SPRING KERNEL
LOW LEVEL CONTROL

Lory Molesky
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

Working Document
December 1990

ABSTRACT

This document focuses on the low level control mechanisms for the Spring Kernel. Currently, the Spring Kernel (a scheduler and multiple dispatchers) is running "on top of VRTX" as a user process. VRTX hinders the Spring Kernel from assuming sufficient control of low level mechanisms, and therefore should be eliminated.

Being a real-time kernel, we desire that the Spring Kernel have minimal latency. In the current dispatcher implementation, polling cannot be avoided because of the lack of low level control. A software architecture which avoids polling, thus reducing latency, is proposed and an implementation plan is described. Although multiprocessor aspects complicate this process, by using one hardware clock to drive all other software clocks, we can achieve synchronized time, thus synchronized events.

1 Implementation Plan

In this section, we intend to enumerate implementation tasks which comprise the next versions of the Spring kernel, V2 and V3. The intent of V2 is to replace VRTX with version 1 of the Spring kernel. V3 will merge the memory management effort with V2.

V1 consists primarily of the scheduler processes on the SP (System Processor) board and the dispatcher processes on the AP's (Application boards). Using synthetic workloads, synthetic tasks arrive dynamically, are scheduled by the scheduler, and are executed by the dispatchers. This code runs "on top of VRTX". In V1, there is essentially only one process per board. Thus, we can run the entire V1 code as a single kernel image. Application tasks are simulated by a VRTX sc_delay() statement – this can be replaced by the notify system call discussed in section 2

The V2 implementation will not have true tasks or processes for a number of reasons. First, the implementor can focus on the low level details of supporting various hardware devices without complicating these issues with the higher level task/process models. Second, the activity of the task/process/memory management implementation can be carried out in parallel by a separate party.

V2 will be aimed at eliminating VRTX, yet keeping most of the BSP (Board Support Package). The present V1 software will essentially replace VRTX. The current BSP performs initialization of both hardware devices (the Z8036 timer chip and the serial I/O ports) and VRTX. The BSP sections supporting hardware will be kept, the VRTX initialization will be replaced with V2 specific initializations. Thus, the V2 kernel will run entirely in supervisor (privileged) mode and there will be no user level processes.

1.1 Spring V2 Implementation Steps

1. Create a stripped BSP (w/o VRTX) supporting only a prototype timer interrupt routine.
   The purpose of this first step is to eliminate VRTX, yet keep some hardware support software. This kernel will essentially be composed of one ISR supporting the Z8036 periodic timer interrupt. In response to the Z8036 timer interrupt, the timer interrupt service routine will update some global variable. The 135bug package and/or the board support package can be used to verify the correct operation of the timer ISR.

2. Alter the timer routine to conform with a multiprocessor real-time architecture.
   This is discussed in detail in section 2.

3. Add support of serial I/O
   Each 136 board has two serial ports, these are used by V1 to output diagnostic information. It is questionable whether serial I/O is really needed. However, if it is easy, it should be done.

4. Sink the V1 dispatcher/scheduler into the V2 kernel.

5. Define a kernel opcode vector table

   In order to obtain privileges (to execute in supervisor mode) operating system calls are generally implemented as exception handlers. At our disposal is the TRAP instruction [2]. By calling the TRAP instruction with an argument vector between 0 - 15, sixteen different exceptions can be generated. These sixteen different exceptions correspond to the EVT numbers 32 - 47, as illustrated in table 2 in appendix A.

   Since there are only 16 different TRAP arguments at our disposal, it is common practice to reserve one TRAP vector for OS calls. In addition, some method of defining the opcode (a register, or known memory location) is used to distinguish between different system calls. This amounts to a two level table – entry to the EVT distinguishes a system call from other system exceptions, from there, the opcode distinguishes the specific system call.

   Defining the stack format and the registers used is a major part of the specification of the system call interface. In addition, as VRTX has done, we can define an OVT (Opcode Vector Table) [1]. This is basically a set of conventions for parsing the opcodes, and generating errors when illegal opcodes are passed.

   In our current operating environment, TRAP 0 is used for VRTX system calls, TRAP 1 for RTSCOPE calls, and TRAP 15 for 135Bug calls.

Wherever possible, the gnu C compiler will be used to download source code into the RTL memory. However, the BSP package is at odds with this goal, being written in an non-gnu 68020 assembler dialect. This is a minor problem and can be overcome by either rewriting the BSP to a gnu dialect, or using an existing object code converter.

1.2 Spring V3 Implementation Steps

It is envisioned that the 135Bug will not be compatible with the new memory management scheme. If this is the case, once the memory management scheme is worked out, we can begin work on a sequential (as opposed to parallel) debugger. Constructing this debugger will probably be a significant effort, yet necessary to make future progress. Among the obstacles confronting the debugger implementor will be:

1. The 135Bug is Hardwired

   The current debugger, 135bug, is in firmware. When the 136 boards are powered up, the 135bug is instantiated as the onboard monitor. The 135bug binary image is in EPROM, memory mapped from 0xfff00000 to 0xfff1ffff. As an example of the tight coupling of the 135bug package to the hardware, consider the function of reset (pressing the red button on any 136 board).

> Port 1 and port 2 are reconfigured to their default state. The breakpoint table is cleared. The offset registers are cleared. The target registers are invalidated. Input and output character queues are cleared. On-board devices (timer, serial prots, etc.) are reset. [3]

Although some actions of the reset function will likely be common to any debugger (such as clearing I/O queues and on-board devices), the offset registers are specific to 135Bug.

The quickest (albeit somewhat dirtiest) path to constructing a new debugger will be to load it "over" the 135bug package. The steps envisioned are, boot the machine as normal, reprogram the MMU, load the new debugger package. These steps will constitute a new base software set.

2. We Do Not Have the Source for the 135bug Package

2 Kernel Events

This section describes a framework for the construction of time based kernel primitives on a Spring (multiprocessor) node. On a Spring node, we desire the notion of time to be consistent between the SP and AP's. We also wish to avoid the technique of polling, instead, having time based events be interrupt driven.

In the current implementation of the Spring Kernel (a user process), dispatchers poll a software clock when in a pending state. The VRTX software clock is a memory location that is updated each time the Z8036 generates an interrupt (every 10 msec.). The pending state occurs when there is an available task i on a dispatch queue, but its sst (scheduled start time) has not arrived (as can occur due to some other guaranteed task j with a resource conflict with task i being scheduled prior to i). Local software clocks are synchronized with the master software clock by a modification of VRTX; one hardware clock, located on the SP, updates all software clocks whenever the timer interrupt is generated (every 10 milliseconds). Even polling the local software clock in the dispatcher increases the response time of the dispatcher to a change in the software clock, thus increasing the dispatch time (and thus the worst case computation time of a task).

When the underlying VRTX kernel is replaced by the Spring kernel it is possible to eliminate such coarse polling. This can be achieved by constructing *event driven* low level kernel mechanisms. Suppose that an application will be idle until some time in the future, t_1, at which time application task T_i has been scheduled to run. Instead of polling until time t_1, the dispatcher will wait until time t_1 by

1. The dispatcher sets semaphore s
2. The dispatcher issues a system call notify(t,s). When time t has elapsed, the dispatcher process is notified by the clearing of semaphore s.

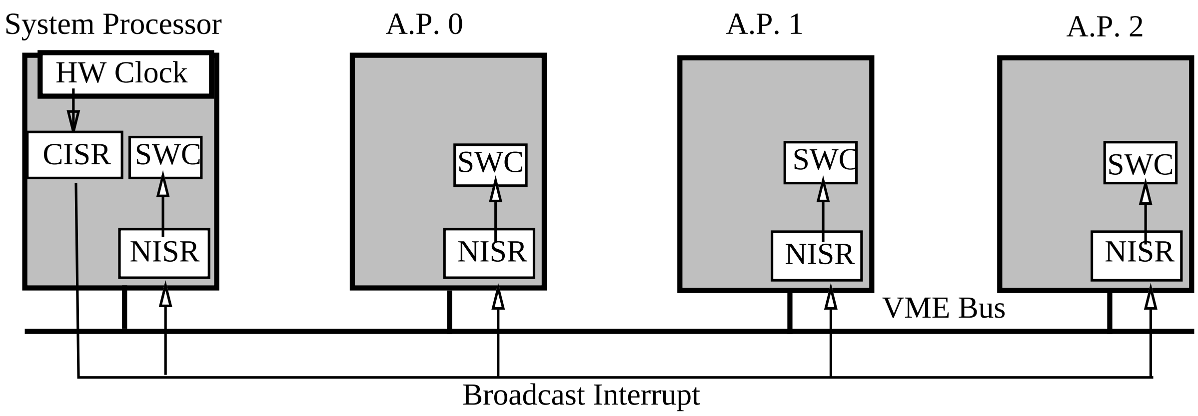

Figure 1: One Hardware Clock is the Event Source for all Boards on a Node. NISR: Notify Interrupt Service Routine; CISR Clock Interrupt Service Routine; SWC: Software Clock.

3. The dispatcher spins on semaphore s

The underlying kernel implements notification with a combination of a queue of events (ordered on time) and an interrupt service routine (NISR), triggered by the hardware clock (the Z8036). Besides being responsible for updating the software clock, each time the NISR is invoked it checks the notify queue for an event which has elapsed (current time, as indicated by the software clock, is greater than or equal to the time stored on the queue). Once an event on the notify queue has elapsed, the semaphore associated with this event is cleared, and the event structure is deleted from the notify queue. The clearing of the semaphore enables the process pending on the notify() to resume.

On a Spring node, the execution of tasks by the dispatcher must be based on either a master clock or a local synchronized clock. In order to achieve this local synchronized software clocks, we use one hardware clock per node, as illustrated in figure 1. The hardware clock, residing on the SP board, generates periodic interrupts (say once per millisecond). An interrupt service routine, CISR on the SP board is responsible for generating a broadcast interrupt [1]. This broadcast interrupt invokes another interrupt service routine, denoted NISR in the figure 1, on each board (the SP and the AP's). Each NISR is responsible for updating the software clock on its local board, and for managing the notify queue.

Thus, the low level kernel mechanisms described enable us to circumvent polling by dispatchers. There is one NISR per board, each NISR can be viewed as being directly triggered by a master clock (per multiprocessor). For all events relevant to a board, this NISR performs time based event notification via access to the notify queue.

[1] On the MVME136A, referencing the lm0 bit causes an interrupt to occur on all boards. Being a broadcast interrupt, only one VME bus cycle is required to interrupt all boards.

One extension to this scheme can also be implemented. In addition to updating the SW clocks on each slave board (AP), the HW clock can be reinitialized. These slave HW clocks would not generate periodic interrupts, but would be used to obtain fine resolution timings. This will give us a finer granularity notion of time on the slave boards. This extension may have some subtlties – clock drift must be accounted for, as well as the delay between the SP hardware clock interrupt and the reinitialization of the AP hardware clocks. This finer granularity time would not be compatible with the notify queue (section 2), since conditions are tested based on an interrupt generated by the hardware clock on the SP.

2.1 More Sophisticated Uses of the Notify Queue

Given that each processor's NISR is driven by a master hardware clock, we examine other potential uses for the Notify Queue in the context of the Spring Kernel.

1. Guaranteed tasks that exceed their deadline:
   Prior to executing a guaranteed task, a dispatcher can place an entry in the notify queue, based on the sft (scheduled finish time), indicating that if the application task is not finished by this time, some corrective action must be taken. The semantics of the notify must be altered slightly to include the address of a function to call in order to take corrective action. Of course, once the application task completes, the dispatcher explicitly performs a delete of this entry from the notify queue.

2. System Processes that exceed their deadline:
   These tasks are handled in a similar fashion to Guaranteed tasks. Consider the scheduler on the SP. If this task does not complete before its worst case computation time, the guarantee algorithm should be terminated, and the original schedule should be reinstated.

3. Dispatching when resource reclaiming:
   When resource reclaiming is used, a dispatcher in a pending state also uses *Reclaim_delta* [2] to determine if its sst has arrived. This computation could also be accommodated in the notify routine - instead of comparing the software clock against the time stored in the first entry of the notify queue, this first entry must be diminished by the value of *Reclaim_delta*.

2.2 Predictability Considerations

Any computation performed in the NISR must be bounded. Since invocation of the NISR will occur periodically, (per software clock tick) the scheduler and dispatchers can account for this periodic computation (with a bandwidth argument).

[2] *Reclaim_delta* is the total amount of time reclaimed at a particular instant in time.

2.2.1 The Notify Queue

How many entries can be in the notify queue? This is a fundamental question pertaining to predictability. If the answer is Ntasks, where Ntasks is the number of tasks in the system, it seems that we must cap the maximum number of tasks in the system. The problem stems from the semantics of the notify queue – the queue entry with the smallest timestamp must be checked on each hardware clock interrupt. Either this check requires a search, or the queue must be ordered on the timestamps. In either case, the complexity of either enqueueing or dequeueing will be a function of Ntasks (O(Ntasks) or O(log Ntasks), depending on the data structures used).

In the proposed (non-preemptive) V2 implementation, it appears that all we need is one entry in the notify queue at any one time. The tasks on the dispatch queues are already ordered on time, so, as long as the head of the dispatch queue executes on time, the other dispatch queue tasks will not be jeapardized. Thus predictability is maintained.

Other requirements may dictate that some function of Ntasks events be present in the notify queue. This may arise in a non-preemptive environment, or through support of ITC. The details of these requirements are unclear at this time, but this issue should be kept in mind. This is not a problem inherent in the proposed notify queue implementation – it is one inherent in the existence of a function of Ntasks events which need checking at one point of runtime.

Given a more than one task limit on the notify queue, the implementation of the queue can be performed as a linked list, ordered on time. The dequeueing and notify cost can be made O(1) while the enqueueing cost is O(N). If the Notify Queue has unique timestamps (the time component of the queue entry), then dequeueing is predictable (O(1)), since only the head will be examined at each software clock tick. If a timestamp has arrived, it must be deleted and the head pointer advanced. This is an O(1) operation.

Appendix

A

This table was constructed by combining information in page 6-5 of [2], and page 4-15 of [4]. Of particular interest are the "user defined vectors", numbers 63 - 255. A number of these have been assigned to 136 hardware functions, such as the broadcast interrupt ($LMIRQ^*$) and the abort switch ($ABORT^*$) located on the front panel of each 136 board.

vector number decimal	hex	vector offset (hex)	Assignment
0	0	000	Reset: Initial Interrupt Stack Pointer
1	1	004	Reset: Initial Program Counter
2	2	008	Bus Error
3	3	00c	Address Error
4	4	010	Illegal Instruction
5	5	014	Zero Divide
6	6	018	CHK, CHK2 Instruction
7	7	01c	cpTRAPcc, TRAPcc, TRAPV Instructions
8	8	020	Privilege Violation
9	9	024	Trace
10	a	028	Line 1010 Emulator
11	b	02c	Line 1111 Emulator
12	c	030	(Unassigned, Reserved)
13	d	034	Coprocessor Protocol Violation
14	e	038	Format Error
15	f	03c	Uninitialized Interrupt
16	10	040	
	through		(Unassigned, Reserved)
23	17	05c	
24	18	060	Spurious Interrupt
25	19	064	
	through		Levels 1 - 7 Interrupt Auto Vectors
31	1f	07c	
32	20	080	
	through		Trap 0 to 15
47	2f	0bc	
48	30	0c0	
	through		(Unassigned, Reserved)
63	3f	0fc	
64	40	100	$ABORT^*$
65	41	104	$ACFIRQ^*$
66	42	108	$TMRIRQ^*$
67	43	10c	$SFIRQ^*$
68	44	110	$SIOIRQ^*$
69	45	114	$SIGHP^*$
70	46	118	(Unassigned)
71	47	11c	$VSBIRQ^*$
72	48	120	(Unassigned)
73	49	124	(Unassigned)
74	4a	128	$LMIRQ^*$
75	4b	12c	$SIGLP^*$
76	4c	130	
	through		Remaining User defined vectors (192)
255	ff	3fc	

Figure 2: The EVT (Exception Vector Table)

References

[1] Inc. Hunter & Ready. *VRTX/68020.* Hunter & Ready, Inc., 1986.

[2] Motorola Inc. *MC68020 32-Bit Microprocessor User's Manual.* Prentice-hall, Englewood Cliffs, N.J., 1985.

[3] Motorola Inc. *MVME135BUG 135Bug Debugging Package.* Motorola Inc., 1986.

[4] Motorola Inc. *MVME135, MVME135-1, MVME135A, MVME136, and MVME136A 32-Bit Microcomputers User's Manual.* Motorola Inc., 1989.