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)

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)

© R. Bettati

Timing, Scheduling Latency, and Preemption
(Real-Time Performance of Linux)

- **Among others: "A Measurement-Based Analysis of the Real-Time Performance of Linux"** (L. Abeni , A. Goel, C. Krasic, J. Snow, J. Walpole) [RTAS 2002]

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^{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^{SJ})
  - Task is not highest in scheduling queue.
- **Non-Preemptable Portions** (L^{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^{tick} = 10ms.
- How about decreasing T^{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 max. amound 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

```
void prune_dcache(int count)
{
        spin_lock(&dcache_lock);
        for (;;) {
            struct dentry *dentry;
            struct list_head *tmp;

            tmp = dentry_unused.prev;

             if (tmp == &dentry_unused)
                    break;
            list_del_init(tmp);
            dentry = list_entry(tmp, struct dentry, d_lru);

            /* If the dentry was recently referenced, don't free it. */
            if (dentry->d_vfs_flags & DCACHE_REFERENCED) {
                    dentry->d_vfs_flags &= ~DCACHE_REFERENCED;
                    list_add(&dentry->d_lru, &dentry_unused);
                    continue;
            }
            dentry_stat.nr_unused--;

            /* Unused dentry with a count? */
            if (atomic_read(&dentry->d_count))
                    BUG();

            prune_one_dentry(dentry);
            if (!--count)
                    break;
        }
        spin_unlock(&dcache_lock);
}
```

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

```
void prune_dcache(int count)
{
        DEFINE_RESCHED_COUNT;

redo:
        spin_lock(&dcache_lock);
        for (;;) {
                struct dentry *dentry;
                struct list_head *tmp;

                if (TEST_RESCHED_COUNT(100)) {
                    RESET_RESCHED_COUNT();
                    if (conditional_schedule_needed()) {
                            spin_unlock(&dcache_lock);
                            unconditional_schedule();
                            goto redo;
                    }
                }

                tmp = dentry_unused.prev;

                if (tmp == &dentry_unused)
                    break;
                list_del_init(tmp);
                dentry = list_entry(tmp, struct dentry, d_lru);

                /* If the dentry was recently referenced, don't free it. */
                if (dentry->d_vfs_flags & DCACHE_REFERENCED) {
                    dentry->d_vfs_flags &= ~DCACHE_REFERENCED;
                    list_add(&dentry->d_lru, &dentry_unused);
                    continue;
                }
                dentry_stat.nr_unused--;

                /* Unused dentry with a count? */
                if (atomic_read(&dentry->d_count))
                    BUG();

                prune_one_dentry(dentry);
                if (!--count)
                    break;
        }
        spin_unlock(&dcache_lock);
}
```

Test Programs

- Measuring L^{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()` ?)

Timer Latency

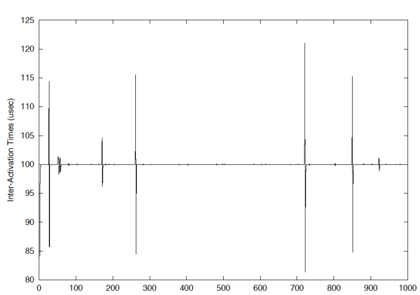


Figure 1. Inter-Activation times for a task that is woken up by a periodic signal with period $100\mu s$ on a high resolution timer Linux.

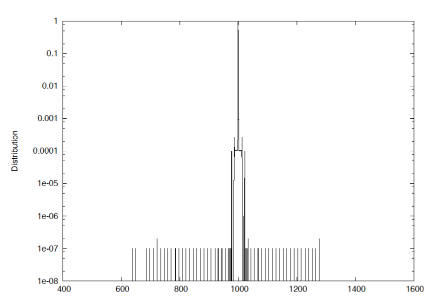


Figure 2. PDF of the difference between inter-activation times and period, when $T = 1000\mu s$.

Test Programs

- Measuring L^{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.
- **Procfs 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.

OS Non-Preemptable Section Latency

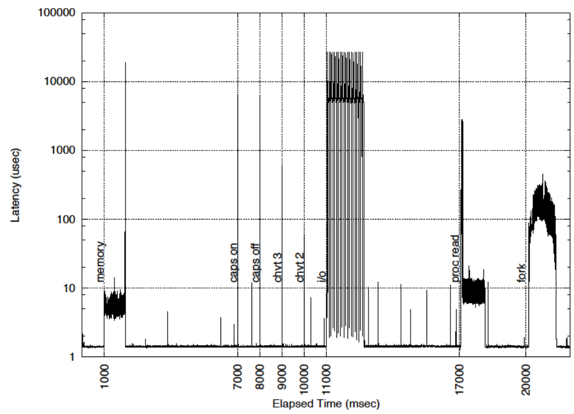
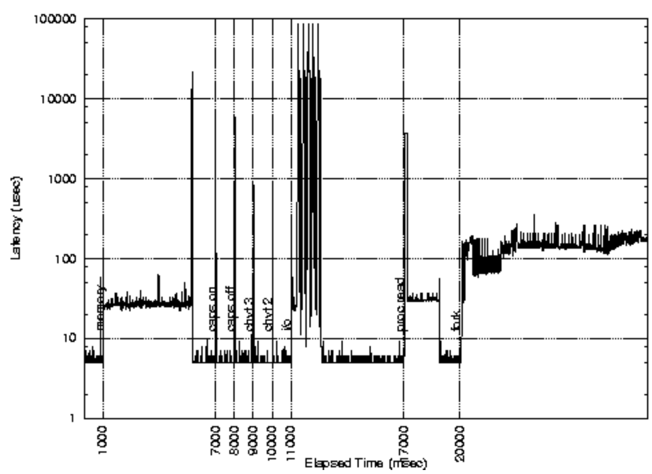


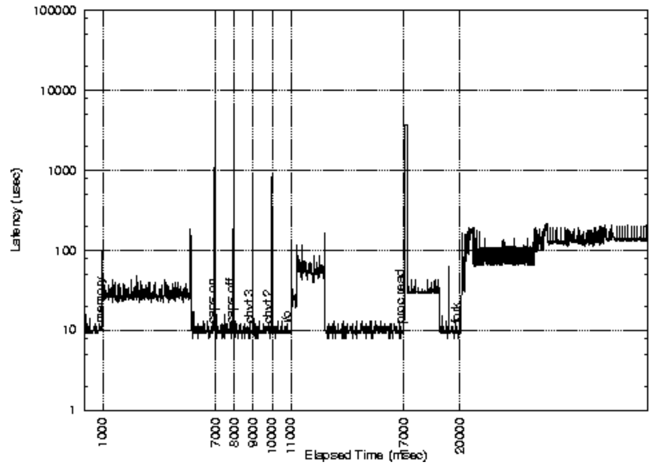
Figure 3. OS non-preemptable section latency measured on a high-resolution timer Linux. This test is performed with heavy background load.

Background Load Tests



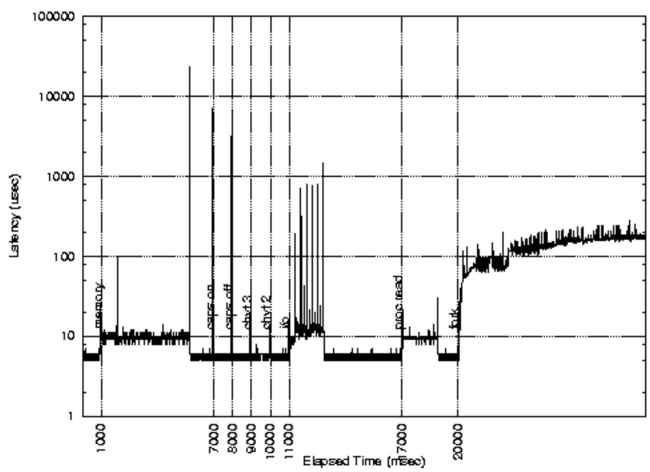
Standard Linux

Background Load Tests



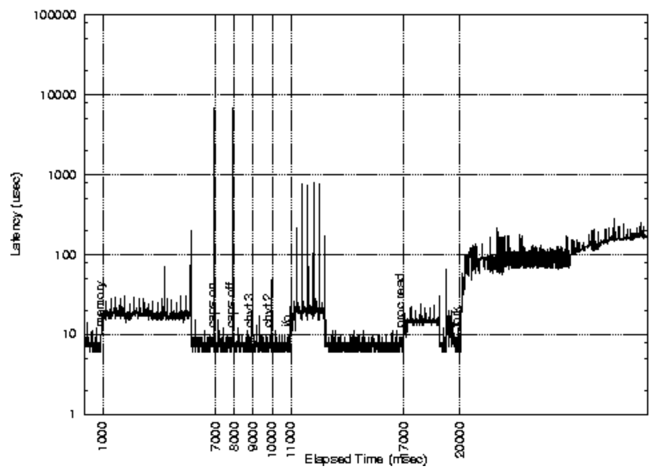
Low-Latency Kernel

Background Load Tests



Preemptable Kernel

Background Load Tests



Lock-Breaking Preemptable Kernel

OS Non-Preemptable Portion Latency

	Memory Stress	Caps-Lock Caps-Lock	Console Switch	I/O Stress	Procfs Stress	Fork Stress
Monolithic	18212	6487	614	27596	3084	295
Low-Latency	63	6831	686	38	2904	332
Preemptable	17467	6912	213	187	31	329
Preemptable Lock-Breaking	54	6525	207	162	24	314

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

	Memory Stress	I/O Stress	ProcFS Stress	Fork Stress
Monolithic	18956	28314	3563	617
Low-Latency	293	292	3379	596
Preemptable	18848	392	224	645
Preemptable Lock-Breaking	239	322	231	537

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

Non-Preemptable Portion Latency

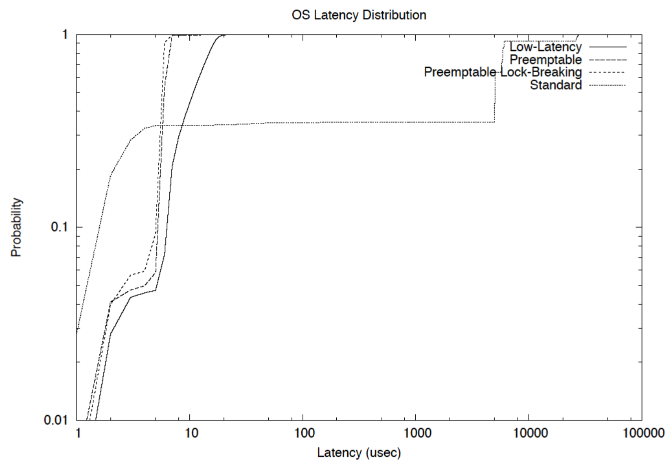


Figure 4. CDF of the latency measured on different versions of Linux (with high resolution timers). This test is performed with the I/O stress in background.

Latencies

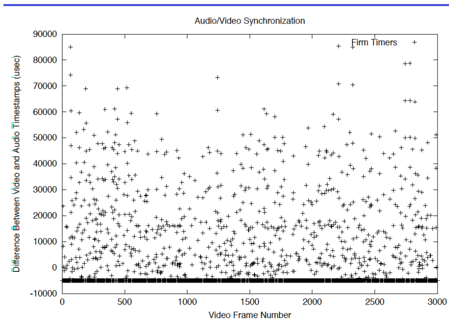


Figure 5. Audio/Video Skew on standard Linux. Heavy kernel load is run in the background.

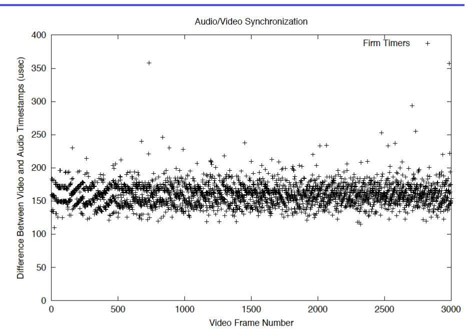


Figure 6. Audio/Video Skew for lock-breaking preemptable Linux with high resolution timers. Heavy kernel load is run in the background. The Audio/Video skew is clustered around 0, and the maximum skew is less than $400us$ (note that the scale is different from Figure 5).

Inter Frame Times

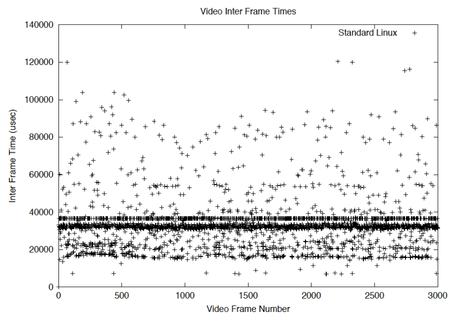


Figure 7. Inter-Frame times for standard Linux. Heavy kernel load is run in the back-ground.

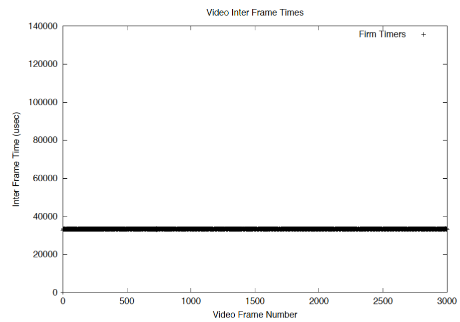


Figure 8. Inter-Frame times for lock-breaking preemptable Linux with high resolution timers. Heavy kernel load is run in the back-ground.

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

  J.Nieh, J.G.Hanko, J.D. Northcutt, G.A.Wall.

  "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.

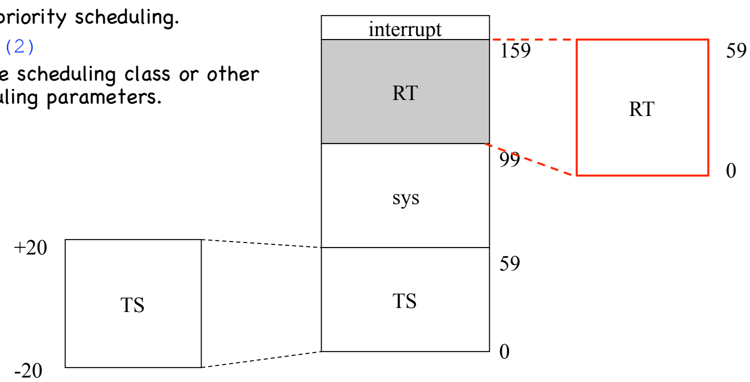
Kernel Dispatch Latency

- 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…

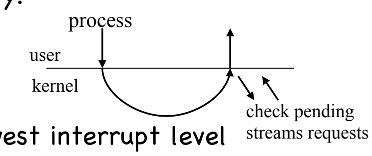
Scheduling Classes

- *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 it, 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

J.Nieh, J.G.Hanko, J.D. Northcutt, G.A.Wall.
"SVR4 UNIX Scheduler Unacceptable for Multimedia Applications." NOSSDAV '93.
URL: `http://www.cs.columbia.edu/~nieh/#publications`

- 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.

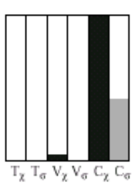
Experiment: The Baseline

Application	Measurement	Mean	Std. Dev.
Typing	Latency between character arrival and rendering to frame buffer	38.5 ms	15.7 ms
Video	Time between display of successive frames	112 ms	9.75 ms
Compute	Time to execute one loop iteration	149 ms	6.79 ms

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

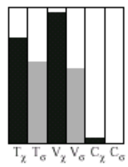


a.) All in SVR4 TS

- 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.

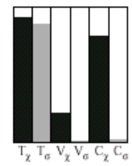
What can the System Administrator do?



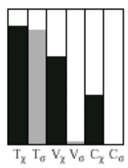
b.) SVR4 TS, Nice
(X+20,C-20)

Increase priority of X-Server, decrease priority of batch task

In addition, decrease priority of VIDEO a bit



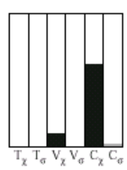
c.) SVR4 TS, Nice
(X+20,V-5,C-20)



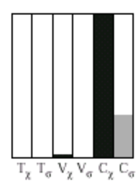
d.) SVR4 TS, Nice
(X+20,V-10,C-20)

Decrease priority of VIDEO a little bit more.

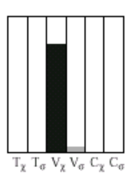
Play with RT Class



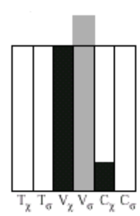
e.) Video in RT



f.) X-server in RT

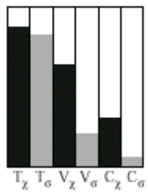


g.) Video and X-server
in RT, P(V)>P(X)



h.) Video and X-server
in RT, P(X)>P(V)

Result: New TS Class



i.) All in New Time-sharing 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.

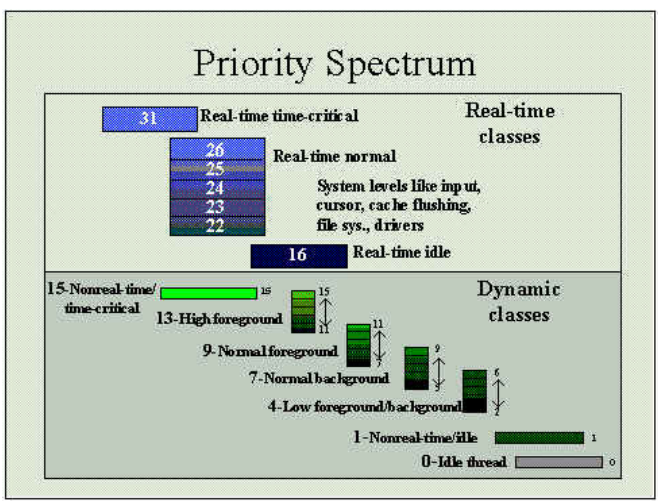
Operating Systems Issues for Real-Time

- 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?

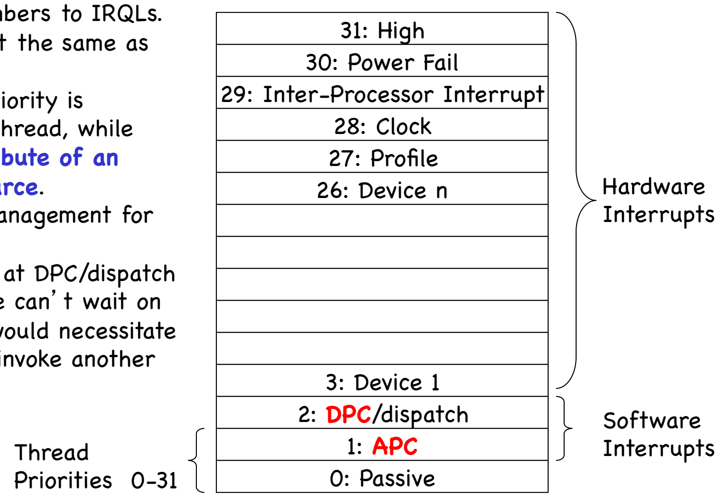
- Reading: **"Inside Microsoft Windows 7",** (Solomon, Russinovich, Microsoft Programming Series)
- **"Real-Time Systems and Microsoft Windows NT"** (MSDN Library)
- **"Windows XP with RTX - The off-the-shelf platform for Integrated Communication Equipment"** (www.venturcom.com)

Priorities in Windows NT/2000/XP/7/8/...

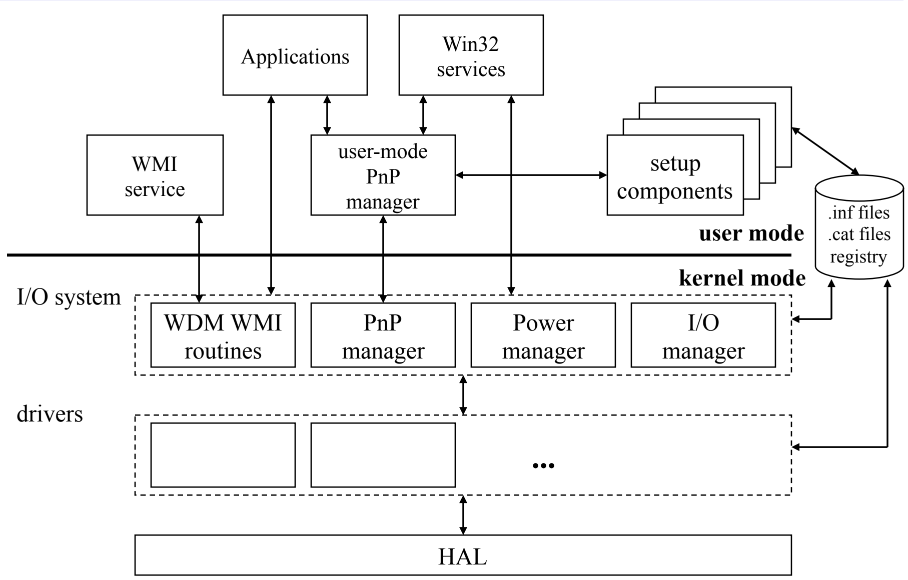


Priority Levels vs. Interrupt Levels

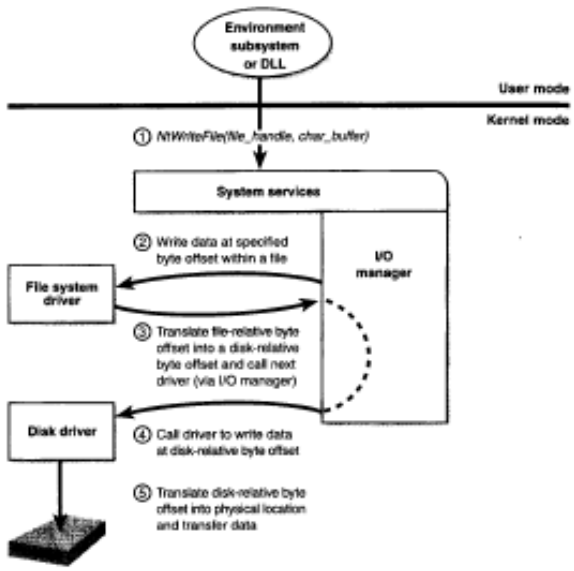
- The HAL maps hardware-interrupt numbers to IRQLs.
- IRQLs are not the same as IRQs in x86.
- Scheduling priority is attribute of thread, while **IRQL is attribute of an interrupt source**.
- Lazy IRQL management for slow PICs.
- Code running at DPC/dispatch level or above can't wait on object if so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



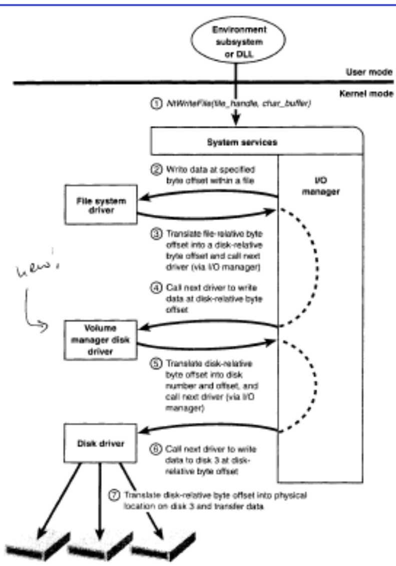
IO System Components (Windows 2k)



Device Driver Layering

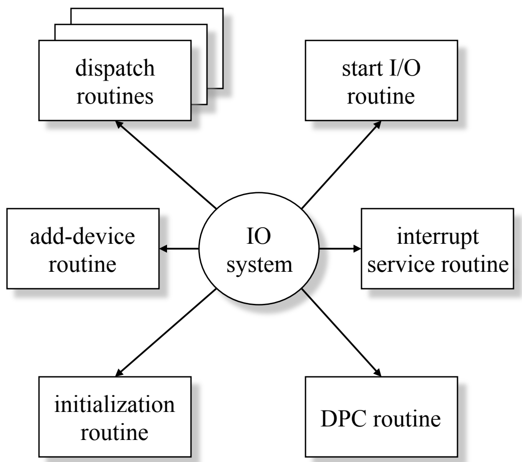


Device Driver Layering (2)



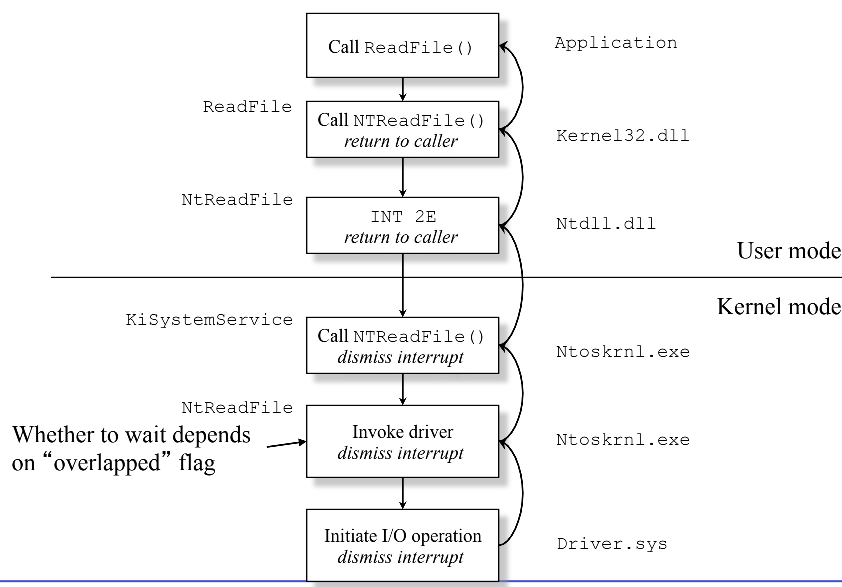
Primary Device Driver Routines

- 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:

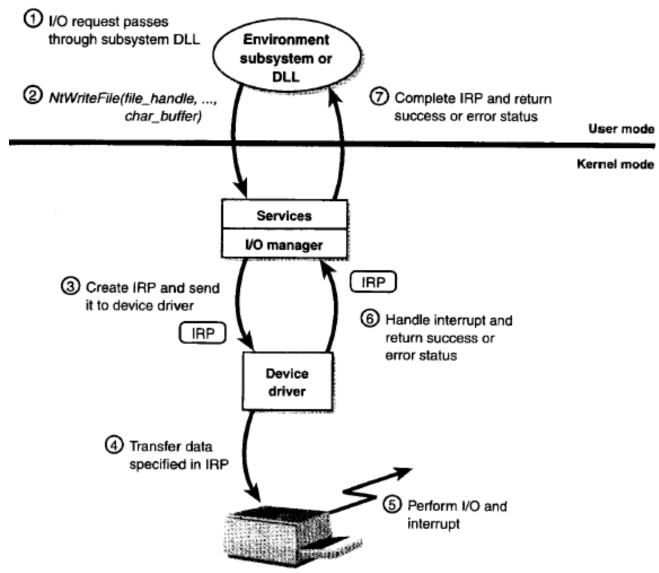


- Initialization routine This routine initializes hardware and sets up data structures used by the driver at startup time.
- Interrupt service routine (ISR) This routine handles an interrupt on the device that the device driver controls.
- Deferred processing call (DPC) One or more DPCs handle non-time-critical processing for the driver.
- System thread Some, but not all, drivers will have a system thread for very low- priority work.

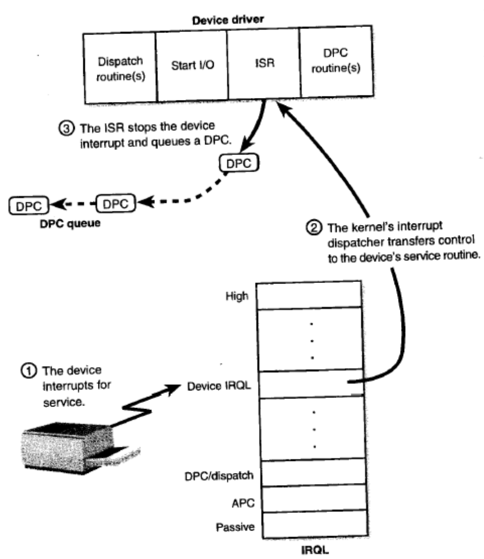
Control Flow for an IO Operation



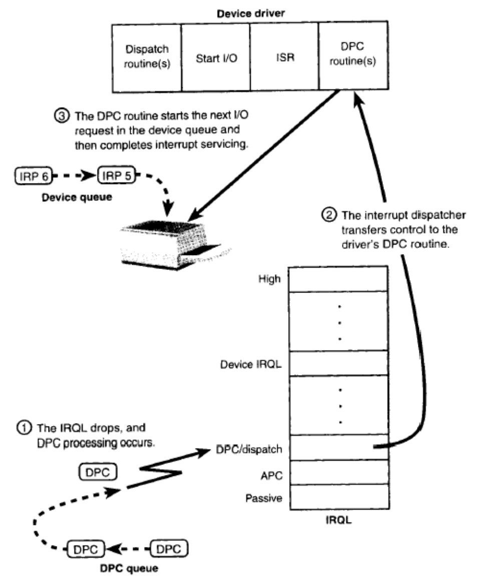
Queueing and Completing a Synchronous Request



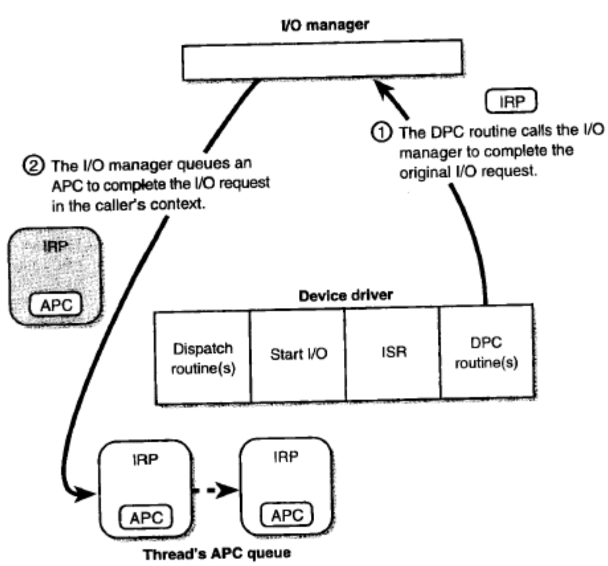
Servicing a Device Interrupt (only Phase I)



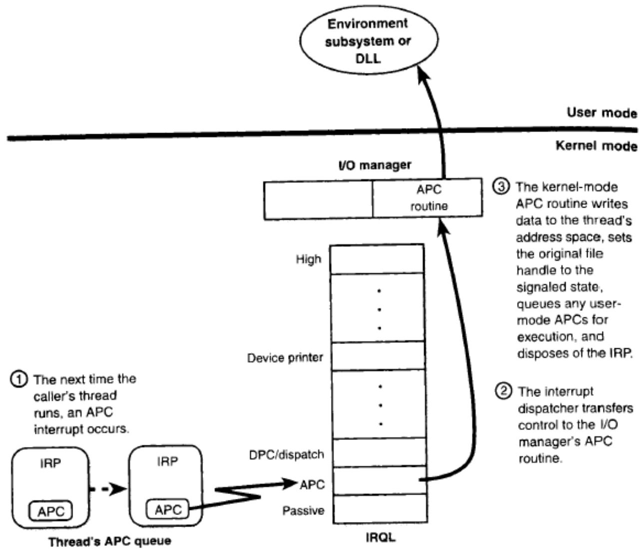
Servicing a Device Interrupt (Phase II)



Completing an I/O Request (Phase I)



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

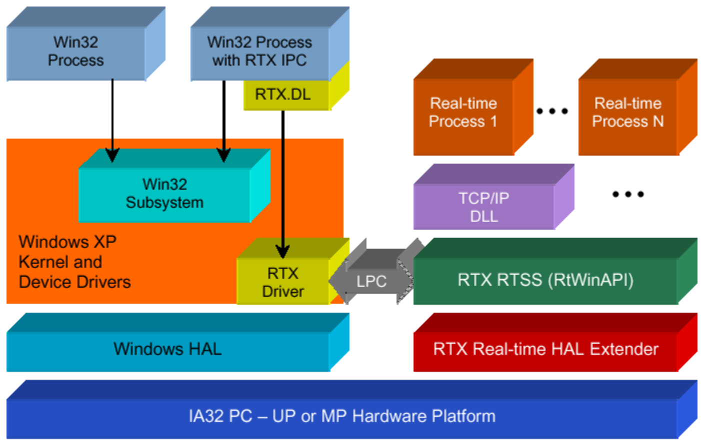


Figure 1: RTX Architecture