Workers and event loops

This all starts from this code and leads to broken toolbox.

Because of a Worker using Atomics.wait, the worker thread is paused on this particular C++ line:
https://searchfox.org/mozilla-central/source/dom/workers/WorkerPrivate.cpp#3033-3034

```
WorkerPrivate::DoRunLoop {
      ...
      // Process a single runnable from the main queue.
      NS_ProcessNextEvent(mThread, false);
```

From my investigation, the C++ code is stuck pending on this line.
One surprise is that the second argument `aMayWait` seems to indicate if the call should pause or not. And I would assume this call should not block, while it does.
I've not tried to debug this deeper within `NS_ProcessNextEvent` as this is very generic code used with about anything…

SpinEventLoopUntil and how breakpoints work on the main thread

It is interesting to see how `SpinEventLoopUntil` is implemented in C++.
It actually uses this same NS_ProcessNextEvent method, but with a true second argument:
(permalink)

```
      while (!aPredicate()) {
            bool didSomething = NS_ProcessNextEvent(thread, true);
```

Note that this `SpinEventLoopUntil` method is the one also used by Debugger C++ API
(permalink)

```
      if (!SpinEventLoopUntil([&]() { return mNestedLoopLevel <
nestLevel; })) {
```

This is the API I recently described, which is a key component to implement breakpoints.
This is called from here in the thread actor universe:
(permalink)

```
      xpcInspector.enterNestedEventLoop(this);
```

(permalink)

```
      this._nestedEventLoop.enter();
```

(permalink)

```
      return this.threadActor._pauseAndRespond(frame, reason);
```

(permalink)

```
      bool ok = CallMethodIfPresent(cx, handler, "hit", 1,
scriptFrame.address(), &rv);
```

(permalink)

```
if (DebugAPI::hasBreakpointsAt(script, REGS.pc)) {
  if (!DebugAPI::onTrap(cx)) {
```

Basically, the JS engine would call breakpoint.js if a breakpoint is registered for the current line of JS being executed. This code will run from the debuggee thread, and will be ultimately paused by the call to `NS_ProcessNextEvent(thread, true);` which will resume once the “spin event loop until” condition switches.

The condition will be switched when we call `xpcInspector.exitNestedEventLoop(this)` which, I suppose, is called from another thread. This is *not* triggered from the debugee thread, instead, it is resumed from a RDP request, itself being probably spawn from the nsISocketTransportService thread.
And it is interesting to note that nsISocketTransportService is having a very similar for..loop, also using `NS_ProcessNextEvent`!
(permalink)

```
NS_ProcessNextEvent(mRawThread);
```

Having a distinct thread, via the socket service is probably what explains why we can still execute stuff in the content process, while the “main thread” is paused.

Breakpoints on the worker thread

Now, in the worker thread, we can’t use any XPCOM, and so can’t use `xpcInspector` interface. Nor can we use SpinEventLoopUntil as we don’t have access to Services either. Instead for fake xpcInspector in order to call another eventEventLoop method:
https://searchfox.org/mozilla-central/source/devtools/shared/worker/loader.js#477-480

```
enterNestedEventLoop: function(requestor) {
requestors.push(requestor);
scope.enterEventLoop();
```

(permalink)

```
void WorkerDebuggerGlobalScope::EnterEventLoop() {
  MOZ_KnownLive(mWorkerPrivate)->EnterDebuggerEventLoop();
```

(permalink)

```
WorkerPrivate::EnterDebuggerEventLoop(){
  ...
  std::queue<RefPtr<MicroTaskRunnable>>& debuggerMtQueue =
  ccjscx->GetDebuggerMicroTaskQueue();
  while (mControlQueue.IsEmpty() &&
    !(debuggerRunnablesPending = !mDebuggerQueue.IsEmpty()) &&
    debuggerMtQueue.empty()) {
      WaitForWorkerEvents();
  }
  ProcessAllControlRunnablesLocked();
```

Instead of using `NS_ProcessNextEvent`, we have this `WaitForWorkerEvents()` (permalink)

```
void WorkerPrivate::WaitForWorkerEvents() {
  mCondVar.Wait();
```

And this waits for a new task to be dispatched. mCondVar is a kind of mutex, which will be released anytime we dispatch a new task in the worker:

- a regular task, from WorkerThread::Dispatch (worker script will be run in such task type)
- a debugger task, from WorkerPrivate::DispatchDebuggerRunnable
- a control task, from WorkerPrivate::DispatchControlRunnable

Each task type is having its own queue. This allows `WorkerPrivate::DoRunLoop` and `WorkerPrivate::EnterDebuggerEventLoop` to triage the various task in a precise order.

What next?

- Is it expected that `NS_ProcessNextEvent(mThread, false);` can be blocked by Atomics.wait?
- Should debugger scripts in the worker thread run in a distinct thread, like what we do for main thread debugging ?
- RemoteAgent setup and incoming Bidi architecture may help us mitigate such issues by having some code running from the main thread.
- Any other idea? Questions?

Meeting notes

- The discussion is about broken Toolbox when debugging/attaching Workers that might be frozen by using Atomic.wait
- Toolbox can be broken only when you open Debugger panel since it's listening to worker targets - and the worker is stack on Atomic.wait()
- It isn't only about the Debugger - there is also the Console panel involved.
- See this code
  - There are multiple event loops
  - There is GC involved
  - Looks like we are also executing pieces of JS code
-