A Primer on Worker Debugging

Introduction

This document gives an overview of worker debugging for people. It's intended target audience is people who want to start hacking on worker debugging, but don't know where to start. That said, this document is not intended to be comprehensive; worker debugging is a very complicated subject, and to give a comprehensive overview would require many more pages than I can afford to write right now. Instead, the intent is to give just enough information to allow one to get started.

For the sake of exposition, I have organized this document into three major sections. The first section gives a brief overview of how workers are implemented. The second section gives a quick overview of the debugger API, and how it can be used. Finally, the third section looks into how the debugger API can be used in workers, which is the essence of worker debugging.

If you are reading this, and you have additional questions, feel free to e-mail me at: ejpbruel@gmail.com

Workers

Introduction

In this section, we will give a brief overview of how workers are implemented. We will go over the most important concepts, and the classes that implement them.

A **worker** is an object that executes JavaScript code in its own thread. This allows JavaScript code to be executed in the background. We will refer to the thread in which a given worker is executing JavaScript code as the **worker thread**.

A worker can be created on either the main thread or another worker thread. If a worker is created on the main thread, it is known as a **top-level worker**. Otherwise, the worker is a **child worker**, and the worker that created it is known as its **parent worker**. We will refer to the thread on which a worker was created as its **parent thread**. The parent thread is either the main thread, or the thread in which the parent worker is executing JavaScript code.

WorkerPrivate

A worker is represented by a **WorkerPrivate**, which is defined here:

https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerPrivate.h#890

A WorkerPrivate represents both the worker DOM object (which lives in the parent thread), and the state of the worker thread. As such, each WorkerPrivate lives on two threads. Notice that WorkerPrivate derives from a template base class called **WorkerPrivateParent**, which is defined here:

https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerPrivate.h#144

(It is not clear to me why WorkerPrivateParent has to be a template, since it is only ever instantiated with WorkerPrivate as the template parameter. Most likely, this is a leftover from an earlier iteration of the worker implementation).

WorkerPrivateParent contains all the state that may be accessed from either the parent or worker thread, and therefore must be mutex protected. In contrast, WorkerPrivate contains all the state the may only be accessed from the worker thread, and therefore doesn’t have to be mutex protected.

Event Queues

Each worker has an implicit **event queue** (associated with its thread), to which other threads can dispatch **runnables**. Except for a few cases, all communication between worker threads happens by means of dispatching runnables. The main purpose of the shared state in WorkerPrivateParent is to coordinate dispatching runnables between the parent and worker thread. Normally speaking, only parent and child workers can directly dispatch runnables to each other, but there are a few exceptions (such as a worker dispatching a runnable to the main thread).

In addition to the implicit event queue, each worker has several *explicit* event queues. The **control event queue** is used to send **control runnables** to a worker. As their name implies, control runnables are used to control the lifetime of a worker. For instance, when the parent thread wants to terminate a worker, it does so by sending a control runnable. As such, control runnables take precedence over normal runnables, and (by using the interrupt callback) will be processed even when the worker is frozen, or spinning in an infinite loop.

In addition to the control event queue, a worker can have one or more **synchronous event queues**. These are created on demand when a worker dispatches a runnable to another thread, and then needs to block until it gets back a reply. An example of this is when loading the main worker script. This needs to happen on the main thread (although the reason why is not clear to me). The worker thread sends a runnable to the main thread requesting it to load the main worker script, and then enters a **synchronous event loop**, which blocks on a synchronous event queue until the main thread dispatches a runnable notifying the worker thread that the script has been loaded.

The main event loop for each worker is defined here:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerPrivate.cpp#4522

Similarly, synchronous event loops are defined here:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerPrivate.cpp#5338

Most of the code in those functions deals with blocking the worker thread when the queue is empty, waking up the worker thread when a new runnable is dispatched, disabling/re-enabling.

WorkerRunnable

Every runnable that can be dispatched to a worker is a subclass of **WorkerRunnable**, which is defined here:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerRunnable.h#32

A WorkerRunnable defines several things that are common to every runnable that can be dispatched to a worker. In particular, each worker runnable has what is known as a **target** and a **busy**

behavior. The target of a runnable determines whether it is sent to the parent thread or the worker thread. The busy behavior of a runnable determines whether it increments the **busy count** of its target. The busy count is a counter that ensures that a worker stays alive until it reaches zero.

Each concrete subclass of WorkerRunnable must implement the method **WorkerRun**, which defines the main behavior of each runnable. It is declared here:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerRunnable.h#159

In addition to WorkerRun, WorkerRunnable defines several overridable methods that are automatically run before/after a runnable is dispatched/run. These methods are typically used to assert that the runnable is used correctly, i.e. dispatched/run from the correct thread.

There are many different subclasses of WorkerRunnable. Among the more interesting are **WorkerControlRunnable**, which represents control runnables, and WorkerSyncRunnable, which represents runnables dispatched to a synchronous event queue. They are declared here:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerRunnable.h#273
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerRunnable.h#212

WorkerScope

In addition to event queues, each worker also has a single global object. All JavaScript code executed by a worker is executed in the context of that global object. The global object is represented by a subclass of **WorkerGlobalScope**, which is defined here:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerScope.h#47

WorkerGlobalScope defines several APIs that are common to every worker. For each class of workers, such as shared workers or service workers, there is a subclass of WorkerGlobalScope, such as **SharedWorkerGlobalScope** or **ServiceWorkerGlobalScope** that defines APIs specific to that class of workers. When a WorkerPrivate is created for a worker, it eventually ends up calling **GetOrCreateGlobalScope**, which creates an instance of the appropriate subclass of WorkerGlobalScope. See:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerPrivate.cpp#6441

Only certain APIs are defined directly on the WorkerGlobalScope classes. Others are defined indirectly using DOM bindings, which are created here:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerPrivate.cpp#6464

RuntimeService

The final piece of the puzzle is the runtime service. When a worker is created, it registers itself with the runtime service. The runtime service is responsible for assigning a thread to each worker object. This implies that there may be more worker objects than threads, and some workers may not (yet) have threads assigned to them.

Debugger API

Introduction

In the previous section, we gave a brief overview of how workers are implemented. In this section, we will give a quick overview of the debugger API, and how it can be used.

The SpiderMonkey debugger API allows one to inspect the runtime state of a JavaScript program at a given moment in time. The debugger API is organized into a set of **shadow objects**, each of which refers to a particular entity in of the execution state. For instance, there are shadow objects that refer to a frame on the stack, a closure environment, an object on the heap, etc. The object referred to by a shadow object is known as its **referent**.

The debugger API is designed to be used from JavaScript, and as such has a JavaScript interface.

Usage

To use the debugger API, we must first create an Debugger object:

```
let dbg = new Debugger();
```

Note that the Debugger constructor is not available from JavaScript by default: for security reasons, it must be explicitly defined on a given object with a call to **JS_DefineDebuggerObject** (which is part of the SpiderMonkey API).

Once we have a Debugger object, our next step is to tell it which globals we want to debug. We will refer to such globals as **debuggees**. Because debugging carries a performance cost, the debugger will only observe compartments belonging to the debuggees. To add a global as debuggee, we call the **addDebuggee** method on our Debugger object:

```
dbg.addDebuggee(global);
```

When this global is added as debuggee, its compartment is switched to debug mode. This may cause existing scripts in that compartment to be recompiled. These scripts are recompiled in such a way that the debugger can easily install callbacks in them after they are compiled. To accomplish this, we insert a NOP instruction between each pair of adjecent instructions between which a debugger callback could be invoked. When necessary, this NOP instruction can be replaced with a CALL instruction that invokes the appropriate callback.

Note that a debuggee cannot be the global in which the Debugger object lives. Doing so would cause the debugger to observe itself, something which it is not prepared for! One important implication of this is that in order to use the debugger API, we need at least two globals: one for the debugger, and one for the debuggee.

Once the debugger API is observing our debuggee, we can install a callback to be invoked whenever something interesting happens. For instance, we can tell the debugger to call a callback whenever the debuggee executes a debugger statement:

```
dbg.onDebuggerStatement = function (frame) {
    ...
};
```

Note that this callback takes a frame as argument. This is a **Debugger.Frame** object, which is a shadow object referring to a stack frame. From here, we can use the properties and methods defined on Debugger.Frame to obtain further s about the execution state.. For instance, if the current frame is a call frame, we can obtain a shadow object referring to the callee as follows:

```
dbg.onDebuggerStatement = function (frame) {
   let callee = frame.callee;
};
```

The callee of a call frame is a function object. Objects are reflected over by a **Debugger.Object** object, which is a shadow object referring to an object on the heap. If an object is a function object, we can obtain a shadow object referring to its lexical environment as follows:

```
dbg.onDebuggerStatement = function (frame) {
   let callee = frame.callee;
   let environment = callee.environment;
};
```

The above gives a very brief overview of how the debugger API is used. This treatment is not meant to be comprehensive, but rather to give some idea about how the debugger API is used. For more information on the debugger API can be used, and the functionality it provides, please refer to the debugger API documentation, which is much more comprehensive: https://developer.mozilla.org/en-US/docs/Tools/Debugger-API

Shadow objects

Each Debugger object maintains its own set of shadow objects. The debugger API guarantees that for a given Debugger object, a shadow object for a given entity will be unique. That is, for a given Debugger object, there will never be more than one shadow object referring to the same entity.

To maintain this invariant, each Debugger object maintains a weak map from entities in the debuggee, to the shadow objects that refer to these entities. This allows the debugger API to always return the same shadow object for a given entity in the debuggee, even if those shadow objects were obtained through different API calls.

Another thing the debugger API guarantees is that each shadow object keeps the entity it refers to alive. For instance, a Debugger.Object referring to an object on the heap will keep that object alive even if there are no other references to that object in the debuggee.

However, shadow objects will only keep the entities they refer to alive as long as the Debugger object by which they are owned is still alive: if either the shadow object or the Debugger object that owns it is no longer reachable, the entity will no longer be kept alive..

Pausing

We have looked at how the debugger API can be used to inspect the runtime state of the JavaScript engine. However, in doing so, we've glanced over an important detail: how do we actually pause the JavaScript engine, so its runtime change doesn't change out from under us? Perhaps surprisingly, the debugger API does not provide a way to pause the JavaScript engine. Instead, we need to rely on external platform APIs.

The naive way to pause the JavaScript engine would be to simply pause whatever thread it is running in. However, this glances over another important detail: because SpiderMonkey is not thread-safe, he debugger API is designed to be an inter-thread API. This means that it needs to be used from the same thread as the one we are debugging. One important consequence of this is that the debugger and debuggee are both executing in the same thread.

Because of this, we can't actually pause the thread in which the JavaScript engine is running. Doing so would not only pause the debuggee, but the debugger as well, making the latter unresponsive. The best we can do is create the *illusion* that the JavaScript engine is paused, but making sure that no debuggee code is executed while we are 'paused'. Moreover, to ensure that the debugger can still respond to events,, we need to do this in such a way that debugger code can still be executed.

How do we create such an illusion? To answer this, we must first understand that whenever a debugger calback is called, this call from within debuggee code. That is, at some point during the execution of the debuggee code, instead of executing the next instruction, SpiderMonkey decides to call a debugger callback instead. Recall how this is implemented by replacing a NOP instruction with a CALL instruction.

Our goal is to make sure that no debuggee code is executed while we are 'paused', but still allow debugger code to run. One obvious way in which debuggee code can be executed is if we return from the debugger callback. Consequently, we should never return from a debugger callback unless we are no longer paused.

However, this poses a problem. While the debugger is paused, we still want it to be able to respond to events, such as the user clicking the resume button. Events are handled by the main event loop, but this doesn't run until *after* we return from the debugger callback. Clearly, these two requirements are at odds with each other.

The way out of this conundrum is this: instead of returning from the debugger callback, we spin up a so called **nested event loop**. Within this nested event loop, we process events that are targeted to the debugger (such as a mouse click on the resume button), but not events that are targeted to the debugger (such as a debuggee timer timing out). The latter would cause debuggee code to be executed, thus breaking the illusion that the JavaScript engine is paused. To prevent busy waiting, a nested event loop should block until an event comes in. This also prevents the debugger callback that entered the nested event loop from returning.

Whenever an event argeted to the debugger comes in, the nested event loop will process it. This may cause further debugger code to be executed, which allows the debugger to respond to whatever caused the event. For instance, if the event was caused by the user clicking the resume button in the debugger API, the debugger responds by calling a function that tells the nested event loop to exit on the next iteration.

After we exit a nested event loop, once we finish processing the event, we fall back to the nested event loop, which then immediately exits. At this point, we are back in the original debugger callback. Once we return from there, the debuggee will continue executing.

Finding globals

As a final note: although the debugger API is capable of finding every global on the heap, and adding these globals as debuggees, we typically only want to add a certain subset of all globals as debuggees. For instance, globals are typically grouped by tab, and we only want to debug one tab at a time. However, the debugger API itself doesn't have a notion of tabs. That means we need external platform support to tell which globals should debugged and which not.

Any implementation of a debugger that is based on the SpiderMonkey debugger API needs to provide these two functions to the debugger API:

1. Pausing the JavaScript engine, most likely using nested event loops.
2. Deciding which globals need to be debugged.

Worker Debugging

Introduction

In the previous sections, we gave a brief overview of how workers are implemented, and how the debugger API can be used. In this section, we will take a look at how the debugger API can be used in workers.

As we have seen, the debugger API is a JavaScript API that can be used to observe the execution state of the JavaScript engine. The debugger API will only observe the execution state of compartments it has been explicitly told to observe (by adding the corresponding global as a debuggee). Because the debugger API does not allow a debugger to observe its own compartment, we need at least two globals: one for the debugger, and one for the debuggee.

Moreover, the debugger API cannot be used to pause the JavaScript engine. For this, we need additional platform APIs. This is further complicated by the fact that the debugger API has to be used from the same thread as the debuggee. As a result, we cannot actually pause the thread in which the JavaScript engine runs without making the debugger unresponsive as well. Because of this, we need to use a nested event loop to create the illusion that the thread is paused.

In order to use the debugger API in workers, we thus need to add the following features to workers:

1. A separate global for the debugger.
2. Nested event loops to simulate thread pauses.

With these features in place, we can use the debugger API in workers. However, since the debugger UI runs in the main thread, we also need a way to discover workers, initialize a debugger in a worker, and communicate between this debugger and the main thread.

Finally, we need to add some additional features to workers, because the debugger implementation relies on them. In particular, we need to be able to:

1. Create a sandbox and load a subscript in it.
2. Schedule a callback to be executed on the next tick of the event loop.

WorkerDebuggerGlobalScope

The separate global for the debugger is represented by an instance of the class **WorkerDebuggerGlobalScope**, which is defined here:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerScope.h#305

When a debugger is initialize in a worker, an instance of WorkerDebuggerGlobalScope is created here:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerPrivate.cpp#6476

For performance reasons, an instance of WorkerDebuggerGlobalScope is only created if a debugger is initialized. This is unlike the subclasses of WorkerGlobalScope, an instance of which is always created for every worker.

When workers were first implemented, they were written under the assumption that there is only ever one global in a worker. As a result, adding the possibility to create additional globals will lead to several problems elsewhere, as we will see later on.

WorkerDebuggerRunnable

To simulate a thread pause, we need a way to create nested event loops. While we are in a nested event loop, we should only process runnables that are targeted to the debugger, while runnables that are targeted to the debuggee are ignored. In what follows, we will refer to runnables targeted to the debugger as **debugger runnables**. Conversely, we will refer to runnables targeted to the debuggee as **normal runnables**.

In order to distinguish between debugger runnables and normal runnables, we always dispatch debugger runnables to a separate event queue, known as the **debugger event queue**. It is defined here:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerPrivate.h#915

To determine whether a runnable is a debugger runnable or a normal runnable, we call the method **IsDebuggerRunnable** on a runnable, which is defined here:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerRunnable.h#111

If this method returns true, the runnable is dispatched to the debugger event queue. Otherwise, it is dispatched to the normal event queue. See:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerRunnable.cpp#108

The class **WorkerDebuggerRunnable** is a subclass of WorkerRunnable, which defines a default implementation of IsDebuggerRunnable that always returns true. In almost all cases, a debugger runnable should inherit from this class. However, there are a few exceptions, such as ScriptExecutorRunnable, which can be used as both a debugger runnable and a normal runnable:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/ScriptLoader.cpp#1870

As we mentioned earlier, having more than one global can lead to problems in workers. One of these is that it is no longer obvious which global a runnable should be executed in. In most cases, we make sure to enter the compartment of the correct global before a runnable is processed, so we can just execute the runnable in the current global. For those cases where a current global does not

exist, each defines a default global to be used here:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerRunnable.h#114

The code that determines which global to execute it in can be found here:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerRunnable.cpp#291

As a final remark, one particular runnable, CompileDebuggerScriptRunnable, is used to create the debugger's global. As such, we cannot enter the correct global before this runnable is processed, since the global doesn't exist until *after* the runnable has been processed. Similarly, CompileDebuggerScriptRunnable cannot have a default global, since it does not yet exist.

Nested Event Loops

While a worker is spinning its main event loop, it processes both normal and debugger runnables (as well as control runnables):
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerPrivate.cpp#4557

In contrast, when a worker is spinning a nested event loop, it only processes debugger runnables (as well as control runnables):
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerPrivate.cpp#5619

Since only normal runnables can cause JavaScript code in the debuggee to run, by only processing debugger runnables, we create the illusion that the thread is paused, while ensuring that the debugger can still respond to events. By still processing control runnables as well, we ensure that a worker can still be terminated, even when it is paused.

Since we want to be able to use nested event loops from JavaScript, we define two JavaScript functions on WorkerDebuggerGlobalScope, **enterEventLoop** and **exitEventLoop,** which enter and exit an event loop, respectively:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerScope.h#352

WorkerDebuggerManager

Since the debugger UI runs in the main thread, we need a way to discover workers on the, initialize a debugger in a worker, and communicate between this debugger and the main thread. The **WorkerDebuggerManager** service keeps track of all workers in the system:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerDebuggerManager.h#29

Every time a worker is created, it registers itself with the WorkerDebuggerManager:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerPrivate.cpp#2428

Conversely, every time a worker is destroyed, it unregisters itself with the WorkerDebuggerManager:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerPrivate.cpp#2442

The WorkerDebuggerManager can be accessed from the main thread. However, workers can only be accessed from either their own thread or the parent thread. For non top-level workers, the parent thread is no the same as the main thread. In order to access workers from the main thread, each worker is represented by an instance of the class **WorkerDebugger** on the main thread:

https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerPrivate.h?q=WorkerDebugger&redirect_type=direct#850

A WorkerDebugger provides the bridge between the main thread and the worker thread, and can be used from either of these threads.

Both WorkerDebuggerManager and WorkerDebugger implement an XPIDL interface that allows you to use these classes from JavaScript. In particular, WorkerDebuggerManager defines methods to list all workers in the system, and to set up a listener to be called when this list changes:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/nsIWorkerDebuggerManager.idl#17

Similarly, WorkerDebugger defines methods to initialize a debugger in a worker, post a message to the debugger in the worker thread and to set up a listener to be called when the debugger sends a message to the main thread:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/nsIWorkerDebugger.idl#18

Additional features

There are many complications to using nested event loops. A particularly glaring one is caused by the fact that our implementation of the debugger in Gecko makes heavy use of promises. Although promises are implemented for workers, the current implementation is not aware of the difference between debugger code and debuggee code.

As a result, promises in debuggee code and promises in debugger code will both schedule the execution of their promise handlers on the normal event queue. Consequently, these handlers will never called while the debugger is paused, not even for promises used by the debugger itself.

It is possible to rewrite the implementation of promises in workers so that they are aware of whether they are created by debugger code or debuggee code, and consequently which event queue they should use to schedule the execution of their runnables. However, such a rewrite is not trivial, and therefore has not been done yet.

As a temporary workaround, the debugger currently uses a JavaScript implementation of promises for the debugger if it detects that it is running in a worker. This implementation still needs a way to schedule promise handlers on the debugger event queue. To support this, we define a JavaScript function, **setImmediate**, on WorkerDebuggerGlobalScope:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerScope.h#363

In addition to promises, the debugger also uses a JavaScript implementation of modules (in the form of CommonJS modules). This implementation needs a way to create a sandbox for each module, and then load the module script inside that sandbox. To support this, we define two JavaScript functions, **createSandbox** and **loadSubScript**, on WorkerDebuggerGlobalScope:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerScope.h#341

Last, but not least, we need both a way to receive messages from and send messages back to the WorkerDebugger in the main thread. To support this, we define a JavaScript function, **postMessage**, and a **message** event handler, on WorkerDebuggerGlobalScope:
https://dxr.mozilla.org/mozilla-central/source/dom/workers/WorkerScope.h#358