Thread, saved it in the current_list local variable and then used current_list for a scan.

Key points (for Crashing Thread):

- assert_locked_or_safepoint(Threads_lock) is true at the time of the acquire.
- The Crashing Thread is exiting and it is trying to safely delete itself.
- Prior to its Thread::smr_delete() call it did a Threads::remove() call which may have resulted in the Threads::smr_free_list() call that deleted the ThreadsList.

The following transaction diagram shows the thread interactions:

JVM_Interrupt Thread	smr_free_list() Thread	Crashing Thread
:	Threads::remove()	
:	grab Threads_lock	
acquire_stable_list()	:	
<do work>	xchg_ptr()	

:	`smr_free_list() {`	
:	`ThreadScanHashtable *scan_table = ...`	
:	`ScanHazardPtrGatherThreadsListClosure scan_cl(scan_table)`	
:	`Threads::threads_do(&scan_cl) {`	

:
:
:
:
:
	`// Closure's do_thread() does not`
	`// acquire() _threads_hazard_ptr.`
	`// We may see NULL here instead of`
	`// the non-NULL _threads_hazard_ptr!`
	`}`	

:	`while (current != NULL) {  // walk to-be-freed-list`	
:	`if (!used)`	
:	`delete current`	
:	`}`	
:	`}`	
:	`drop Threads_lock`	
:	:	`Threads::smr_delete() {`
:		`grab Threads_lock`
:		`is_a_protected() {`
		`ThreadScanHashtable *scan_table = ...`
:		`ScanHazardPtrGatherProtectedThreadsClosur (scan_table)`
:		`Threads:threads_do() {`
:		`ScanHazardPtrGatherProtectedThreadsClosure::d {`
:		`current_list = acquire(_threads_hazar`
`release_stable_list()`		:
		`ThreadsList:threads_do() {`
		`current = *current_p; // SIGSEGV!`

The above scenario is nicely illustrated, but it's wrong! The above analysis proposes that a NULL to non-NULL transition was missed because of a missing ld_ptr_acquire(). The code that transitioned the _threads_hazard_ptr from NULL to non-NULL was:

```
OrderAccess::release_store_ptr_fence((void* volatile*)&self->_java_threads_do_hp, (void*)threads);
```

Because of the fence part in the above call, a ld_ptr_acquire() in ScanHazardPtrGatherThreadsListClosure::do_thread() is not necessary for the NULL to non-NULL transition to be seen. The theory illustrated by the above transaction diagram hinges on the NULL pointer being seen by smr_free_list() Thread while the non-NULL pointer is seen by JVM_Interrupt() Thread and the Crashing Thread. The theory is proven wrong by eliminating the possibility of two different views of the _threads_hazard_ptr value.

Analysis Round 2

Erik has come up with a different theory to explain how the Crashing Thread got a bad ThreadsList.

The JVM_Interrupt Thread calls Threads::acquire_stable_list() (now called acquire_stable_list_fast_path()) which loops until it acquires a validated hazard pointer (ThreadsList).

Key points (for JVM_Interrupt Thread):

- acquire_stable_list() (now called acquire_stable_list_fast_path()) doesn't use any locks
- the current threads list is stored in the 'threads' local
  - This has to happen before the smr_free_list() Thread has updated the current threads list to a new value.
  - The delay time between storing the current threads list in the 'threads' local and publishing the 'threads' local as a hazard ptr in the _threads_hazard_ptr field is arbitrary:
    - The current threads list (_smr_java_thread_list) could have changed just once or any number of times.
    - The value we have in the 'threads' local could still be the same ThreadsList that hasn't been deleted yet, or it could be the same ThreadsList that has been deleted (our scenario), or it could be another ThreadsList that has been reallocated at the same start memory address (A-B-A problem), or it could be freed memory, or it could be something else entirely.
    - The important point is we have a hazard ptr in the 'threads' local that we have not yet validated (critical step 2), but we published it.
- release_store_ptr_fence() is used to publish the 'threads' local
  - This has to happen after the smr_free_list() Thread has done its scan for in-use hazard pointers.
  - This has to happen before the Crashing Thread has set its 'current_list' local variable.
- ld_ptr_acquire() is used to validate the 'threads' value against the current threads list
- if the current threads list has changed, then we loop around and try again

The Threads::smr_free_list() Thread is not nailed down to a specific thread because that action/event happened in the past so we only have a log message and no stack trace or calling thread identity.

Key points (for smr_free_list() Thread):

- assert_locked_or_safepoint(Threads_lock) was true when smr_free_list() was called
- The specified ThreadsList was pre-pended to the to-be-freed list.
- ScanHazardPtrGatherThreadsListClosure was used to gather the current hazard ptrs (ThreadsList references) into scan_table.
- The JVM_Interrupt Thread has not yet put a value in its _threads_hazard_ptr field.
- The to-be-freed list is walked and all ThreadsLists that are not mentioned in scan_table (in smr_free_list()) are freed.

The Crashing Thread acquires a non-NULL _threads_hazard_ptr field from the JVM_Interrupt Thread, saved it in the current_list local variable and then used current_list for a scan.

Key points (for Crashing Thread):

- assert_locked_or_safepoint(Threads_lock) is true at the time of the acquire.
- The Crashing Thread is exiting and it is trying to safely delete itself.
- Prior to its Thread::smr_delete() call it did a Threads::remove() call which may have resulted in the Threads::smr_free_list() call that deleted the ThreadsList.

The following transaction diagram shows the thread interactions:

JVM_Interrupt Thread	smr_free_list() Thread	Crashing Thread
`acquire_stable_list() {`	`Threads::remove() {`	
`do {`	`  grab Threads_lock`	
`threads = _smr_java_thread_list;`	`  new_list = ...`	
`: // _smr_java_thread_list changed!`	`  xchg_ptr(new_list, &_smr_java_thread_list)`	
`    : // But we'll detect it and fix it on the next loop...`	`  smr_free_list() {`	
`    :`	`    ThreadScanHashtable *scan_table = ...`	
`    :`	`ScanHazardPtrGatherThreadsListClosure scan_cl(scan_table)`	
`    :`	`    Threads::threads_do(&scan_cl)`	
`    :`	`    while (current != NULL) {  // walk to-be-freed-list`	
`    : // We haven't stored 'threads' yet so the`	`      if (!used)`	

: // smr_free_list() thread deletes it...	delete current	
:	}	
:	}	
:	drop Threads_lock	Threads::smr_de
:		grab Threads_
:		is_a_protecte
		ThreadScanH
:		ScanHazardP (scan_table)
:		Threads:thr
release_store_ptr_fence(&self->_threads_hazard_ptr, threads);		ScanHazardPtrGa {
// The 'threads' value that is stored in _threads_hazard_ptr // above is validated by load_ptr_acquire() below, but // ld_ptr_acquire() in the crashing thread can get that bad // value and use it for a scan before we loop around and // replace it with a value that is going to validate.		current_ ld_ptr_acquire(// We'v _threads_hazard // here because // we t
} while (load_ptr_acquire(&_smr_java_thread_list) != threads);		current_
}		for (_threads; curre
<do work>		cur
release_stable_list()		
// The above release_stable_list() will set the // _threads_hazard_ptr field to NULL.		

The above theory has been through several rounds of design review and the fix has been through a few rounds of code review. The summary of the fix is:

- Change Threads::acquire_stable_list() (now called acquire_stable_list_fast_path()) to tag a hazard pointer that has not yet been verified to be a stable hazard ptr.
- Change the scanning threads to recognize tagged hazard pointers and deal with them appropriately.
  - ScanHazardPtrGatherProtectedThreadsClosure recognizes the tagged hazard pointer and attempts to clear it because it cannot be safely followed.
  - ScanHazardPtrPrintMatchingThreadsClosure recognizes the tagged hazard pointer and can safely ignore it.
  - ScanHazardPtrGatherThreadsListClosure recognizes the tagged hazard pointer and can safely use an untagged version of it.

The relevant code quoted in the other subsections of this wiki that are not related to this bug have been updated to show the fixed version of the code.

See "Acquire-Scan Race" above for the transaction diagram that shows how the race was fixed.

Section 11 - Performance Testing Ideas

nsk/monitoring/stress/thread/cmon001 Might Provide Interesting Data

At one point in the this project, we ran into a big performance problem with the nsk/monitoring/stress/thread/cmon00[1-3] tests.

Here are some interesting points:

- cmon001 is the real test.
- cmon00[23] are wrappers around cmon001 with different options.
- The test launches 4000 threads per iteration for 50 iterations so 20000 threads over a single test run.
- We should be able to measure a lot of different things with this test.

Revisit Threads::is_a_protected_JavaThread()

When Dan chased down the performance problem with the cmon001-3 tests, he had an idea to improve performance by changing just is_a_protected_JavaThread() to use the linear scan because it could return "true" on first sighting instead of having to gather all the JavaThread pointers after gathering all the hazard pointers... This idea should be revisited at some point.

Section 12 - Remaining Work

This section lists the work that remains to be done to complete this project:

- Finish examining other JavaThread pointer uses to determine if ThreadListHandles are needed.
  - Original searches were done with:
    - java_lang_Thread::thread()
    - Threads::first()
    - Threads:find_java_thread_from_java_tid()
    - Threads:includes()
  - What other coding patterns might yield JavaThread* usage?