Issues and Alternatives for Real-Time Systems

Don Wells
(prepared for NRAO RT Workshop at AOC 1999-04-12)
dwells@nrao.edu

April 9, 1999

Contents

1 Why are we having this meeting?

Here are four reasons: (any others?)

- several major projects starting up need background and guidance in making big decisions with longterm implications for NRAO
- several innovative approaches now have some operational experience that needs to be shared
- recent tech developments like Linux and Java suggest new approaches to building RT systems that are integrated into the networked world
- RT groups at NRAO sites have tended to work in isolation, so that we have had some NIH problems, which is a luxury that in an era of tight budget and fewer people we can no longer afford.

2 RT experiences

Some of the participants in this meeting have been doing RT for *decades*, and have multiple experiences. I hope that insights gained from these experiences will come out during the course of this workshop. My own experiences include:

- (1967-69) Telescope and instrument RT applications in Fortran with shared common blocks under an IBM RTOS; probably similar to Modcomp systems at NRAO at that time
- (1970,1984..) a number of RT systems in assembly language with my own interrupt routines and no RT-OS; the modern equivalent is 'embedded' controller programming
- (1972-76) large telescope, instrument and data display RT applications in Forth, very similar to Forth systems at the old 36-foot and early 12-meter.
- (1982) a 'soft' RT application (a slow state machine) under VMS in Fortran
- (1988-92) a big system using C under VxWorks with networking.

A number of my designs have had to operate in a production mode and be maintained by other people after I departed, so I have worried about long-term maintenance issues. In my experience, 'documentation' in the classic sense was not necessarily the critical issue for ensuring long lifetimes for my RT code, contrary to much popular wisdom; I acknowledge that the popular wisdom may be more valid today. B.Clark, Langston, Sowinski

3 RT Design Alternatives

- Networked versus Embedded
  - Networked systems facilitate development support on state-of-the-art platforms while enabling RT systems to support remote debuggers, remote login, X-clients, NFS clients and RPCs for custom services. Pisano, Blachman, Brandt, etal
  - Embedded systems are fine for static applications that need to be robust and don't need to evolve. The VLBA Correlator contains more than 100 embedded CPUs which act as high speed front-end processors, almost dedicated at the one-task-equals-one-CPU level.
- Synchronous versus Asynchronous
  - Synchronous (ticking) state-machine RT systems can be designed with *proofs* that they will meet "hard" deadlines while managing resources. This is the classic, conservative design choice.
  - Asynchronous RT systems can achieve higher performance statistically, but perhaps(?) at the expense of poorer worst-case performance. They depend on a pervasive distributed absolute clock and do a just-in-time scheduling analysis.

  We need to discuss this alternative thoroughly at this workshop. The GBT M&C approach appears to *guarantee* correct synchronous activity in its highly distributed system. *Should we build all new NRAO systems with this paradigm?* M.Clark, Blachman, D'Addario, etal

- Lightweight Threads versus Memory Protection
  - Lightweight threads are the task abstraction of VxWorks and POSIX Pthreads: multiple instances of any C function can be spawned as tasks, each with its own stack space for dynamic variables, but sharing static variables. Task initiation is quick and cheap, but the tasks all execute in the same memory address space, and a wild pointer in one task can trash the memory of another task or maybe even overwrite the kernel. Tasks are often ephemeral in this style. For example, VLBA Correlator "jobs" spawn an ensemble of tasks which terminate at completion of the job; a new job can initiate its tasks in idling state at low priority while prior job completes.
  - Each task executes in its own protected memory space in many older RT OSes. This is safer, but task startup is slower and it is harder to share code and data between tasks. Tasks tend to do multiple jobs in this style, and to be permanent.

  We need to discuss the pros and cons of this RT design alternative during this workshop, because OSes of both types still exist (I wish we had a list of RTOSes versus type), and code designed for one style may be hard to port to the other. Ford, Brooks, etal

- Semaphores-for-Everything versus Interrrupt-Lockout
  - VxWorks and Pthreads and any SMP OS guard all critical regions with semaphores, and interrupts are almost never inhibited.
  - Interrupts are disabled during kernel critical regions in many older RT OSes. This produces a variable latency in context switching, and usually statistically poorer responsiveness overall.

  The author dislikes the typical uses of the word "Deterministic" in RT contexts, especially by vendor salespeople! The word is misleading, because RT latencies are always variable. Older systems which inhibited interrupts in the kernel often claimed to be deterministic, but they never were. The real issue is the *shape of the histogram* of latencies: the goal in RT design is to bound the worst-case tail of the histogram of latency. Ford

4 Strategy/Policy/Philosopy Issues

- Table-based designs simplify documentation and inter-personal commununication:

  > Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.[1]

  The point is that people tend to understand tables *intuitively*. There is an equivalence between DBMS relations, table script language and pointer-linked C structures in the VLBA Correlator architecture[2] which provides a good example of table-based design technique. B.Clark

- State-machine notation, with transition rules, simplifies documentation and inter-personal communication. It appears that people tend to understand state-transition rules *intuitively*. This concept made a key contribution in the VLBA Correlator project circa 1991. Shelton

- The role of inheritance of methods of object classes in the GBT should be explored during this workshop – I would like to hear the GBT designers tell us the pros and cons of their concepts. For example, does the automatic inheritance and invocation of destructors eliminate memory leaks? Does the inheritance of exception mechanisms enable warm restarts by sending soft resets to clear hangups (deadlocks), perhaps with multiple levels of reset? M.Clark, Glendenning

[1] F. P. Brooks, Jr., *The Mythical Man-Month (Essays on Software Engineering)*, 1975, on p.102 in a section titled "Representation is the Essence of Programming".

[2] see VLBA Correlator Memo 95, September 1989, in section 2.2 titled "Tuples-Scripts-Structs".

- Cheap PCs with a rich market of controller cards raise real questions about any further investment in VME crates. Industrial RT controllers are exploiting this market. Somes, Vanosdol
- What about Windows NT? The pros and cons are similar to those that applied to VMS ten years ago: the richness of the environment is seductive, with its synergistic combination of RT and non-RT applications, but there are real risks associated with dependence on a single company whose strategic goals (primarily non-RT) may be inconsistent with ours. Creager

5 Nasty technical problems, some unsolved

- Memory leaks are a perennial problem. I would like to hear discussion of approaches to detecting/preventing memory leaks in RT systems. My own most successful leak elimination campaign depended on coding wrappers for `malloc()` and `free()` which maintained private lists of allocations.

- Cache memory is(was?) a frequent source of trouble, especially in the presence of intelligent device controllers. Modern multiple-CPU shared-memory systems prevent this problem by invalidating cached values; their mutexes are global.

- Priority inversion can cause mysterious failures: a priority inversion bug caused the periodic resets which occurred when the Mars Pathfinder spacecraft was first operating on Mars two years ago.[3] I wonder if priority inversion is known not to be a problem in the VLA Modcomp systems. I also wonder whether the VLA RT team knew about priority inversion when they built the VLA in early 70s. Sowinsky, B.Clark Priority inversion is the one piece of RT theory which I did not know in the 70s (I first heard of it in 1988); the fact that so fundamental an issue was unknown to me in the 70s makes me slightly nervous that maybe there are other RT concepts that I don't know about. Is priority inversion a problem only in synchronous state machines, are asynchronous systems immune to it? M.Clark

[3] see "What Happened on Mars?" at `http://www.cs.cmu.edu/Groups/real-time/mars.html`

- RT device driver availability is a perennial problem for RT designers, especially for RTOSes like VxWorks. Hoyle, Brooks
- RT network security has been a lurking problem for some years now; we have depended too much on the fact that our RT OSes are not well known. "..Security through obscurity is no security at all.."[4] In the end, in a fully networked world, RT systems must adopt security models which are just as rigorous as the ones which non-RT systems have been been forced to adopt. We should discuss this subject in this workshop. Hunt

[4] William Lefebrve, *Sun Spots*, 1989-04-25

- Debugging concurrent programs can be frustrating:

  > "..I *still* don't know what the paradigm should be for debugging a distributed and concurrent program... I mean, when you've got an application running across a number of machines, how should you think about it? When error messages come from deep in the bowels of the system, asynchronously, how are you going to know what they mean?"[5]

  - Running two copies of the debugger helps when debugging problems with lowlevel IPC and semaphores – step the two windows through the state changes. This approach doesn't help in a truly large complex system where too many things are happening.
  - Millisecond-precision timestamp logs really helped with VLBA Correlator debugging at a certain stage circa 1991. A commercial example is `TNF_PROBE_N()` and `TNFview`.[6] Granados
  - Thread-aware graphical debugger technology[7] is now available
  - Note the "Debug Mutexes" package.[8] The idea is to do a temporary global replace of "`_mutex_`" with "`_dmutex_`" to get features like recording the "owner" of a mutex, list of all threads sleeping on a mutex, detection of non-owner-unlock, list of all mutexes, count usage, etc.

[5] Bill Joy, in an interview in *Unix Review*, April 1988, p.67
[6] see [LB98, p.251]. TNF=Trace Normal Form.
[7] Sun Visual Workshop [LB98, p.247]
[8] see [LB98, p.111–112,249–250]

- There is no generally accepted benchmarking package for RT systems. Some years ago the CMU RT group had a benchmark called "Hartstone"[9] for evaluating Ada-based RT systems for DoD applications. I haven't heard anything about it recently. I am a little surprised that nobody has produced a version of Hartstone for POSIX threads environments; such a thing could probably be portable and automatically adaptable to a variety of environments. The concept of Hartstone was execution of multiple tasks at different priorities executing at different "harmonic" (the "H" of Hartstone) frequencies, with detection of failures to meet schedules. Of course, this approach only applies to synchronous 'ticking' RT systems, not to asynchronous systems.

- The goal of producing proofs of correctness for RT systems has proved to be elusive, as far as I know. Proofs of simple RT algorithms are easy, even trivial (CS profs have filled the journals with them), but proofs for interesting *system* problems are *hard*, if not impossible. Actually, in the real world, it is the unanticipated problems that bring systems down (consider the Mars Pathfinder case). One model of systems design is to think in terms of watchdog mechanisms that detect failures of design assumptions, and intervene to bring systems back into a safe operating state within finite time. The philosophy is that failure is inevitable, and so the issue is whether your design "fails safe". The RT systems which are known by the author to be robust in production use are all of this type. Heald

[9] CMU-CS-90-110, "Distributed Hartstone Real-Time Benchmark Suite", Clifford W. Mercer, Yutaka Ishikawa, Hideyuki Tokuda, March 1990. Several of the authors are still at CMU's *Real-Time and Multimedia Laboratory*; see `http://www.cs.cmu.edu/Groups/real-time/`.

6 Evolution of RT

- Is RT Unix in our future?
  - Some traditional "RT Unix" systems are said to have high performance (LynxOS, Venix) [Brooks, D'Addario, Langston]
  - Any Unix-like OS (e.g. Solaris, Linux, AIX, etc) which claims to support "SMP" [Symmetric Multi-Processing] should be presumed to have a hard kernel with threads and semaphores and preemptive priority, and therefore to be capable of hard RT, until proven otherwise. This RT application concept combines the robustness of VxWorks 'hard RT' at high priority with the richness of a general-purpose networked Unix environment running at lower priority, all in the same hardware box under the same OS kernel (which might have multiple CPUs to speed up multiple threads). [Yodaiken]
  - There are now POSIX standards for RT features, and these appear to be in the process of becoming universal in Unix systems and in many"hard" RT kernels (like VxWorks), so that portable RT applications may eventually be practical. Is there any NRAO experience with POSIX Pthreads?
  - Maybe we could build our RT applications as device drivers, rather than Posix threads, under Linux? Would this be a prudent policy for NRAO? [Folkers]

- Four theses for debate:
  - Some RT systems have multiple CPUs today; almost all will be concurrent in the end.
  - Many RT systems are networked today, all will be in the end. Brandt,Hunt
  - Many user interfaces and multimedia systems are multi-threaded RT systems today, all will be in the end (ubiquitous RT). A good example is the Netscape Web browser, which is obviously multi-threaded. Cornwell
  - Ubiquitous GPS technology means distributed RT systems everywhere will know standard time.
- 'hard'-RT vs 'soft' – if underlying hardware systems all contain embedded controllers maybe most non-embedded RT will be 'soft' in the future. For example, the VLBA Correlator has more than 100 programmable embedded controllers which synchronize at sub-millisecond level, and the high-level RT system ticks at only 7.6 Hz and has opportunity to precompute and buffer many of its timecritical operations at lower priorities. Rowen, M.Clark, Cornwell

- Java may become a candidate for serious RT designs in the future; we should review this possibility during this workshop. See:
  - `www.chai.hp.com` ("..HP's ChaiVM is an independently developed small-footprint implementation compliant with the Java[tm] Virtual Machine Specification. It is targeted for embedded devices running real-time operating systems..")
  - `http://java.sun.com/products/embeddedjava/` and `http://java.sun.com/pr/1999/03/pr990301-03.html` ("IBM Leads Industry Experts to Define Java Technology Specifications for Real-Time Extensions")
  - `http://www.wrs.com/press/html/jworks.html` (Wind River Systems Introduces Personal JWorks and WindPower TurboJ)

  Scott

References

[LB98] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Pthreads*. Sun Microsystems (Prentice-Hall), Mountain View, CA, 1998. ISBN:0-13-680729-1, QA76.76.T55.L49.