Threads Library Overview

Scott McKay & Roger Jarrett

1. Introduction

The purpose of the Dylan threads library is to provide Dylan applications with multiple threads of control in a single address space, locking and synchronization facilities, and support for “fluid” bindings of settable “locations”. The threads must also cooperate properly with Dylan’s memory management and GC modules. This proposal is based in part on the Apple thread proposal.

2. Basic Functionality

Dylan threads are intended to provide multiple thread of control in a single address space. Each thread runs in its own dynamic environment, with its own control stack. Conditions cannot be signalled across threads.

It is highly desirable to implement Dylan threads by using OS-supplied threads or calling into a native threads library, so that hybrid applications using threads work properly.

We plan to support the following thread functionality:

- Creation and destruction of threads.
- Enabling and disabling threads.
- Thread blocking and wakeup functions.
- Preventing any other threads from running.
- Utility functions for examining threads.

We plan to support the following synchronization functionality:

- A simple, high level API for blocking locks.
- Light-weight “spin” locks.
- Lower-level non-blocking mutexes.
- Lower-level blocking semaphores.
- Monitors.
- Events and event flags.
- Message queues.

We plan to support the following functionality for fluid binding:

- A way to indicate that a settable slot or module variable is subject to fluid binding with the **fluid-bind** macro.
- A **fluid-bind** special form that binds a module variable or settable slot.
- Transparent access to a fluidly bound variable or slot.

3. Open Issues

This proposal uses “mutex” to mean a primitive, non-blocking lock, and “semaphore” to mean a primitive, blocking lock. Apple uses this terminology, too. Windows/NT uses “mutex” to mean a lock across processes (not just threads!) and “semaphore” to mean a lock across processes that doesn’t care what thread (or threads) actually owns the lock. What do we do?

Do we want to provide coroutining primitives? I bet most common thread libraries don’t, so it will be hard to simultaneously provide this and use the native thread library. Do we want to try to specify `call/cc`?