

# Setting Your Daemon Free

**Alex Kodat**  
**Sirius Software Inc.**



BSD Daemon Copyright 1988 by Marshall Kirk McKusick.  
All Rights Reserved.



Sirius Software, Inc.

# Quick Refresher on Daemons

- Threads that perform work on behalf of a master threads
- Run on Sirius SDAEMON threads
  - Usually IODEV 15
  - These threads also do other terminal-less work such as act as web servers, socket servers, SirFact dump reading servers, etc..
- Old way of controlling them was via `$comm<xxx>` functions
- New cool way is via the Daemon class



# Using the Daemon Class

- When new instance of object created, an SDAEMON thread bound to the new object:

```
%daemon = new
```

- Master thread can then control the daemon thread via method calls:
  - Run: Run a command or set of commands on daemon thread as if someone was typing them into daemon
  - Open: Open one or more files
  - Miscellaneous utility methods



# Flow of Control with Daemons

1. Normally only the master or daemon can be running at a given time
2. When master sends request to daemon, the master stops and waits
3. Daemon processes the request and when it gets back to Model 204 command prompt and it has no more input, it stops and lets the master run
  - If daemon gets to non-command-level input prompt and no input available, it does a user restart
4. Master runs until it has something else to request from the daemon, at which point it goes back to 2.
5. Daemons can have their own daemons that work the same way.



# Sharing Data Between Master and Daemon

- Daemons and masters cannot have record locking conflicts between them
  - Since they can't run at the same time, one can't be updating a record while the other is looking at it
- Daemons and masters can optionally share transactions
  - So updates on master and daemon committed or backed out together
  - Possible, again, because threads can't run at the same time and so be updating the same record at the same time
  - Transactions really “owned” by master thread, so can continue even if daemon goes away



# Passing Data Between Masters and Daemons

- Master passes input stream to daemon and daemon responds with output stream
- Master can also pass objects to daemon and daemon can pass back result objects
  - In both cases, objects passed by DeepCopy
    - Good news: Daemon can't update master object and vice versa
    - Bad news: Daemon can't update master object and vice versa
    - Maybe “family” objects over the horizon
  - For large objects (such a big recordsets) this might have less than ideal performance
    - Though most copies pretty efficient



# Example of a Daemon Application

```
%recordset is object recordset in foobar
%program is object stringlist
%daemon is object daemon
...
%daemon = new
%daemon:open('FOOBAR')
%daemon:run('*LOWER')
%program = new
text to %program
    b
    %recordset is object recordset in foobar
    find records to %recordset where
end text
... build dynamic Find statement
text to %program
    end find
    %(daemon):returnObject(%recordset)
end
end text
%daemon:run(%program, , %recordset)
for each record in %recordset
    ... and so on
```



# But Suppose What's Running on the Daemon Can Take a Long Time?

- Call to an external system
- Big complicated Find that might do lots of I/O
- Reading/writing of a sequential dataset
- Can the master thread do something else while the daemon is doing its thing?
  - Like maybe get another daemon to do something else?
- Yes! As of Sirius Mods 7.0. The RunAsynchronously method makes this possible.



# The RunAsynchronously Method (cont..)

- Returns immediately after request sent to daemon
- At that point, both the master and daemon can run at the same time
  - On an MP/204 system, this means literally at the same time
  - On a non-MP/204 system, they can overlap I/Os
- Because of this, RunAsynchronously cannot be applied to a transactional daemon
- Also because of this, an asynchronously running daemon can encounter a record-locking conflict with the master
  - If there'd be a conflict at the time of the RunAsynchronous, the request is cancelled



# The RunAsynchronously Method (..cont)

- Has same input parameters as the Run method
- But has no output parameters (return object and info object)
- And does not produce an output Stringlist
  - So method is a subroutine, not a function
- Since input command/Stringlist passed by copy, as is any input object, it doesn't matter what happens to the originals as request runs asynchronously



# Asynchronously Running Daemons

- If daemon object discarded (maybe when master logs off), a running asynchronous daemon is BUMPed
- A master can have several daemons running asynchronously at once
- A master cannot apply any methods (other than Discard) against an asynchronously running daemon
  - Trying results in request cancellation
- If an asynchronously running daemon gets back to command-level input and it has no more input data, it waits for the master
  - But it is still considered to be “asynchronous” until master notices



# But How to Know When an Asynchronous Daemon is Done and to Retrieve its Output?

- The WaitAsynchronous method
  - The only valid method (other than Discard) when a daemon is running asynchronously
- If the daemon is finished, it returns immediately
- If the daemon is still running, it waits for the daemon to get to its stop point
  - Command-level with input stream used up
- Returns the output stream like the Run method
- Has output and info object output parameters like the Run method



# Run vs. RunAsynchronously and WaitAsynchronous

- RunAsynchronously and WaitAsynchronously behave like a Run method split in half
  - Inputs go with RunAsynchronously
  - Outputs go with WaitAsynchronous
- So **almost** any Run method can be (easily) split into two methods and so be run asynchronously
  - There can't be record locking conflicts between daemon and master
  - Daemon can't be transactional
    - This will be fixed, there's no reason the daemon can't run non-transactionally when asynchronous
    - In fact, there's no reason a daemon can't switch back and forth from being transactional and not



# Example of a Run Converted to Asynchronous Calls

```
%daemon    is object daemon  
%output    is object stringlist
```

```
...
```

```
%daemon = new
```

\* The following was the old synchronous call

```
* %output = %daemon:run('V')
```

\* But now we do it asynchronously

```
%daemon:runAsynchronously('V')
```

```
%output = %daemon:waitAsynchronous
```



# A Little More Interesting: Getting Two XML Streams in Parallel

```
%daemon1 is object daemon
%daemon2 is object daemon
%custid1 is string len 10
%custid2 is string len 10
%response is object xmlDoc
...
%daemon1 = new
%daemon2 = new
* Old code:
* %daemon:run('GETCUST ' with %custid1, %response)
* %(customer):dealWithResponse(%response)
* %daemon:run('GETCUST ' with %custid2, %response)
* %(customer):dealWithResponse(%response)

%daemon1:runAsynchronously('GETCUST ' with %custid1)
%daemon2:runAsynchronously('GETCUST ' with %custid2)

%daemon1:waitAsynchronous(%response)
%(customer):dealWithResponse(%response)

%daemon2:waitAsynchronous(%response)
%(customer):dealWithResponse(%response)
```



# Running Things In Parallel

- Fire off all the things that can be run in parallel with `RunAsynchronously`
- Perhaps do additional work on the master thread
- Then wait for each asynchronous request to finish with `WaitAsynchronous`
  - The elapsed time for the request will be more or less the time required for the longest request running in parallel
- Currently no way to wait on first request that completes
  - Probably coming soon to a Mods release near you
  - Won't help much if all requests must complete
  - Might be useful for controller thread that divvies out batch work



# Things That are Good to Run in Parallel

- Multiple network requests
- Big complicated finds
  - Though need some better tools for splitting finds on group file boundaries
  - Destructive copy option for recordsets would make bitmap copies more efficient and prevent doubling of CCATEMP usage for bitmaps
  - So maybe this is premature
- Time-consuming batch processes
  - Again, better tools for subsetting records would help
- Utility tasks
  - Backups, CREATEs, etc..



# Things That Don't Make Sense to Run in Parallel

- Fast-running code
  - Fewer than 50 milliseconds need not apply
  - Not worth the overhead/trouble of thread-switching
  - Unless hundreds of these in a request
- CPU-intensive code if not running MP/204
  - Parallel requests will simply take turns monopolizing the CPU



# What About Fire and Forget Daemons?

- Traditionally done with \$Commbg
- Now, can be done with the RunIndependently method
  - As of Sirius Mods 7.0



# The RunIndependently Method

## Sets a daemon free

- Daemon object null after a RunIndependently
  - No way to rendezvous with daemon after a RunIndependently
  - No way to retrieve outputs from independent daemon
- Daemon will keep running even if master thread logs off
- Can run in parallel with master or other independent daemons, of course
  - So, can encounter record locking conflicts with former master
- Once daemon running independently, no longer counted against master's MAXDAEM limit



# The RunIndependently Method

- Returns no outputs
- Takes same inputs as Run and RunAsynchronously
  - Input stream and input objects
- Not allowed on transactional daemons
- Has same record locking rules as RunAsynchronously
  - Daemon can't have record locking conflicts with master
    - Or request cancellation at time of method invocation



# Differences Between \$Commbg and RunIndependently

- \$Commbg allows one input stream and nothing else
- RunIndependently allows arbitrarily complex interactions between master and daemon before daemon becomes independent
  - Including passing objects
- \$Commbg will queue up the request if no daemon threads available
  - Is this important/useful to anyone?
- RunIndependently only works when you already have a daemon thread so no queuing issues
- \$Commbg had a non-independent mode which is unnecessary for RunIndependently



# Problem: Suppose a Thread Goes Feral and Starts Too Many Independent Requests?

- Can eat up all the SDAEMON threads in an Online
  - So interfere with stuff like Janus Web Server
- Already a problem with \$Commbg
- So, before Sirius Mods 7.0 there could never be more background requests running at a time than one-half or 10 fewer than number of SDAEMON threads, whichever is greater
  - So if number of SDAEMON threads  $> 20$ , then limit was number of SDAEMON threads minus 10
  - Otherwise, one-half the number of SDAEMON threads



# The MAXBG System Parameter

- Allows customization of the maximum background/independent requests running at one
- Defaults to the old limit (one-half, ten-fewer, etc.)
  - For backward compatibility
- Probably want set lower than default in big systems to make sure threads available for other things
- If \$Commbg about to exceed limit, it queues up
  - Might never run
- If RunIndependently about to exceed limit, it causes request cancellation



# Using Asynchronous/Independent Daemons for QA

- Can be used to test record-locking conflict parts of code
- Can be used to drive a web port
  - Via HttpRequest class
- Anyone care for a tn3270 client class?
- Of course, this requires some work
  - Driving interesting work automatically
  - Validating results
  - A presentation for next year



# Summary

- New daemon capabilities available in Sirius Mods 7.0
- Familiarize yourself with them
  - RTFM



# Questions? Comments?



Sirius Software, Inc.