Operating Systems Issues for Real-Time

- Timing, Scheduling Latencies, and Preemption (example: Linux)
- Scheduling Policies (example: Solaris)
- Device Driver Architectures for Real-Time (example: Windows)
- Integration of Hard Real-Time and General-Purpose OS Architectures (example: Windows / Linux)
Timing, Scheduling Latency, and Preemption
(Real-Time Performance of Linux)


OS Latency

Definition [OS Latency]

Let $T$ be a task belonging to a time-sensitive application that requires execution at time $t$, and let $t'$ be the time at which $T$ is actually scheduled; we define the OS latency experienced by $T$ as $L = t' - t$. 
Sources of OS Latency

- **Timer Resolution** ($L_{\text{timer}}$)
  - Timer are generally implemented using a periodic tick interrupt. A task that sleeps for an arbitrary amount of time can experience some timer resolution latency if its expected activation time is not on a tick boundary.

- **Scheduling Jitter** ($L_{\text{SJ}}$)
  - Task is not highest in scheduling queue.

- **Non-Preemptable Portions** ($L_{\text{NP}}$)
  - Latency can be caused by non-preemptable sections in kernel and in drivers. (e.g. ISRs, bottom halves, tasklets).

Timer Resolution

- Standard Linux timers are triggered by a periodic tick interrupt.
- On x86 machines it is generated by the Programmable Interval Timer (PIT) with period $T_{\text{tick}} = 10\text{ms}$.

- How about decreasing $T_{\text{tick}}$?

- High-resolution timers using aperiodic interrupt capabilities in modern APICs (Advanced Programmable Interrupt Controller).

- Timer resolution possible in range of 4–6musec.
Non-Preemptable Section Latency

- **Standard Linux:**
  - Monolithic structure of kernel.
  - Allows execution of at most one thread in kernel. This is achieved by disabling preemption when an execution flow enters the kernel, i.e., when an interrupt fires or when a system call is invoked.
  - Latency can be as large as 28ms.

- **Low-Latency Linux:**
  - Insert explicit preemption points (re-scheduling points) inside the kernel.
  - Implemented in RED Linux and Andrew Morton’s low-latency patch.

- **Preemptable Linux:**
  - To support full kernel preemptability, kernel data must be explicitly protected using mutexes or spinlocks.
  - Linux preemptable-kernel patch disables preemption only when spinlock is held.
  - Latency determined by maximum amount of time for which a spinlock is held plus maximum time taken by ISRs, bottom halves, and tasklets.

- **Preemptable Lock-Breaking Linux:**
  - Spinlocks are broken by releasing spinlocks at strategic points.

---

**Preemptable Lock Breaking: Example**

```c
void proc_xactlock count()
{
    spinlock_break(dcache_lock);
    struct dentry tmp;  /* dentry structure */
    struct dentry *dentry;
    spinlock_release(dcache_lock);
    /* Wait for lock to be acquired. */
    while (spinlock_is_locked(dcache_lock))
    {
        sleep();
    }
    spinlock_break(dcache_lock);
}
```

- This function reclaims cached dentry structures in `fs/dcache.c`.
- High-latency point.
- Why count iterations at all?

© R. Bettati
Test Programs

- Measuring $L_{\text{timer}}$:
  - Run test task on lightly loaded system, to avoid $L_{np}$.
  - Set up a periodic signal (using $\text{itimer}()$)

- Measuring $L_{np}$:
  - Run test task against background tasks
  - Test Task:
    - Read current time $t_1$
    - Sleep for a time $T$
    - Read time $t_2$, and compute $L_{np} = t_2 - (t_1 + T)$

- How to read $t_1$ and $t_2$? ($\text{gettimeofday}()$ ?)

Timer Latency

Figure 1. Inter-Activation times for a task that is woken up by a periodic signal with period $100\mu s$ on a high resolution timer Linux.

Figure 2. PDF of the difference between inter-activation times and period, when $T = 1000\mu s$. 

© R. Bettati
Test Programs

- Measuring $L_{\text{timer}}$:
  - Run test task on lightly loaded system, to avoid $L_{np}$.
  - Set up a periodic signal (using `itimer()`)  

- Measuring $L_{np}$:
  - Run test task against background tasks
  - Test Task:
    - Read current time $t_1$
    - Sleep for a time $T$
    - Read time $t_2$, and compute $L_{np} = t_2 - (t_1 + T)$
  - How to read $t_1$ and $t_2$? (gettimeofday()?)

Measuring $L_{np}$

- **Memory Stress**:
  - Page fault handler invoked repeatedly.
- **Console-Switch Stress**:
  - Console driver contains long non-preemptable paths.
- **I/O Stress**:
  - Systems calls that move large amounts of data between user and kernel space, or from kernel memory to hardware peripherals.
- **Procs Stress**:
  - Concurrent access to `/proc` file system must be protected by non-preemptable sections.
- **Fork Stress**:
  - New processes created inside non-preemptable section and requires copying of large amounts of data.
  - Overhead of scheduler increases as number of active processes increases.
Figure 3. OS non-preemptable section latency measured on a high-resolution timer Linux. This test is performed with heavy background load.

Background Load Tests

Standard Linux
Background Load Tests

Low-Latency Kernel

Preemptable Kernel

© R. Bettati
Background Load Tests

OS Non-Preemptable Portion Latency

<table>
<thead>
<tr>
<th></th>
<th>Memory Stress</th>
<th>I/O Stress</th>
<th>ProcFS Stress</th>
<th>Fork Stress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic</td>
<td>18212</td>
<td>6487</td>
<td>614</td>
<td>27596</td>
</tr>
<tr>
<td>Low-Latency</td>
<td>63</td>
<td>6831</td>
<td>686</td>
<td>38</td>
</tr>
<tr>
<td>Preemptable</td>
<td>17467</td>
<td>6912</td>
<td>213</td>
<td>187</td>
</tr>
<tr>
<td>Preemptable Lock-Breaking</td>
<td>54</td>
<td>6525</td>
<td>207</td>
<td>162</td>
</tr>
</tbody>
</table>

Table 1. OS non-preemptable section latencies (in μs) for different kernels under different loads (test run for 25 seconds).

<table>
<thead>
<tr>
<th></th>
<th>Memory Stress</th>
<th>I/O Stress</th>
<th>ProcFS Stress</th>
<th>Fork Stress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic</td>
<td>18956</td>
<td>28314</td>
<td>3563</td>
<td>617</td>
</tr>
<tr>
<td>Low-Latency</td>
<td>293</td>
<td>292</td>
<td>3379</td>
<td>596</td>
</tr>
<tr>
<td>Preemptable</td>
<td>18848</td>
<td>392</td>
<td>224</td>
<td>645</td>
</tr>
<tr>
<td>Preemptable Lock-Breaking</td>
<td>230</td>
<td>322</td>
<td>231</td>
<td>537</td>
</tr>
</tbody>
</table>

Table 2. OS non-preemptable section latencies (in μs) for different kernels under different loads (tests run for 10 hours).
Non-Preemptable Portion Latency

Figure 4. CDF of the latency measured on different versions of Linux (with high resolution timers). This test is performed with the I/O stress in background.

Latencies

Figure 5. Audio/Video Skew on standard Linux. Heavy kernel load is run in the background.

Figure 6. Audio/Video Skew for lock-breaking preemptable Linux with high resolution timers. Heavy kernel load is run in the background. The Audio/Video skew is clustered around 0, and the maximum skew is less than 400us (note that the scale is different from Figure 5).
Inter Frame Times

Figure 7. Inter-Frame times for standard Linux. Heavy kernel load is run in the background.

Figure 8. Inter-Frame times for lock-breaking preemptable Linux with high resolution timers. Heavy kernel load is run in the background.

Operating Systems Issues for Real-Time

- Timing, Scheduling Latencies, and Preemption (example: Linux)
- Scheduling Policies (example: Solaris)
- Device Driver Architectures for Real-Time (example: Windows)
- Integration of Hard Real-Time and General-Purpose OS Architectures (example: Windows / Linux)
(Some) Real-Time Operating Systems Issues

- (Some, random) Issues with Real-Time OSs

- Problems with the design of general-purpose real-time capable OS: Solaris


  “SVR4 UNIX Scheduler Unacceptable for Multimedia Applications.” NOSSDAV ‘93.

  URL: http://www.cs.columbia.edu/~nieh/#publications

So, You want to make your OS Real-Time?!

- Making general-purpose OS real-time capable:
  - Scheduling of tasks in kernel should be deterministic. Kernel should be free from unbounded priority inversion.
  - Deterministic dispatch latency.
  - Allow for mixed-mode applications: real-time and non-real-time components.
  - Appropriate for multiprocessor machines.
  - Provide standard interface to user, such as POSIX.
Kernel Dispatch Latency

- Historically: unbounded dispatch latency caused by non-preemptible kernel.
  - Solution 1: Well-defined preemption points. (?)
  - Solution 2: Fully synchronize access by kernel code to kernel data structures.
    - Reduces set of non-preemptible portions in kernel.
    - Kernel is multithreaded.

more about this later...

Scheduling Classes

- **Time-Sharing** class:
  - round robin scheduling.
- **Sys** class:
  - fixed priority scheduling,
  - not accessible by the user.
- **Real-Time** class:
  - fixed priority scheduling.
- `priocntl(2)`
  - Change scheduling class or other scheduling parameters.
Priority Inversion

- Priority inversion happens due to
  - non-preemptable portions
  - access to synchronization objects
  - “hidden scheduling”
- Synchronization Objects (mutex, r/w locks)
  - Solution: basic priority-inheritance protocol
- Hidden Scheduling
  - Work done asynchronously in kernel on behalf of threads without regard to their priority.
  - Example: streams processing
  - Example: timeouts done at lowest interrupt level
  - Solution: Move this code into kernel threads running at sys priority level.

Priority Inheritance

- Primitives:
  - pi_willto(thread) impose priority of argument thread onto all threads that block if, directly or indirectly
  - pi_waive() release priority inheritance
- The function pi_willto() is called after the thread has been put to sleep in the queue associated with the synchronization object. The information about the synchronization object can therefore be recovered.
Priority Inheritance and R/W Locks

- Priority inheritance for readers/writers locks:
  - when writer owns the lock: no problem
  - when readers own the lock:
    • potentially many “owners”; not practical to keep pointer from resource to every thread that owns it
    • Solution: define a single “owner-of-record”, which is only thread that inherits priority.

Applicability of SunOS 5.0 for Multimedia Applications

- Objectives of real-time OS for general-purpose workstations
  - Provide real-time guarantees without reducing general capabilities of workstations
  - Manage resources so that other applications can operate correctly.
  - SunOS 5.0 (SVR4) provides real-time static-priority scheduler.

- Question: How well are resources managed?
Experimental Evaluation: Overview

- Platform
  - Sun Sparc10
  - Solaris 2.2
  - Scheduling classes (RT class, TS class, SYS class)

- Experiment (measurement) criteria:
  - Interactive:
    - minimize average and variance between user input and response
    - Typing, cursor motion, mouse selection <= 50 - 150 ms.
  - Continuous media:
    - Minimize difference between average display rate and desired display rate.
    - Minimize variance of display rate.
  - Batch:
    - "Minimize difference between actual time of completion and minimum time of completion when whole machine is dedicated."

Experiment: Workload

- 3 classes of workload

- Interactive: (editors, GUIs)
  - TYPING: Emulate a user typing, and display characters on the screen.

- Continuous media: (television, teleconference)
  - VIDEO: Capture data from digitizer board and display through x-windows server.

- Batch: (compilations, scientific computation)
  - make: Repeatedly fork and wait for small processes to complete.

- Instrumentation of application and system software components does not measurably change the performance.
Experiment: The Baseline

<table>
<thead>
<tr>
<th>Application</th>
<th>Measurement</th>
<th>Mean</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typing</td>
<td>Latency between character arrival and rendering</td>
<td>38.5</td>
<td>15.7</td>
</tr>
<tr>
<td></td>
<td>to frame buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Video</td>
<td>Time between display of successive frames</td>
<td>112</td>
<td>9.75</td>
</tr>
<tr>
<td>Compute</td>
<td>Time to execute one loop iteration</td>
<td>149</td>
<td>6.79</td>
</tr>
</tbody>
</table>

Table: Application Baseline Values

- What is a well-behaved system?
  - Concurrent applications should make some progress
  - No case where system fails to respond to operator input
  - User should exercise wide range of influence over system behavior.

Experiment 1: Run all tasks in TS class

- Window system is no longer accepting input events from mouse or keyboard.
- Command interpreter not permitted to run.
- System blocked by batch-job
  - Identified as I/O intensive interactive job.
  - Gets priority boosts for sleeping.
- Window server develops backlog of service requests. As it works down its queue, it gets identified as compute bound.

- Table entries are relative to baseline (tall is better)
- T: TYPING character latency
- V: time between display of successive frames for VIDEO.
- C: time for one iteration in COMPUTE.
What can the System Administrator do?

- Increase priority of X-Server, decrease priority of batch task.
- In addition, decrease priority of VIDEO a bit.
- Decrease priority of VIDEO a little bit more.

Play with RT Class

- Video in RT.
- X-server in RT.
- Video and X-server in RT, P(V)-P(X).
- Video and X-server in RT, P(X)-P(V).
Result: New TS Class

- Removes anomalies of identifying batch jobs as interactive and vice versa.
- Ensures that each process makes steady progress.
- Reduces feedback interval
- Included in Solaris 2.3.

Operating Systems Issues for Real-Time

- Timing, Scheduling Latencies, and Preemption (example: Linux)
- Scheduling Policies (example: Solaris)
- Device Driver Architectures for Real-Time (example: Windows)
- Integration of Hard Real-Time and General-Purpose OS Architectures (example: Windows / Linux)
Windows NT Family and Real-Time?

- Reading: “Inside Microsoft Windows 7”, (Solomon, Russinovich, Microsoft Programming Series)

- “Real-Time Systems and Microsoft Windows NT” (MSDN Library)

Priority Levels vs. Interrupt Levels

- The HAL maps hardware-interrupt numbers to IRQLs.
- IRQLs are not the same as IRQs in x86.
- Scheduling priority is attribute of thread, while IRQL is attribute of an interrupt source.
- Lazy IRQL management for slow PICs.
- Code running at DPC/dispatch level or above can’t wait on object if so would necessitate scheduler to invoke another thread.

Thread Priorities 0–31

Hardware Interrupts
- 31: High
- 30: Power Fail
- 29: Inter-Processor Interrupt
- 28: Clock
- 27: Profile
- 26: Device n

Software Interrupts
- 3: Device 1
- 2: DPC/dispatch
- 1: APC
- 0: Passive

IO System Components (Windows 2k)

Applications
Win32 services
WMI service
user-mode
PnP manager
setup
components
user mode
I/O system
I/O manager
kernel mode

WDM WMI routines
PnP manager
Power manager

drivers

HAL

© R. Bettati
Device Driver Layering

1. Write file (file handle, char_buffer)
2. Write data at specified byte offset within a file
3. Translate file-relative byte offset into a disk-relative byte offset and call next driver (via I/O manager)
4. Call driver to write data at disk-relative byte offset
5. Translate disk-relative byte offset into physical location and transfer data
**Primary Device Driver Routines**

- NT/2000 device drivers run entirely within the system process and have access to all hardware through the HAL. A typical device driver will have several components:
  - **Initialization routine** This routine initializes hardware and sets up data structures used by the driver at startup time.
  - **Interrupt service routine (ISR)** This routine handles an interrupt on the device that the device driver controls.
  - **Deferred processing call (DPC)** One or more DPCs handle non-time-critical processing for the driver.
  - **System thread** Some, but not all, drivers will have a system thread for very low-priority work.

**Control Flow for an IO Operation**

1. **Application**
   - Call `ReadFile()`
   - Call `NTReadFile()`
   - `return to caller`

2. **User mode**
   - `NTReadFile`
   - `INT 2E return to caller`
   - `System Service`

3. **Kernel mode**
   - `Ntoskrnl.exe`
   - `Driver.sys`
   - `Ntoskrnl.exe`

Whether to wait depends on "overlapped" flag.
Queueing and Completing a Synchronous Request

1. I/O request passes through subsystem DLL
2. NewFile/Read file: char_buffer
3. Create IRP and send it to device driver
4. Transfer data specified in IRP
5. Handle interrupt and return success or error status
6. Complete IRP and return success or error status

Servicing a Device Interrupt (only Phase I)

1. The device interrupts for service.
2. The device interrupt dispatcher transfers control to the device's service routine.
3. The ISR stores the device interrupt and queues a DPC.
Servicing a Device Interrupt (Phase II)

1. The DPC routine starts the next I/O request in the device queue and then completes interrupt servicing.
2. The interrupt dispatcher transfers control to the driver's DPC routine.
3. The IRQ signal and DPC processing paths.
4. The APC routine signal DPC queue.

Completing an I/O Request (Phase I)

1. The I/O manager queues an APC to complete the I/O request in the caller's context.
2. The DPC routine calls the I/O manager to complete the original I/O request.
Completing an I/O Request (Phase II)

Memory Management

- **Paging I/O** occurs at a lower priority level than the real-time priority process levels. Paging within the real-time process is still free to occur, but this really ensures that background virtual memory management won’t interfere with processing at real-time priorities.

- Windows NT permits an application to **lock** itself into memory so that it is not affected by paging within its own process. This allows even very large processes (such as raster image processing, where some processes are over 100MB) to lock all their memory down into physical memory and avoid the overhead of paging, while allowing the rest of the system to function normally.

- Windows NT memory management allows for **memory mapping**, which permits multiple processes—even device drivers and user applications—to **share the same physical memory**. This results in very fast data transfers between cooperating processes or between a driver and an application. Memory mapping can be used to dramatically enhance real-time performance.
Windows NT/2000/XP/… and Real-Time Processing

- Windows NT/2000/XP/… 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.
- System designer must bound the length of device’s ISR and DPC in the worst case.
- Embedded versions of Windows NT/2000/XP… 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.

Operating Systems Issues for Real-Time

- Timing, Scheduling Latencies, and Preemption (example: Linux)
- Scheduling Policies (example: Solaris)
- Device Driver Architectures for Real-Time (example: Windows)
- Integration of Hard Real-Time and General-Purpose OS Architectures (example: Windows / Linux)
Real-Time Executives - Example:
VenturCom RTX Architecture

Figure 1: RTX Architecture