EECS 571 “Principles of Real-Time Embedded Systems”

Lecture Note #15:

RT extensions/applications of general-purpose OSs

General-Purpose OSs for Real-Time

- Why? (as discussed before)
  - App timing requirements are *not* hard
  - Costs less
  - Large user base
- How?
  - App developers should know sources of unpredictability
  - Use app SW architecture that minimizes use of problematic OS features.
  - Use sched and resource access-control strategies.
- Two most widely-used OSs: Windows NT and Linux

OS Requirements for Real-Time Apps

- Multi-threaded and preemptable
- Notion of thread priorities must exist
- Support for synchronization mechanisms
- Priority inheritance or ceiling must exist
- Predictable/fast interrupt latency

Windows NT

- Strengths: threads, priority interrupts, events, sufficiently-fine timer and clock resolution.
- Weaknesses:
  - Large memory footprint
  - Weak support for RT scheduling and resource-acces control
  - Unpredictable interrupt-handling and IPC mechanisms
- Approach: Avoid system services that can introduce unpredictable and prolonged blocking, and keep processor utilization low, and provide priority-inversion control at the user level.

Windows NT Scheduler

- Designed for good avg performance for time-shared apps
- Only 32 priority levels!
  - 0--15: variable-priority class for threads of time-shared apps. OS can boost priority and adjust sched quantum of a thread. (9 Idle, Normal, High classes.)
  - 16--32: real-time priority class. OS never adjusts (thus static) priorities of threads in this class
- Small # of priority levels) low schedulable processor untilization, why?
- Many kernel-mode threads run at real-time priority level 16) High-priority RT threads may delay system threads (for memory managers, file systems, etc.)

Jobs, Scheduling Classes and FIFO Policy

- NT 4.0 doesn't support FIFO with equal priority but NT 2K gives the FIFO option to choose.
- A job may contain multiple processes but each process can belong to only one job
- Can set (e.g., exec time or memory usage) limits of a job object that controls all of its processes. NT 2K terminates an offending job.
- NT 4.0 allows a user thread to specify only 7 out of 16 real-time priority levels, while NT 2K makes all 16 RT priority levels available to a user thread.
- NT 2K offers 9 job scheduling classes; class 9 is for RT applications.
- Do all processes of a job belong to the same scheduling class? Yes (no) if the sched class limit is enabled (disabled).

Resource Access Control

- NT doesn't support priority inheritance
- 9 2 ways at user level to (incompletely) control priority inversion in a uniprocessor.
  - Employ user-level non-preemptive critical section (NPCS). How can NPCS be achieved in NT 4.0 (supports RR for equal priority tasks) and NT 2K (supports FIFO for equal priority tasks)?
  - Use the ceiling priority protocol, i.e., a thread holding any resource executes at the highest priority ceiling of all the resources it holds. For NT 4.0, restrict RT threads to have even priority (16, 18,…, 30), why?

Interrupt Handling

- Two-level (split) interrupt handling
  - Interrupt Service Routine (ISR)
  - Deferred Procedure Call (DPC)
- ISRs can be preempted by higher priority ISRs
- DPCs queued and handled by the I/O Manager
- All DPCs run in FIFO order at the same priority lower than HW interrupt priorities but higher than that of scheduler/dispatcher.

) A higher-priority thread can be blocked by DPCs in response to the interrupts caused by lower-priority threads.
- DPCs *cannot be preempted* by other DPCs

Interrupt Event Flow

- Interrupt occurs
- Processor calls the dispatcher
- OS calls the ISR
- ISR does critical work
- Device driver calls a DPC function
- Exit ISR
- Pending DPC is scheduled
- After completion of all DPCs, user applications resume

Rehashing Problems with NT

- Very small number of real-time priority levels
- No priority inheritance, so priority inversion can occur
- Kernel threads are not preemptable by user threads
- Interrupt management can take an unbounded amount of time
- Scheduler is not fair when dealing with mouse, keyboard or display events
- Minor problems:
  - For embedded applications, memory footprint is huge
  - Licensing is expensive

Recommended Solutions

- DPCs should have many priority levels
- DPCs should be preemptable by higher priority DPCs
- Third party drivers should be configurable (ISR and DPC priority)
- Third party drivers should provide the maximum time it can take during a ISR or DPC
- System call times should be specified to the developer

Commercial Solutions

- Use NT as it is, but with care
- Implement a Win32 API on top of a RTOS
  - QNX using the Willow library
  - VxWorks using the Willow library

- Make NT and a RTOS (or part of it) coexist
  - Imagination Systems (HyperKernel)
  - Radisys (INtime with RT API)
  - Venturcom (RTX, KPX, RTAPI)

Implementation of Win32 API on Top of RTOS

- Advantages:
  - Portability between real-time and non-real time approaches
  - Small footprint
  - Real-time behavior from these RTOSs is known
- Disadvantages:
  - Standard NT applications cannot be used
  - No NT device drivers
  - NT graphic devices cannot be used
  - Limited expandability
  - Special tools for development and compilations

Make NT and a RTOS Coexist

- Possible approaches:
  - Modify the HAL by intercepting interrupts and including a small scheduler or RTOS
  - Run NT as one of the tasks on top of a RTOS
- Advantages:
  - Compatibility with NT is maintained
  - Protection for the RT tasks may be included in the RTOS
- Disadvantages:
  - Non-portability unless a RT-API is provided
  - Device drivers for NT cannot be used in the RT part (non-predictable timing guarantees)
  - Many task levels and context definitions may exist

Real-Time Networking

- Also want predictability in the network subsystem
- How to achieve it?
  - Quality of Service (QoS)
  - Admission control protocols (RSVP)
  - Priority packet scheduler

Current Implementations

- Windows NT 4.0 has no support for QoS
- Windows NT 2K has:
  - The RSVP protocol has been implemented
  - APIs for QoS and Admission Control are in place
- Now available:
  - Traffic control API (packet scheduler)
  - QoS API

Linux and Real-Time

There are several problems when using Linux for RT apps

- Interrupts are disabled by Linux subsystems when they are in CS
- Coarse timer resolution (10 ms)
- Time-sharing, dynamic priority adjustment
- Non-preemptable kernel but changed to "preemptable" in 2003 (2.2.16)
- Virtual memory.

Linux Scheduling

- Provides processes with choices among SCHED_FIFO, SCHED_RR, or SCHED_OTHER policies
- SCHED_FIFO and SCHED_RR are for RT processes scheduled with fixed priorities
- Processes with SCHED_OTHER are scheduled on a time-sharing basis with priorities lower than RT processes.
- Nine 100 priority levels, and one can determine max and min priorities of a schedling policy using *sched_get_priority_min()* and *sched_get_priority_max(),* and size of the time slices given to processes scheduled by RR using *sched_rr_get_interval().* One can also change these parameters.

Interrupt Handling in RT-Linux

Interrupts are first captured by the real-time executive

- If Linux kernel enables interrupts, interrupts are passed to Linux kernel which runs ISR.
- If Linux kernel disables interrupts, interrupts are queued in the real-time executive. When Linux re-enables interrupts, all pending interrupts are passed to Linux kernel, which runs ISR

  => interrupts are never disabled.

RT-Linux Applications

- RT-Linux applications consist of:
  - Hard RT component that consists of one or more real-time tasks
  - Non-real-time component that consists of one or more non-real-time Linux processes.
- Linux processes and real-time tasks communicate via *fifos* or *shared memory*
- Real-time tasks cannot access any system services such as X Window, networking, and disk. They can't make any syscalls.

KURT Linux

- Increased the timer resolution to μ sec level without significant interrupt overhead
- Modified Linux scheduler to include static plan-based scheduling
- Full set of system services available, but using them may result in scheduling distortion:
  - Some subsystems support real-time better than others
  - Disk subsystem is the largest source of distortion.

KURT Linux, cont'd

Suited for soft RT apps and consists of 2 parts:

- Core that takes care of scheduling real-time events
- Real-time modules (RTMods) which implement functionality of a specific real-time task. The only built-in real-time module is *Process RTMod*, which provides the user processes with syscalls for registering and unregistering KURT real-time processes, as well as suspending the calling process until next time it is to be scheduled

KURT Core

- Responsible for scheduling all real-time events
  - Adds timer for each event
  - Calls the appropriate RTMod when an event occurs
- Provides syscalls to
  - switch the kernel between real-time and normal mode
  - schedule real-time events.

KURT Scheduling

- Explicit, static plan-based scheduler:

  RT events (i.e., invocation of RT Mods) are specified in a schedule file and passed to KURT core before running

- Three modes of operation
  - Normal mode: no real-time features
  - Real-time sharing mode: both RT and non-RT processes are allowed to run
  - Real-time exclusive mode: only RT processes are allowed to run.

UTIME: Microsecond Resolution On-Demand Timers

Two approaches to increasing timer resolution in Linux

- *Increase timer interrupt rate*
  - Unnecessary overhead as interrupts need to be serviced even when no events are scheduled
  - Observation: There is a significant disparity between temporal resolution and frequency of events
- *Interrupt only when there are events scheduled*

UTIME, cont'd

- Timer chip is programmed in one-shot (instead of periodic) mode --- timer chip has to be re-programmed to generate the next interrupt
- Within the ISR, timer chip is re-programmed to interrupt just in time to service the next event
- Fake events are scheduled every 10 ms in absence of any scheduled events

UTIME in Detail

- Uses both HW clock and the Pentium timestamp counter
- Uses the Pentium timestamp counter to maintain the SW clock which is programmed to interrupt periodically at system boot
- During initialization, UTIME reads and stores the timestamp counter periodically and calculates the length of a *jiffy* (clock interrupt period) in terms of the number of timestamp cycles per jiffy and per second. It then calibrates the timestamp counter and reprograms the clock to run in one-shot mode.
- Whenever a timer interrupt occurs, the ISR first updates the SW clock based on the timestamp readings at current time and the previous timer interrupt. It then queues timer functions to be executed at current timer interrupt, finds the next timer expiration time from the timer queue, and sets the clock to interrupt at that time.
- Exec time of timer ISR in Linux with UTIME is several times larger than in the standard Linux.