Threads in SML

1. Overview

This document is intended to explain how threads are used in Soar 8.6 and later (this document is being written against 8.6.3). It assumes you already have a passing familiarity with both Soar and the SML interface language. This is advanced reading, for those who want to understand everything that's going on "under the hood".

2. What Threads Exist

As with all things related to SML, we need to divide up the world into client-side threading and kernel-side threading.

The kernel side threading is relatively simple. The kernel is either run in the client's thread or in its own separate thread. This choice is made by the client when it initializes the Soar kernel by calling either Kernel::CreateKernelInCurrentThread() or Kernel::CreateKernelInNewThread(). A remote connection (Kernel::CreateRemoteConnection()) doesn't affect the way the kernel is run, this is determined by the local client that created the kernel initially. If the kernel is running in its own thread I'll call that thread the KernelThread.

The client side is potentially more complex. First, the client application may inherently be multi-threaded, such as a Java GUI app or it may be a simple single-threaded C++ program (e.g. a command line utility). Second, the calls to start Soar running (e.g. RunAllAgentsForever) block, so many clients will choose to execute this call in a separate thread in order to keep the rest of the application responsive. I'll call this the RunThread, which means the thread where the run call is initiated. Finally, SML itself starts up (by default) a thread called the EventThread. This thread is intended to keep the client responsive without a lot of work by the application developer.

This gives us a picture of the overall set of threads:

	Runs in	**Description**
KernelThread	Kernel	Keeps the kernel responsive to external commands (i.e. ones coming in over a socket connection).
EventThread	Client SML	Used to service incoming events sent by the kernel if Soar is not running (i.e. productions are not firing). This thread is optional but is created by default.
RunThread	Client	Point that run commands are executed. This thread is created by the client, either explicitly with a new thread or implicitly by calling run from the client's main thread.

Client Threads	Client	Other threads that the client application may have for its own use. Typical examples are window manager threads in GUI apps.

3. Why Have Multiple Threads At All?

As we're about to dive into the complexity of the threading in SML, one obvious question is why not just have a single thread and make everyone's life easier. Well, the answer is that would make the SML/kernel developer's life easier but at the cost of making the client/application developer's life harder. Let me explain why that is more fully.

3.1. Why Have a Kernel Thread?

With Soar 8.6 a single Soar kernel can have multiple clients connected to it at once. A common situation is an environment in Java that has a local connection to the kernel itself. A debugger is connected remotely to the kernel and a logging application is also remotely connected. Commands can be sent to the kernel over a socket. The question is, how and when should the kernel check this socket for new commands?

In the single-threaded model, the kernel is running in the same thread as the client that created it (in our example, the Java environment). In this case, the client is required to poll the socket periodically to see if new commands have arrived. This is exactly what the method Kernel::CheckForIncomingCommands() does and clients that call CreateKernelInCurrentThread() are required to call this periodically. But making such "periodic" calls is often difficult for a client. In the case of the Java app the user would need to start some sort of timer and poll across to the kernel whenever it went off. If they fail to realize that this must be done, then the debugger would fail to function at all, leading to a lot of "why doesn't the debugger respond" problems. In other clients, making periodic calls may be really complicated. Simple command line utilities tend to take the form: get command from the user, do some work, print some results, get another command. Writing these would be much more challenging if the client is not allowed to block while getting input from the keyboard.

So the result is that we recommend running the kernel in its own thread in most cases. That separate thread checks for new commands coming in from either the socket or the local client. In this model, commands are always run in the kernel's thread. That means if the local client calls "run" what actually happens is that command is placed on a message queue and the kernel thread then pulls the command from that queue and executes it. That's important to understand if you're trying to debug a problem at the kernel level as the client's function call just adds a message to a queue and you usually need to break the execution at the point that the kernel's thread has picked up the message (e.g. in KernelSML::ProcessCommand()). You won't see the client's triggering call higher up in the call stack at that point as it's in a different thread.

3.2. Why Have an Event Thread?

When Soar is running, it is constantly generating a stream of events. These events are posted to each client, allowing the client to know what's going on during the course of a run. The EventThread exists to help clients remain responsive to these incoming events. The first thing to understand about Soar events is that they are synchronous calls. That is to say, the kernel blocks during the call to send each client a message that a particular event has occurred. The kernel only continues executing once that call completes. It may not be immediately obvious that events need to be synchronous (indeed early on we thought they might not

be) but in turns out to be useful they must be. For example, consider an agent that wishes to take some action when a new phase starts so it registers for the start phase event. If the call is not synchronous, then by the time the event arrives the kernel may now be at a much later execution point. This would limit clients to knowing what had happened, but not doing things at particular times (and this turns out to be 95% of the actual usage).

So given that event calls are synchronous and the kernel blocks waiting for them to complete, how do we ensure that clients are responsive. In the case where the kernel and the client are running in the same thread there is really no problem. The event call is just a function callback and the callback executes in the kernel's thread. But when the kernel and client are running in separate processes connected by a socket, the client needs to check for new messages arriving on the socket. As with the kernel thread, this could be done by requiring the client to periodically poll for new events coming in on the socket (and again this mode is supported by having clients call Kernel::CheckForIncomingEvents()). But this periodic polling can be complicated as explained earlier. In fact, the situation is even worse for the client as it needs to be as fast as possible responding to incoming events or the entire system's performance will degrade. This is where the EventThread comes in. Its job is to check the socket for new events and make the appropriate callback to the client code. This keeps the client responsive and most people will never stop to wonder how a callback inside one process is being triggered by a kernel running in another process without their doing anything.

There is however a wrinkle in this behavior. When a run call is made by a client (on what we're calling the RunThread) that call blocks. While it is blocked it is checking for incoming messages and dispatching them, while it waits for the message to indicate that the run has completed to come through. This means that when a run is triggered by a client callbacks will occur on the RunThread. When that client did not initiate the run (e.g. somebody tied "run" in the debugger) or when Soar is not running (e.g. during an init-soar event), events are handled by the EventThread. This means that callbacks can be called on different threads at different times.

3.3. Callback threading and blocking on run

Let's discuss that a little more. First of all, having the calls come back on different threads isn't ideal. It means a client developer could have different bugs that occur when Soar is run from the client or when Soar is run remotely because the threading is different. In practice, this hasn't really been an issue. On a larger front though it suggests maybe the "run" calls should not in fact block. If they didn't block, the client's life would be easier (they wouldn't need to create a thread when they started a run) and all callbacks would come in on the event thread. The fact that run does block is largely an artifact of how we used to design environments. The main run loop used to look something like:

```
While (!stopped)
{
      Run-agents-1-step()
      Update-world()
}
```

With this model we really want the run call to block so that each single step was a single call (writing this where Run-agents-1-step is non-blocking would be much harder). But over the development of 8.6 we learned that we really didn't want this model and instead we now have a model more like this:

RegisterCallback(update-after-output-event, updateWorld)

```
While (!stopped)
{
      Run-agents-forever()
}
```

This new event driven model means that having "run-agents-forever" be a non-blocking call wouldn't really present a problem at all. In fact it would probably simplify matters for the client side developer. However, it's a very significant adjustment (breaking lots of existing code) and so far we haven't acitvely considered making this change.

4. Locking and Thread-Safety

Given that there are these multiple threads in the system you may need to understand how locking is implemented, to ensure thread safety across the system. The approach we've taken is to assume that the kernel is itself not thread-safe at all. E.g., we'll assume it's not safe to remove a production at the same time a preference is being created as the result of another production firing. Therefore, the approach is to only allow a single command to execute in the kernel at a time.

In practical terms, this means even if multiple clients are connected through remote connections their commands are queued up, together with the commands that come from the local connection and executed one at a time. The code for this is in ReceiverThread.cpp. This ensures that the kernel side only executes one command at a time.

We also assume that the messaging code is not necessarily thread safe, so we have the client block from sending two messages at the same time. This is achieved with the `m_ClientMutex` mutex object defined in the Connection class. This decision has a couple of implications.

First, it allows the EventThread and the RunThread in the client to live in harmony. A run call will wait for the current event to be processed before it starts and once the run has begun, this lock forces the EventThread to wait for the run to complete. As a result, all events during a run (started by that client) come back on the RunThread and all events outside of a run come back on the EventThread.

Second, the presence of a lock on the client makes interruption a bit tricky. If a client issues a "stop" command from a different thread than the RunThread it will block, waiting for the run to complete which is not very helpful. The solution is to have the client issue the stop in an event handler callback, as that callback will always occur on the RunThread if the client is currently running Soar. (If you really want to follow the different cases, the callback might be called by the EventThread but only if the client has not called "run", in which case the stop call will go through because the RunThread will not be holding the lock or at least not holding it for a long time).

You might wonder, given this model how does the debugger print out state information during a run (e.g. after every 5 decisions). The answer is that it does the work in a callback so the print commands execute on the same thread as the original run command.

So while we do not support multiple commands being executed simultaneously (where a second command starts up at an arbitrary time during the first command), we do support multiple commands being executed in a stack (or with reentrancy) where one command is partially complete (such as a run) and another command is executed in the middle of the first command. The keys to this are that the run is paused while this happens (it's in an event handler to be more precise) and the second command executes at a known time (inside an event handler) so the state of the system is well defined. We often end up with large sequences of calls going back and forth inside an initial trigger (init-soar is famous for this) as one action triggers an event, which triggers another action and so on. But it's all within a single logical stack of commands, rather than having multiple simultaneous threads of execution.