CROSS PLATFORM SIMULATION OF THE INTERRUPT SCHEMA USED WITHIN THE GNU MODULA-2 RUNTIME SYSTEM

GAIUS MULLEY

School of Computing,
University of Glamorgan,
CF37 1DL, UK
E-Mail: gaius@gnu.org

Abstract: This paper reports on a technique used to simulate the behaviour of module priorities, interrupts and context switching within a Modula-2 runtime system. The technique presented here is highly portable as the complete system is written in C and Modula-2. This implementation of the Modula-2 runtime system supports processor context switching, interrupt service routines without requiring any assembly language source code. The runtime system has been ported to the x86, Opteron, Athlon 64, Alpha, Itanium processors running GNU/Linux, Sparc based Solaris, PowerPC MacOS, x86 Open Darwin and the x86 processor running FreeBSD.

Keywords: Modula-2, interrupts, pthreads, simulation.

INTRODUCTION

The motivation for producing a Modula-2 front end for GCC includes both providing a migration path for legacy source code and providing a robust compiler for production systems. Currently there are only a few commercial Modula-2 compilers being actively maintained. Code which was written ten or fifteen years ago may still be compiled by older commercial (possibly unmaintained) compilers, a number of these compilers generate 16 bit code. While the 32 bit x86 processors remain, compilers targeting these processors may be run in compatibility mode. However time is running out as the computing industry is switching to 64 bit microprocessors [AMD, 2003a]. While x86 emulation or 16 bit backwards compatibility or running 32 bit code on a 64 bit platform is possible they all have serious drawbacks. In order for the older source to run natively the source code will either have to be translated into another high level language or alternatively a Modula-2 compiler which can target these new generation of microprocessors will have to be acquired. GNU Modula-2 suits this purpose as it has the advantage of being closely tied to GCC. Not only does this produce excellent code and good architectural and operating system coverage but it also utilises many of the GCC features. For example GNU Modula-2 can invoke the C preprocessor to manage conditional compilation; in-lining of `SYSTEM` procedures, intrinsic functions, memory copying routines are also exploited; access to assembly language using GCC syntax is also provided. GNU Modula-2 also supports sets of any ordinal type (memory permitting).

HISTORY OF MODULA-2

The first implementation of Modula-2 became operational on a PDP-11 in 1979 and the initial language definition was published as a technical report [Wirth, 1980]. The language Modula-2 is similar to that of Pascal but extends many of its concepts by introducing the module construct, access to low level programming features and processes. Over subsequent years four major revisions were made. The first three by Wirth and the last by the International Standards Institute. Over the years these have become informally known as PIM2, PIM3, PIM4 and ISO respectively [Wirth, 1983; Wirth, 1985; Wirth, 1989; ISO/IEC, 1996].

MODULA-2 AND INTERRUPT PRIORITIES

One of the important features of Modula-2 is that it provides all the necessary primitives to implement a microkernel either in language constructs or system procedures [Wirth, 1976]. The language allows for modules specified to run with a specific interrupt mask. This has the effect that any procedure declared within a module will also inherit this interrupt mask.

Thus when an exported procedure is invoked it will automatically set the processor interrupt mask to that of its parent module and restore the previous processor interrupt mask before returning. For example in figure 1, it can be seen that procedure `foo` is declared in the inner module which was specified to operate with an interrupt mask of 7 whereas the outer module was specified to operate with an interrupt mask of 0. When the procedure `foo` is called from the outer module the current interrupt mask is saved and set to 7. After `foo` returns the interrupt mask is restored back to 0 again.

```
MODULE outer[0] ;

   MODULE inner[7] ;
   EXPORT foo ;

   PROCEDURE foo ;
   BEGIN
   END foo ;

   END inner ;

BEGIN
   foo
END outer.
```

Figure 1: example of a procedure associated with an interrupt mask

MODULA-2 AND PROCESSES

The `SYSTEM` module as defined by Wirth [Wirth, 1983; Wirth, 1985; Wirth, 1989] provides four procedures which can be used to create a process, context switch between two processes and switch to another process should an interrupt occur. These `SYSTEM` procedures are particularly elegant as they provide high level mechanisms from which the programmer can coordinate process control. By using these four procedures and the module priority schema a complete microkernel executive can be written without any assembly language source code [Wirth, 1983]. This makes Modula-2 an extremely effective language for teaching real-time systems or microkernel implementation. Of course the module interrupt mask priority mechanism and the `SYSTEM` procedures all form part of the Modula-2 runtime system and conventionally these are implemented in assembly language and are provided by the compiler vendor. Nevertheless from the compiler user's perspective no additional assembly language is required to implement a process executive. The prototypes for these `SYSTEM` module procedures are given in figure 2. The procedure `NEWPROCESS` instantiates the procedure represented by parameter `p` into a process `new`. Whereas the procedure `TRANSFER` context switches from process `p1` to process `p2`. The procedure `IOTRANSFER` initially context switches from process `first` to `second` however when an interrupt occurs it saves the current processor volatile environment in `second` and then context switches back to the process `first`. The `LISTEN` procedure briefly removes the processor interrupt mask.

```
PROCEDURE NEWPROCESS (p: PROC;
                      a: ADDRESS;
                      n: CARDINAL;
                      VAR new: PROCESS)

PROCEDURE TRANSFER (VAR p1: PROCESS;
                    p2: PROCESS)

PROCEDURE IOTRANSFER (VAR First,
                           Second: PROCESS;
                      InterruptNo: CARDINAL)

PROCEDURE LISTEN
```

Figure 2: the `SYSTEM` procedures which coordinate process activity

RELATED WORK

In modern systems lightweight processes or threads are often used in event driven paradigms. Lightweight processes or threads share the address space of the parent process and are created by allocating a new stack to a new thread. The thread libraries normally fall into two categories: preempt-able and non preempt-able thread libraries. Preempt-able thread libraries nearly always require assistance from the kernel [Bernard, 2000] so that the thread scheduler can context switch away from a processor bound thread. This assistance although enhancing the fairness of the thread scheduler unfortunately is not yet portable across operating systems.

The GNU Pthread library is a very portable POSIX/ANSI-C based library for UNIX based platforms which provides non-preemptive priority-based scheduling for multiple threads of execution. All threads in this library use the same address space of the application but each thread has its own program counter, stack, signal mask and `errno` variable. The thread scheduling is achieved through cooperation between the scheduler and the threads being managed.

Previous work has shown that Ada Tasking can be implemented through a Pthread library [Giering, 1993]. The Ada tasking semantics are fairly complex, nevertheless Pthreads were sufficiently flexible to allow implementation of Ada tasks without users having to directly call upon the services of Pthreads.

The Modula-2 process primitives are completely different to the Ada task model and the Modula-2 primitives operate at a lower level by dictating which contexts to switch and which interrupts to serve. Neither do they define the method of IPC.

DEFINITION OF THE PROBLEM

The problem facing the GNU Modula-2 project is how can module priorities and the process manipulation facilities be re-implemented so that they are portable and execute in user address space on a multiuser operating system?

The requirement for portability between operating systems makes the GNU Pthread [Engelschall, 2005] library seem attractive yet this also presents the problem of simulating interrupts as the Pthread library is non-preemptive.

SIMULATION OF INTERRUPTS

This section describes how interrupts in user mode on a multiuser operating system can be simulated and coordinated with the Modula-2 process primitives.

Fortunately the Pthread library contains low level context switching primitives and these allow for a straightforward implementation of `NEWPROCESS` and `TRANSFER`. The procedure `NEWPROCESS` is implemented by calling `pth_uctx_create` and `pth_uctx_make`. These two Pthread primitives create a process context. Each Modula-2 process is represented by a single Pthread context with an associated interrupt priority mask. In GNU Modula-2 the definition for the `PROCESS` type is shown in figure 3 together with the interrupt priority range.

```
PROCESS  = RECORD
              context: ADDRESS ;
              ints   : PRIORITY ;
           END ;

PRIORITY = [0..7] ;
```

Figure 3: definition of `PROCESS` and `PRIORITY`

The implementation of `NEWPROCESS`, `TRANSFER` and `IOTRANSFER` are shown in figures 4, 5 and 7.

There are three categories of interrupts currently simulated in this runtime system: input, output and clock interrupts. The input and output interrupts are simulated by providing a mapping to a file descriptor and the clock interrupts are simulated through the use of a relative ordered time ascending list. A module `SysVec` provides procedures to map file descriptors onto simulated interrupt vectors and it also implements an interrupt dispatcher. This dispatcher is called whenever the simulated interrupt mask is altered and the duty of the dispatcher is to build the set parameters and time parameters for `pth_select`. The values to the set parameters are derived from the active interrupt list which were populated by successive calls to `IncludeVector` via the `IOTRANSFER` procedure. The compiler was modified to generate runtime procedure calls to this dispatcher whenever one procedure is about to call another procedure associated with a different interrupt mask (as in figure 1).

```
PROCEDURE NEWPROCESS (p: PROC; a: ADDRESS;
                      n: CARDINAL;
                      VAR new: PROCESS) ;
TYPE
   ThreadProcess = PROCEDURE (ADDRESS) ;
VAR
   ctx: ADDRESS ;
   tp : ThreadProcess ;
BEGIN
   localInit ;
   tp := ThreadProcess(p) ;
   IF pth_uctx_create(ADR(ctx))=0
   THEN
      Halt(__FILE__, __LINE__, __FUNCTION__,
           'unable to create user context')
   END ;
   IF pth_uctx_make(ctx, a, n, NIL, tp, NIL,
                    illegalFinish)=0
   THEN
      Halt(__FILE__, __LINE__, __FUNCTION__,
           'unable to make user context')
   END ;
   WITH new DO
      context := ctx ;
      ints    := currentIntValue ;
   END
END NEWPROCESS ;
```

Figure 4: implementation of `NEWPROCESS`.

```
PROCEDURE TRANSFER (VAR p1: PROCESS;
                    p2: PROCESS) ;
VAR
   r: INTEGER ;
BEGIN
   localMain(p1) ;
   p1.ints := currentIntValue ;
   currentIntValue := p2.ints ;
   IF p1.context=p2.context
   THEN
      Halt(__FILE__, __LINE__, __FUNCTION__,
           'switching to the same process')
   END ;
   currentContext := p2.context ;
   IF pth_uctx_switch(p1.context,
                      p2.context)=0
   THEN
      Halt(__FILE__, __LINE__, __FUNCTION__,
           'unable to context switch')
   END
END TRANSFER ;
```

Figure 5: implementation of `TRANSFER`.

```
IOTransferState =
   RECORD
      ptrToFirst,
      ptrToSecond: POINTER TO PROCESS ;
      next: POINTER TO IOTransferState
   END ;
```

Figure 6: declaration of `IOTransferState`.

Every call to `IOTRANSFER` results in a new `IOTransferState` being constructed and this is kept on the callers stack so that the context of `first` process can be restored when the interrupt is serviced. The procedure `IOTRANSFER` initially saves the current processes context into `first` and then restores the context belonging to process `second`. The `IOTransferState` is constructed and initialised appropriately. A pointer to this record, (`q` in figure 8) is created when calling `AttachVector`. The procedure `AttachVector` associates `q` with the particular interrupt number. When the interrupt is serviced the dispatcher context switches back to process `second` by invoking `TRANSFER` and passing the relevant fields of `q`.

```
TRANSFER(q^.ptrToSecond^, q^.ptrToFirst^)
```

The last `SYSTEM` procedure `LISTEN` simply listens to all pending interrupts briefly before returning. `LISTEN` is easily implemented by a call to the interrupt dispatcher after unmasking all interrupts. The simulated system module also provides a non standard procedure `ListenLoop` which exhibits the same behaviour as:

```
LOOP
   LISTEN
END
```

except that it allows the interrupt dispatcher to block waiting for a simulated interrupt to occur thus respecting the underlying operating system.

The implementation works surprisingly well, it only amounts to 1600 lines of code and it allows input, output and time based interrupts to be simulated. These modules now form part of the GNU Modula-2 runtime system and they provide a method whereby microkernel executives originally designed for stand alone systems can be simulated under UNIX like operating systems.

```
PROCEDURE IOTRANSFER (VAR First,
                      Second: PROCESS;
                      InterruptNo: CARDINAL)
VAR
   p: IOTransferState ;
BEGIN
   localMain(First) ;
   WITH p DO
      ptrToFirst  := ADR(First) ;
      ptrToSecond := ADR(Second) ;
      next := AttachVector(InterruptNo,
                           ADR(p))
   END ;
   IncludeVector(InterruptNo) ;
   TRANSFER(First, Second)
END IOTRANSFER ;
```

Figure 7: implementation of `IOTRANSFER`.

GNU MODULA-2 SOURCE CODE

At the time of writing GNU Modula-2 0.5 has been released. This front end can be grafted onto GCC-3.3 source tree. The code break down for GNU Modula-2 release 0.5 is shown in table 1 and the line count for all other languages is taken from GCC-3.3 [Weinberg, 2003].

Category	
Core compiler	250,000
Back ends	410,000
rs6000	40,000
x86	42,000
Front ends	480,000
C	861,000
Ada	298,000
Java	127,000
C++	105,000
Modula-2	124,000
Runtime libraries	
Java	274,000
ObjC	8,200
f77	11,000
Modula-2	28,300
Modula-2 interrupt schema	4,200

Table 1: lines of code by category

The modules at the core of the Modula-2 process implementation and interrupt simulation are `SYSTEM`, `SysVec` and `pth`. The later is only an interface file which maps Modula-2 procedures and parameters onto their C counterparts. The source for these three modules amounts to approximately 1600 lines of code. The total process runtime library code which includes a higher level executive and timerhandler is 4200 lines.

CONCLUSIONS AND FURTHER WORK

In conclusion this technique has been very successful as it has been widely ported to many different UNIX like operating systems and different processor architectures [AMD, 2003b; Intel, 2000]. This work demonstrates that the GNU Pthread library is capable of being used to implement the low level primitives: `NEWPROCESS`, `TRANSFER`, `LISTEN` and `IOTRANSFER`. In the future this technique will be extended to implement the ISO Modula-2 process model.

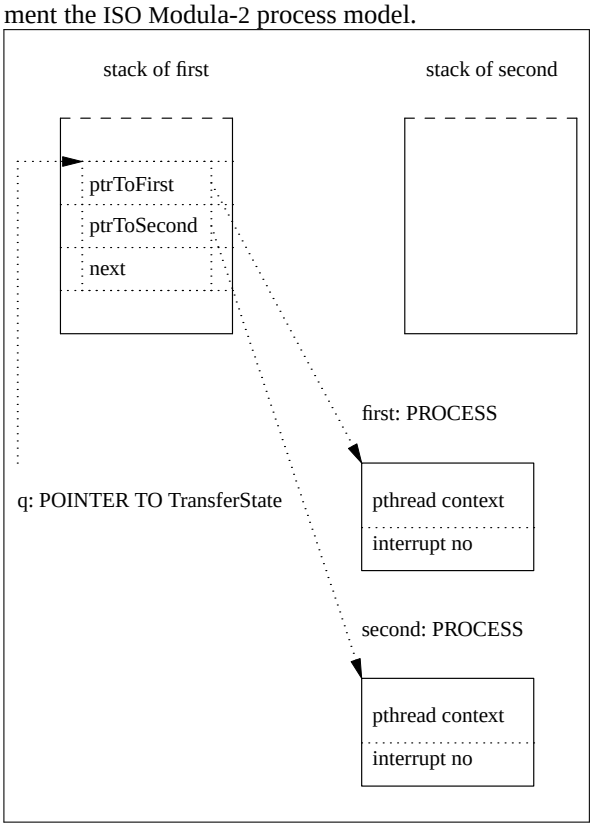

Figure 8: data structure diagram showing key values just before the call to `TRANSFER` in `IOTRANSFER`.

ACKNOWLEDGEMENTS

The author would like to thank all the readers and contributors of the GNU Modula-2 mailing list for their patience and many valuable bug reports, repeated test builds and patches over the last six years. Many people and organisations have been very generously providing access to computing equipment, which has enabled an extensive list of ports. In particular, Benjamin Kowarsch, John Calleys, Keith Verheyden for access to Power PC hardware. The University of Glamorgan research group MoCoNet for access to a dual Opteron system and the University of Glamorgan's Information Security Research Group (ISRG) for access to dual Itanium, Ultra Sparc I and top end 32 bit Intel hardware.

Finally a great debt of thanks is owed to the Free Software Foundation without which the source code to GCC would not be free to read, modify and redistribute.

REFERENCES

AMD 2003, *AMD x86-64 Architecture Programmer's Manual*, AMD, USA.

Wirth N. 1980, "MODULA-2," Institut für Informatik, Eidgenössische Technische Hochschule, Zürich.

Wirth N. 1983, *Programming in Modula-2,* 2nd Edition, Springer Verlag.

Wirth N. 1985, *Programming in Modula-2,* 3rd Corrected Edition, Springer Verlag.

Wirth N. 1989, *Programming in Modula-2,* 4th Edition, Springer Verlag, New York.

ISO/IEC 1996, "Information technology - programming languages - part 1: Modula-2 Language," ISO/IEC 10514-1.

Wirth N. 1976, "Design and Implementation of Modula," Software Practice and Experience, 7, Pp. 67-84.

Bernard J. 2000, "Using the Clone() System Call," The Linux Journal.

Giering E.W., Mueller F. and Baker T.P. 1993, "Implementing Ada 9X Features using POSIX Threads: Design Issues," Proceedings of the Conference on TRI-Ada, Pp. 214-228, ACM Press, Seattle, WA, USA. ISBN 0-89791-621-2.

Engelschall R.S. 2005, *GNU Pth - The GNU Portable Threads*, Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Weinberg Z. 2003, "A Maintenance Programmer's View of GCC," GCC Developers Summit, Pp. 257-268, Ottawa, Canada.

AMD 2003, *Software Optimization Guide for the AMD Opteron Processor,* AMD, USA.

Intel 2000, *UNIX System V Application Binary interface; IA-64 Processor ABI Supplement,* Intel, USA.

BIOGRAPHY OF AUTHOR

Gaius Mulley is a senior lecturer at the University of Glamorgan. He is the author of GNU Modula-2 and the groff html device driver `grohtml`. His research interests also include performance of microkernels and compiler design. He obtained a Bsc(Hons) and PhD in Computer Science from the University of Reading and later worked for Meiko Scientific.