Multi-threading programming

Logtalk supports multi-threading programming on selected Prolog compilers. Logtalk makes use of the low-level Prolog predicates that interface with POSIX threads (or a suitable emulation), providing a high-level set of predicates that allows programmers to easily take advantage of modern multi-processor and multi-core computers without caring about the details of creating, synchronizing, or communicating with threads. Logtalk multi-threading programming integrates with object-oriented programming by allowing the optional creation of per object threads, enabling objects to send and receive asynchronous messages or to call local predicates concurrently.

Enabling multi-threading support

Multi-threading support may be disabled by default. It can be enabled on the Prolog configuration files of supported compilers by setting the read-only compiler flag `threads` to `on`.

Object threads

In order to automatically create and set up an object's thread the `threaded/0` object directive must be used:

```
:- threaded.
```

The thread is created when the object is loaded or created at runtime. The thread for the pseudo-object `user` is automatically created when Logtalk is loaded (provided that multi-threading programming is supported and enabled for the chosen Prolog compiler).

The message queue associated with an object's thread receives both goals to be proved asynchronously and the replies to the *threaded calls* made from within the object itself.

Multi-threading built-in predicates

Logtalk provides a small set of built-in predicates for running goals in new threads and for retrieving goal results. These predicates provide high-level support for multi-threading programming, covering the most common use cases.

Proving goals asynchronously using threads

A goal may be proved asynchronously using a new thread by calling the Logtalk built-in predicate `threaded_call/1`. The term representing the goal is copied, not shared with the thread. Therefore, any variable bindings resulting from the goal proof must be retrieved by calls to the built-in predicates `threaded_exit/1` and `threaded_exit/2`.

For example, assume that we want to find all the prime numbers in a given interval, `[N, M]`. We can split the interval in two parts and then span two threads to compute the primes numbers in each sub-interval:

```
prime_numbers(N, M, Primes) :-
    M > N,
    N1 is N + (M - N) // 2,
    N2 is N1 + 1,
    threaded_call(prime_numbers(N, N1, [], Acc)),
    threaded_call(prime_numbers(N2, M, Acc, Primes)),
    threaded_exit(prime_numbers(N, N1, [], Acc)),
    threaded_exit(prime_numbers(N2, M, Acc, Primes)).

prime_numbers(N, M, Acc, Primes) :-
    ...
```

Note that a call to the `threaded_call/1` predicate is always true (assuming a callable argument). The `threaded_exit/1` calls block execution until the results of the `threaded_call/1` calls are sent to the object's thread. In a computer with two or more processors (or with a processor with two or more cores) the code above can be expected to provide better computation times when compared with single-threaded code for sufficiently large intervals.

Retrieving asynchronous goal proof results

The results of proving a goal asynchronously in a new thread may be retrieved by calling the Logtalk built-in predicates `threaded_exit/1` and `threaded_exit/2` within the same object where the call to the `threaded_call/1-2` predicate was made.

The `threaded_exit/1-2` predicates allow us to retrieve alternative solutions through backtracking. For example, assuming a `lists` object implementing the usual `member/2` predicate, we could write:

```
| ?- threaded_call(lists::member(X, [1,2,3])).

X = _G189
yes

| ?- threaded_exit(lists::member(X, [1,2,3])).

X = 1 ;
X = 2 ;
X = 3 ;
no
```

Note that, in this case, the `threaded_call/1` and the `threaded_exit/1` calls are made within the pseudo-object *user*. The implicit thread running the `lists::member/2` goal suspends itself after providing a solution, waiting for the request of an alternative solution; the thread is automatically terminated when the runtime engine detects that further backtracking to the `threaded_exit/1` call is no longer possible.

Calls to the `threaded_exit/1` predicate, and, by default, calls to the `threaded_exit/2` predicate, block the caller until the object's thread receives the reply to the asynchronous call. The option `peek` may be used to check if a reply is already available without removing it from the thread queue. Using this option, the `threaded_exit/2` predicate call succeeds or fails immediately without blocking the caller. However, keep in mind that repeated use of this option is equivalent to polling a thread queue, which may severely hurt performance. The `threaded_exit/2` predicate supports an

additional option, `discard`, which allows us to discard all matching replies available on the object's thread waiting to be retrieved.

One-way asynchronous calls

Sometimes we want to call a goal in a new thread without caring about the results. This may be accomplished by using the built-in predicate `threaded_call/2` with the option `noreply`. For example, assume that we are developing a multi-agent application where an agent may send an "happy birthday" message to another agent. We could write:

```
threaded_call(agent::happy_birthday, [noreply]), ...
```

The call succeeds with no reply of the goal success, failure, or even exception ever being sent to the object making the call.

Atomic goals and asynchronous calls

Proving a goal asynchronously using a new thread may lead to problems when the goal implies side-effects such as input/output operations or modifications to an object's database. For example, if a new thread is started with a similar goal before the first one finished its job, we may end up with mixed output or a corrupted database. In these cases, we may call the built-in predicate `threaded_call/2` using the option `atomic` in order to ensure that the object which will span the thread for proving the goal argument will not accept any other asynchronous request until the first one terminates. For example, assume two predicates for writing, respectively, even and odd numbers in a given interval to the standard output. Given a large interval, a goal such as:

```
| ?- threaded_call(odd_mumbers(1, 100000)),
     threaded_call(even_numbers(1, 100000)).

1 3 2 4 6 8 5 7 10 ...
...
```

will most likely result in a mixed up output. Adding the option `atomic` to the calls solves the problem:

```
| ?- threaded_call(odd_mumbers(1, 100000), [atomic]),
     threaded_call(even_numbers(1, 100000), [atomic]).

1 3 5 7 9 11 ...
...
2 4 6 8 10 12 ...
...
```

In this case, the pseudo-object `user` will only execute one goal at a time. Note that, in a more realistic scenario, the two `threaded_call/1` calls would be made concurrently from different objects.

Competing goals

Usually, the argument of the `threaded_call/1` predicate is a single goal. The predicate also accepts as an argument a conjunction or a disjunction of goals. The semantics of a conjunction of goals is simply the conjunction of the `threaded_call/1` calls of each individual goal. I.e. a call such as:

```
threaded_call((G1, G2, ...)), ...
```

is equivalent to writing:

```
threaded_call(G1), threaded_call(G2), ...
```

A disjunction of goals is interpreted as running a set of *competing* goals, each one in its own thread. The first thread to terminate successfully leads to the termination of the other threads. This is useful when we have several methods to compute something, without knowing a priori which method will be faster. For example, assume that we have defined a predicate `try_method/2` that calls a chosen method, returning its result. We may then write:

```
method(Result) :-
    ...,
    threaded_call(
        (   try_method(method1, Result)
        ;   try_method(method2, Result)
        ;   ...
        )),
    threaded_exit(try_method(Method, Result)),
    ...
```

The `threaded_exit/1` call will return both the identifier of the fastest method and its result. Note that, when setting up competing goals such as in the example above, we must ensure that the argument of the `threaded_exit/1` call unifies with each individual goal in the disjunction used as argument on the `threaded_call/1` call. This ensures that the `threaded_exit/1` call will return the results of the first method to complete, no matter which one.

Atomic predicates

Predicates and grammar rule non-terminals may be declared as *atomic* by using the `atomic/1` object and category directive. Proving a query to an atomic predicate (or atomic non-terminal) is protected by a mutex, thus allowing for easy thread synchronization. For example:

```
:- atomic(db_update/1).   % ensure thread synchronization

db_update(Update) :-      % predicate with side-effects
    ...
```

The `atomic/1` directive must precede any local calls to the atomic predicate (or atomic non-terminal) in order to ensure proper compilation. In addition, as each Logtalk entity is independently compiled, this directive must be included in every object or category that contains a definition for the described predicate, even if the predicate declaration is inherited from another entity, in order to ensure proper compilation.

Whenever possible, atomic predicates should be coded as deterministic predicates in order to avoid deadlocks. Nevertheless, Logtalk supports both deterministic and non-deterministic atomic predicates (and atomic non-terminals). In those cases where the predicate (or grammar rule) is defined in the same object (or catehory) where the predicate is declared dynamic, Logtalk takes advantage off any existing `mode/2` directives in order to generate the most appropriated mutex handling code.

Last updated on: October 26, 2005