Thread-Safety in Parallel Computation

option lock

A new option, lock, can be added to any procedure. There can be only one thread running a procedure with option lock at a time. If a second thread tries to run a procedure with option lock, then the second thread will block until the first thread's procedure is done. Other threads are free to run any other procedure.

```
> p := proc(n)
      option lock;
      printf("Running p %d\n",n);
      Threads:-Sleep(1);
      printf("Done p %d\n",n);
  end proc:
> Threads:-Task:-Start(()->NULL,Task=[p,1],Task=[p,2]);
Running p 2
Done p 2
Running p 1
Done p 1
```

Notice that the first procedure that begins to run starts *and finishes* before the second procedure can begin. Without option lock, you would see two "Running" statements printed before two "Done" statements.

Option lock is now used on over 10,000 procedures in Maple's library of mathematical algorithms. This is a small fraction of the total number of procedures, the wider number already being thread-safe.

CodeTools:-ThreadSafetyCheck

The new ThreadSafetyCheck command in the CodeTools package can be used to help identify global state in a procedure or module. This is helpful in determining if your package is thread-safe or not.

Here is a module that implements a counter. The **Jump** procedure simply increments the counter by a given increment, although it does it by ones in a loop. This example is meant to be a simplification of a procedure that iterates through a longer process.

```
> Counter := module()
      local count := 0;
      export Reset := proc() count := 0; end proc;
      export Jump := proc(n)
          local i, j;
          for i from 1 to n do
              for j from 1 to 1000 do (* delay *) od;
              count := count + 1;
          end do;
          return count;
      end proc;
  end module:
```

Running this in serial we see that the internal counter is incremented by the given stride.

```
> Counter:-Jump(10);
```

$$10$$

```
> Counter:-Jump(90);
```

$$100$$

Let's **Jump** by 1000 in two different threads.

```
> Counter:-Reset();
```

$$0$$

```
> Threads:-Task:-Start((a,b)->[a,b], Task=[Counter:-Jump,1000],
```

```
  Task=[Counter:-Jump,1000]);
```

[1894, 1994]

The counter should be 1000 + 1000 = 2000, but the results above are strange numbers. The two threads interfered with each other. A quick test using the ThreadSafetyCheck command in the CodeTools package warns us of the presence of a global variable.

```
> CodeTools:-ThreadSafetyCheck( Counter );
```

Warning, proc Jump uses lexical count

1, 1

One way to fix this is to add option lock. This prevents two instances of **Jump** from running at the same time.

```
> CounterLock := module()
      local count := 0;
      export Jump := proc(n)
          option lock;
          local i;
          for i from 1 to n do
              count := count + 1;
          end do;
          return count;
      end proc;
  end module:
> Threads:-Task:-Start((a,b)->[a,b], Task=[CounterLock:-Jump,
  1000],Task=[CounterLock:-Jump,1000]);
```

[2000, 1000]

Locking in the above example is unsatisfying because it limits parallelism. Another way to fix this algorithm is to make the counter be a thread-local variable -- a different variable on each thread.

```
> CounterThreadLocal := module()
      local count::thread_local := 0;
```

```
    export Jump := proc(n)
        local i;
        for i from 1 to n do
            count := count + 1;
        end do;
        return count;
    end proc;
end module:
```

```
> Threads:-Task:-Start((a,b)->[a,b], Task=[CounterThreadLocal:-
  Jump,1000],Task=[CounterThreadLocal:-Jump,1000]);
```

$$[1000, 1000]$$

CodeTools:-ThreadSafetyCheck does analysis on the given procedure or module and warns about the following:

- global variables declared in a procedure's global statement
- global variables appearing on the left side of an assignment statement
- lexically scoped variables appearing on the left side of an assignment statement

For example:

```
> M := module()
    global G1, G2, G3;
    local L1, L2, L3, P1, P2;
    L1 := 1;
    G1 := 2;
    P1 := proc()
       L2 := 2*L3*G2;
       G3 := 1;
    end;
    P2 := proc()
       global G4;
       NULL;
    end;
end:

CodeTools:-ThreadSafetyCheck( M );
```

```
Warning, proc P1 uses lexical L2
Warning, proc P1 uses globals [G3]
Warning, proc P2 has declared globals [G4]
                                    2, 2
```

Notice the following variables are not flagged:

- G1 - globals declared in a module are not flagged
- G2 - globals declared in a module are not flagged even if they are used in a procedure (as in P1)
- L1 - locals and exports assigned at the top-level of a module get that assignment when the module is created and thus are not flagged
- L3 - locals that are not found on the lhs of an assignment are not flagged
- P1, P2 - same as L1

Analysis of each procedure is done based on just the statements inside the present procedure, not on it's children. Therefore, it is insufficient to simply apply option lock to any procedure flagged during CodeTools:-ThreadSafetyCheck.

For example, here is a situation that would not be thread-safe:

```
> M := module()
      local data, FetchData, ProcessData;
      export Compute;

      FetchData := proc( file )
          data := ImportMatrix(file);
      end;

      ProcessData := proc()
          data[..,1];
      end;

      Compute := proc( file )
          FetchData(file);
          ProcessData();
```

```
      end;
  end module:
  CodeTools:-ThreadSafetyCheck( M );
```

```
Warning, proc ModuleApply uses lexical i
Warning, proc FetchData uses lexical data
Warning, proc ProcessData uses lexical data
                                  3, 71
```

(Note: the ModuleApply here is actually ReadBinaryL5File:-ModuleApply -- something used by ImportMatrix.)

Adding option lock to **FetchData** and **ProcessData** is not sufficient. The parent function, **Compute**, also needs option lock (or some other mechanism) in order for this to be thread-safe. Otherwise, if threads A and B call Compute simultaneously, the call to FetchData in thread A may run in between the calls to FetchData and ProcessData in thread B: in that case, ProcessData in thread B will see the data from thread A.

How to Make your code Thread-Safe

Pattern 1: Read-only Constant

This is a situation where the code actually is thread-safe, but you want to resolve the warning.

```
> p := proc()
      global `debugger/max_width`;
      do_stuff;
  end:
  CodeTools:-ThreadSafetyCheck( p );
```

```
Warning, proc p has declared globals [`debugger/max_width`]
                                  1, 1
```

The appropriate fix is to add option threadsafe; to this procedure. This is a declaration that has no runtime significance, but does prevent ThreadSafetyCheck from warning about this procedure.

```
> p := proc()
```

```
      option threadsafe;
      global `debugger/max_width`;
      do_stuff;
  end:
  CodeTools:-ThreadSafetyCheck( p );
```

0, 1

Pattern 2: Initialization of Constants

A package module requires some initialization to set read-only constants.

```
> m1 := module()
      local a;
      local init := proc()
          a := 4;
      end proc;
      export user := proc()
          a*a;
      end proc;
      init();
  end module:
  CodeTools:-ThreadSafetyCheck(m1);
```

Warning, proc user uses lexical a
Warning, proc init uses lexical a

2, 2

There are several good options to solve this:

1. Move your init code to ModuleLoad.

```
> m1 := module()
      local a;
      local ModuleLoad := proc()
          a := 4;
      end proc;
      export user := proc()
```

```
        a*a;
    end proc;
    ModuleLoad();
end module:
CodeTools:-ThreadSafetyCheck(m1);
```

$0, 2$

2. Initialize your code in the module body.

```
> m1 := module()
    local a := 4;
    export user := proc()
        a*a;
    end proc;
end module:
CodeTools:-ThreadSafetyCheck(m1);
```

$0, 2$

3. Use the protect command to mark the variable as protected -- this declares it as read-only.

```
> m1 := module()
    local a;
    local init := proc()
        a := 4;
        protect('a');
    end proc;
    export user := proc()
        a*a;
    end proc;
    init();
end module:
CodeTools:-ThreadSafetyCheck(m1);
```

0, 2

4. Declare the variable as thread_local.

```
> m1 := module()
      local a::thread_local;
      local init := proc()
          a := 4;
      end proc;
      export user := proc()
          a*a;
      end proc;
      init();
  end module:
  CodeTools:-ThreadSafetyCheck(m1);
```

0, 2

5. Declare all procedures that use the variable as option threadsafe.

```
> m1 := module()
      local a;
      local init := proc()
          option threadsafe;
          a := 4;
      end proc;
      export user := proc()
          option threadsafe;
          a*a;
      end proc;
      init();
  end module:
  CodeTools:-ThreadSafetyCheck(m1);
```

0, 2

Pattern 3: The code is NOT thread-safe

In this situation you know the code is not thread-safe, but you want to get rid of the warning. Just mark it with option lock.

```
> p := proc()
          option lock;
          global a;
          a := 4;
       end:
  CodeTools:-ThreadSafetyCheck(p);
```

0, 1

Pattern 4: define_external

The define_external command allows you to link to compiled procedures in external DLLs or shared libraries. In order to use an external procedure in a package that may be run on different operating systems, it was often the case that a "trampoline" pattern was used. The first invocation would create the link and then reassign itself.

```
> pkg := module()
      export RandPerm := proc()
          unprotect(RandPerm);
          RandPerm := define_external("mstring_randperm",MAPLE,
  LIB=ExternalLibraryName("mstring"));
          protect(RandPerm);
          procname( args )
      end proc;
  end module:
  CodeTools:-ThreadSafetyCheck(pkg);
```

Warning, proc RandPerm uses lexical RandPerm

1, 1

In a multi-threaded environment, this pattern could lead to the external routine being linked in a second time. The define_external command now will accept the base name of the external library file, and apply standard transformations as ExternalLibraryName would. So, LIB="mstring" will map to "mstring.dll" on Windows, and to "libmstring.so" on Mac OS X and Linux. Saving such procedures will remember the original definitions so they can still be used in a platform-independent way.

```
> pkg := module()
      export RandPerm := define_external("mstring_randperm",MAPLE,
  LIB="mstring");
  end module:
  CodeTools:-ThreadSafetyCheck(pkg);
```

$$0, 2$$