

Implementing dependable process control applications using BCPL coroutines and Cintpos



Martin Richards^{*,†}

Computer Laboratory, William Gates Building, JJ Thomson Ave, Cambridge CB3 0FD, U.K.

SUMMARY

This paper outlines a method of implementing large and complex process control applications where extreme reliability is required and where the application is expected to have a long life of perhaps between 20 and 50 years. The approach relies on the use of an interpretive abstract machine that supports a multi-tasking operating system. The entire system including the operating system and the compiler for the systems implementation language are simple enough to be maintained as part of the application and so will continue to exist as long as needed. The abstract machine is simple enough to be easily re-implemented on new hardware as the need arises. This paper suggests using Cintcode as the abstract machine supporting an interpretive implementation of the Tripos portable operating system called Cintpos using BCPL as the systems implementation language. This paper pays particular attention to how synchronization primitives can be implemented using BCPL coroutines and the multi-tasking features of the Cintpos portable operating system and why such code will remain reliable throughout the long lifetime of the process control application.

KEY WORDS: process control, dependability, systems programming, coroutines, BCPL, Cintpos

1. Introduction

There are many process control applications that can be expected to have a long life, and in some reliability is of great importance with a cost of system failure possibly measured in tens or hundreds of thousands of Pounds per minute. The complete system must be constructed

*Correspondence to: Martin Richards, Computer Laboratory, William Gates Building, JJ Thomson Ave, Cambridge CB3 0FD, U.K.

†E-mail: mr@cl.cam.ac.uk

with great care so that component failures will not cause the entire system to fail. Often such systems run on multiple computers so that if one fails, another can automatically take over without interrupting the process being controlled.

The proposal explained in this paper is to implement the application using a simple language supported by an equally simple multi-tasking portable operating system all running on an interpretive abstract machine that can easily be re-implemented on any new hardware as the need arises during the lifetime of the application. More specifically, BCPL was chosen as the implementation language running on the Cintpos portable operating system all implemented on the Cintcode abstract machine. The abstract machine, the BCPL compiler and Cintpos are treated as part of the application and are each easily maintained by those who maintain the application code.

2. BCPL and Cintpos

The Tripos Portable Operating System [1] was first implemented in 1978 and ran on a variety of mini-computers of the day, such as the PDP-11, the Data General Nova and machines based on the Motorola 68000 processor. It was used at Cambridge and elsewhere for several years providing a framework for operating system and networking research. When the hardware on which Tripos ran became obsolete much of the original Tripos source code was archived.

Tripos was implemented in BCPL[2, 3] which still has a machine independent implementation freely available via my home page[4].

Tripos was later revived, mainly for historical reasons, when the 32 bit BCPL Cintcode system[5] was developed. The Cintcode abstract machine was enhanced to include interrupts, and the Tripos kernel re-implemented in BCPL rather than its original assembly language. This demonstration version of Tripos was called Cintpos and was made freely available via my home page. More recently, as machines became faster, it became clear that this interpretive system could form a good basis for dependable process control applications, and so the original “toy” version of Cintpos was further developed to improve its reliability and to include new features such as TCP/IP connectivity and better debugging aids. It is still freely available via the web[6], even in its current experimental form.

The reasons why the BCPL Cintpos system provides a good environment for writing high quality process control applications are summarised as follows:

1. The Cintpos operating system is simple and clearly defined. It has few kernel primitives but sufficient to provide the multi-tasking and synchronisation facilities needed for real time process control.
2. The whole system including the kernel primitives are implemented in BCPL that is itself supported by a clean and simple interpretive abstract machine (Cintcode) that can easily be re-implemented to run on any hardware. Currently this abstract machine is implemented in C but in years to come could be re-implemented in whatever language is appropriate at the time. The BCPL to Cintcode compiler is simple and easy to maintain being little more than 5000 lines of source code and able to re-compile itself in a fraction of a second.

3. BCPL coroutines[7] can be combined with the Cintpos multi-tasking facilities to provide an excellent way to control the scheduling and synchronization needed in process control.

This paper shows how features such as mutexes, condition variables, semaphores, signalling and waiting primitives can be implemented easily in BCPL within the Cintpos system. These very important and sometime subtle primitives will thus be guaranteed to behave identically throughout the lifetime of the application and not be subject to the inevitable differences arising from changes in standards, re-implementation of the primitives and operating system upgrades that are bound to happen in the next 50 years. It is hard to predict what Windows, Java, Linux or Pthreads will be like that far in the future especially if developments continue at the rate they have for the last 50 years.

The entire system is implemented in BCPL including the Cintpos Kernel, all the resident tasks, the libraries and the application code. The examples given below are in BCPL but these should be easy to understand for anyone familiar with C. The main differences are that BCPL is typeless with every variable and vector element being 32 bits long. The BCPL operator `!` is used for indirection like monadic `*` in C and is also used as a dyadic operator for vector subscription. The operator `@` will form a pointer to a variable just as monadic `&` does in C. `TEST` is used instead of `IF` in the BCPL version of the `if-then-else` construct found in C and `LOOP` is equivalent to `continue` in C. A complete description of BCPL is in Richards[3].

2.1. Cintpos Kernel Primitives

Cintpos is a simple multi-tasking operating system. Its tasks in modern parlance would be called threads since they are pre-emptive and share the same address space. Tasks communicate with each other and with devices by sending and receiving packets which are small vectors (or arrays) of 32-bit fields. The first field is called `pkt_link` and is used to form lists of packets, the next (`pkt_id`) holds the identity of the task or device to which the packet is addressed, and the third field (`pkt_type`) holds the type (or purpose) of the packet. The next two fields (`pkt_res1` and `pkt_res2`) hold the results of a request and finally there are up to six fields (`pkt_arg1` to `pkt_arg6`) which hold the arguments of a request. Since BCPL is typeless all these fields can contain values of any kind, such as integers, strings, pointers or even functions.

The call `qpkt(pkt)` will append the given packet to the work queue of its specified task or device. The identity of the sender replaces the `pkt_id` field so it can be returned by just a simple call of `qpkt`. When a task is ready to process a packet it calls `taskwait()` to de-queue the first packet from its work queue. If the work queue is empty, the task is suspended until a packet arrives.

Normally packets are thought of as requests sent to devices or server tasks, and the reply is usually returned using the same packet. While a server task is processing the request, the client is free to continue with other work but often immediately suspends itself in `taskwait` awaiting the reply. In practice, inter-task communication is particularly efficient in Cintpos.

Tasks have distinct priorities and the scheduler ensures that the runnable task with the highest priority has control. For example, as the result of an interrupt, a device returns a packet to a client task. If this task is currently suspended in `taskwait` and of higher priority than the currently running task, it will be given control. The scheduling algorithm is simple

to implement and has predictable properties. Greater fairness can be arranged within the application by dynamically changing task priorities, but this is rarely needed.

As an illustration of the use of `qpkt` and `taskwait`, we will arrange for a packet to be bounced between two tasks. One task will execute the following infinite loop:

```
qpkt(taskwait()) REPEAT
```

This will cause it to suspend itself in `taskwait` until a packet arrives, when the packet will be immediately returned to the sender using `qpkt`. `REPEAT` causes these operations to be repeated indefinitely. This bounce task can be exercised by another task running the following code:

```
{ LET link, id, type, res1, res2 = notinuse, bouncetaskid, ?, ?, ?
  LET pkt = @link
  writef("Sending a packet 10 million times to task %n*n", bouncetaskid)
  FOR i = 1 TO 10_000_000 DO
    { qpkt(pkt) // Send the packet
      taskwait() // Wait for it to return
    }
  }
  writef("Done*n")
}
```

The variable `pkt` points to the first of five consecutive words of memory named `link`, `id`, `type`, `res1` and `res2` which form the packet to be repeatedly sent to the bounce task. A safety check in `qpkt` requires the `link` field to have value `notinuse` (`=-1`). The `id` field is set to the identity of the bounce task. The usual packet fields `type`, `res1` and `res2` have been included although they are not used in this demonstration. As can be seen, this program will send the packet to the bounce task and wait for it to return 10 million times. When run on a 1 GHz machine using a Cintpos interpreter with all debugging aids enabled it takes about 108 seconds to complete, indicating that control can pass from one task to another in about half a micro-second. This is faster than the time to execute a single machine instruction on the machines on which Tripos first ran.

With the current BCPL implementation of `qpkt` and `taskwait`, it takes the execution of about 121 Cintcode instructions from the moment `qpkt` is entered to the moment `taskwait` in the destination task returns with the packet. Although this figure depends slightly on the relative priorities of the two tasks, it is a fair measure of the cost of transferring control from one task to another. The above figure was obtained using single step execution available in the standard Cintpos interactive debugger.

A typical task in Cintpos is the file handler which accepts requests from clients to open, read, write and close files. These typically cannot be processed instantaneously since there may be real time delays while the disk device reads or write sectors. Much of the time the file handler will be suspended in `taskwait` waiting for either a new request from a client or a reply from the disk or possibly the clock device. Tasks such as this are called *multi-event* tasks since they cannot predict what kind of packet will arrive next. A command language interpreter (CLI), on the other hand, is a *single-event* task since it has a single thread of execution and always knows what packet it is expecting to receive next. Whenever it sends a request it waits for the reply before proceeding. In such tasks, the calls of `qpkt` and `taskwait` are usually invoked in the wrapper function `sendpkt` whose definition is as follows:

```

LET sendpkt(link, id, type, r1, r2, a1, a2, a3, a4, a5, a6) = VALOF
{ UNLESS qpkt(@link)      DO abort(181)
  UNLESS taskwait() = @link DO abort(182)
  result2 := r2
  RESULTIS r1
}

```

In BCPL, function arguments are called by value and are laid out in consecutive memory locations and so behave like an initialized vector. In `sendpkt` the arguments form the packet to be sent eliminating the need to allocate, initialize and later release another vector for this purpose. The expression `@link` yields the pointer to this packet and is the appropriate argument for `qpkt`. If `qpkt` fails, possibly because of a bad task (or device) identifier, it will return `FALSE` causing `abort(181)`. Otherwise, `sendpkt` goes on to call `taskwait` suspending the task until the packet is returned. This implementation checks that the expected packet is indeed the one received.

The normal convention is for tasks and devices to place their results in the fields `r1` and `r2` of the packet. These are returned, respectively, as the result of `sendpkt` and in the global variable `result2`.

The code in the sender task given above could have been implemented using `sendpkt` as follows:

```

{ writef("Sending a packet 10 million times to task %n*n", bouncetask)
  FOR i = 1 TO 1_000_000 DO sendpkt(notinuse, bouncetaskid)
  writef("Done*n")
}

```

This is more convenient code but is slightly slower because of the extra safety checking done in `sendpkt`. All the standard library functions that communicate with other tasks or devices use `sendpkt`, including, for example, the `delay` function whose definition is as follows.

```
LET delay(ticks) BE sendpkt(notinuse, clkdevidev, ?,?, ticks)
```

A packet sent to the clock device returns to the sender after the specified number of ticks have taken place. A task can suspend itself for 5 seconds by executing:

```
delay(5*ticksperssecond)
```

Notice that the calls of `sendpkt` above have fewer arguments than its definition expects. BCPL permits this, giving the unspecified arguments undefined values. As an aside, the BCPL formatted output function `writef` is also variadic and is related to the C function `printf` which extends and improves `writef`, but the implementation of `printf` is more complicated since the size of C arguments depend on their types and so cannot be treated as elements of a vector.

2.2. Other Kernel Primitives

Although most of the flavour of Cintpos derives from the `qpkt-taskwait` mechanism, it is worth briefly noting the other kernel primitives. Tasks can be created and deleted using `createtask` and `deletetask`. The priority of a task can be changed using `changepri`. A

task can be suspended by a call of `hold` causing it to stop execution until explicitly released by a call of `release`. These are analogous to the deprecated functions `suspend` and `resume` in Java. Tasks each have a field of 32 flag bits that can be set and tested using `setflags` and `testflags`, and memory can be allocated and freed using `getvec` and `freevec` which are somewhat similar to the C functions `malloc` and `free`. Modules of compiled code can be loaded and unloaded using `loadseg` and `unloadseg`.

Cintpos does not provide the function `dqpkt` which, in Tripos, was used to retrieve previously sent packets.

3. BCPL coroutines

Like most block structured languages, BCPL uses a stack to hold function arguments, local variables and anonymous results. It uses an area called the global vector (analogous to FORTRAN's Blank COMMON) to hold non-local quantities accessible to all separately compiled sections of code. Each task has its own global vector and root stack but it is often convenient for a task to have separate runtime stacks so that different activities can proceed in pseudo-parallelism. This facility is provided by BCPL coroutines[7] which were designed in 1978. The idea of coroutines was not new since they first appeared in a paper by M.E. Conway[8] in 1963. Although few modern languages support them, there seems to be a growing realisation of their importance, see, for instance, Adya et. al.[9].

The specification of BCPL coroutines has remained unchanged since 1978 and is likely to remain unchanged for the foreseeable future and so can safely be used in applications expected to have a long life. A coroutine is created by a call of the form: `createco(fn, size)` where `fn` is the main function of the coroutine and `size` is the size of its stack. The result is a pointer to the newly created coroutine stack. Control can be passed to a coroutine by the call: `callco(cptr, arg)` where `cptr` is the result of a previous call of `createco` and `arg` is a value to be passed to the coroutine. The first time this happens `fn` will be applied to `arg`. If this function returns a result, `res` say, the coroutine will suspend itself and control passed back to the calling coroutine causing `callco` to yield the value `res`. The implementation of `createco` actually leaves the newly created coroutine suspended in the `cwait` call of the following infinite loop:

```
c := fn(cwait(c)) REPEAT
```

A few locations near the base of a coroutine stack hold system data including `fn`, `size`, `c`, and `parent` which points to the calling coroutine, if any. A coroutine with a null parent pointer is said to be inactive. The call `cwait(res)` suspends the current coroutine leaving it inactive and passing control and the value `res` to the parent. The function `callco` is only permitted to transfer control to inactive coroutines. There is a chain of coroutines from the current coroutine through the parent links to the outermost coroutine (the root) which is identified by the special parent link value of -1. The length of the parent link chain increases by one on every call of `callco` and decreases by one on every call of `cwait`. All coroutines not on this chain are inactive.

The function `resumeco` takes the same arguments as `callco` but has a subtly different effect. It causes the specified coroutine to resume execution exactly as `callco` does, but the parent

of the calling coroutine becomes the parent of the called coroutine. The length of the parent chain thus remains unchanged. Two typical and important uses of `resumeco` are given later in this paper in the definitions of `die` and `coread`.

A coroutine within a task will keep control until it explicitly calls one of `callco`, `cawait` or `resumeco`, and so can safely manipulate data it shares with other coroutines in the task without having to use synchronization primitives such as semaphores, or mutexes, provided the shared data is left in a consistent state when it gives control to another coroutine. Sometimes a coroutine wishes to retain exclusive access to some resource, such as a file, for a longer period and this will require the use of synchronization primitives but, as will be seen, these are easy to implement. Sharing resources between tasks is possible but requires calls of Cintpos kernel primitives such as `qpkt` and `taskwait` which are necessarily more costly. Any inactive coroutine can be deleted using `deleteco`.

As an example we will use coroutines to re-implement the bounce demonstration described above. Firstly two coroutines `bounce_co` and `sender_co` are created by the following code:

```
bounce_co := createco(bouncefn, 200)
sender_co := createco(senderfn, 200)
```

where `bouncefn` and `senderfn` are defined as follow:

```
LET bouncefn(val) BE val := cawait(val) REPEAT

LET senderfn(count) BE
{ writef("Calling the bounce coroutine %n times*n", count)
  FOR i = 1 TO count DO callco(bounce_co, i)
  writes("done*n")
}
```

The sender coroutine can be started by:

```
callco(sender_co, 10_000_000)
```

This completes its 20 million control transfers between the two coroutines in about 24 seconds indicating that the coroutine primitives are many times faster than `qpkt` and `taskwait`. More specifically, in the current implementation only 10 Cintcode instructions are executed between the start of `callco` in `senderfn` and the return from `cawait` in `bouncefn`, and 11 between the start of `cawait` in `bouncefn` and the return from `callco` in `senderfn`. Remember that it takes about 121 instructions to pass control from one task to another using `qpkt` and `taskwait`. As an aside, `bouncefn` could have been defined as the identity function `LET bouncefn(val) = val`, since body of `createco` already contains a suitable loop.

There is a benchmark test call Cobench[10] that tests the efficiency of BCPL-style coroutines implemented in a variety of languages. It demonstrates that interpretive BCPL is not much slower than compiled C and very much faster than a Java version using threads.

4. Multi-event tasks

As was mentioned in section 2.1, tasks can be classified as single- or multi-event. A single-event task typically uses calls of `sendpkt` as defined above to request a service from another task,

suspending itself until the request is answered. Single-event tasks are thus like conventional single threaded programs. Conversely, a multi-event task has an organization that resembles the event loops found in many windowing systems. It typically has a main loop in which `taskwait` is called suspending the task until the next packet arrives. The packet is then passed to the coroutine that was waiting for it, or given to a main coroutine if the packet is a new request. If the main coroutine is busy processing a previous request, the packet is queued and processed later. Many tasks within Cintpos and within process control applications are best organized in this fashion. Rather than implementing this mechanism afresh for each multi-event task, it is convenient to invoke the library function `gomultievent` which is defined as follows.

```

LET gomultievent(maincofn, size) = VALOF
{ LET mainco = createco(maincofn, size)
  LET oldsendpkt, wkq = sendpkt, 0
  UNLESS mainco RESULTIS FALSE // Unsuccessful return
  multi_count, mainco_busy := 1, FALSE
  sendpkt, pktlist := sndpkt, 0 // Enter multi-event mode
  callco(mainco, 0) // Ask mainco to start everything up

  WHILE multi_count>0 DO // Start of the multi-event loop
  { LET pkt = taskwait() // Wait for a packet
    LET co = findpkt(pkt) // Find which coroutine, if any, owns it
    IF co DO { callco(co, pkt); LOOP }
    IF mainco_busy DO // If the main coroutine is busy
    { LET p = @wkq // append the packet onto the
      WHILE !p DO p := !p // end of the work queue
      !pkt, !p := 0, pkt
      LOOP
    }
    { callco(mainco, pkt) // Give the packet to the main coroutine
      IF mainco_busy | wkq=0 BREAK
      pkt := wkq // De-queue a waiting request
      wkq := !pkt
      !pkt := notinuse
    } REPEAT // Process the waiting request
  }

  sendpkt := oldsendpkt // Return to single event mode
  deleteco(mainco) // and delete the main coroutine
  RESULTIS TRUE // Successful return
}

```

This function uses the following variables in the current task's global vector:

`multi_count`. This holds an indication of the amount of work still to be done in multi-event mode. It is typically incremented whenever a multi-event coroutine is created and decremented when it dies. On reaching zero, the task can return to single-event mode.

`mainco_busy`. This is initialised to `FALSE` and is only `TRUE` when the main coroutine is busy processing a request packet. It determines whether a new request packet can be processed now or must be queued.

`pktlist`. This holds an initially empty list of nodes giving the mapping between packets and the coroutines that own them. This list is used by `findpkt(pkt)` to find the coroutine, if any, that is waiting for the given packet.

When running in multi-event mode, the standard version of `sendpkt` is replaced by the multi-event version `sndpkt`. So this becomes the version called (indirectly) by all standard library functions such as `writef` when used in multi-event coroutines. The definition of `sndpkt` is as follows.

```
AND sndpkt(link, id, type, r1, r2, a1, a2, a3, a4, a5, a6) = VALOF
{ LET ocis, ocos, ocurrentdir = cis, cos, currentdir
  // The following three variables form the pktlist node.
  LET next, pkt, co = pktlist, @link, currco
  pktlist := @next // Insert [next,pkt,co] as first node in pktlist
  UNLESS qpkt(pkt) DO abort(181) // Dispatch the packet
  UNLESS cowait()=pkt DO abort(182) // and wait for the reply.
  // Restore the saved global state
  cis, cos, currentdir := ocis, ocos, ocurrentdir
  result2 := r2 // Recover the two results
  RESULTIS r1
}
```

As with `sendpkt`, the arguments of `sndpkt` form the packet to be used. Since multi-event coroutines may wish to call standard library functions such as `writef` and `delay` that indirectly call `sndpkt`, some of the task's global variables must be saved and restored by `sndpkt`. These are: `cis` the currently selected input stream, `cos` the currently selected output stream and `currentdir` the currently selected filing system directory. Next, `sndpkt` creates a node containing a link to the next node in the list, the packet and the current coroutine, and inserts it at the head of `pktlist`. The packet is then dispatched using `qpkt` with a safety check to ensure that it was valid. Unlike `sendpkt`, it uses `cowait` to wait for the packet to be returned, knowing that this will be sent by `callco` from within `gomultievent`.

The call of `findpkt` in `gomultievent` will search `pktlist` for a specified packet, de-queuing it if found. Its definition is as follows:

```
LET findpkt(pkt) = VALOF
{ LET a = @pktlist // a is the address of the next link word
  { LET p = !a
    UNLESS p RESULTIS 0 // The packet was not found.
    IF p!1 = pkt DO { !a := !p // Remove from pktlist and
                      RESULTIS p!2 // return the coroutine.
                    }
    a := p
  } REPEAT
}
```

If `findpkt` finds a matching node in `pktlist`, it unlinks it and returns the corresponding coroutine pointer. Notice that the saved global variables `cis`, `cos` and `currentdir` are restored in `sndpkt`, and that the space for both the `pktlist` node and the packet itself will be released when control returns from `sndpkt`. This is clearly more efficient than using a general purpose space allocator.

After creating the main coroutine and initializing the global variables `multi_count` and `mainco_busy`, `gomultievent` enters multi-event mode by overriding `sendpkt` and setting `pktlist` to zero. It then calls `main_co` to create and start the multi-event coroutines. Control returns to `gomultievent` when `main_co` has completed its initialization and is ready to receive

request packets. Typically, the main coroutine activates the multi-event coroutines as necessary but the conventions of how this is scheduled is application dependent and not relevant to `gomultievent`. All that is required is for `multi_count` to become zero when the main and all the multi-event coroutines agree to return to single-event mode.

Careful study of the event loop in `gomultievent` will show that, when a packet is received by `taskwait`, if owned by a waiting coroutine it will be given to that coroutine. If not, it must be a request packet and will be given to the main coroutine for processing, but if it is currently busy it will be placed at the end of the work queue (`wkq`).

When the main coroutine completes the processing of a request, it sets `mainco_busy` to `FALSE` and calls `cawait`. If this happens when the work queue is non empty, it is immediately given the next packet to process. It is up to the application programmer to decide which requests to handle entirely in the main coroutine and which to sub-contract to other multi-event coroutines, or even other tasks. This decision depends on the amount of parallelism the application requires.

5. Coroutine suicide

Sometimes a coroutine in a multi-event task completes the job it was given and must delete itself. It is not possible for a coroutine to reliably delete itself using the call `deleteco(currco)`, since the stack frame that was active just before the call of `deleteco` resides in the coroutine stack that is being returned to free store (and possibly reallocated and used by another task). The local variables and more importantly the function return link information is thus not valid after `deleteco` returns. A reliable solution is to create a killer coroutine whose sole purpose is to delete other coroutines. It can be created by the following assignment.

```
kill_co := createco(deleteco, 100)
```

Any other coroutine wishing to commit suicide should call:

```
resumeco(kill_co, currco)
```

since this causes `kill_co` to delete the current coroutine before giving control to its parent. Note that the loop inside `createco` allows `kill_co` to be used repeatedly. Sometimes it is convenient to define the suicide function as follows:

```
LET die() BE resumeco(kill_co, currco)
```

Since this mechanism is required so often, it is normally best to create and delete `kill_co` in `gomultievent`.

6. Synchronization primitives

Process control applications typically take input from many sources such as keyboards, bar code readers, external communication lines and sensor devices of all kinds. These generate data

asynchronously and at unpredictable times. The output of the system is typically written to files and sent down communication lines possibly to devices such as printers, displays, robots or other computers. Often different parts of the system must access shared resources such as disk files or shared data structures in memory. So that different parallel activities can proceed correctly, access to these shared resources must be controlled using synchronization primitives. Many such primitives have been proposed in the literature and provided either directly in programming languages or supplied in library packages. A few examples are the communication primitives in Occam[11], synchronized objects and methods in Java[12], mutexes and condition variables in the POSIX Pthreads library[13] and the similar but different facilities available in the WIN32 API[14]. These mechanisms are fairly new and their specifications are still being revised. Different implementations of them often vary in subtle ways.

This section shows how a variety of synchronization primitives can be implemented efficiently using coroutines running within multi-event tasks, and having the advantage that once implemented they will remain the same unless the application programmer wishes them to change. They rely on the abstract machine being a uniprocessor even though the system as a whole may be running on multi-processor hardware.

6.1. Occam Style Channels

A channel in Occam provides a synchronized way of transmitting a value from one Occam process to another. One process can execute an input statement to read the value from a channel while another executes an output statement to send the value down the same channel. An input statement will not complete until an output statement on the same channel is executed, and likewise, an output statement will not complete until an input statement on the same channel is executed. When both processes reach the communication point, the value is transferred and then both processes continue independently.

We use coroutines running in multi-event mode to model the Occam processes and use the functions `coread` and `cowrite` to provide the synchronous communication between them. A channel is represented by a channel word that will contain a pointer to the first coroutine to reach a call of `coread` or `cowrite` involving the channel. If no such call has yet been made, the channel word is zero. The definitions of `coread` and `cowrite` are as follows.

```
LET coread(ptr) = VALOF TEST !ptr
  THEN { LET cptr = !ptr
    !ptr := 0 // Clear the channel word
    RESULTIS resumeco(cptr, currco) // Get value from cowrite
  }
  ELSE { !ptr := currco // Set channel word to this coroutine
    RESULTIS cwait() // Wait for value from cowrite
  }

LET cowrite(ptr, val) BE TEST !ptr
  THEN { LET cptr = !ptr
    !ptr := 0
    callco(cptr, val) // Send val to the waiting coread
  }
  ELSE { !ptr := currco // Wait for coread to be ready
    callco(cwait(), val) // Send val to coread
  }
```

In both functions, `ptr` is a pointer to a channel word. If the channel word is zero, neither `coread` nor `cowrite` is waiting to communicate, otherwise it points to the first coroutine making the call of `coread` or `cowrite`. A value can be passed when control passes from one coroutine to another and this is how the value is passed from `cowrite` to `coread`.

If `coread` is executed first, it will find that the channel word is zero and will set it to point to itself by `!ptr := currco`, and will then become suspended in `cwait` waiting for `cowrite` to send a value. When `cowrite` is called it will find the channel word is non zero and so knows that it points to the waiting coroutine. All it has to do is clear the channel word (`!ptr := 0`) and send the value by the calling `callco(cptr, val)`.

If, however, `cowrite` is called first, the implementation is somewhat more subtle. As before the coroutine suspends itself, but this time in the statement `callco(cwait(), val)`, after updating the channel word to point to itself. At this moment `cowrite` is waiting to receive the identity of the `coread` coroutine. When `coread` is called, it notices that `cowrite` is ready to send a value which `coread` obtains by calling `resumeco(cptr, currco)`. This gives `cowrite` the identity of the `coread` coroutine so that the call `callco(cwait(), val)` in `cowrite` knows where to send the value. At the same time the parent of `coread` becomes the parent of `cowrite`, so that when `coread` next suspends itself, control will return to `cowrite`. This scheme has the merit that execution preference is given to `coread` which typically results in the slightly more efficient communication described earlier.

The mechanism just described is efficient since there is little code to execute and, as we have seen, the coroutine primitives are themselves efficient. However, the implementation is subtle and has other subtle implications. Firstly, these functions must be used within multi-event coroutines under the control of the event loop in `gomultievent`. Typically, `gomultievent`'s main coroutine would declare and initialize the channel word before creating both the producer and consumer coroutines. These would then be started successively using `callco`.

Assuming the consumer was started first, it will run until it chooses to suspend itself, typically in the `cwait` call in `coread`. This will return control to the main coroutine which will call `callco` to start the producer which would typically run until it reaches a call of `cowrite`. This will find that the channel word is non zero and so immediately passes a value to the consumer which would be given control at that moment. Control will return to the producer when the consumer coroutine next calls `cwait`. This may not be, as expected, in `coread`, but could be in `sndpkt` if the consumer, for example, invokes a service from the file handler by calling `writef`. If this happens the consumer will become suspended in `sndpkt` waiting for a reply. In the mean time, control will pass to the producer coroutine allowing it to make progress. The reply from the file handler will not reach the consumer until it is received by `taskwait` in `gomultievent` and this will not happen until all the multi-event coroutines of the task are suspended. Normally multi-event coroutines require very little CPU time but, if one does, it would probably be wise for it to subcontract the work to a lower priority task so as not to interfere with the multi-event scheduling. Provided none of the coroutines require much CPU time they will rapidly all become suspended, causing the event loop to call `taskwait`.

7. Stream locks

It is common for several multi-event coroutines to wish to write messages to a log file. Clearly only one should be allowed to write to the file at a time. To ensure this happens, a coroutine wishing to write to the log should call `lock_logfile` which only returns when it has obtained exclusive access to the log file. When this coroutine has finished outputting its message, it releases the lock by calling `unlock_logfile`. These two functions can be defined as follows.

```
LET lock_logfile() BE
  TEST log_wait_queue = 0
  THEN log_wait_queue := -1          // Mark as locked
  ELSE { LET link, co = 0, curcco    // Make lock node [link, co]
        TEST log_wait_queue=-1
        THEN log_wait_queue := @link // Make a list of length one
        ELSE { LET p = log_wait_queue // or append the lock node
              WHILE !p DO p := !p     // to the end of the queue.
              !p := @link
            }
        cwait() // Suspend until unlock_logfile() is called
        // We now own the lock and log_wait_queue will be non zero
      }

LET unlock_logfile() BE
  TEST log_wait_queue = -1
  THEN log_wait_queue := 0          // Mark as unlocked
  ELSE { LET co = log_wait_queue!1  // Dequeue the first lock node
        log_wait_queue := !log_wait_queue
        UNLESS log_wait_queue DO log_wait_queue := -1
        callco(co) // Give control to the first coroutine
      }
```

These have been optimised on the assumption that the lock is almost always free and even when locked other coroutines are very rarely waiting for it. The variable `log_wait_queue` is zero when the file is unlocked, it equals -1 if the current coroutine owns the lock and no others are waiting for it, otherwise it contains a list of coroutines waiting for the lock. If `lock_logfile` finds that `log_wait_queue` is zero, it just sets it to -1 and returns. If `unlock_logfile` finds `log_wait_queue` is -1 it resets it to zero and returns. In the rarer case when `lock_logfile` finds that `log_wait_queue` is non zero it appends a node to the `log_wait_queue` being careful with the -1 marker and then suspends itself in `cwait`. If `unlock_logfile` finds `log_wait_queue` is not equal to -1, it must contain a list of waiting coroutines. It dequeues the first lock node in the queue and transfers control to its coroutine using `callco`, being careful to set `log_wait_queue` to -1 if no other coroutines are waiting. The provision of this kind of locking mechanism would clearly be much less efficient if required to work between threads especially if running on multi-processor hardware.

8. Condition variables

Condition variables can be used when an activity must wait for some condition involving shared variables to become satisfied. When another activity changes a value that might cause

the condition to be satisfied, it must awaken the first activity so that it can re-evaluate the condition. This can be implemented for use within a multi-event task using a single word for the condition variable and two functions `wait` and `notify`. Assuming `condwaitlist` is the condition variable, it should be initialized to zero, and can be used in code of the form:

```
UNTIL <complicated condition> DO wait(@condwaitlist)
```

When a variable in the condition is updated by another coroutine it should call:

```
notify(@condwaitlist)
```

This will awaken every coroutine waiting on the condition variable so that they can all re-evaluate the condition and possibly continue normal execution. Simple implementations of `wait` and `notify` are given below.

```
LET wait(ptr) BE
{ // These form a waitlist node [link, cptr]
  LET link, cptr = !ptr, currco
  !ptr := @link // Insert the node at the head of the list
  cwait()       // Suspend until the waiting condition
}              // may have changed.

LET notify(ptr) BE
{ // Wakeup all coroutines on the given wait list
  // so that they can each re-evaluate the condition.
  LET p = !ptr
  !ptr := 0 // Clear the condition wait list
  WHILE p DO { LET cptr = p!1; p := !p; callco(cptr) }
}
```

A condition variable is just a, possibly empty, list of coroutines containing nodes of the form (*link*, *cptr*), where *link* is either zero or points to another node and *cptr* is the coroutine pointer to a coroutine suspended in `wait`. As can be seen, `wait` simply inserts a node at the start of the wait list pointed to by `ptr`. Notice that the wait list node is formed from two adjacent local variables which cease to exist as soon as control returns from `wait`. This is safe since by then `notify` will have finished with the wait list node in question.

Every coroutine on a wait list can be woken up by calling `notify`. This simply executes `callco` for every coroutine in the list. But, to avoid looping indefinitely, it must first reset the condition variable to zero. The implementation of `wait` given above causes the most recent coroutine to wait to be the first to be released. In rare situations where this strategy is not appropriate, `wait` could be modified to append the node to the end of the list.

Observe that `notify` and `wait` are analogous to `pthread_cond_broadcast` and `pthread_cond_wait` in the Pthreads library[13], and for `notifyall` and `wait` in Java, but are much more efficient since coroutines are non pre-emptive.

9. Discussion

The synchronization mechanisms just described show how easily they can be implemented using coroutines within multi-event tasks under Cintpos, and it should be clear that many other synchronization primitives could be implemented with similar ease.

As we have seen in the discussion of the Occam channel example, the detailed flow of control between the coroutines is not always easy to follow and this may cause some users to have doubts about whether they actually work. The following observations should help.

- A well written multi-event task will have coroutines that never require much CPU time before transferring control to another coroutine. Such transfers may result from direct calls of `cwait`, `callco` or `resumeco`, or these may also be called indirectly via calls to the synchronization primitives or library functions (such as `writef` or `delay`) that involve calls of `sndpkt`.
- Whenever a multi-event coroutine becomes suspended it will have transferred control to another multi-event coroutine or returned to the main event loop. Whenever a coroutine is suspended, there must be a reference to it somewhere in the system so that it can be resumed later. This reference may be the parent link of another coroutine, or in a `pktlist` node, or in one of the queues maintained by one of the synchronization primitives, such as `log_wait_queue` used in `lock_logfile`.
- The main coroutine will create most of the coroutines used in a multi-event task and schedule work for them depending on what request packets are received.
- Provided all these coroutines have been implemented correctly, whenever a packet is received by `taskwait` in the event loop, control will pass briefly through a sequence of one or more coroutines before returning to the event loop where the task will typically suspend itself in another call of `taskwait`. The exact order in which control passes from one coroutine to another should not matter, just as the scheduling processes in a conventional operating system should not affect correct working of the system as a whole.

10. Final comments

The strategy suggested in this paper for the implementation of complex real time process control applications that are expected to have lifetimes exceeding 20 or even 50 years is as follows.

- Base the entire system on a simple interpretive abstract machine implemented easily in any suitable language and run on any hardware. The abstract machine should include an interrupt mechanism and should be able to handle asynchronous devices such as clocks, disks and communication lines.
- Use an implementation language with a simple compiler to implement the entire application including a multi-tasking operating system kernel and all the standard tasks such an operating system needs. The operating system, the implementation language, compiler and debugging aids should be simple enough to be easily maintained in-house by a single person. This strategy greatly reduces the dependence on outside vendors to supply the operating system, compiler and synchronisation primitives.
- Implement the necessary synchronization primitives as part of the application to retain control of their precise specification and behaviour. These are best provided using multi-event coroutines running cooperatively within separate tasks.

- Be aware that, if the system is implemented in a currently fashionable language under a current modern operating system purchased from an outside vendor, then it will be large, complex and probably bug-ridden and that maintenance for these critical parts of the system is not likely to be available in 50 years time since the language, operating system and vendor may not then exist. Consider how few operating systems and languages of as little as 35 years ago are still available and properly maintained today.

The hope is that this paper demonstrates that using BCPL coroutines running under the Cintpos portable operating system is a good basis for implementing a complex process control application expected to have a long life.

ACKNOWLEDGEMENTS

I would like to acknowledge the work of the many systems programmers in the Ford Motor Company (Europe) who have implemented a significant process control system in BCPL running under Tripos that has been used successfully for more than 20 years. Many of their ideas have been incorporated into this paper, but any mistakes are entirely my own.

REFERENCES

1. Richards M, Aylward AR, Bond P, Evans RD, Knight BJ. The Tripos portable operating system for minicomputers. *Software-Practice and Experience* June 1979; **9**:513–527.
2. Richards M, Whitby-Strevens C. *BCPL - the language and its compiler*. Cambridge University Press: Cambridge, 1979.
3. Richards M. The BCPL user manual. www.cl.cam.ac.uk/users/mr/bcplman.ps.gz [24 March 2004].
4. Richards M. Home page. www.cl.cam.ac.uk/users/mr [24 March 2004].
5. Richards M. BCPL Cintcode distribution. www.cl.cam.ac.uk/users/mr/BCPL/bcpl.{tgz,zip} [24 March 2004].
6. Richards M. BCPL Cintpos distribution. www.cl.cam.ac.uk/users/mr/Cintpos/cintpos.{tgz,zip} [24 March 2004].
7. Moody K, Richards M. A coroutine mechanism for BCPL. *Software-Practice and Experience* February 1980; **10**:765–771.
8. Conway ME, *Design of a seperable transition-diagram compiler*. CACM July 1963; **6** (7):369–408
9. Adya A, Howell J, Theimer M, Bolosky WJ, Douceur JR, *Cooperative Task Management without Manual Stack Management* Proceeding of the 2002 Usenix Annual Technical Conference, June, 2002
10. Richards M. A benchmark test for BCPL style coroutines. www.cl.cam.ac.uk/users/mr/Cobench/cobench.{tgz,zip} [24 March 2004].
11. INMOS. Occam Programming manual. *INMOS Ltd* 1983.
12. Flanagan D. *Java in a nutshell*. O'Reilly, 2002.
13. Lewis B, Berg DJ. Multithreaded programming with Pthreads. *Sun Microsystems* 1998.
14. Pretzold C. Programming Windows: The definitive guide to the Win32 API. *Microsoft Press* 1999.