Non-Blocking Interfaces Made Easy

Tom
Glod

#LiveCodeGlobal

Session Content:

- Introduction
- Example File
- Why Interfaces Freeze
- Build a Scheduler
- Slow it Down with Wait
- Time Stuff
- Send in Time
- Pseudo Callbacks with Wait Command
- Screen Lock Unlock
- What Blocks What Doesn`t
- Things to Remember
- Further Resources

Tom Glod
Tom@MakeShyft.com
Developer @ MakeShyft R.D.A

#LiveCodeGlobal

Introduction

What is a blocking interface?

An interface that blocks user input or interaction while a long or intensive handler is being executed

A block most often occurs with large chunks of code running in long recurring repeat loops

What is a non-blocking interface?

An interface that continues to respond to user interaction (typing, mouse movements, clicks, drags) while it is calculating or updating data in the background, or updating aspects of the interface itself.

Blocking is always the default behavior.

GLOBAL

#LiveCodeGlobal

Example File

Our example file features a LC Datagrid containing the Stocks available on the Toronto Stock Exchange.

The file simulates real-time, continuous, non-blocking updates to the visible rows.

The non-blocking aspect is demonstrated by the ability to continue to interact with interface items despite any background work that is happening.

The new data is simulated, not obtained from real data sources.

Demo stack in running on old Intel dual core G2250 which gets 2800-3000 on Geekbench. Relatively slow processor.

GLOBAL

#LiveCodeGlobal

Why Interfaces Block

repeat with x =1 to SomeLargeNumber

… do something

… do something

… do something

… do something

… do something

end repeat

}

Livecode Engine runs on a single clore and is a single process to the operating system.

100% Processor Core Utilization until loop is complete. Computer executes as fast as possible.

If long enough, and intensive enough, the interface blocks.

Windows may even be “(not responding)”

GLOBAL

#LiveCodeGlobal

Build a Scheduler

- Its easy to build a reliable scheduler which executes your program in a way that maximizes Livecode's ability to let messages interweave in the message path. Eventually all the messages are handled and all the code executes.
- A properly built scheduler will never lose track of its processes.
- Performance hit, but Livecode is fast, at times really fast.

Time Stuff

When building a scheduler it helps to know how long things take. Build timing into your scheduler and you can time everything. Without any additional work. Overhead is incredibly low.

LC GLOBAL

#LiveCodeGlobal

Slow it Down with Wait

repeat with x =1 to SomeNumber

… do something

… do something

… do something

… do something

wait x milliseconds with messages

end repeat

GLOBAL

#LiveCodeGlobal

Send In Time

1. send “Message” to stack “MyStack”

SENDS THE MESSAGE AND WAITS FOR IT TO FINISH EXECUTING, ONLY THEN CONTINUES

2. send “Message” to stack “MyStack” in 0 milliseconds

CONTINUES HANDLER & LETS CURRENT HANDLER FINISH BEFORE MESSAGE IS SENT

3. send “Message” to stack “MyStack” in x milliseconds

CONTINUES HANDLER & LETS x AMOUNT OF TIME PASS BEFORE MESSAGE IS SENT.

LC GLOBAL

#LiveCodeGlobal

Pseudo Callbacks with Wait Command

```
send “UpdateSomething ” & SomeUniqueID in 0 milliseconds

put the result into SchedulerArray[SomeUniqueID][“message #”]

wait until SchedulerArray[SomeUniqueID][“status”] is not empty with messages

.. check result of update

.. do something
```

When your handler continues …. You know the scheduled task completed because the status variable was set for that particular one. Until this is set, this “thread” of this function or command waits BUT the engine is continuing to send and receive messages.

GLOBAL

#LiveCodeGlobal

Pseudo Callbacks with Wait Command

```
On UpdateSomething MyUniqueID
        put the milliseconds into SchedulerArray[MyUniqueID]["start time"]
        .. do stuff
        … do more stuff
        put some_array_variable into SchedulerArray[MyUniqueID]["result"]

        If something = true then
                put "OK" into SchedulerArray[MyUniqueID]["status"]
        else
                put "ERROR" into SchedulerArray[MyUniqueID]["status"]
        end if

        put the milliseconds into SchedulerArray[MyUniqueID]["end time"]

End UpdateSomething
```

LC GLOBAL

#LiveCodeGlobal

Screen Lock

Speed up execution by locking the screen and unlocking it only every x amount of milliseconds. 33 milliseconds for a 30fps experience.

```
On UpdateSomething MyUniqueID
        lock screen
        … do something
        … do something

End UpdateSomething

On UnlockScreenEvery
        unlock screen
        send “UnlockScreenEvery” to me in 33 milliseconds
End On UnlockScreenEvery
```

GLOBAL

#LiveCodeGlobal

What Blocks What Doesn`t

In Livecode the following commands are non-blocking

- Move Command
- Load URL (if different domain)
- (TsNet HTTP Requests are faster and don’t have the limitation above)
- Read from Process
- Write To Process
- Read from Socket
- Write To Socket

#LiveCodeGlobal

Things to Remember

When building programs that run on an infinite loop and require continuous updates to its user interface without blocking it.

- Build it in from the beginning if you can
- Write your code in small manageable chunks. The longer your handler, the more chance it can block the interface.
- Build a way to debug & test to make sure your scheduler isn't losing messages.
- Build handlers for failure, to make sure they always finish
- Optimize second
- Setting custom properties is MANY times slower than setting a variable.
- USE FULL REFERENCES TO CONTROLS (the long id)

GLOBAL

#LiveCodeGlobal

Further Resources

Rob Pike - Concurrency is Not Parrarelism

https://www.youtube.com/watch?v=cN_DpYBzKso

Mailing List Threads:

http://lists.runrev.com/pipermail/use-livecode/2015-April/213203.html

Richard Gaskin's Idle Hour Stack

http://fourthworld.net/channels/lc/IdleHour.rev

Concurrent Non-Blocking Backend?

GoLang & NodeJS