COMPUTING SCIENCE

On the Comparison of Two Threads Packages

C.R. Snow

TECHNICAL REPORT SERIES

No. 420 March, 1993
On the Comparison of Two Threads Packages

C.R. Snow

Abstract

During an exercise in creating an implementation of the Modula-3 threads package on an Encore Multimax computer running the UMAX operating system, in particular making use of the "native" threads package EPT, some interesting subtle differences were observed between the semantics of apparently similar operations in the two packages. This paper attempts a detailed comparison between these two sets of facilities, and discusses the feasibility (or otherwise) of mapping from one set to the other and vice versa. We conclude that this is not possible without the assistance of some additional primitives.

© 1993 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details

SNOW, Charles Richard
On the Comparison of Two Threads Packages.
[By] C.R. Snow
(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no.420)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract

During an exercise in creating an implementation of the Modula-3 threads package on an Encore Multimax computer running the UMAX operating system, in particular making use of the "native" threads package EPT, some interesting subtle differences were observed between the semantics of apparently similar operations in the two packages. This paper attempts a detailed comparison between these two sets of facilities, and discusses the feasibility (or otherwise) of mapping from one set to the other and vice versa. We conclude that this is not possible without the assistance of some additional primitives.

About the author

Dr Snow has been a lecturer in the Department of Computing Science since 1970. He was a Visiting Scientist with Xerox System Development Division in Palo Alto, California, between 1978 and 1979.

Suggested keywords

CONCURRENT PROGRAMMING
MODULA-3
THREADS

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 001.6425
U.D.C. 519.687 681.322.06
On the Comparison of Two Threads Packages

C. R. Snow
Department of Computing Science
University of Newcastle upon Tyne, UK
e-mail: C.R.Snow@newcastle.ac.uk

Abstract: During an exercise in creating an implementation of the Modula-3 threads package on an Encore Multimax computer running the UMAX operating system, in particular making use of the "native" threads package EPT, some interesting subtle differences were observed between the semantics of apparently similar operations in the two packages. This paper attempts a detailed comparison between these two sets of facilities, and discusses the feasibility (or otherwise) of mapping from one set to the other and vice versa. We conclude that this is not possible without the assistance of some additional primitives.

Keywords: Threads, Concurrent Programming, Modula-3

1 Introduction.

Current thinking in the provision of facilities for concurrent programming has tended to move away from the Unix-style "heavyweight" process notion, in which a process consists of a separate address space and the code to run in that address space. Rather, the present view is that the code to be run as a parallel activity to the main processing of a program can be specified separately from the specification of the address space upon which it will act. The tendency therefore is to specify concurrent activity in terms of separate threads of control, leaving the definition of the method of interaction between such threads as an orthogonal concern.

The threads of control approach is not new, since it was discussed by Conway (1960) and by Dennis and Van Horn (1963) in the early 1960's, but perhaps the first high level programming language to incorporate such facilities was Mesa (Lampson and Redell 1980, Xerox 1985). Mesa called its separate threads "processes", but since the mechanism was essentially similar to a "concurrent procedure call", all of the usual facilities in relation to access to global variables are available; the only respect in which the process owns a private address space is that it creates its own stack frame in which to save its parameters and local variables. Additional stack frames belonging to the procedures called by the new thread are also private, of course.
A number of systems now offer a threads, or lightweight process facility, such as the SunOS system (Sun 1988) and the Apollo system (Apollo 1987). This paper addresses the comparison between the Brown Threads Package (Doeppner 1987), now available on the Encore Multimax system under the name EPT (Encore Parallel Threads, Encore 1988), and the Threads package supplied as one of the required interfaces of the Modula-3 language (Nelson 1991).

EPT is a library of routines designed to run under the Multimax operating system UMAX, and although in principle it may be linked with programs written in any language, it is clear that C was the language in the designer's mind.

Modula-3 is a new language, derived from a number of earlier languages, including Modula (Wirth 1977) and Modula-2 (Wirth 1980), Mesa, and Modula-2+ (Rovner 1985). One of the principal features of all of these languages is the provision of type-checkable interfaces which allow programs and systems to be constructed in a modular fashion. One incidental consequence of this development is that it is possible for a language system to include a number of useful modules providing implementations for library interfaces. Modula-3 has taken this concept one stage further by specifying that certain interfaces are required. One such is the Threads interface, to be described in more detail in section 2.

Birrell (Birrell 1989) provides an introduction to programming in threads, using the Modula-3 threads package for the examples given, and in an appendix to the report he also gives a brief comparison of the Modula-3 package with various other threads packages. A comparison with EPT is not included however, neither does he address the detailed issues of the comparison as we do in this report.

2 Modula-3 Threads

The Modula-3 Threads package might be regarded as containing a minimal set of facilities. It contains the basic calls required for creating new threads, for joining with concurrent threads, for acquiring and releasing locks and for synchronisation using condition variables. It also provides a limited mechanism for sending asynchronous signals to other threads. Most other threads packages contain these basic primitive operations, though perhaps with different
semantics, but frequently offer a large number of additional facilities. The basic data types involved in the use of the Threads package are the handle on a thread, which is the value returned by a call to Thread.Fork and is of type Thread.T,* Mutex which controls access to sections of code where mutual exclusion is required, and Condition, which controls the sequencing of access to shared data structures.

2.1 Thread Creation

In order to create a new thread in Modula-3, a call is made to the procedure Fork, which takes as its (single) parameter an object of type Closure. By making use of Modula-3’s object-oriented programming facilities, it is possible to pass to Fork all of the information it requires to start a new thread. Objects from the class Closure contain a method called apply, a procedure which constitutes the code which the thread will run, and which has as its only parameter the object to which it belongs. Since the parameter of Fork may be an object which is a sub-type of Closure, and hence inherits this method, parameters required by apply, and indeed results to be returned, may be passed to the code of the newly executing thread as part of this object. Thus the requisite Modula-3 code would appear as:

\[
\begin{align*}
\text{VAR } & \text{obj : } \text{Thread.Closure} := \text{NEW (Thread.Closure, apply := DoSomething);} \\
& \quad \text{newThread : } \text{Thread.T}; \\
\text{PROCEDURE } & \text{DoSomething (o : Thread.Closure) : REFANY =} \\
& \quad \text{BEGIN} \\
& \quad \quad \text{END DoSomething;} \\
& \quad \quad \text{...} \\
& \quad \text{newThread := Thread.Fork (obj);} \\
& \quad \text{...}
\end{align*}
\]

Notice that the procedure DoSomething takes a single parameter which has the type Closure, and returns a value of type REFANY. The subtyping rules of Modula-3 thus enable the

* There is a convention in common use by the Modula-3 community in which if an interface Int exports a main type (frequently an object type) that is used by most if not all of the operations of the interface, then the type is named T, and is therefore referred to by clients of the interface as Int.T.
procedure to return a value of any object type, giving a great deal of flexibility in what may be returned from the thread.

2.2 Thread Re-synchronisation

Re-synchronisation of threads is achieved by use of a procedure Join. The Modula-3 Join procedure takes a thread identity as a parameter. It is therefore necessary for the parent thread to specify which of its child threads it is attempting to join with. This may be illustrated by the following modified form of the previous example:

```plaintext
VAR obj : Thread.Closure := NEW (Thread.Closure, apply := DoSomething);
  newThread : Thread.T;
  result : REFANY;

PROCEDURE DoSomething (o : MyClosure) : REFANY =
BEGIN
  ...
  END DoSomething;
  ...
  newThread := Thread.Fork (obj);
  ...
  (* Parent thread running in parallel with the newly created thread. *)
  ...
  result := Thread.Join (newThread);
  ...
```

Thus results may be returned as a result of the Join call. They may also be returned by the inclusion of a suitable field in the Closure object which was originally passed to Fork. Notice that Forks and Joins are paired (statically) because the returned result of a Fork call is used as the parameter of a corresponding Join call.

We observe that there are no facilities in the Modula-3 threads interface for creating what might be called a detached thread, neither is there any way of detaching a thread once it has been created. That is, once a child thread has been created, it will only completely disappear and release the resources it holds when either it Joins with its parent, or the parent terminates. It could be argued that this would be a useful addition to the interface. However, it is the case that Modula-3 has a garbage collector as part of its run-time system. It is therefore sufficient for a
parent thread to fork a new thread and then 'lose' the handle. When the thread terminates, the
garbage collector will then automatically reclaim the storage associated with the thread.

2.3 Mutual Exclusion

The Modula-3 language itself provides a minimal amount of support for concurrency by
providing a LOCK statement. This allows a structured form of statement which is controlled by
a Mutex. This statement provides mutual exclusion of a block of statements from other
statements requiring access to the same Mutex. This could have been achieved, admittedly in a
less structured and hence less secure way, by a pair of calls to the procedures Acquire and
Release. The LOCK statement is the only language feature providing support for concurrency,
all other concurrency facilities being provided through a set of additional routines - albeit via a
required interface.

As has just been mentioned, the Threads interface provides two procedures Acquire and
Release for the purpose of providing mutual exclusion, which take a Mutex as their parameter.
Both the LOCK statement and the use of Acquire and Release are illustrated in the next
examples (in which it is assumed that all the necessary Forking has already been done).

```
VAR mu := Thread.NewMutex ();

LOCK mu DO

END

VAR mu := Thread.NewMutex ();

Thread.Acquire (mu);

Thread.Release (mu);

```
2.4 Thread Scheduling

The only thread scheduling mechanism (by which we mean that threads may wish to suspend themselves following the discovery that they are unable to continue) provided by the Modula-3 threads package is through primitives which implement a variant of the monitor concept (Hoare 1974). The mutual exclusion properties of the monitor are handled, as described in the previous section, by the LOCK statement or by explicit use of Acquire and Release, and the thread scheduling is provided for through the use of conditions. There are however some crucial differences between the Modula-3 approach and what might be called the "pure" Hoare monitor. Firstly, a Modula-3 condition is not tied to a particular Mutex (or monitor) and can be created by what might be called a "free" declaration, as in:

\[
\text{VAR cond := Thread.NewCondition ()}
\]

which declares and initialises a new variable (called cond) of type Thread.Condition. (In practice, of course, the Mutex and any conditions associated with it will all appear within the same module, and encapsulation and association will be achieved through a more general program structuring mechanism). An association is made between a Mutex and a condition at the point where a thread discovers that it must be held up because the state of some data structure prevents the thread from continuing. This is achieved by a call to the procedure Wait.

\[
\text{Thread.Wait (mu, cond)};
\]

The parameters are a Mutex mu, and a condition cond, and it is regarded as an error for any thread to make this call if it does not already hold mu, i.e. by having successfully called Acquire (mu), or, equivalently, successfully entered a LOCK statement which has mu as its lock variable. The Wait operation will release the Mutex mu, and suspend itself on the queue associated with the condition cond. This must clearly be done as a single indivisible operation.
Following the execution of a Wait operation, the calling thread is suspended, and can only be re-activated by the action of another thread. The most common method by which this is achieved is for a thread to call the operation Thread.Signal or Thread.Broadcast. These operations take a condition variable as their only parameter, and cause the re-awakening of threads waiting on the condition. The difference between Signal and Broadcast is that Signal will awaken only one waiting thread, whereas Broadcast awakens each thread which is waiting on the condition.

In the original description of the monitor, Hoare explicitly states that if a thread (Hoare calls threads processes) is signalled, then it must resume immediately. This is because a Wait is usually performed within a conditional statement, and if the thread is required to wait, the test is not performed again following the re-awakening of the thread. That is, it is assumed that following the Signal, the circumstances which forced the thread to wait no longer obtain, and that the thread may continue without further testing. Unless the signalled thread resumes immediately, this cannot be guaranteed.

The Modula-3 approach differs substantially from the pure monitor style in that it no longer regards a Signal as an imperative to the underlying scheduling mechanism, but merely as a hint (Lampson and Redell 1980). In Modula-3, therefore, on being awoken following a Wait, the thread must re-check the state of the computation to determine whether or not it can proceed. Thus, while it is sufficient to write

\[
\text{if ( predicate ) then condition.wait}
\]

within a monitor (where predicate represents the circumstances under which the computation may continue), the Modula-3 threads package requires the programmer to write

\[
\text{WHILE NOT predicate DO Thread.Wait (mu, cond) END;}
\]

A further consequence of the Modula-3 approach is that it is possible to include a Broadcast facility. Clearly it is necessary for the first action of any thread which has been newly
awakened from a *Wait* must be to reclaim the monitor lock. In fact, as described by Hoare, there is a sense in which the monitor lock is handed over from the signaller to the signalled thread. In these circumstances, an attempt to awaken more than one thread with a single operation is clearly dangerous.

In Modula-3, in contrast to the monitor, the newly-awakened thread has to complete for the Mutex with all other threads which may be attempting to acquire it.

### 2.5 Asynchronous Inter-Thread Communication

Inter-thread communication, as described in the previous sections, requires the co-operation of both of the communicating parties. Either one thread calls for a *Join* and another terminates, or two threads participate in a *Signal/Wait* style exchange.

There are however, instances where asynchronous communication is desirable, and the Modula-3 threads package provides for a limited form of this. Modula-3 offers an exception handling mechanism, where exceptions may be raised in a procedure, and may be handled by any level of the calling code that offers a handler for that exception. In particular, procedure declarations may specify not only the form of their parameters lists and their returned results, but also which exceptions may be raised as a result of a call to that procedure. Generally speaking, this mechanism is used to allow exceptions arising in procedure calls to be handled appropriately within a single sequential thread. However, the Threads interface contains the declaration of an exception *Alerted*, and associated with each thread will be a (conceptual) boolean variable (which we shall call *alerted*), initially false, indicating whether or not another thread has attempted to alert this thread.

Threads may therefore alert other threads by calling the procedure

```
Thread.Alert (th)
```

which causes the boolean *alerted* to be set to true within the thread *th*. However, the mechanism is limited in the sense that *Alerted* will only be raised when the target thread recognises that
alerted has been set, and this will only happen as the result of an AlertWait or an AlertJoin call, or by the thread explicitly testing to see whether it has been alerted by calling a procedure TestAlert. AlertWait and AlertJoin are versions of Wait and Join, which will terminate (i.e. return to their callers through the exception mechanism) if the suspended thread is the target of an Alert call by another thread, either prior to the call of AlertWait or AlertJoin, or while the thread is awaiting the completion of these calls.

3 Encore Parallel Threads

The Encore Parallel Threads package as offered on the Encore Multimax computer operating under UMAX 4.3 is derived from a package developed by Doeppner at Brown University (Doeppner 1987). The package provides not only a rich set of facilities for the writer of multi-threaded programs, but also provides access to lower level functions, thereby enabling the programmer to exercise greater control over the operation of the program. These lower level facilities include the ability to reschedule threads, and to create and manipulate queues. In this section, however, we shall confine ourselves to a discussion of the facilities for multi-threaded programming.

3.1 Thread Creation

When a Modula-3 program begins to run, it is assumed that an environment to support multi-threading already exists. In the EPT package, however, the programmer is given the responsibility of starting up the Threads environment by calling a function named THREADgo. This provides the opportunity for the user to specify certain parameters of the program, including the total shared memory to be used and the priority at which the main thread will run. Having created the multi-threaded environment, the program is free to create additional threads by means of the function THREADcreate, each call of which creates a separate, parallel, thread to run a function supplied to THREADcreate as an argument, as in:
\( th = \text{THREADcreate}(\text{func}, \text{args}, \text{argsize}, \text{detached}, \text{stacksize}, \text{priority}) \)

As with \textit{Fork} in Modula-3, so this function returns a handle for the newly created thread to the caller.

\textit{THREADcreate} offers the user the opportunity for parameters to be passed to their respective functions using the parameters \textit{args} and \textit{argsize}. This can either be done by passing a pointer to the parameter list into the function, or by copying the arguments to the stack of the newly created thread. In this way, the parameters will either be shared between the threads, or will be private to the new thread. This also provides the facility whereby different calls of \textit{THREADcreate} with different functions as their parameters can be given different, and differently structured, argument lists.

By contrast with the Modula-3 package, the \textit{THREADcreate} function allows the user to specify that the new thread will be \textit{ATTACHED} or \textit{DETACHED} using the parameter \textit{detached}. A detached thread does not resynchronise with its creating thread, or parent, at the end of its execution. If a detached thread terminates, it just disappears. An attached thread, on the other hand, is expected to resynchronise with its parent, and possibly to return results.

\section*{3.2 Thread Resynchronisation}

When an attached thread reaches the end of its execution, it may well have some values to return to its parent, and it cannot therefore die completely. The parent is required to resynchronise with its child threads using the \textit{THREADjoin} function:

\[ th = \text{THREADjoin}() \]
As we can see, this function takes no parameters, and simply joins the parent thread with the
next child thread to terminate, which may imply having to wait. The thread joined has to be a
child of the parent, and if no such thread exists, the function returns immediately with the value
NIL. This approach means that a) any terminating child thread will release the parent thread,
and not a specified thread as in the Modula-3 case, and b) any value that the terminating child
wishes to return must be obtained by a separate mechanism, which is achieved through the use
of a function THREADreturnvalue, which takes a thread identity as its parameter, and returns
the (integer) value which was the parameter of the thread's exit call. This may then be followed
by a call to the function THREADfree to release the storage associated with the thread.

3.3 Inter-Thread Communication

EPT offers a number of ways in which threads can control their interaction, in order, for
example, to prevent simultaneous access by more than one thread to sensitive shared data
structures. However, we begin by describing the features which most closely resemble those of
Modula-3, as already described.

3.3.1 Monitors

The EPT package provides a monitor-like facility which is more closely related to the monitor
of Hoare than is the corresponding Modula-3 mechanism. Once again, however, the
programmer is expected to provide explicit locking on entry to critical functions, and the
corresponding unlocking on exit. Thus, two functions

\[
\text{THREADmonitorentry (mon, manager)} \quad \text{and} \quad \text{THREADmonitorexit (mon)}
\]

are provided for these purposes. The parameter mon in each case is the handle of the monitor,
and the parameter manager in THREADmonitorentry allows the caller the option of managing
the monitor control block space, which can be (and in this work, is) done by default.
Monitors à la Hoare also contain condition variables which are used to allow synchronising communication between concurrent processes calling operations within the monitor. These are usually implemented in the form of queues, and the operations wait and signal can be performed on them, the first allowing the calling process to place itself on a condition queue, and the second for removing a process from the condition queue. Within a monitor, conditions are regarded as local variables, and therefore cannot even be referred to outside the monitor, and hence there is no possibility of attempts be made to reference conditions in inappropriate circumstances.

The EPT package cannot rely on such language features to enforce the correct use of monitors and conditions, and hence a number of checks must be made at the time of calling the functions to ensure, as far as is possible, that monitors and conditions are being used correctly. Since the condition variables are associated with the monitor, the function

\[ mon = \text{TREDmonitorinit} (conds, \text{resetfunc}) \]

which is responsible for creating the necessary monitor data structure, requires that the programmer should specify the number of conditions which the monitor will use as the parameter \( \text{conds} \). (The second parameter, \( \text{resetfunc} \), provides for correct recovery should a thread be aborted while within the monitor. This feature is optional, and has not been used in this work). The value returned is again a pointer which is only used as an identifier for the monitor.

Conditions are not specified by name, by merely by an integer in the range \( 0..n-1 \), where \( n \) is the declared number of conditions. In all of the functions operating upon conditions, one of the parameters has to be the monitor to which the condition belongs. Clearly, if a condition is specified in relation to a monitor, and the condition number is greater than \( n-1 \), then an error will be indicated.

The actual functions provided for handling the monitor synchronisation assume Hoare-style semantics for conditions. That is, whenever a thread discovers that it is unable to proceed while within a monitor, it blocks itself by using the function
\[ res = \text{THREADmonitorwait} (mon, cond) \]

which takes two parameters, a monitor identifier \( mon \) and a condition number \( cond \). Clearly it is necessary for the calling thread to release the monitor lock as part of this operation. The value returned is simply a mechanism whereby the caller is informed about erroneous use of this function, such as in the case where \( cond \) is outside the declared range of condition numbers for the monitor \( mon \).

When a thread is awoken following a wait, it is assumed that the circumstances which cause the thread to wait have been changed, so that the thread may continue without further testing. The implication of this is that if another thread signals to a waiting thread, then the newly awoken thread must be allowed to re-enter the monitor immediately, and hence the signalling thread must leave the monitor. If however the signalling thread has not completed its processing within the monitor, it must give still priority to the signalled thread, and must therefore wait to re-enter the monitor, which it does by waiting on another condition.

Thus, in order for a thread to signal to a waiting thread, two functions are provided,

\[ res = \text{THREADmonitorsignalandexit} (mon, cond) \]

for use in the case where signalling is followed immediately by exit from the monitor, and

\[ res = \text{THREADmonitorsignalandwait} (mon, cond1, cond2) \]

The parameters are the monitor identifier \( mon \) and the condition number \( cond \) (or \( cond1 \)) of the condition being signalled, with the second function requiring an additional condition number \( cond2 \) upon which the signaller will wait. In each case, the result returned is merely an indication of whether or not the call has been correctly made.
3.3.2 Semaphores

In addition to the functions described in section 3.3.1 to implement a Hoare-style monitor mechanism, EPT also offers semaphores as an alternative mechanism for thread synchronisation (Dijkstra 1967).

As originally proposed, the semaphore is a simple data structure consisting essentially of a counter and a queue. Within EPT, there are two operations which can be applied to a semaphore, \textit{THREADpsem}, which decrements the counter if its value is positive, otherwise the calling thread is placed on the queue, and \textit{THREADvsem}, which inspects the queue and, if not empty, marks the thread at the head of the queue as being ready to run. If the queue is empty, then the counter is incremented by one. These two functions correspond exactly, of course, to Dijkstra's \textit{P} and \textit{V} operations. A further function called \textit{THREADseminit} is provided for the purpose of creating a new semaphore and initialising the value of its counter. Thus

\[
sem = \textit{THREADseminit}(ct);
\]

will create a new semaphore with initial value of \textit{ct} for its counter, and will return a value of type \textit{SEMAPHORE}, which is, like the type \textit{THREAD}, a pointer to a control block, but which is only used as a handle by which the semaphore can be identified.

3.4 Asynchronous Inter-thread Communication

EPT makes use of the full Unix signal handling mechanism to allow threads to communicate asynchronously with each other. Threads may respond to Unix-style signals in one of three ways:

\begin{itemize}
\item[i.] a thread may establish an exception handler in much the same way as Unix processes do. The arrival of a signal will cause the exception handler to be called, and when it has finished executing, control is returned to the point at which the handler was established.
\end{itemize}
ii. the thread receiving the signal is frozen, and a child thread is created which runs on the same stack as the frozen parent. The parent is unfrozen as soon as the child terminates. This behaviour is not unlike the call of a procedure, except that the parent and child threads have different handles, and may be manipulated (by appropriate functions) independently of one another.

iii. a completely new detached thread running alongside the original is created.

For each pre-defined Unix signal, a thread may specify which of these three methods of handling signals it wishes to employ. However, only two of these methods may be used by threads wishing to cause asynchronous communication to occur with another thread. One of these is to call the function `THREADraiseexception`, which, as expected, causes the exception handler in the target thread to be invoked. Clearly, if no exception handler has been established, the raising of an exception is catastrophic.

Alternatively, a thread may asked to freeze another thread, using the function `THREADfreeze`. This will cause the target thread to be frozen, and a new (child) thread to be created which runs on the stack of the frozen parent. This is similar to response ii. above, but there is no requirement for the target thread to have made any provision for handling the signal. The action to be taken is specified in the call of `THREADfreeze`.

4 Comparison and Implementation Issues

4.1 Fork

Broadly speaking, the Fork call in the two packages behaves in the same way, in that in each case a procedure (function) is passed to the Fork procedure, and this causes that procedure to run as a separate thread. There are also mechanisms which allow parameters to be passed and results to be returned, although these are different in the two cases. In the EPT package, the parameters may be passed in using the parameter list of `THREADcreate`, and the flexibility of C means that parameter lists of any size and type may be given. Only a single integer variable may be returned from a thread, but with sufficient care, it is possible to coerce this to and from a
pointer type, which offers the opportunity for more flexible results to be passed. Presumably, however, most communication between a child thread and its parent would take place by some other means, such as placing messages in, and retrieving them from, a shared memory region.

In Modula-3, the type mechanism prevents parameters from being passed in a similar way, but the subtyping mechanisms prove very useful in this situation. As has been described, the parameter to a call of Thread.Fork is a Closure, which is an object with a single method, the procedure apply. When apply is called, it requires a single parameter of type Closure. But it is also permissible for Thread.Fork to be passed an object which is a subtype of Closure, providing the opportunity for additional values (i.e. additional parameters for apply) to be passed as part of the argument object. Similarly, results may be returned using the same mechanism. However, we also observe that the method apply returns a value of type REFANY. Again the subtyping provides us with a mechanism whereby values of any type may be returned, provided they are subtypes of REFANY; i.e. any pointer type.

Modula-3 does not offer the concept of a priority for running new threads, and neither is a distinction made between attached and detached threads. The assumption is made that all Modula-3 threads may eventually join with their parents, and must all be started as attached threads, but as mentioned earlier, threads may effectively become detached by virtue of the parent discarding the handle of a child thread.

4.2 Locking

The operation of acquiring a lock in Modula-3 threads has exactly the same meaning as entering a monitor in EPT threads. Similarly, exiting from an EPT monitor is equivalent to releasing a Modula-3 lock. The only difference is language-related, in that Modula-3 provides the structured LOCK statement.
4.3 Wait/Signal

The differences between Modula-3 threads and EPT with regard to *Wait* and *Signal* revolve around the issue of whether the Hoare-style monitor semantics are adopted, or whether to regard *Signal* as being no more than a hint.

Full Hoare-style monitor semantics require that when the signal operation is invoked, control is passed immediately from the signaller to the signalled process, implying that the monitor lock is relinquish by the signaller in favour of the signalled. The signaller then has to wait until the monitor lock is released, at which time it may or may not have priority over new processes waiting to enter the monitor. Frequently, the process invoking the signal operation would wish to leave the monitor anyway, so that an optimisation which allows the signalling process to avoid waiting to re-enter the monitor and then immediately leave it again would be useful.

In EPT, there is an operation which allows the programmer to specify such an optimisation called `THREADmonitorsignalandexit`. There is also an additional function to provide for the situation where the signalling process does not wish to leave the monitor. This operation is called `THREADmonitorsignalandwait`, and it allows the signalling thread to suspend itself "inside" the monitor by waiting on a different condition variable. Again this differs slightly from Hoare's original proposal, in that the signaller must wait for another process to signal this second condition explicitly, rather than wait simply for the monitor lock to become free.

The Modula-3 Threads package on the other hand, takes the view that it does not wish to impose such scheduling constraints on the wait/signal mechanism, and by so doing is unable to guarantee that when a signalled thread is given the opportunity to run following a *Wait*, that the condition which caused it to wait (and which presumably was removed by the signalling thread) has not been re-imposed between the signal being sent, and the signalled thread resuming.
4.4 Join

The noticeable difference between the join mechanism in the two packages is merely to do with the parameter which the join operations require. This is simply that the \texttt{EPT THREADjoin} function takes no parameter, and returns the identity of the joined thread as its result. This means that the (parent) thread calling the join operation will continue as soon as any of its children terminates. Also, this form of join operation requires that a separate function must be used to retrieve any results that the terminating thread wishes to return to its parent.

By contrast, the Modula-3 form of join operation takes a parameter which is the handle of a thread, and therefore the caller will only continue following the termination of a specific thread. This does allow for a thread to return results within the same call.

4.4 Asynchronous Communication

It may be seen from the earlier discussion that the EPT form of asynchronous inter-thread communication is much more extensive and flexible than the corresponding Modula-3 facilities. Not only is there more flexibility in the way in which a thread can respond to asynchronous communication from another thread (or elsewhere), but such events may occur at any time. Thus, an asynchronous request from a thread for action in another thread will be acted on immediately.

In the Modula-3 case, there is only one mechanism for generating asynchronous events in another thread, namely to flag the target thread as alerted, and the target thread will only respond at certain points in its own execution. Under these circumstances, it is perhaps a little misleading to describe this form of inter-thread communication as truly 'asynchronous'.

However, the handling of alerts is dealt with entirely within the language, by making use of the exception handling facilities provided, whereas the EPT mechanisms rely on the fact that the package is designed to run in conjunction with the UMAX Operating System, and unashamedly uses the Unix-style signal generation and handling for its implementation.
5 Mapping Issues

In attempting to provide a comparison between two sets of facilities, it is sometimes instructive to consider what would be involved in mapping between the two approaches, i.e. to provide an implementation of each in terms of the primitives offered by the other. In what follows, we shall adopt the convention that references to EPT functions will be written entirely in lower case, but without the prefix THREAD, as in wait, and references to Modula-3 operations will always begin with an upper case letter, as in Wait (although there may be other upper case letters in the name, as for instance in AlertWait).

The creation of threads in each of the two packages is very similar, except that the EPT package relies on the fact that strict type checking is not enforced in the C language in order to provide flexibility in creating threads with different numbers and types of parameters. In Modula-3, this aspect is handled by the subtyping (type inheritance) mechanism. This however is purely a language issue, only peripherally related to the semantics of the threads primitives.

More interesting from a concurrency point of view are the differences in the handling of thread joining, of the wait/signal styles of thread intercommunication and of the asynchronous communication facilities.

5.1 Locking

There is a direct equivalence between the two approaches as far as locking is concerned. This comes about because the EPT notion of a monitor is, as far as the straightforward access to the monitor is concerned, the same as the Hoare monitor, which simply entails the acquisition of the appropriate monitor lock. Thus, the EPT monitorentry and monitorexit procedures are exactly equivalent to the Modula-3 Acquire and Release operations, which, as we have already seen, may also be expressed in terms of the Modula-3 language's LOCK construct.
5.2 Wait/Signal

A more interesting comparison concerns the treatment of Modula-3's \textit{Wait} and \textit{Signal} (or their equivalents). As we have already observed, the EPT monitor implements an almost "pure" monitor as defined by Hoare, but this carries with it implications on the way in which the various threads are scheduled. Pure monitor semantics, and EPT, allow for the monitor lock essentially to be passed directly from one thread to another, whereas the Modula-3 threads treats signalling as a hint. This enables a certain degree of flexibility in the way in which the processes are scheduled following the signal, and in particular, it allows for the introduction of a \textit{Broadcast}, or multiple \textit{Signal}, and for the possibility of a \textit{Wait} terminating for reasons other than an explicit \textit{Signal}, such as an exception.

We first consider the problem of mapping the Modula-3 \textit{Wait} and \textit{Signal} operations into the corresponding EPT primitives. In this, we attempt to provide an implementation of the Modula-3 operations, using only the facilities provided by EPT.

The important feature of this approach is to consider the way in which the possession of the monitor lock changes between the threads as the various operations are invoked. In both packages, the execution of a wait operation causes the calling thread to release the monitor lock. The difference occurs when the signal operation is considered. In Modula-3, a thread calling \textit{Signal} will retain the monitor lock, whereas in EPT it must relinquish the lock to the signalled thread. We must therefore attempt to devise a mechanism whereby the lock can be passed back to the signalling thread after the waiting thread has been released, but before it is allowed to continue.

We propose that a Modula-3 condition be represented by four EPT conditions and a counter. In addition, whenever a Modula-3 condition is created (Modula-3 provides a \textit{New} operation to create new conditions), a thread is started which manages the handling of the mutual exclusion lock, and ensures that the lock is returned to the signaller whenever a signal operation is invoked.
Thus, suppose we wish to create a new Modula-3 condition variable \( c \). This will cause the creation of four EPT condition variables, \( c_1, c_2, c_3 \) and \( c_4 \), and a counter \( ct \). A new thread will also be created, whose action may be described by the following:

\[
\begin{align*}
\text{monitorentry (m);} \\
ct := 0; \text{wait} (m, c1); \\
\text{loop} \\
\quad \text{wait} (m, c2); \\
\quad \text{signalandexit} (m, c3); \\
\quad \text{monitorentry} (m); \\
\quad \text{signalandwait} (m, c4, c1); \\
\quad ct := ct - 1;
\end{align*}
\]

(The variable \( m \) is intended to signify an EPT monitor handle, which corresponds directly with a Modula-3 Mutex). With such a structure, the action of the Modula-3 \textit{Wait} operation consists of incrementing the counter \( ct \) by one, and performing the EPT operation $\text{signalandwait} (m, c1, c4)$.

The operation \textit{Signal} then becomes simply:

\[ \text{signalandwait} (m, c2, c3); \]

The "condition managing thread" begins by initialising \( ct \) to zero, and then waits on condition \( c1 \). The thread then continues only when a \textit{Wait} operation on \( c \) is performed. When this does occur, the counter is incremented by one (the significance of which will be explained shortly), the managing thread is restarted, and the caller is suspended on the EPT condition \( c4 \). The managing thread is now in possession of the monitor lock, the lock having been (briefly) in the possession of the now-waiting thread. On entering its (infinite) loop, the managing thread immediately waits on \( c2 \), releasing the lock as it does so.

The requirement is that when another thread signals on \( c \), it does so while retaining the lock. Since this would not normally occur in the EPT scheme, it is necessary for the monitor lock to be yielded up to a thread which will pass it back to the signaler as soon as the signalled thread has been made ready to run. This effect is achieved by replacing the call to signal on \( c \) by a $\text{signalandwait} (m, c2, c3)$ call. This allows the manager thread to resume, but which immediately passes the monitor lock to the signaler and exits (i.e. continues to execute without the lock).
At some later stage, when the monitor lock becomes free again, the manager thread will re-acquire it, signal on \( c4 \) (on which the waiting thread is now waiting) and reduce the count of the number of threads waiting on \( c \) by one. This counter is not important when considering only the \textit{Signal} operation, but when we turn to consideration of the \textit{Broadcast} operation, we note that there is no way in the EPT package of discovering whether or not a thread is waiting on a particular condition. All we know is that if a signal operation is performed on a condition, but no threads are waiting on that condition, then the call is a null operation. It is therefore necessary for the number of threads waiting on a condition to be recorded explicitly, hence the inclusion of the counter, and the \textit{Broadcast} operation is simulated by:

\[
\text{while } ct > 0 \text{ do} \\
\quad \text{signalandwait} (m, c2, c3); \\
\quad ct := ct - 1 \\
\text{end;}
\]

We now consider the problem of providing the mapping in the reverse direction, i.e. providing a simulation of the EPT functions in terms of the Modula-3 set. It appears however that this is not possible, as may be demonstrated by the following argument:

The definition of \textit{Wait} requires that the monitor lock be held at the time of calling the \textit{Wait} operation. Furthermore, the action of the \textit{Wait} operation causes the monitor lock to be released. Since the operation \textit{signalandexit} must re activates a waiting thread (if there is one) immediately, it is required that the signalling thread should in some sense be permitted to pass ownership of the lock to this waiting thread, rather than releasing it. But the thread calling the \textit{Wait} cannot continue until the the lock has been reacquired, and hence it must be released by the signalling thread, with the possibility that another thread may intervene between the signal and the wait.

\section*{5.3 Thread Joining}

The Modula-3 \textit{Join} mechanism has the virtue that the strict type checking of returned values from a terminating thread can be enforced. Since the EPT \textit{join} call does not specify which child thread is to be joined, separate calls are required for joining and for retrieving the returned value, whereas Modula-3 performs these two operations through a single call. There is some difficulty with the Modula-3 style of joining in the case of the parent wishing to join with some,
but not all, of its children, where the number of children, but not their identities, is of concern to the parent. In the Modula-3 style, there is no way of directly expressing the equivalent of

\[
\text{for } (i=0; i < n; i++) \ \text{THREADcreate (.....);} \\
\text{...for } (i=0; i < m; i++) \ \text{THREADjoin ();}
\]

where \(m < n\). That is, the parent thread creates \(n\) children, but only wishes to join with \(m\) \((< n)\) of them. If required, however, this effect can be achieved in Modula-3 by forking an \textit{envelope} thread for each child, which is responsible both for forking the actual child thread, and for joining with it when it terminates. On termination of the real child, the envelope thread may then use some other mechanism, which is independent of the identity of the child (or the envelope), to indicate termination of the child to the original parent, and to return any results from the child thread.

Mapping the Modula-3 \textit{Join} into the EPT \textit{join} is a little more complex, since it requires that when a Modula-3 \textit{Join} is requested, it must first be ascertained whether the thread supplied as a parameter has already joined (in the EPT sense) or not. If it has, then \textit{Join} can return immediately, otherwise a \textit{join} must be invoked. The EPT \textit{join}, however, will succeed when any child thread terminates, not necessarily the thread which is the parameter of the \textit{Join}. Thus, whenever a \textit{join} call returns, the identity of the terminating thread must be noted and compared with that of the required thread. If they are the same, then the \textit{Join} returns to its caller, otherwise, the identity of the terminating thread must be saved, and a new \textit{join} requested. To provide this mapping, it is necessary for a data structure to be maintained, which keeps a record of all those threads which have joined in the EPT sense, but not yet done so in the Modula-3 sense.

5.4 Asynchronous Communication

Providing an implementation of EPT asynchronous inter-thread communication in terms of the facilities offered by Modula-3 is clearly impossible. There is no provision in Modula-3 for a thread to cause an interruption of another thread unless the target thread is willing at least to the
extent of using AlertWait and AlertJoin in place of Wait and Join, and possibly calling TestAlert.

On the other hand, it is relatively easy to incorporate into the implementation of Wait and Join the inspection of a variable representing the alerted state of a thread. There is however a minor complication in that it is assumed that if a thread has called AlertWait or AlertJoin, but that alerted is FALSE at the time of the call, then the calling of Alert by another thread will cause an immediate effect in the target thread. If however the target was involved in an EPT wait or join, then it will be necessary for the Alert operation to cause an exception in the target thread in order to cause the wait or join operation to be abnormally terminated.

6. Conclusions

Although it is interesting from a comparison point of view to investigate the extent to which the facilities offered by one package may be implemented using those of another, when the exercise in conducted in earnest with a view to producing a real implementation, it is useful, and more efficient, to appeal to other mechanisms which may be available.

The hidden agenda behind this work is connected with an attempt to port the Modula-3 system which is available from the DEC Systems Research Laboratories in Palo Alto California to an Encore Multimax System running the UMAX Operating System. The port itself has been successful completed, and the Modula-3 system can be used to compile and run Modula-3 programs, but the implementation of the Threads module which implements the Threads interface supplied with the system assumes a "standard" Unix system platform, and hence simulates the concurrency by time-division scheduling within a single process.
Since UMAX offers a threads package (i.e. EPT), it was thought that it might be sensible to try to build an implementation of the Threads module using the native threads package. In fact, the facilities provided by EPT consist of a set of high level routines, such as those described above, and also a collection of lower level functions which allow the manipulation of more primitive data structures such as queues. So, although we have argued that the implementation of Modula-3 Wait, Signal and Broadcast operations required the introduction of a "condition manager thread", in practice it was more natural and simpler to implement these functions using EPT's lower level facilities.

The implementation of Modula-3 threads facilities using the primitives offered by EPT is by no means all that is necessary to provide the required implementation module. There are also other issues connected with the port which require attention. Two of these are the way in which exceptions are handled and the operation of the garbage collector.

Exception handling in Modula-3 operates on a per thread basis (i.e. exception propagation cannot cross thread boundaries) and hence when an exception is raised within a thread, the exception propagation mechanism must be able to find the thread's stack. This means that the exception mechanism must be aware of how the stack of an EPT thread may be located, which in turn requires a deeper knowledge of the internal behaviour of the EPT package than appears in the publication describing the facilities of the package.

Garbage collection is concerned with the retrieval of regions of memory which have been allocated to programs for use but are no longer required. A system which includes a garbage collector encourages the programmer not to be concerned about returning memory to the operating system when it is no longer required. The garbage collector is responsible for retrieving such areas of memory once they are no longer accessible (i.e. no program can reference them any longer). It is a difficult problem to build a garbage collector which will work in the presence of concurrent processing when shared memory is in use, and typically garbage collectors will require all processes and threads to be halted temporarily while the garbage collector is running.
For a system which runs within a single Unix process, and is responsible for its own scheduling of concurrent threads, the problem of halting all threads is straightforward; it is simply a matter of temporarily disabling the scheduling mechanism. A system running on top of a multi-threaded environment may not offer that option, and so an additional complication associated with providing the required implementation is concerned with signalling to the various concurrent threads to ask them to suspend their operations until the work of the garbage collector is complete. We look forward with interest to future developments in the area of concurrent garbage collectors (DeTreville 1990).

7 References

[Apollo 1987]

[Bartlett 1988]

[Birrell 1989]

[Cardelli 1991]
[Conway 1963]

[Dennis 1966]

[DeTreville 1990]

[Dijkstra 1968]

[Doeppner 1987]

[Encore 1988]

[Hoare 1974]

[Nelson 1991]

[Rovner 1985]

[Sun 1988]

[Wirth 1977]

[Wirth 1985]

[Xerox 1985]