JDK Flight Recorder
How to root cause Stop-the-World pauses

Liberica JDK
Revision 1.0
October 17, 2023

bellsoft

BellSoft software contains open source software. Additional information about third party code is available at https://bell-sw.com/third_party_licenses. You can also get more information on how to get a copy of source code by contacting info@bell-sw.com.

THIS INFORMATION MAY CHANGE WITHOUT NOTICE. TO THE EXTENT PERMITTED BY APPLICABLE LAW, BELLSOFT PROVIDES THIS DOCUMENTATION “AS IS” WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT WILL BELLSOFT BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, LOST INVESTMENT, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF BELLSOFT IS EXPRESSLY ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE.

The use of any software product referenced in this document is governed by the applicable license agreement, which is not modified in any way by the terms of this notice.

Alpaquita, Liberica and BellSoft are trademarks or registered trademarks of BellSoft Corporation. The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Java and OpenJDK are trademarks or registered trademarks of Oracle and/or its affiliates. Other trademarks are the property of their respective owners and are used only for identification purposes.

Contents

1. Introduction

This document focuses on Stop-the-World pauses in the JVM and how JDK Flight Recorder (JFR) can help you identify the root cause.

Before moving on to details, we need to make some clarifications on terminology. The Stop-the-World (STW) pause is a general term for a state of the JVM when all application threads are suspended for the duration of a specific internal JVM operation.

In the HotSpot JVM used in OpenJDK, there is a more specific term: **safepoint**.

Citing the HotSpot Glossary, a safepoint is defined as:

> "*A point during program execution at which all GC roots are known and all heap object contents are consistent. From a global point of view, all threads must block at a safepoint before the GC can run. (As a special case, threads running JNI code can continue to run, because they use only handles. During a safepoint they must block instead of loading the contents of the handle.) From a local point of view, a safepoint is a distinguished point in a block of code where the executing thread may block for the GC. Most call sites qualify as safepoints. There are strong invariants which hold true at every safepoint, which may be disregarded at non-safepoints. Both compiled Java code and C/C code can be optimized between safepoints, but less so across safepoints. The JIT compiler emits a GC map at each safepoint. C/C code in the VM uses stylized macro-based conventions (e.g., TRAPS) to mark potential safepoints.*"

Safepoints are utilized not only for GC but also for other JVM operations. Below are key categories of JVM operations causing safepoints:

- Garbage collector related;
- JIT compiler related;
- Bias locking related;
- Diagnostic related (e.g., thread dump, heap dump, profiling, and debugging).

Besides explicit VM operations, certain cleanup tasks related to internal JVM housekeeping are delayed until the next safepoint. As there is no guarantee that a VM operation will run regularly, safepoint could be initiated without any explicit VM operation.

Effective pause time associated with a safepoint can be split into three components:

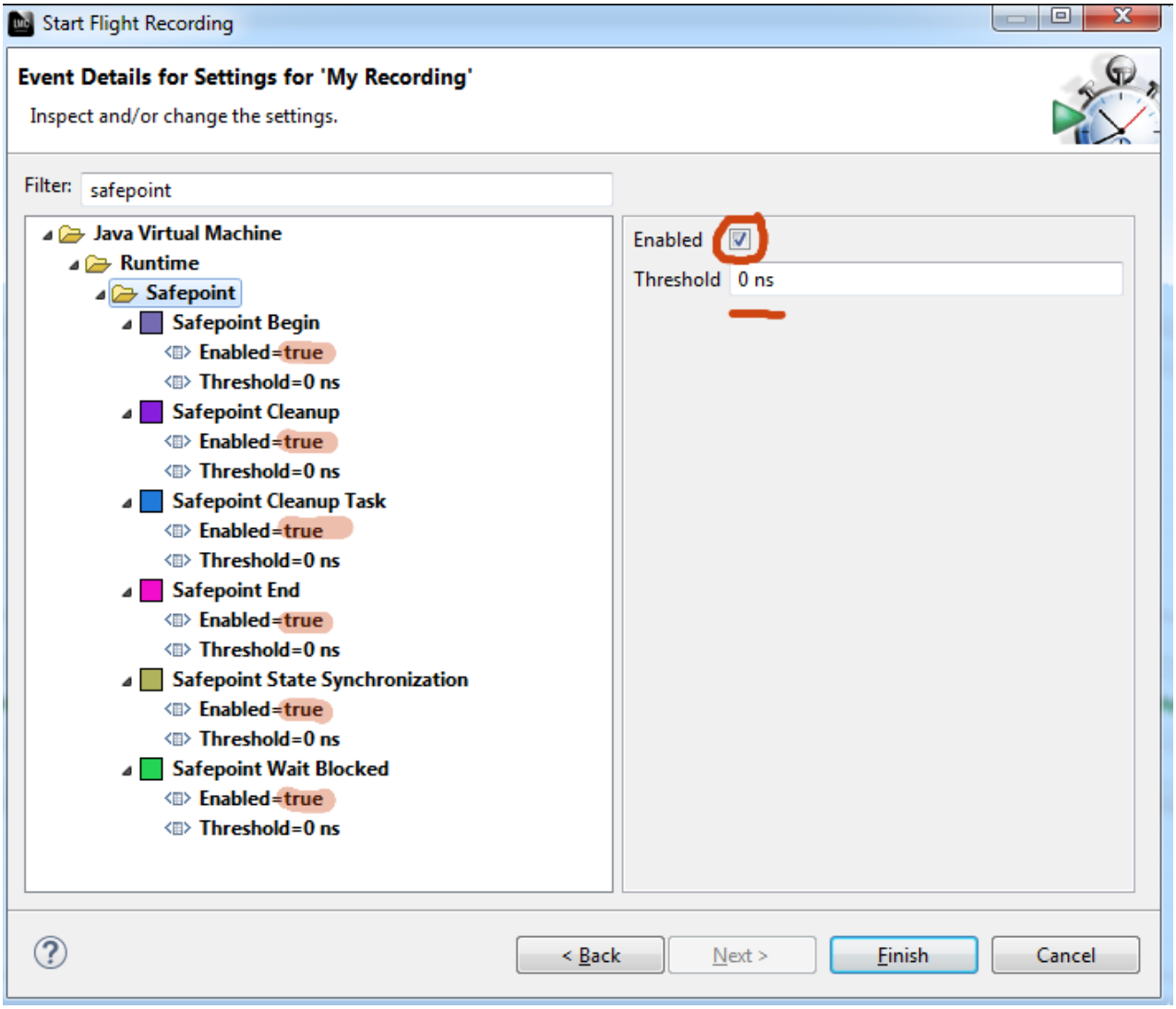
- Entering safepoint time — the JVM needs all application threads to arrive at a safepoint, which does not happen instantly. This period is unrelated to the type of operation to be executed at a

safepoint.

- Executing tasks pending for the next safepoint.
- VM operation time — operations obviously need some time to complete. This component is operation dependent.

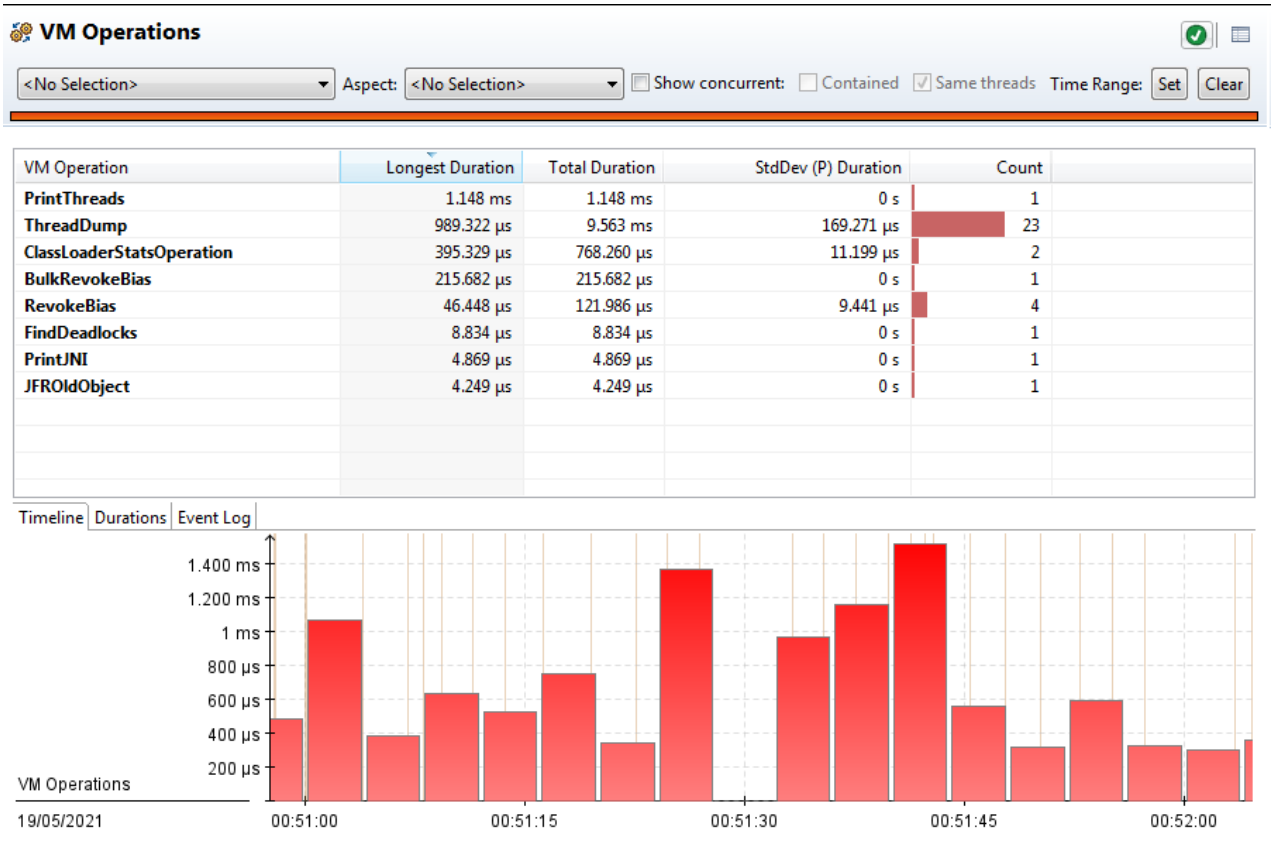
2. Safepoints in JDK Flight Recorder

JDK Flight Recorder has multiple events related to safepoints, yet some are disabled by default. You can enable a safepoint-specific event in the **Start Flight Recording** window. Switch to low-level configuration mode, type **safepoint** in the filter, and select **Enabled** next to all options.



Once a recording is done, you can find the following views with information related to safepoints.

The standard report available in Mission Control is **VM Operations** under the **JVM Internals** category.

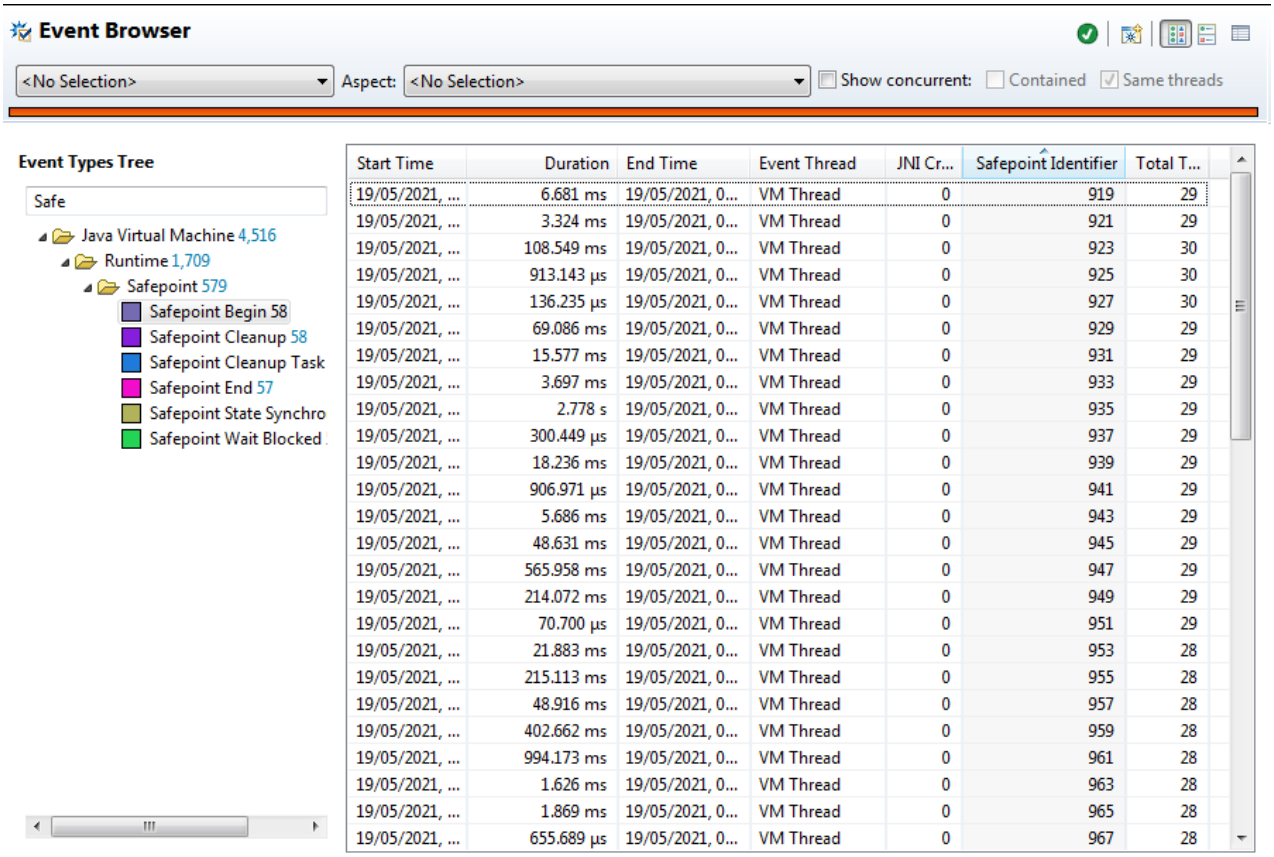


This report presents a summary of VM operations grouped by type. It is also possible to see individual events on the **Event log** tab.

Keep in mind that event durations in this report are just durations of VM operations and do not reflect the duration of the Stop-the-World pause.

Other safepoint-related events do not have predefined reports in Mission Control so far. We will use **Event Browser** to view them. You can enter **safepoint** into the filter on top of the event type tree to see relevant events.

All event types in the screenshot below are disabled by default, so you need to enable them explicitly before starting Flight Recorder.



Start Time	Duration	End Time	Event Thread	JNI Cr...	Safepoint Identifier	Total T...
19/05/2021, ...	6.681 ms	19/05/2021, 0...	VM Thread	0	919	29
19/05/2021, ...	3.324 ms	19/05/2021, 0...	VM Thread	0	921	29
19/05/2021, ...	108.549 ms	19/05/2021, 0...	VM Thread	0	923	30
19/05/2021, ...	913.143 µs	19/05/2021, 0...	VM Thread	0	925	30
19/05/2021, ...	136.235 µs	19/05/2021, 0...	VM Thread	0	927	30
19/05/2021, ...	69.086 ms	19/05/2021, 0...	VM Thread	0	929	29
19/05/2021, ...	15.577 ms	19/05/2021, 0...	VM Thread	0	931	29
19/05/2021, ...	3.697 ms	19/05/2021, 0...	VM Thread	0	933	29
19/05/2021, ...	2.778 s	19/05/2021, 0...	VM Thread	0	935	29
19/05/2021, ...	300.449 µs	19/05/2021, 0...	VM Thread	0	937	29
19/05/2021, ...	18.236 ms	19/05/2021, 0...	VM Thread	0	939	29
19/05/2021, ...	906.971 µs	19/05/2021, 0...	VM Thread	0	941	29
19/05/2021, ...	5.686 ms	19/05/2021, 0...	VM Thread	0	943	29
19/05/2021, ...	48.631 ms	19/05/2021, 0...	VM Thread	0	945	29
19/05/2021, ...	565.958 ms	19/05/2021, 0...	VM Thread	0	947	29
19/05/2021, ...	214.072 ms	19/05/2021, 0...	VM Thread	0	949	29
19/05/2021, ...	70.700 µs	19/05/2021, 0...	VM Thread	0	951	29
19/05/2021, ...	21.883 ms	19/05/2021, 0...	VM Thread	0	953	28
19/05/2021, ...	215.113 ms	19/05/2021, 0...	VM Thread	0	955	28
19/05/2021, ...	48.916 ms	19/05/2021, 0...	VM Thread	0	957	28
19/05/2021, ...	402.662 ms	19/05/2021, 0...	VM Thread	0	959	28
19/05/2021, ...	994.173 ms	19/05/2021, 0...	VM Thread	0	961	28
19/05/2021, ...	1.626 ms	19/05/2021, 0...	VM Thread	0	963	28
19/05/2021, ...	1.869 ms	19/05/2021, 0...	VM Thread	0	965	28
19/05/2021, ...	655.689 µs	19/05/2021, 0...	VM Thread	0	967	28

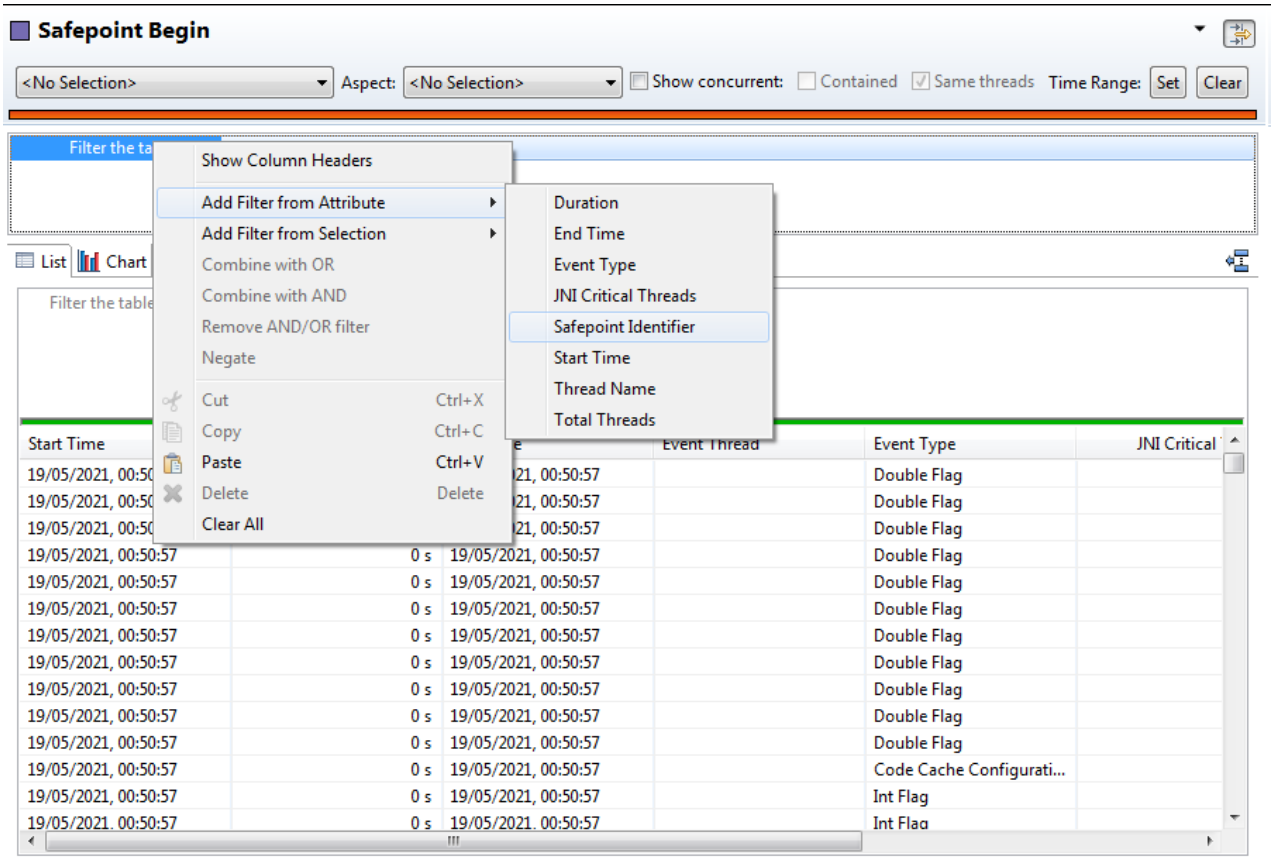
The **Safepoint Begin** event is the one stirring the most interest: its duration is the total duration of the Stop-the-World pause.

Information about safepoint is scattered among multiple events. Regardless, different events related to the same safepoint could be correlated by the **Safepoint Identifier** column.

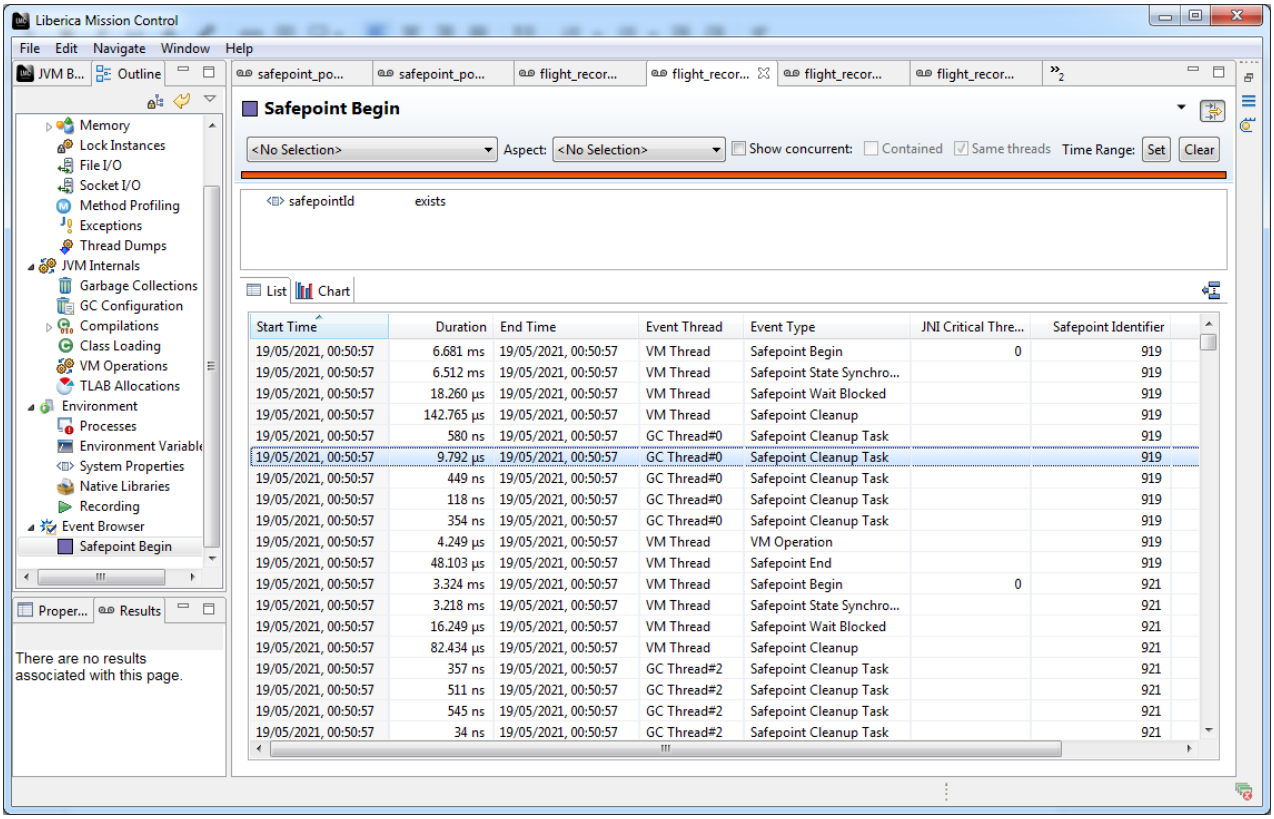
We can use custom filters to make digging through safepoint details more ergonomic.

Right-click on the **Event Browser** node at the **Outline** side view on the left and choose **New Page** > **Custom Page**, then select the **Safepoint Begin** event type to create a new custom report.

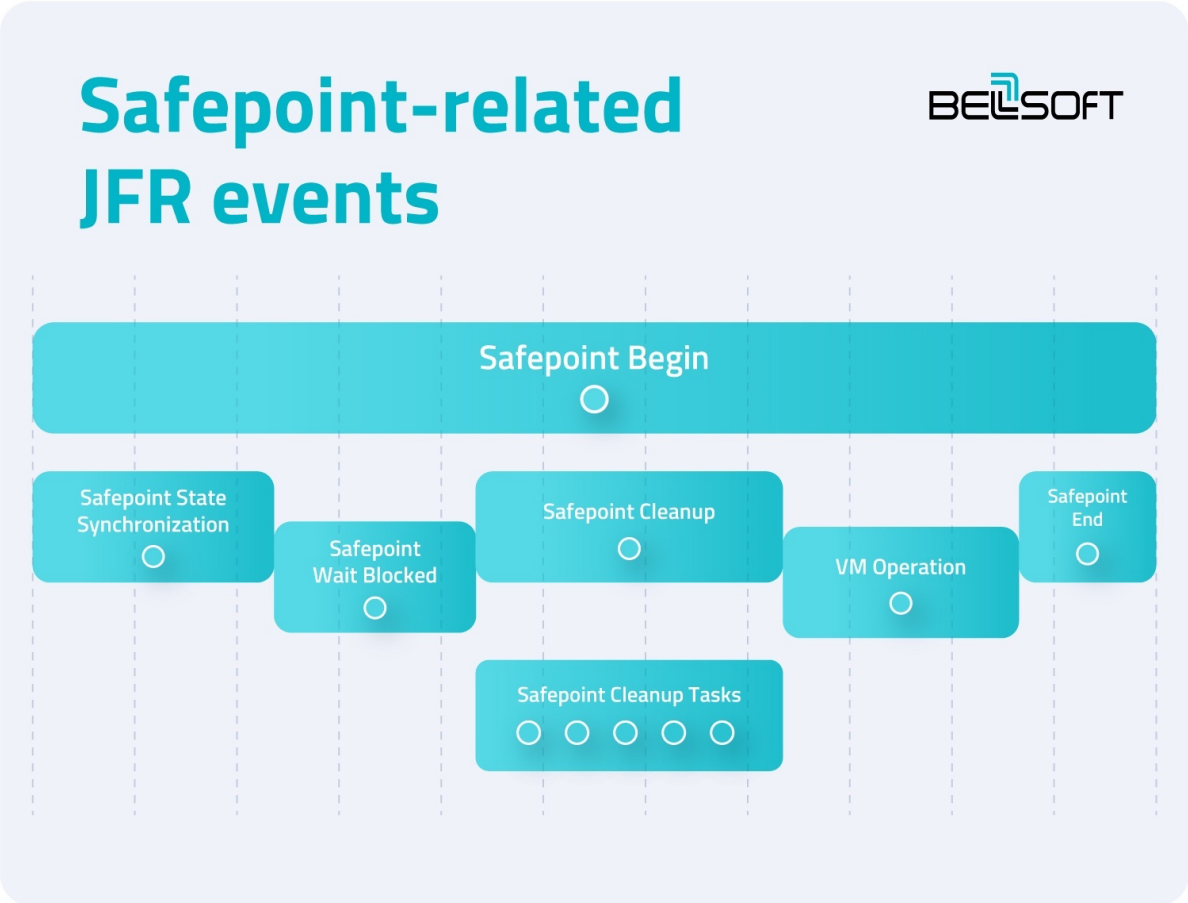
A new node will appear in the **Outline** side view; select it. You will see a report showing only **Safepoint Begin** events. Now, right-click and delete the existing filter node in the upper section of the report. Instead of filtering events by type, we will filter all events with the `Safepoint Identifier` attribute. Right-click and choose **Add Filter from Attribute** > **Safepoint Identifier**. Lastly, change the type of the created filter node from == to **exists**.



The event log (section at the bottom) contains all safepoint-related events, including the **VM Operation** event. Now sort by **Start Time**, and all events associated with the same safepoint will be grouped.



The diagram below shows relations between different JFR events affiliated with a safepoint.



Normally the VM operation is the main factor defining the duration of the STW pause. But sometimes, phases before the operation turn out to be problematic.

3. Entering the safepoint

Entering the safepoint is a fairly sophisticated process. The JVM needs to stop every Java thread at the point where all local variables are predictably stored at stack (many VM operations need to traverse stack for heap references). A cooperative safepoint protocol is used to coordinate running threads to enter a safepoint state.

Generally, each Java thread could be in one of the following conditions:

- A thread is blocked with API in the runtime library (it's not in the RUNNABLE state). In this case, no additional actions are required.
- A thread is RUNNABLE and is running in the interpreter. In this case, the thread could be suspended on the next bytecode operation. The JVM replaces the interpreter jump table with special jump table variants where each byte code index is mapped to enter a safepoint routine.
- A thread is RUNNABLE and is running JIT-compiled code. This is the most tricky scenario — the compiler could do a lot of operation reordering and load/store optimizations. The compiler has to emit special safepoint checks. The thread will suspend itself once it reaches the next checkpoint after the safepoint protocol is initiated by the JVM.
- A thread is RUNNABLE and is running native code via JNI. Such threads don't need to be stopped, and they may continue to run during the JVM Stop-the-World pause. However, if this kind of a thread returns to calling Java method, calls Java method, or tries to access data in Java heap via JNI API, it will be blocked until the safepoint ends.
- A thread is RUNNABLE and is running JVM intrinsic code or the `JavaCritical` JNI method. Such thread cannot be stopped until it will complete the current operation and enter the JIT-compiled code section.

All application threads need to be stopped. Until each and every thread enters a safepoint, the operation cannot start. Or put it another way—even if just one thread is stuck and isn't able to enter for some time, it will effectively freeze the JVM.

4. Why it may take a long time to enter a safepoint

All Java threads in the RUNNABLE state should enter a safepoint. Although RUNNABLE, in JVM terms, does not necessarily mean that a thread is actually running on CPU at the moment.

The RUNNABLE thread in Java could be:

1. actually running on a CPU,
2. scheduled to run by the OS but waiting for a CPU slice,
3. blocked due to blocking the OS call,
4. blocked due to a page fault.

Case 1 is the normal condition. The thread can proceed with the safepoint protocol.

Case 2 is possible if the host is starved of CPU resources. In Linux, it's hard to monitor how long threads spend in the scheduler queue. Therefore, we can rely only on indirect metrics like total host CPU usage for post mortem diagnostics for such conditions. Remember that CPU starvation means overall degradation of performance, not just slowed-down VM operations.

Case 3 means the thread is blocked on an explicit OS call, which will be wrapped via JNI so that such thread will be exempt from the safepoint protocol.

Case 4 is a different story. Unlike Case 3, a page fault may block execution at any instruction. The JVM cannot add any safety net for these cases. The thread has to be resumed by the kernel and continue execution to, first, finish the safepoint protocol and then let it proceed with the VM operation.

It is obvious that Case 4 is a most problematic condition here, having the potential to cause abnormally long safepoint time.

What are common reasons for page faults in Java applications?

Thrashing

System-wide swapping (aka thrashing) could be one reason. As a rule, the JVM should keep its heap and a few other vital data structures resident in memory. If JVM's memory starts to be paged out, it hits app performance quite badly. The GC also does not like memory being swapped out, so this condition

has the double impact of the STW pauses.

Swapping, in turn, has two causes:

- Host memory shortage — physical memory is not enough for all processes running on the host.
- Container resident memory limit — it is possible to configure resident memory usage limit per container. This limit may lead to thrashing for a specific container even if the host has enough free memory.

Memory-mapped I/O

The Java runtime library has an API for memory-mapped disk I/O. "Memory-mapped" means that reading or writing to a file on disk are regular memory operations in application code. The actual content is loaded from disk to memory page on first access and eventually synchronized back to disk if modified. The kernel is responsible for moving data. It's worth pointing out that for memory-mapped I/O, moderate page fault rate is expected.

Just one thread busy with memory-mapped I/O is enough to impact the safepoint for the JVM globally.

A memory mapping approach should reduce I/O overhead by eliminating OS calls, but in the JVM case, it can introduce worse problems interfering with the safepoint protocol.

We recommend avoiding memory-mapped I/O for services that are response time critical and process requests in parallel. Often, libraries (e.g., Lucene) support both memory-mapped and non-memory-mapped I/O. You need to assess the impact of the JVM STW pause before enabling the memory-mapped options in your middleware.

Long intervals between safepoint checks in JIT-compiled code

Above, we have described situations where Java threads are prevented from running by the OS. However, it's possible for threads to run on CPU without yet completing the safepoint protocol in a timely manner.

The issue could be with the code produced by the JIT compiler. It should place checks for safepoint in the emitted code.

Each method call site is typically eligible for a safepoint check (a method should always ensure its local variable is appropriately placed on the stack before calling another method).

Loop back branch is another natural point for a safepoint check, even though a safepoint implies a barrier in operation reordering, thus forcing many optimizations. A check for the safepoint may be omitted for "counted" loops (loops with an index variable of type int). The compiler takes risk for potentially long periods between checks in runtime and better general execution performance.

Besides the "counted" loop, JVM intrinsics can also have high upper bounds for execution time. For instance, when you create a huge array instance, array memory is zeroed at the moment of allocation with no safepoint possible until it’s done.

Fortunately, such situations are rare enough. The JIT compiler is well-balanced in the JVM, and chances to hit this problem class are close to zero in typical Java code.

Below are several situations where such problem can be witnessed:

- operations on unreasonably long strings;
- extensive Java BigNumber manipulation;
- handmade statistics library working on long arrays of samples.

5. Monitoring safepoints without JFR

JDK Flight Recorder is not the only tool to monitor safepoints. The JVM has multiple options to enable logging safepoint-related information to the console.

`-XX:+PrintGCApplicationStoppedTime` — Contrary to its name, this option will let the JVM log all STW pauses, related or unrelated to garbage collection, to GC logs. If enabled, you'll see lines like this:

```
Total time for which application threads were stopped: 0.0007555 seconds,
Stopping threads took: 0.0005893 seconds
```

The message above tells you little about the nature of the safepoint, though.

`-XX:+PrintSafepointStatistics -XX:PrintSafepointStatisticsCount=1` — Such a combination of flags will let the JVM print more details after each safepoint. The output will look as follows:

```
     vmop       [threads: total initially_running wait_to_block]  [time: spin
block sync cleanup vmop] page_trap_count

9.220: ThreadDump   [     17     8       8    ]  [    1 19  21   0   0 ] 8
```

This output contains much more details, including VM operation, thread counts, and time (in milliseconds) spent in the pre-safepoint phases. It also contains the number of page traps during safepoint, which is very handy to identify I/O-related issues.

A downside here is formatting, which is hard to parse. The idea behind this option is to log a summary of passing safepoints in the table form.

JDK 11 made available new structures logging system, and safepoint logging could be enabled with the following option: `-Xlog:safepoint*=debug`

The result will look like this:

```
[13.173s][debug][safepoint      ] Safepoint synchronization initiated. (18
threads)
[13.173s][info ][safepoint      ] Application time: 2.0725501 seconds
[13.173s][debug][safepoint      ] ... found polling page loop exception at pc =
0x0000000012975dbc, stub =0x000000000aed7100
[13.173s][debug][safepoint      ] ... found polling page loop exception at pc =
0x0000000012975dbc, stub =0x000000000aed7100
```

```
[13.173s][debug][safepoint      ] ... found polling page loop exception at pc =
0x0000000012975dbc, stub =0x000000000aed7100
[13.173s][debug][safepoint      ] ... found polling page loop exception at pc =
0x0000000012975dbc, stub =0x000000000aed7100
[13.173s][debug][safepoint      ] ... found polling page loop exception at pc =
0x0000000012975dbc, stub =0x000000000aed7100
[13.173s][debug][safepoint      ] ... found polling page loop exception at pc =
0x0000000012975dbc, stub =0x000000000aed7100
[13.173s][debug][safepoint      ] ... found polling page loop exception at pc =
0x0000000012975dbc, stub =0x000000000aed7100
[13.209s][debug][safepoint      ] ... found polling page loop exception at pc =
0x0000000012975dbc, stub =0x000000000aed7100
[13.209s][debug][safepoint      ] Waiting for 8 thread(s) to block
[13.210s][info ][safepoint      ] Entering safepoint region: ThreadDump
[13.210s][info ][safepoint,cleanup] deflating idle monitors, 0.0000004 secs
[13.210s][info ][safepoint,cleanup] compilation policy safepoint handler,
0.0000004 secs
[13.210s][info ][safepoint,cleanup] updating inline caches, 0.0000004 secs
[13.210s][info ][safepoint,cleanup] resizing system dictionaries, 0.0000004
secs
[13.210s][info ][safepoint,cleanup] purging class loader data graph, 0.0000000
secs
[13.210s][info ][safepoint,cleanup] safepoint cleanup tasks, 0.0000809 secs
[13.210s][info ][safepoint      ] Leaving safepoint region
[13.210s][info ][safepoint      ] Total time for which application threads were
stopped: 0.0375086 seconds, Stopping threads took: 0.0372714 seconds
```

6. Conclusion

Stop-the-World (aka safepoints) pauses are fairly frequent in the JVM, and not all of them are related to garbage collection. Normally non-GC safepoints have submillisecond duration and go unnoticed.

Still, there are conditions that cause abnormal STW pauses due to a long *time to safepoint*.

Built-in JDK Flight Recorder profiles do not have detailed safepoint-related events disabled. If you suspect a non-GC-related STW pause, enabling this event in Flight Recorder is the first step in your investigation. Besides metrics collected by JFR, we also recommend looking at OS metrics to identify situations such as thrashing and CPU starvation.

JDK Flight Recorder

How to root cause Stop-the-World pauses

bellsoft