Windows NT and Ardence/VenturCom RTX eXtension

Hugues Mounier
Université Paris Sud
Real time requirements

- Requirements that real-time systems have typically come to provide:
  - A multithreaded, preemptive scheduler with a large number (typically 64-256) of thread priorities.
  - Predictable thread synchronization mechanisms.
  - A system of priority inheritance.
  - Fast clocks and timers.
NT shortcomings

- Too few thread priorities.
- Opaque and non-deterministic scheduling decisions.
- Priority inversion, particularly in interrupt processing.
- Interrupt management
Needs for a real time extension

- Preempt NT anywhere, at least outside critical NT interrupt-processing code.
- Defer NT interrupts and faults while running real-time tasks.
- Process real-time interrupts while running real-time tasks.
RTX at a glance
What is Ardence RTX

- A real-time extension for windows NT
- Two schedulers: one for RTSS, one for NT
- RTSS sched preempts NT sched
- RTX provides:
  - Bounded system call response time
  - Larger thread priority space
  - Synchronisation mechanisms
  - Control of priority inversion
RTSS and Windows NT

- Real-time Subsystem is an extension to Windows NT
  - NT Kernel is unmodified
  - RTSS is loaded as a standard NT device driver
  - Uses NT Device Driver Interfaces and special purpose interfaces added to the Hardware Abstraction Layer (HAL)

- RTSS programs execute in the protected kernel space
Ardence RTX Architecture

Figure 1: RTX Architecture
RTSS as an NT driver

- Preempting high-level IRQ activity of NT and its drivers for unbounded periods of real-time activity may seem dangerous.

- Yet, NT is designed to handle them:
  - high-IRQL events intrude on lower-IRQL ones,
  - bus-mastering by DMA peripherals may defer even the highest-level interrupt processing,
  - PCI devices may stall CPU accesses to the I/O space (see RTX and Interrupt Latency).
RTSS as an NT driver

- Thus, from the NT point of view, RTSS activity that steals its cycles is equivalent to taking and coming back from an interrupt.
- Such an event is well handled by NT, regardless of its duration.
RTSS HAL

HAL with extensions for RTSS

- **Clock services**
  - Faster rates
  - Synchronization with NT clock

- **Interrupt services**
  - Isolation of RTSS and NT interrupts
  - Inter-environment interrupts

- RTSS thread switch support

- Inter-environment switching
RTAPI

- Real-time Application Programming Interface (RTAPI)

- Two types
  - New calls
    - Looks like Win32 calls, e.g. returns a HANDLE type
    - RtAttachInterruptVector
  - Similar to Win32
    - Accesses the RTSS implementation of a familiar Win32 object
    - RtCreateSemaphore
RTAPI

- Win32 Calls implemented in RTSS
  - CreateThread
  - Sleep
- All “Rt” interfaces are available in both environments
- All RTSS calls are deterministic (i.e. have bounded execution time),
- Except object creation calls
RTSS objects

- Available to Win32 programs for communication with RTSS processes
- Synchronization Objects
  - Semaphore
  - Mutex
- Shared Memory Object
- Common objects
  - Registry, File System objects
- Strictly local objects
  - Threads, objects derived from threads
Process and thread management
NT thread states

Thread States

- *Initialized*
- *Ready*
- *Standby*
- *Running*
- *Waiting*
- *Transition*
- *Terminated*
- *Unknown*
NT thread states

- *initialized*
- *transition*
- *ready*
  - Creation
  - Stacked into memory
- *standby*
  - Preemption or quantum expiration
  - Preemption
  - Execution
  - Waiting on an object or suspended
- *running*
  - Execution
  - Pending execution
  - Waiting completion
  - Preemption or quantum expiration
  - Termination
- *terminated*
- *unknown*
NT thread scheduling

- Threads scheduled without regard to processes
  - 32 Priority levels
  - 0 = lowest priority
- 31 = highest priority
- 32 ready queues
- Round-robin for highest-priority non-empty ready queue
NT thread priorities

- Dynamic threads: priority levels 0-15
- Real-time threads: priority levels 16-31

Process
  - Base priority class
  - Determines a narrow range for the base priorities of its threads

Threads
  - Base priority level
  - Determines the exact base priority within a priority class
  - Base priority class + Base priority level = base priority
NT thread priorities

Priority changes

- Only for dynamic threads
- Priority boosts
  - Exiting the wait state
  - Window receives user input
  - Has not executed for a long time
NT thread priorities

Priority changes

• Priority reduction
  – Cannot go below base priority
  – Reduced by one if thread executes for entire quantum
  – Priority returns to previous level after one quantum if thread boosted because had not executed for a long time
RTX Process and threads

- An RTX or Win32 process is launched by CreateProcess
- Either in Win32 environment (.exe file)
- Or in RTSS environment (.rtss file)
- The default stack is 8Ko
- The end of a process in Win32 environment made through ExitProcess
- The end of a process in RTX environment made through rtsskill
RTX thread scheduling

- Priority based preemptive policy with priority promotion to prevent priority inversion.
- The RTSS environment provides for 128 priority levels, numbered from 0 to 127, with 0 the lowest priority.
- The RTSS scheduler will always run the highest priority thread that is ready to run.
- In the case of multiple ready threads with the same priority, the thread which has been ready the longest will run first.
RTX thread scheduling

- RTX is a preemptive system
- An RTSS thread will run until
  - A higher priority ready thread preempts it
  - Or until it voluntarily relinquishes the processor through a Sleep(0)
  - by waiting on a synchronisation object
  - At the return of an interrupt of timer routine
- There is no time-slicing among ready threads at the same priority.
RTX thread scheduling

● Each priority has its own ready queue, a doubly linked list.

● Execution time of insertion (at the end of the list) and removal (from anywhere in the list) is independent of the number of threads on the list.

● A bit array keeps track of which lists are non-empty,

● manipulating this bit array is done by high-speed assembly-coded routines.
Context switch figures

- On a 1GHz Pentium
  - Between 2 RTX processes: 0.4 microsec
  - Between 2 RTX threads: 0.4 microsec
  - Between a Win32 thread and an RTX threads: 2 microsec
Priority Spectrum

Real-time time-critical
31

Real-time normal
26
25
24
23
22

Real-time idle
16

Dynamic classes

Nonreal-time/time-critical
15

High foreground
13

Normal foreground
9

Normal background
7

Low foreground/background
4

Non-real-time/Idle
1

Idle thread
0

System levels like input, cursor, cache flushing, file sys., drivers
RTX priority spectrum

- 128 priority levels for RTX threads (0 to 127)
- 0 is the lowest
- Win32RTX threads (those scheduled by NT but using RTX calls) start their execution in the REALTIME level
Priority inversion

- To deal with priority inversion, *priority promotion*
- For the duration of the time that a low priority thread owns an object for which a high priority thread is waiting,
- its effective priority is promoted to that of the high priority thread.
I/O and memory management
Standard I/O access with RTX

- Standard I/O access is done through Win32 API
- Knowing that NT's file manager is too slow for real time processing
- Solutions:
  - Rewrite your own driver with RTX calls :-(
  - Put real time data in a shared memory buffer with another process reading or writing on disk :-(

Beware! Shared buffer must be large enough to account for disk access time jitter :-0
NT's I/O port access

- NT's privilege levels:
  - PL = 3: user applications
  - PL = 2: OS extensions
  - PL = 1: System Services
  - PL = 0: Kernel

- Any access by a user application to an I/O port raises an exception on instructions IN, OUT, INS, OUTS, CLI, STI
I/O protection leverage

- The I/O space is automatically unlocked for RTX processes
- For Win32RTX processes, RTEnablePortIO and RTDisablePortIO unlocks and locks one or more I/O port
- To any Win32RTX process an I/O protection bitmap is associated
- Each bit of the bitmap corresponds to a port of the I/O space
  - If the bit is set, the I/O port is unlocked
  - If it is unset, the I/O port is masked
I/O APIs

- In addition to RTEnablePortIO and RTDisablePortIO,
- RtReadPortUchar, Ushort, Ulong
- RtWritePortUchar, Ushort, Ulong
NT's memory

- **System memory** : 2Gb
  - Accessed in privileged mode only
  - Non paged part for kernel & system calls
  - Paged part for system DLLs

- **Per process memory** : 2Gb
  - Accessed in privileged and user mode
  - Unique for each process
  - Paged to ensure virtuality of process addresses
Memory optimisation

- Page faults imply non deterministic delays
- One can lock Win32RTX memory parts to avoid page faults:
  - RtCommitLockHeap for the heap
  - RtCommitLockStack for the stack
  - RtLockProcess for all processes sections
  - RtLockKernel for kernel libraries
- RTX processes memory is automatically locked in physical pages
Kernel memory for RTX

- RTSS (around 100Kb) installs itself in the nonpages NT kernel space
- Dynamical allocations for RTX and Win32RTX processes are made in NT's paged system memory
- Beware! The more memory you take, the less is left for standard NT applications to run
Deterministic memory allocation

- RTX processes are in concurrency with NT drivers for memory allocation, a source of undeterminism ...

- If an RTX process needs large portions of memory (30% to 50% of the RAM), then

- Change the MAXMEM value in the boot.ini file, at the `[operating systems]` section (here for 12Mb)

  ```plaintext
  multi(0)disk(0)rdisk(0)partition(1)\WINNT="Microsoft Windows XXX"
  /MAXMEM=12
  ```

- The MAXMEM value must be > 8Mb (the minimum space for NT to run)
Timers and clocks
RTX HAL timer

- RTX's HAL has a timer based on the hardware timer (838.095344ns) upon which RTX timers are built.
- Its period can be initialized in the RTX settings window; the following values are legal:
  100, 200, 500, 1000 microseconds
- RTX applications use this timer to build their own timers.
- The expiration routine associated with a timer is a thread.
- The timer APIs are non-blocking.
RTX settings

- Watchdog
- HAL timer
- Quantum
- Process slots
RTX startup

StartRTX launches managers for:

- HAL extension
- RTX Server
- RTX Subsystem
RTX clocks

- Values given since January 1st 1600 at 12h through:
  - RtGetClockTime & RtSetClockTime
  - RtGetClockResolution
  - RtGetClockTimerPeriod

- Available clocks:
  - CLOCK_1 (or CLOCK_SYSTEM) with a ms resolution
  - CLOCK_2 (or CLOCK_FASTEST) with a 100ns resolution
Timer expirations

• Suppose:
  – A chosen 100 microsec HAL RTX timer
  – A timer period of 300 microsec (i.e. A routine executes every 300mus)
  – The routine's execution time is not fixed, having a 100mus average with peaks of 350mus

• Question: Is the event signaled by the timer memorised during the routine's execution?
Timer expirations

Answer:

- The event remains signaled and the thread is immediately relaunched as soon as it terminates.
- The expiration timer queue has depth 1:
  - If the routine's execution time is between 0 and 600 mus, no expiration is lost.
  - If the execution time is between 600 and 900 mus, one expiration is lost.
  - If the execution time is between 900 and 1200 mus, two expirations are lost, ...
HAL and Interrupts
Real time HAL

- RTX's modified HAL for
  - Adding interrupt isolation between NT and RTSS threads
  - To implement high-speed clocks and timers.
  - To implement shutdown handlers.
Windows NT ’s interrupt cycle

- An Interrupt signal send to the interrupt controller
- The controller informs the CPU of an IRQ
- The CPU ends the current instruction
- The CPU saves the current tasks context
- The CPU masks all Interrupts
- The CPU loads the ISR ’s context
- The CPU executes the ISR
- The ISR re-enables the interrupts
- The ISR ends
- The CPU resets the state of the preceding task and resumes its execution
Interrupt requests levels (IRQL)

- Levels are priorities for interrupts (0 is lowest)
  - 31 HIGH_LEVEL
    - Hardware error interrupt
  - 30 POWER_LEVEL
    - Power failure interrupt
  - 29 REQUEST_LEVEL
    - SMP interprocessor interrupt
  - 28 CLOCK_LEVEL
    - Clock interrupt
  - 12-27
    - PC's interrupts (IRQ) 0 to 15
  - 4-11
    - Reserved for kernel
  - 3 WAKE_LEVEL
    - Software debuggers interrupt
  - 2 DISPATCH_LEVEL
    - Dispatch routine execution
  - 1 APC_LEVEL
    - APC
  - 0 PASSIVE_LEVEL
    - System thread execution
Windows NT and interrupts

- Windows 2000/NT does not prioritize device IRQs in controllable way.
- User-level applications execute only when a processor’s IRQL is at passive level.
- System’s devices and device drivers – not the OS – ultimately determine the worst-case delay.
- This is a problem with off-the-shelf hardware and drivers.
NT interrupts & memory

- System designer must bound the length of device’s ISR and DPC in the worst case.
- Embedded versions of Windows NT/2000 provide control over memory footprint etc, but are not real-time capable.
- Extensions of real-time kernels can be provided through custom extensions of the HAL.
RTX & interrupt latencies

- NT drivers cannot mask RTSS ITs
  - A hook is set in the HAL to catch OUT instructions from the hardware interrupt controller
  - All NT's interrupts are masked while RTSS executes

- HAL modification to eliminate lengthy critical sections (duration > 5microsec)

- A utility detects bus monopolisation
Inter process communication and concurrency
Various IPCs

- Shared memory : in non paged memory
- Semaphores
- Mutex
- Events
- The objects can be shared through their names (a string)
- The object creation is estimated at under 10mus (on 1GHz pentium)
- The object manipulations (e.g. wait/release) is deterministic, bounded by 0.4mus (on 1GHz pentium)
Various IPCs

- **Semaphores**
  - RtCreateSemaphore, RtOpenSemaphore
  - RtWaitForSingleObject, RtReleaseSemaphore

- **Mutexes**
  - RtCreateMutex, RtOpenMutex
  - RtWaitForSingleObject, RtReleaseMutex

- **Shared memory**
  - RtCreateSharedMemory, RtOpenSharedMemory

- **Events**
  - RtCreateEvent, RtOpenEvent
  - RtPulseEvent, RtSetEvent, RtResetEvent
Semaphore example

```c
void mainfunction(void) {
    HANDLE hSem;
    hSem = RtCreateSemaphore(NULL, 0,
                              MAX_COUNT, "MySem");
    // initialize (other threads are waiting)
    RtReleaseSemaphore(hSem, MAX_COUNT, NULL);
    WaitForShutdownMessage();
    RtCloseHandle(hSem);
}
```
void otherProcFunction(void) {
    HANDLE hSem;
    hSem = RtOpenSemaphore(NULL,
                             FALSE,
                             "MySem");
    RtWaitForSingleObject(hSem, INFINITE);
    processData();
    RtReleaseSemaphore(hSem, 1, NULL);
    RtCloseHandle(hSem);
}
Event management

2 event types:

- Manual reset events: the thread is unlocked by RtWaitForSingleObject. Must execute RtResetEvent to reset the event in the non signaled state.
- Automatic reset events: the RtWaitForSingleObject function does an RtSetEvent.
- A thread can set an event in signaled state through RtSetEvent or RtPulseEvent.
Event management

- **Manual** reset case
  - \texttt{RtSetEvent} \hspace{1cm} \texttt{RtWait} \hspace{1cm} \texttt{RtResetEvent}
  - Non sig \hspace{1cm} \text{----------} \hspace{1cm} sig \hspace{1cm} \text{--------------------------} \hspace{1cm} \text{[ ]} \hspace{1cm} \text{[ ]} \hspace{1cm} Non sig

- **Automatic** reset case
  - \texttt{RtSetEvent} \hspace{1cm} \texttt{RtWait}
  - Non sig \hspace{1cm} \text{----------} \hspace{1cm} sig \hspace{1cm} \text{--------------------------} \hspace{1cm} \text{[ ]} \hspace{1cm} \text{[ ]} \hspace{1cm} Non sig

- **Manual** reset case
  - \texttt{RtPulseEvent}
  - Non sig \hspace{1cm} \text{[- ]} \hspace{1cm} sig \hspace{1cm} \text{[- ]} \hspace{1cm} \text{[ ]} \hspace{1cm} Non sig: unlocks \textit{all waiting threads}

- **Automatic** reset case
  - \texttt{RtPulseEvent}
  - Non sig \hspace{1cm} \text{[- ]} \hspace{1cm} sig \hspace{1cm} \text{[- ]} \hspace{1cm} \text{[ ]} \hspace{1cm} Non sig: unlocks \textit{first waiting thread}
Performance
### RTX Latencies

- **Typical latencies for a 1Ghz pentium**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency (μsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt thread: avg., max</td>
<td>2.4, 10</td>
</tr>
<tr>
<td>Timer interrupt: avg., max</td>
<td>2.6, 11</td>
</tr>
<tr>
<td>Context switch (yield): avg.</td>
<td>0.93</td>
</tr>
<tr>
<td>Context switch (semaphore): avg.</td>
<td>1.5</td>
</tr>
<tr>
<td>Thread priority change: avg.</td>
<td>0.79</td>
</tr>
<tr>
<td>Thread yield: avg.</td>
<td>0.79</td>
</tr>
<tr>
<td>Acquire semaphore, uncontested: avg.</td>
<td>0.39</td>
</tr>
<tr>
<td>Acquire semaphore: avg.</td>
<td>1.33</td>
</tr>
<tr>
<td>Win32-to-RTSS semaphore call: avg.</td>
<td>13.3</td>
</tr>
</tbody>
</table>
References
References

3. RTX VenturCom presentation slides