Base Language - Feature #3254

add support for running 4GL on multiple threads in a single session

02/24/2017 01:08 PM - Greg Shah

Status:	WIP	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Greg Shah	**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**vendor_id:**	GCD

Description

Related issues:

Related to Base Language - Feature #3253: add cross-session publish/subscribe	**WIP**
Related to Base Language - Feature #4064: large-raw data type	**New**
Related to Runtime Infrastructure - Feature #4406: server-side REST execution...	**New**
Related to Base Language - Feature #4065: server-side processing of client pl...	**WIP**
Related to User Interface - Feature #4912: move UI portions of the web client...	**New**

History

#1 - 02/24/2017 01:08 PM - Greg Shah

- Related to Feature #3253: add cross-session publish/subscribe added

#2 - 02/24/2017 01:24 PM - Greg Shah

This is one of the ideas listed in 4GL Enhancements.

The approach is to support multi-threading for ABL code. This means that we would enhance the ABL syntax itself to allow 4GL code to be added or changed to use these new features. Since it only could be used with enhanced 4GL, we are free to impose limitations on the way in which one would be able to use it.

The following is a non-exhaustive list of the synchronization and data sharing primitives that I would want to include:

- 4GL PUBLISH/SUBSCRIBE
- generic thread-safe-queues
- event semaphores
- mutex semaphores
- thread-safe BaseDataTypes (always will be NO-UNDO, APIs will be atomic to read/write
- thread-safe temp-table sharing (always NO-UNDO; copy to the other thread and then copy back; will require the temp-tables to be identical classes so that the fast Java-based deepCopy/assign can be used)

It seems to me that this is doable with the following assumptions/limitations:

- each thread needs to run as a separate session (security context), even though it will share the same account as the primary thread
- only the primary (initial) thread can call the client side (e.g. drive the UI), all other threads are completely server-side and non-interactive
- it may be possible to allow access to some non-UI client APIs (streams, file system, memptr... maybe even process launching) but we would have to do it in a way that was asynchronous for the network protocol and the client-side daemons would need to be reworked to be thread safe (my initial idea is to ignore this complication for now)

I guess that this new thread safe form of temp-tables/BDT instances could even be useful for sharing across unrelated sessions, but it is possible to confine these within just the "related sessions" (the primary thread and any sessions from threads that were spawned by that thread, including grandchild threads).

The other synchronization mechanisms could and probably should be accessible server-wide, so long as we implement security controls.

Let me know what you think.

#3 - 02/27/2017 08:03 AM - Greg Shah

I want to point out that the assumption of having a separate security context/session (using the same account) enables a completely independent connection to the database AND an independent transaction/sub-transaction/block properties approach for each thread.

Let's distinguish sessions that were started by an existing session for threading purposes by calling them a related session. Related sessions would automatically have full rights to access all thread-safe resources created by any other related session.

This is in contrast to a cross-session feature, which I will use to refer to access to resources from sessions that are NOT related.

I think it is probably best to implement all of the synchronization and data sharing primitives in a cross-session manner. This will be a useful feature and make for a more generic implementation. The only additional work needed is to implement a security plugin that will control access to these resources. One slight "wrinkle" of this is that these resources are created dynamically by application code. I don't think we've had dynamic resources before. The plugin will have to track the lifetime of these resources and dynamically manage that list and all the permissions.

I think this plugin will only apply to cross-session access. Related session access should be automatic. We may want to expand this to the idea that any code (even in cross-session access) which is run by the same account as the creator of the resource, should be able to access the resource without any additional configuration. The restriction should probably be at the level of cross-session access when code is run in a different account from the creator of the resource.

At a minimum, the permissions bitfield needs to allow CDRW (create, delete, read, write) and the resource name would be the text name of the resource (semaphore name, queue name, variable name, temp-table buffer name).

Does anyone see any flaws in my logic or ideas?

#4 - 07/03/2019 04:39 PM - Greg Shah

For these threads, it may make sense to implement some server-side versions of features that normally are client-side. The following seem quite straightforward to implement:

- memptr (will require libp2j.so)
- process launching/shell commands
- native API calls
- XML input/output to file
- general filesystem access and streams
- sockets
- web services

Program launch, native API calls, memptr and filesystem access will need to be protected by a security manager plugin since any use of these on the server is dangerous.

The only non-UI client-side feature which seems less useful here is the environment access and Windows registry usage. On the other hand, the INI file usage might in fact be useful as well. We may want to just enable all client-side non-UI features and ensure that suitable security controls are in place.

This set of features would allow selective batch programs to run, which would be a substantial performance improvement for some use cases AND it would avoid the need for many different client processes.

The next logical step would be to mark some appserver agents and batch mode sessions as server-only, completely avoiding the need for a client process. In such cases, any use of client-features that was rejected by the security checks would cause the current operation to exit immediately (appserver/rest/wsa) while for the batch session it might abend (like a STOP or QUIT).

The security plugin for securing client-side access should be implemented as a general purpose feature so that customers can lock down 4GL sessions from using specific client features.

The task for this idea is #4065. A related idea is #4064, but it may just be simpler to implement server-side versions of the existing client features.

#5 - 07/03/2019 04:41 PM - Greg Shah

- Related to Feature #4064: large-raw data type added

#6 - 08/22/2019 01:39 PM - Greg Shah

Once we have server-only session support, we should consider how we can use these with REST and/or websockets (javascript appserver) for a lightweight approach (no agents needed) to remote access.

In the REST case, we may have to use a pool of server side sessions that replace agents. See #4406.

In the websockets case, we have a linkage to a long running session, so it is a direct match with a specific server-side session.

#7 - 01/09/2020 04:37 PM - Greg Shah

- Assignee set to Greg Shah

- Start date deleted (02/24/2017)

#9 - 01/09/2020 05:04 PM - Greg Shah

- Related to Feature #4406: server-side REST execution without appserver agents added

#10 - 01/09/2020 05:05 PM - Greg Shah

- Related to Feature #4065: server-side processing of client platform dependencies added

#11 - 01/09/2020 05:09 PM - Greg Shah

Some months ago, Eric noted to me that sharing temp-tables will be a problem since they are private to a single JDBC connection. Since JDBC connections cannot be shared across more than one thread, the current temp-table approach is not compatible.

I expect to exclude temp-tables from multi-threaded usage for now. This would mean that each thread could still have its own private temp-tables, which should work because each thread will have its own security context/transaction state/JDBC connection. But those tables will not be shared with other threads.

#12 - 01/09/2020 05:16 PM - Eric Faulhaber

Greg Shah wrote:

> Since JDBC connections cannot be shared across more than one thread, the current temp-table approach is not compatible.

Note that the same connection restriction applies to persistent tables, too.

#13 - 01/09/2020 05:17 PM - Eric Faulhaber

To clarify, you can access a persistent table across threads, but not over the same connection. A temp-table actually only exists privately on a connection, so it can't be accessed at all from a separate thread.

#14 - 01/09/2020 05:56 PM - Greg Shah

Understood. To implement "temp-tables" that are shared across threads we would have to implement a multi-user database instance that works like _temp. Only the special shared temp-tables would be used in this manner. Presumably, the same DMOs would be used.

To be clear: I'm not planning to implement this **unless** you tell me this is easy to implement.

#15 - 01/10/2020 05:32 PM - Greg Shah

- Status changed from New to WIP

- Subject changed from add support for running ABL on mulitple threads in a single client to add support for running 4GL on multiple threads in a single session

To create a new thread, I was planning to add an optional AS-THREAD [SET handle] extension to the RUN statement. The idea is that you would be able to write code like this:

```
RUN some-internal-proc1 AS-THREAD.
RUN some-internal-proc2 AS-THREAD (INPUT var1).
RUN some-internal-proc3 AS-THREAD SET h.
RUN some-internal-proc4 AS-THREAD SET h (INPUT var1).
RUN ext-proc1.p AS-THREAD.
RUN ext-proc2.p AS-THREAD (INPUT var1).
RUN ext-proc3.p AS-THREAD SET h.
RUN ext-proc4.p AS-THREAD SET h (INPUT var1).
```

It would be subject to the following limitations:

- Types of top-level code to be supported.
  - Only internal and external procedures can be run on threads.
  - The 4GL has no void functions so they can be excluded.
  - Triggers must always/only ever be executed on the "main" thread (the Conversation thread in most cases).
  - We could also implement some Java class based approach to start a new thread, passing in some 4GL OO but we don't currently support inheriting from Java classes, implementing Java interfaces, anonymous inner classes or lambdas. So there is no obvious way to delegate the execution of 4GL OO code. I don't like the idea that the class-based methods are excluded, but I think we must add some enhancements to the direct Java OO support for it to work.
- Only input parameters can be passed.
- We would extend ControlFlowOps with new invokeAsThread[Set][WithMode]() variants. This will allow us to most cleanly manage the procedure manager and other critical state that must be handled for the invocation.
- NONE of these are allowed:
  - PERSISTENT
  - SINGLE-RUN
  - SINGLETON
  - ON SERVER
  - IN
  - ASYNCHRONOUS
  - TRANSACTION DISTINCT
- If specified, the handle would probably point to the procedure that was started. We will probably need to provide a small set of additional things you can do with the returned handle, the most important of which would be to access the Java Thread instance.
- I think the core of the RUN processing needs to be on the caller's thread, except the invocation of the found code itself which would be on an AssociatedThread. By doing it this way, failures (procedure not found, parameter mismatches...) will be reported directly but the actual processing is done on the new thread.

I'm interested in general feedback on the above.

Constantin, I have these specific questions for you:

- Should we exclude having super-procedures on the AssociatedThread instances? Lots of old code might require such a thing, but it is not clear to me if this would cause problems. It is also not clear how one would safely setup the super proc in this case.
- I don't see a good reason for the "thread-procedures" to be persistent, am I missing some advantage?
- Do we need to worry about whether the main thread was run persistently? Are there resource management implications?
- I assume the normal THIS-PROCEDURE processing will work OK here?
- On the AssociatedThread, I assume we will have to do something like StandardServer.invoke(int stopDisp, Isolatable entry). Do you see a better way?

#16 - 01/13/2020 07:17 AM - Ovidiu Maxiniuc

Greg,
I think we should support the thread "JOIN" concept by using an additional option AS-THREAD [SET handle] [ON-FINISH callback-proc] (or CALLBACK in order to reuse a KEYWORD). The callback-proc will be called on main thread when the called procedure end. Probably this callback/notification procedure will have a parameter (or more, to be decided) to let the caller know of the outcome of the execution: whether or not the routine ended with success and possible return value.

#17 - 01/31/2020 09:03 AM - Greg Shah

I'm thinking through the JOIN implementation. I agree it is a good idea.

I also am thinking about the following:

- When the main thread exits, we must kill/interrupt all the associated threads. This is similar to how an operating system process exits when the main thread exits, even if other threads are still active.
- I suspect we need to enforce a particular ordering of session context cleanup (associated threads cleanup first).
- I wonder if we need anything special to cleanly handle the server shutdown?

#18 - 01/31/2020 09:38 AM - Constantin Asofiei

Greg Shah wrote:

- Should we exclude having super-procedures on the AssociatedThread instances? Lots of old code might require such a thing, but it is not clear to me if this would cause problems. It is also not clear how one would safely setup the super proc in this case.

I think we shouldn't exclude, but create a copy of SESSION's super-procedures, and in case of the internal procedure call, the THIS-PROCEDURE's list, also. This way, the caller and callee can both use and manage these lists separately.

- I don't see a good reason for the "thread-procedures" to be persistent, am I missing some advantage?

What we could do (if that is persistent) is let the caller invoke other internal procedures AS-THREAD. As Ovidiu mentioned, we should add a callback procedure (similar to RUN ASYNC options) so that the caller knows when the thread has finished. This callback in turn (if the procedure is ran persistent) allows the caller to invoke internal procedures on that persistent procedure (note that at the time the callback is executed, the callee's thread will be finished, but its resources - from the persistent procedure - must survive).

The same applies if the RUN AS-THREAD is used on internal procedures, for an external program ran persistent (or not). And this raises the question - what happens if someone deletes this procedure handle (or the external program finishes execution), while there are active AS-THREAD invocations on its internal procedures? We should add some attributes to the external program handle, to be able to track the number of active AS-THREAD invocations (for internal procedure) on this external program. Or some method like WAIT-FOR-THREADS to wait until all invocations finish. Or don't allow the external program (which is not persistent) to terminate until all threads finish (for persistent programs, fail the DELETE if there are active threads on internal procedures).

- Do we need to worry about whether the main thread was run persistently? Are there resource management implications?

As for resource management implications - except for the above, we need synchronized access on any resource/variable which is accessed from the internal procedure ran AS-THREAD, and **NOT LOCALLY SCOPED** to this internal procedure; same applies for the external program ran AS-THREAD - any access to a resource/variable not locally scoped must be synchronized. This is because I see all the AS-THREAD invocations sharing the FWD context with the caller, except for the super-procedure limitations.

But, see bellow for more concerns...

- I assume the normal THIS-PROCEDURE processing will work OK here?

Yes, I don't see issues here.

- On the AssociatedThread, I assume we will have to do something like StandardServer.invoke(int stopDisp, Isolatable entry). Do you see a better way?

Hmm... this raises some other questions. What is the 4GL stacktrace for an AS-THREAD invocation? Is it inherited from the caller or it has its own? If the AS-THREAD is considered to have its own 'global block', then yes, I think that API must be used.

But there are other consequences when we consider the 4GL stacktrace and the resources (like temp-tables) created in the caller's context - for example, one can create a temp-table having as template another one created in a previous scope (on the stacktrace), and reference this temp-table by name. The same applies to any 'scoped dictionary' state which is from the caller (aka Scopeable implementations in FWD) - the state kept by these in ScopedDictionary instances will evolve on both the caller and callee threads. And on the caller, these scoped data **must** evolve naturally. I think we need to either allow the callee to not inherit state from the caller (as in, it starts its own 'sub-context'), or impose serious limitations on what it can do when looking for data 'below the scope from it was started' - this may impose resource leaks, like the caller destroys something (naturally, as it goes down the stack as it exits top-level blocks), and the thread saved some resource from that scope.

You mention only INPUT parameters are allowed - but we should exclude HANDLE/COMHANDLE/OO parameters (i.e. which are not BY-VALUE); or let the programmer handle the synchronization, when using these.

#19 - 01/31/2020 09:43 AM - Constantin Asofiei

Constantin Asofiei wrote:

> Greg Shah wrote:
>
> - Should we exclude having super-procedures on the AssociatedThread instances? Lots of old code might require such a thing, but it is not clear to me if this would cause problems. It is also not clear how one would safely setup the super proc in this case.
>
> I think we shouldn't exclude, but create a copy of SESSION's super-procedures, and in case of the internal procedure call, the THIS-PROCEDURE's list, also. This way, the caller and callee can both use and manage these lists separately.

And when adding the resource lifetime mentioned bellow, the callee's copies need to be aware if some program in these lists has been destroyed by the caller (or any other threads). Maybe impose some restrictions on who can delete a resource - only the creator thread?

#20 - 01/31/2020 10:39 AM - Greg Shah

> Should we exclude having super-procedures on the AssociatedThread instances? Lots of old code might require such a thing, but it is not clear to me if this would cause problems. It is also not clear how one would safely setup the super proc in this case.

> I think we shouldn't exclude, but create a copy of SESSION's super-procedures, and in case of the internal procedure call, the THIS-PROCEDURE's list, also. This way, the caller and callee can both use and manage these lists separately.

This makes sense. But my worry is that this pulls in quite a lot of legacy insanity and opens up many cases of unexpected and unsafe resource usage from other contexts.

My instinct is to disallow this. The new threads are related from the perspective that they can share thread safe resources, but all the legacy (unsafe) resources should not be shared.

> I don't see a good reason for the "thread-procedures" to be persistent, am I missing some advantage?

> What we could do (if that is persistent) is let the caller invoke other internal procedures AS-THREAD.

Yes, we **could** but I don't think we **should**. I consider persistent procedures to be a poor design decision in the 4GL. I think they only did this to try to compensate for the lack of OO support in the 90's. It made a mess and is not a good idea. With OO, it is no longer necessary in my opinion.

I prefer not to include this into our threading solution unless you have a reason that it is needed. Is it needed?

> The same applies if the RUN AS-THREAD is used on internal procedures, for an external program ran persistent (or not).

I was thinking that we **should** allow internal procedures to be run AS-THREAD. But your point about accessing the resources of the external procedure is a good one. I think this is a bad idea for any resource except those which have been explicitly made thread-safe (see #3254-2).

I think this is an acceptable limitation, but I need to figure out a way to detect (and raise an error) on such accesses. The point here is that anyone that writes threaded code will be required to carefully consider how data is shared in a thread-safe manner, using the thread-safe facilities we are providing.

> And this raises the question - what happens if someone deletes this procedure handle (or the external program finishes execution), while there are active AS-THREAD invocations on its internal procedures?

I see no reason to support the main thread being run persistently. That is not how multi-threaded applications are written. They either have the main thread do real work until the process exits or they explicitly block the main thread until all other threads complete their work. In Java, both of these models are possible though we have to implement our own version since the main thread of a related session is not actually the main thread of the server process. This is where Ovidiu's idea of implementing JOIN makes sense. The usual implementation basically blocks a given thread until all other threads finish their work and JOIN (un-fork) the main thread. I think in the common case we don't need a callback, but I do see the callback option as useful too.

As noted in #3254-17, I plan that exit of the main thread will kill any remaining related threads.

> We should add some attributes to the external program handle, to be able to track the number of active AS-THREAD invocations (for internal procedure) on this external program. Or some method like WAIT-FOR-THREADS to wait until all invocations finish. Or don't allow the external program (which is not persistent) to terminate until all threads finish (for persistent programs, fail the DELETE if there are active threads on internal procedures).

Yes we will probably want to add some attributes. The WAIT-FOR-THREADS idea is the main thread's counterpart to JOIN. The tricky part is to figure out how flexible to make this. The common case is for the main thread to wait for all related threads. I can think of a more general use case but implementing it would require more complex syntax that identifies which threads to wait for and which thread to join.

> As for resource management implications - except for the above, we need synchronized access on any resource/variable which is accessed from the internal procedure ran AS-THREAD, and NOT LOCALLY SCOPED to this internal procedure; same applies for the external program ran AS-THREAD - any access to a resource/variable not locally scoped must be synchronized.

Yes, but we need to figure out how to detect when an unsafe access is occurring. I don't like the idea that all of this data is moved into context local form. But we might need to know the context in which we are allowed to be accessed.

> This is because I see all the AS-THREAD invocations sharing the FWD context with the caller, except for the super-procedure limitations.

I think we still have something to figure out here. The AssociatedThread today does share the same FWD context as the spawning thread. And using that model would be convenient for sharing resources at the context level. BUT today we also store all of our transaction state, block manager state etc... in that same context. This would mean that all threads would be inside the same transaction scope. This cannot work. At a minimum:

- It breaks all the normal contracts with block-level scoping used by the 4GL.
- JDBC connections cannot be actively used on more than one thread. So the database transaction model would break.

So I think we must design this as separate FWD contexts which share the same FWD account AND in which some well-defined thread-safe resources can be shared between those contexts.

> What is the 4GL stacktrace for an AS-THREAD invocation? Is it inherited from the caller or it has its own? If the AS-THREAD is considered to have its own 'global block', then yes, I think that API must be used.

Yes, each thread MUST have its own global block.

> I think we need to either allow the callee to not inherit state from the caller (as in, it starts its own 'sub-context')

Yes, I think we must find a way to exclude this inheritance.

> You mention only INPUT parameters are allowed - but we should exclude HANDLE/COMHANDLE/OO parameters (i.e. which are not BY-VALUE); or let the programmer handle the synchronization, when using these.

I think it is important to allow OO parameters for sure and probably HANDLE too. COMHANDLE seems less useful since UI processing MUST NEVER happen on the related threads (only the main "Conversation" thread is allowed to call the client).

But we must allow OO parms so that meaningful sharing of data and functionality can occur. I agree, that the programmer is responsible for ensuring such usage is thread safe.

#21 - 01/31/2020 11:12 AM - Constantin Asofiei

Greg Shah wrote:

> My instinct is to disallow this (super-procedures). The new threads are related from the perspective that they can share thread safe resources, but all the legacy (unsafe) resources should not be shared.

I agree we can disallow it; my points were mainly what are the implications if we allow it.

> I prefer not to include this into our threading solution unless you have a reason that it is needed. Is it needed?

I agree, we can exclude the persistent programs case.

> I think this is an acceptable limitation, but I need to figure out a way to detect (and raise an error) on such accesses.

I think we need to track an external program's 'instantiating thread' - this will allow you to identify if an internal procedure is ran on the same thread as its persistent program. And if we track it, we can expose it to 4GL via an attribute; same for 'current thread'.

> I see no reason to support the main thread being run persistently. That is not how multi-threaded applications are written. They either have the main thread do real work until the process exits or they explicitly block the main thread until all other threads complete their work. In Java, both of these models are possible though we have to implement our own version since the main thread of a related session is not actually the main thread of the server process. This is where Ovidiu's idea of implementing JOIN makes sense. The usual implementation basically blocks a given thread until all other threads finish their work and JOIN (un-fork) the main thread. I think in the common case we don't need a callback, but I do see the callback option as useful too.

If the external program can't terminate until all its threads are finished, then I'm OK with checking just the SOURCE-PROCEDURE that is not ran persistent. Otherwise, we need to look on the entire stack.

> As noted in #3254-17, I plan that exit of the main thread will kill any remaining related threads.

That makes sense.

> So I think we must design this as separate FWD contexts which share the same FWD account AND in which some well-defined thread-safe resources can be shared between those contexts.

Yes, with more thinking into this, I think the approach looks more to 'what data we need to allow to be shared' instead of 'what data we do not allow to be shared', as what is shared is significantly less than what is not.

We may consider that this related thread shares with the parent and other related threads only its security data, plus any synchronization and data sharing primitives made available. How is this done? Via NEW GLOBAL SHARED variables? Or only via the INPUT arguments?

#22 - 01/31/2020 01:30 PM - Greg Shah

> We may consider that this related thread shares with the parent and other related threads only its security data, plus any synchronization and data sharing primitives made available. How is this done? Via NEW GLOBAL SHARED variables? Or only via the INPUT arguments?

1. I think that we should introduce a new 4GL annotation @ThreadSafe which can be used to mark internal procedures as allowed to be run on its own thread. We may need something similar for external procedures but this may need a new language statement at the top of the file. The idea is that the conversion can attempt to detect any unsafe resource usage and fail conversion if so found. This annotation would be set as a flag in our name_map.xml.

2. RUN ... AS-THREAD would check the thread-safe flag and disallow execution unless it is present.

3. In thread-safe programs, the only non-local data that will be available will be our specially crafted concurrent resources (examples in #3254-2). This will include atomic no-undo BDT variables. Such resources will be shared using an explicit naming scheme and access will be controlled via security plugin. For example, one will have to create() or open() a new instance of a resource and you will pass a name (e.g. /event_semaphore/<resource_instance_portion>). This is how multiple threads can safe resources. One thread calls create() and the others call open(). The resulting resource instance is truly a shared reference to the same object but each resource implementation will be carefully crafted for thread safety. We will not allow 4GL shared-resource (e.g. DEFINE NEW SHARED or DEFINE SHARED), including global forms. Such resources were never designed for thread safety. INPUT parameters will also be allowed, but they must also be thread safe.

4. It must be possible to write a new OO class in a way that is thread safe. This means we will need to use the annotation here too and more importantly, we must provide locking/synchronization primitives that are sufficient for this purpose. I don't know if we wil to directly expose Java synchronized blocks and Object monitors or if we will provide something a little "higher level". If a class is thread safe then instances can be passed as INPUT parameters.

I do want to be able to run thread safe code as a new thread from a "main thread" that is not itself fully thread safe. This will allow legacy code running on the main thread to still take advantage of additional threads.

Is there a problem with that idea?

#23 - 01/31/2020 01:44 PM - Constantin Asofiei

Greg Shah wrote:

> 1. I think that we should introduce a new 4GL annotation @ThreadSafe which can be used to mark internal procedures as allowed to be run on its own thread. We may need something similar for external procedures but this may need a new language statement at the top of the file. The idea is that the conversion can attempt to detect any unsafe resource usage and fail conversion if so found. This annotation would be set as a flag in our name_map.xml.

Please try to move away from name_map.xml - we have a 321MB name_map.xml for a project. Please plan to emit this annotation at the Java method and (once the correct Java method is resolved) check it from there with reflection. In the future, my goal would be to have the entire name_map.xml just a registry of external programs/4GL classes (and maybe not even this, if we can scan the entire application jar and get all classes with a special annotation - external program or 4GL class).

Also, for 4GL classes, I already rely only on the LegacySignature annotation, to make method overloading work easily.

> Is there a problem with that idea?

Not beside the above.

#24 - 02/11/2020 05:39 PM - Greg Shah

4335a revision 11414 adds the following changes:

- Refactored the AssociatedThread to move code into a base class (ContextAwareThread) which can be used by other thread types (RelatedThread).
- Modified the security context processing to allow RelatedThread to easily create new independent contexts patterned off the current context. This means that the contexts will share subjects, ACLs but will NOT share context local data.
- Created a base class (RelatedResource) for concurrency primitives and a first subclass (EventSemaphore) implementing a resource.

#25 - 02/12/2020 02:21 PM - Greg Shah

I think this is the list of open items:

- Constantin is going to write ControlFlowOps.invokeAsThread() and ControlFlowOps.invokeAsThreadSet(). Because of our simplifying assumptions, I expect that much of the existing invocation implementation is unnecessary. Hopefully a simpler, independent worker can do this job. This would reduce risk and presumably make the implementation easier. The worker will use the new RelatedThread which handles the new context creation patterned off the existing thread's context.
- Implement the following thread life cycle management items:
  - Kill related threads and cleanup when the main thread exits.
  - Ensure that everything exits safely/cleanly on server shutdown.
  - Add JOIN/WAIT-FOR-THREADS capability.
  - Implement any changes needed for related thread session cleanup on normal exit.
- Add concurrency primitives including:
  - thread safe queue
  - mutex semaphore
  - safe/atomic BDTs
  - Can we add some kind of temp-table copying helper?
- #4065 (execute client features on the server), this would include the security aspects.
- Although the context created by the RelatedThread has no network session (and the network socket is not used as its session id), I think we still need to implement an approach to gracefully handle the case where code on that thread attempts to access a client API. This work needs to mesh with #4065 and only needs to cover those APIs that have no server-side version.
- Add 4GL language features to allow creation of thread safe OO 4GL code. Such objects would be allowed as parameters and could be shared between threads.
- Add attributes to the session handle to provide some way to inspect the state of the threading environment, possibly walk the list of threads, read exit codes and similar.

- Add conversion analysis and safety checks to abort conversion when non-thread-safe code is used with threads. We may postpone this last item since it is not really needed for usage.

#26 - 02/12/2020 02:24 PM - Greg Shah

One question I have is whether the blocking methods in the concurrency primitives (e.g. EventSemaphore.await()) should provide an option to translate an InterruptedException to a STOP condition. In the current approach, we return a WaitStatus enum from EventSemaphore.await() and this can be checked for WaitStatus.INTERRUPTED. If we do provide this, I think it should just be an option and not the only/default implementation.

#27 - 02/12/2020 02:25 PM - Constantin Asofiei

Review for 4335a rev 11414-11515:

- RelatedResource - do we want to use ErrorManager.recordOrThrow instead of throw new ErrorConditionException? To allow NO-ERROR mode.

Otherwise, I don't see anything obvious.

#28 - 02/12/2020 02:29 PM - Constantin Asofiei

Greg Shah wrote:

> One question I have is whether the blocking methods in the concurrency primitives (e.g. EventSemaphore.await()) should provide an option to translate an InterruptedException to a STOP condition. In the current approach, we return a WaitStatus enum from EventSemaphore.await() and this can be checked for WaitStatus.INTERRUPTED. If we do provide this, I think it should just be an option and not the only/default implementation.

The STOP condition can be caught and managed by business logic; if this is from a related thread, and the business logic does not manage this, will it terminate the related thread?

#29 - 02/12/2020 02:42 PM - Greg Shah

> RelatedResource - do we want to use ErrorManager.recordOrThrow instead of throw new ErrorConditionException? To allow NO-ERROR mode.

Good question.

My approach with these concurrency primitives was that there would be no additions to the 4GL syntax. Instead, I am implementing all of these as pure Java code, which the 4GL is expected to use directly. With this in mind, I was thinking that we would not support NO-ERROR.

On the other hand, since a standalone expression or assignment can have NO-ERROR specified, we probably do need to honor it. What do you think?

#30 - 02/12/2020 02:42 PM - Greg Shah

> The STOP condition can be caught and managed by business logic; if this is from a related thread, and the business logic does not manage this, will it terminate the related thread?

Yes. This is no different to how the current threads work. It seems reasonable.

#31 - 02/12/2020 02:56 PM - Constantin Asofiei

Greg Shah wrote:

> On the other hand, since a standalone expression or assignment can have NO-ERROR specified, we probably do need to honor it. What do you think?

Yes, I think we should honor NO-ERROR.

#32 - 02/13/2020 02:32 PM - Constantin Asofiei

Greg, do you want me to code the #3254-16 and #3254-17 ideas, too?

#33 - 02/13/2020 02:51 PM - Constantin Asofiei

The support for RUN ... AS-THREAD [SET ht] is in 4335a rev 11418. Please review.

The resource which is supposed to manage the thread is not added.

#34 - 02/13/2020 03:04 PM - Greg Shah

Constantin Asofiei wrote:

> Greg, do you want me to code the #3254-16 and #3254-17 ideas, too?

No, we can handle that later.

#35 - 02/13/2020 05:11 PM - Greg Shah

Code Review Task Branch 4335a Revision 11418

Why was it better to add this into the common CFS.invokeImpl() code instead of implementing a simpler CFS.invokeAsThreadWorker()? Considering that we are deliberately trying to simplify the AS-THREAD assumptions, the CFS.invokeImpl() brings a lot of baggage. And now it is more complicated.

#36 - 02/14/2020 09:54 AM - Constantin Asofiei

Greg Shah wrote:

> Code Review Task Branch 4335a Revision 11418
>
> Why was it better to add this into the common CFS.invokeImpl() code instead of implementing a simpler CFS.invokeAsThreadWorker()? Considering that we are deliberately trying to simplify the AS-THREAD assumptions, the CFS.invokeImpl() brings a lot of baggage. And now it is more complicated.

OK, then how simple do you want AS-THREAD to be? Because all the error processing and target resolution is in invokeImpl()... I don't want to drop that.

We still want to be able to run internal procedures, right?

#37 - 02/14/2020 10:10 AM - Greg Shah

> Because all the error processing and target resolution is in invokeImpl()... I don't want to drop that.

In a perfect world, this processing would be refactored into helpers that could be used from both places.

However, we don't have the time to do an extensive refactoring here. And I don't want to add risk.

> We still want to be able to run internal procedures, right?

Yes.

Leave everything as is for now. I just want you to know the long term objective so that when the time is right, we can refactor the code accordingly.

#38 - 02/14/2020 10:12 AM - Constantin Asofiei

Greg Shah wrote:

> Leave everything as is for now. I just want you to know the long term objective so that when the time is right, we can refactor the code accordingly.

OK. I'll think of improving this next time I'm touching that (probably for the ThreadSafe annotation and more complex argument validation - to allow only INPUT)

#39 - 03/13/2020 03:45 PM - Greg Shah

Task branch 4335a was merged to trunk as revision 11345.

#40 - 09/24/2020 10:01 AM - Greg Shah

- Related to Feature #4912: move UI portions of the web client to the server-side added