11. Kernel Design

- Interrupts and Exceptions
- Low-Level Synchronization
- Low-Level Input/Output
- Devices and Driver Model
- File Systems and Persistent Storage
- Memory Management
- Process Management and Scheduling
- Operating System Trends
- Alternative Operating System Designs

Bottom-Up Exploration of Kernel Internals

Hardware Support and Interface

- Asynchronous events, switching to kernel mode
- I/O, synchronization, low-level driver model

Operating System Abstractions

- File systems, memory management
- Processes and threads

Specific Features and Design Choices

- Linux 2.6 kernel
- Other UNIXes (Solaris, MacOS), Windows XP and real-time systems

11. Kernel Design

- Interrupts and Exceptions
- Low-Level Synchronization
- Low-Level Input/Output
- Devices and Driver Model
- File Systems and Persistent Storage
- Memory Management
- Process Management and Scheduling
- Operating System Trends
- Alternative Operating System Designs

Hardware Support: Interrupts

- Typical case: electrical signal asserted by external device
  - Filtered or issued by the *chipset*
  - Lowest level hardware synchronization mechanism
- Multiple priority levels: Interrupt ReQuests (IRQ)
  - Non-Maskable Interrupts (NMI)
- Processor switches to kernel mode and calls specific *interrupt service routine* (or *interrupt handler*)
- Multiple drivers may share a single IRQ line
  $\longrightarrow$ IRQ handler must identify the source of the interrupt to call the proper service routine

Hardware Support: Exceptions

- Typical case: unexpected program behavior
  - Filtered or issued by the *chipset*
  - Lowest level of OS/application interaction
- Processor switches to kernel mode and calls specific *exception service routine* (or *exception handler*)
- Mechanism to implement *system calls*

11. Kernel Design

- Interrupts and Exceptions
- Low-Level Synchronization
- Low-Level Input/Output
- Devices and Driver Model
- File Systems and Persistent Storage
- Memory Management
- Process Management and Scheduling
- Operating System Trends
- Alternative Operating System Designs

Hardware Interface: Kernel Locking Mechanisms

Low-Level Mutual Exclusion Variants

- Very short critical sections
  - Spin-lock: active loop polling a memory location
- Fine grain
  - Read/write lock: traditional read/write semaphore
  - Seqlock: high-priority to writes, speculative (restartable) readers
  - Read-copy update (RCU) synchronization: zero/low-overhead concurrent readers, concurrent writers in special cases
- Coarse grain
  - Disable preemption and interrupts
  - The "big kernel lock"
    - Non scalable on parallel architectures
    - Only for very short periods of time
    - Now mostly in legacy drivers and in the virtual file system

Hardware Interface: Spin-Lock

Example

- Busy waiting

```
do {
  while (lock == 1) { pause_for_a_few_cycles(); }
  atomic { if (lock == 0) { lock = 1; break; } }
} while (lock == 0);
// Critical section
lock = 0;
// Non-critical section
```

Applications

- Wait for short periods, typically less than $\mathbf{1}\,\mu\mathbf{s}$
  - As a proxy for other locks
  - As a *polling* mechanism
  - Mutual exclusion in interrupts
- Longer periods would be wasteful of computing resources

Beyond Locks: Read-Copy Update (RCU)

Principles

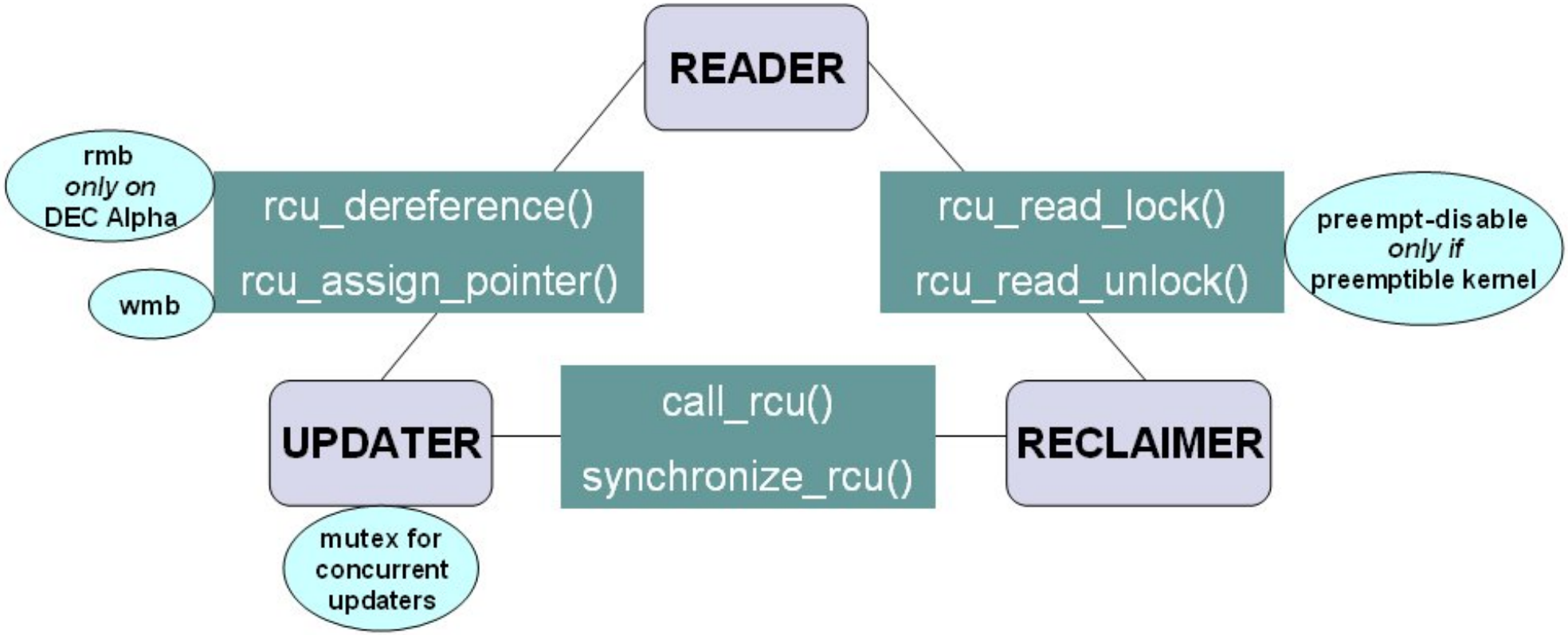
- Synchronization mechanism to improve scalability and efficiency
- RCU supports concurrency between a *single updater and multiple readers*
- Reads are kept atomic by maintaining multiple versions of objects — *privatization* — and ensuring that they are not freed up until all pre-existing read-side critical sections complete
- In non-preemptible kernels, RCU's read-side primitives have zero overhead
- Mechanisms:
  1. Publish-Subscribe mechanism (for insertion)
  2. Wait for pre-existing RCU readers to complete (for deletion)
  3. Maintain multiple versions of recently updated objects (for readers)

Beyond Locks: More About RCU

Programming Interface

- `rcu_read_lock()` and `rcu_read_unlock()`: delimit a RCU "read-side critical" section; important for kernel premption
- `rcu_assign_pointer()`: assign a new value to an RCU-protected pointer, with the proper fences (memory barriers)
- `rcu_dereference()`: return a pointer that may be safely dereferenced (i.e., pointing to a consistent data structure)
- `synchronize_rcu()`: blocks until all current read-side critical sections have completed, but authorize new read-side critical sections to start and finish

Beyond Locks: More About RCU



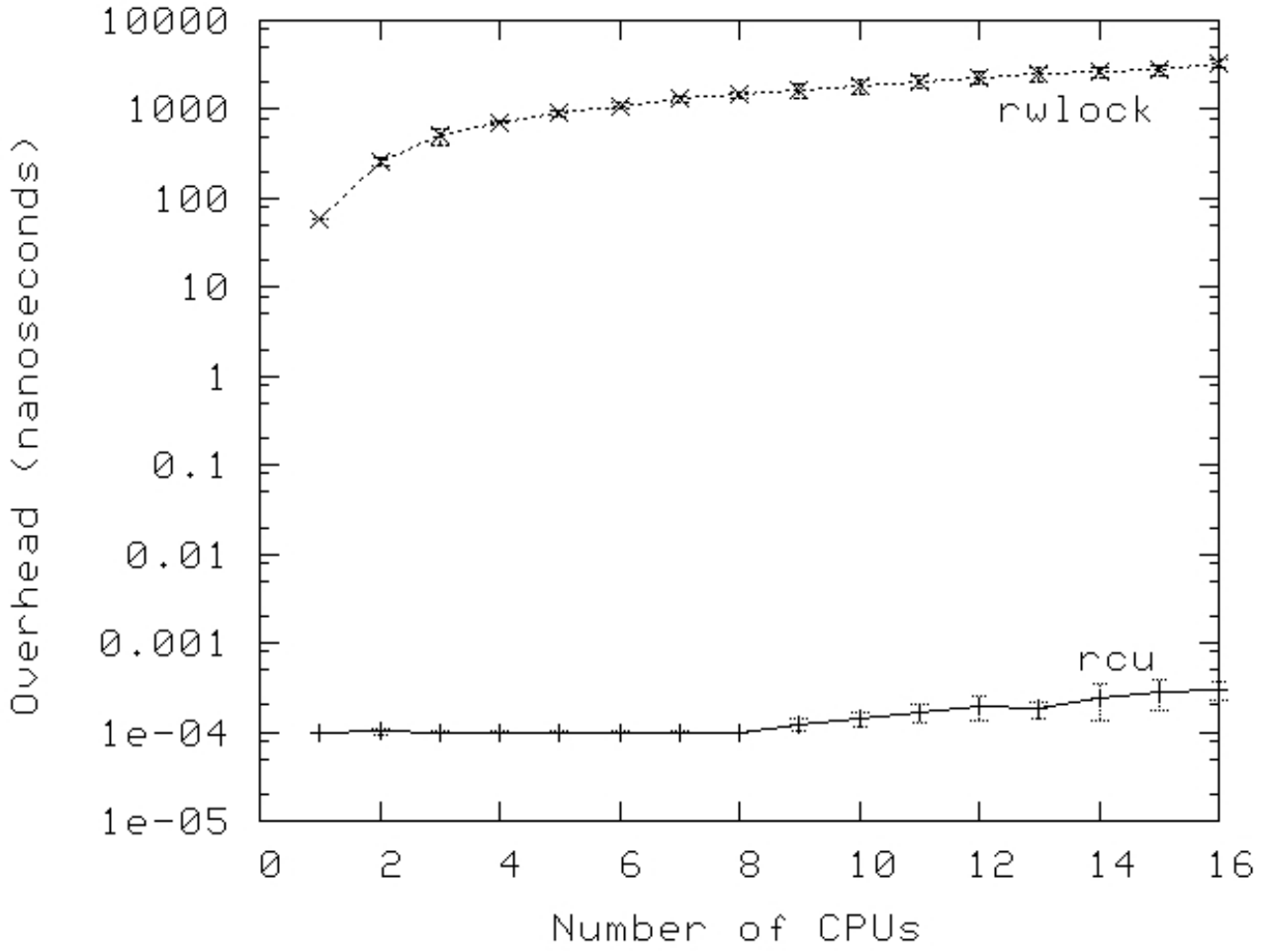
Beyond Locks: More About RCU

Toy but Correct Implementation on Non-Preemptible Kernels

```
void rcu_read_lock(void) { }
void rcu_read_unlock(void) { }
void synchronize_rcu(void) {
  /* force wait when kernel is not preemptible */
  attempt_to_switch_context_on_all_cpus();
}
#define rcu_assign_pointer(p, v) ({ \
    smp_wmb(); \
    (p) = (v); \
  })
#define rcu_fetch_pointer(p) ({ \
    typeof(p) _pointer_value = (p); \
    smp_rmb(); /* not needed on all architectures */ \
    (_pointer_value); \
  })
```

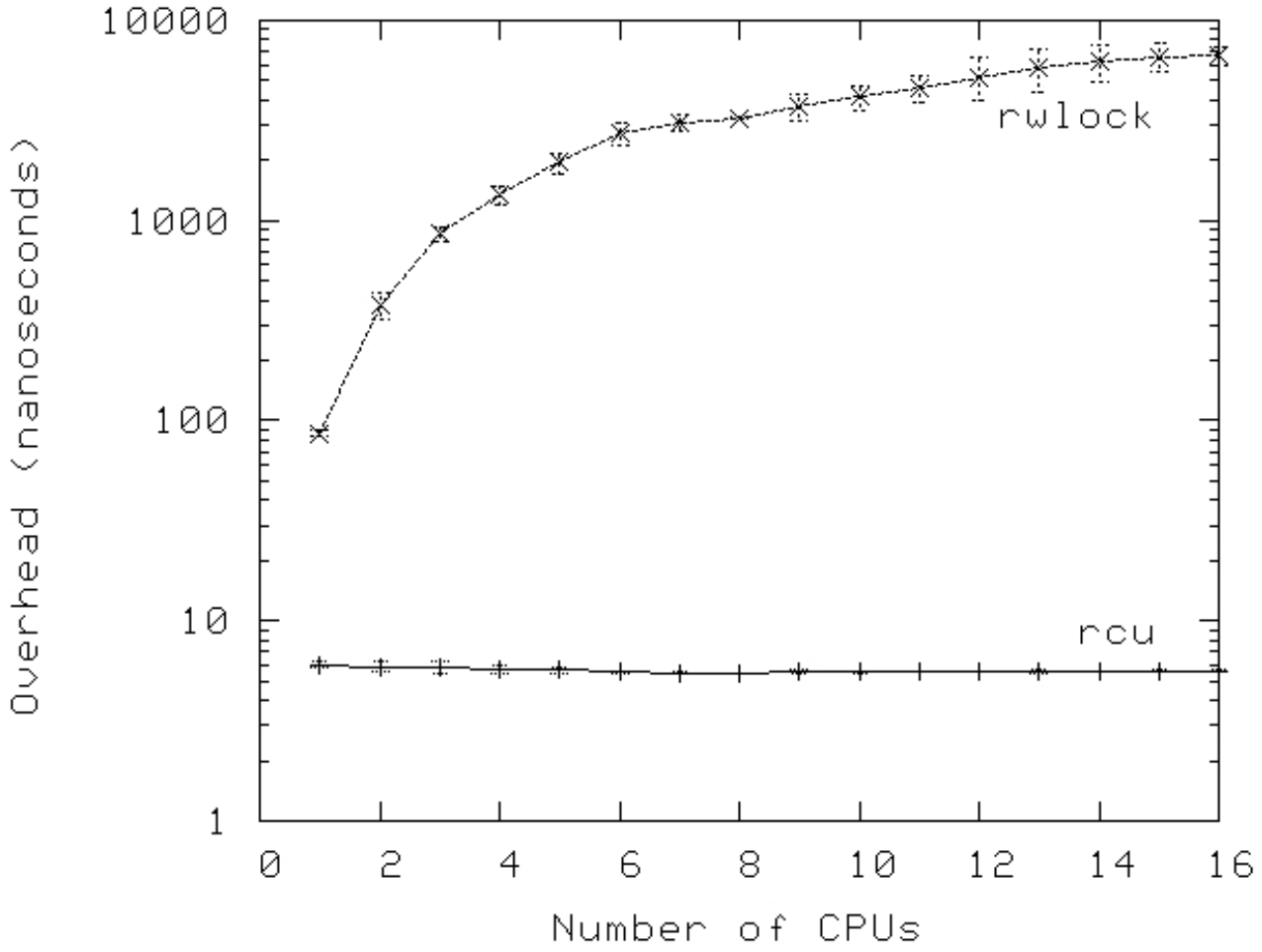
Beyond Locks: More About RCU

Overhead of RCU on *non-preemptible* 16-CPU Intel x86 at 3GHz



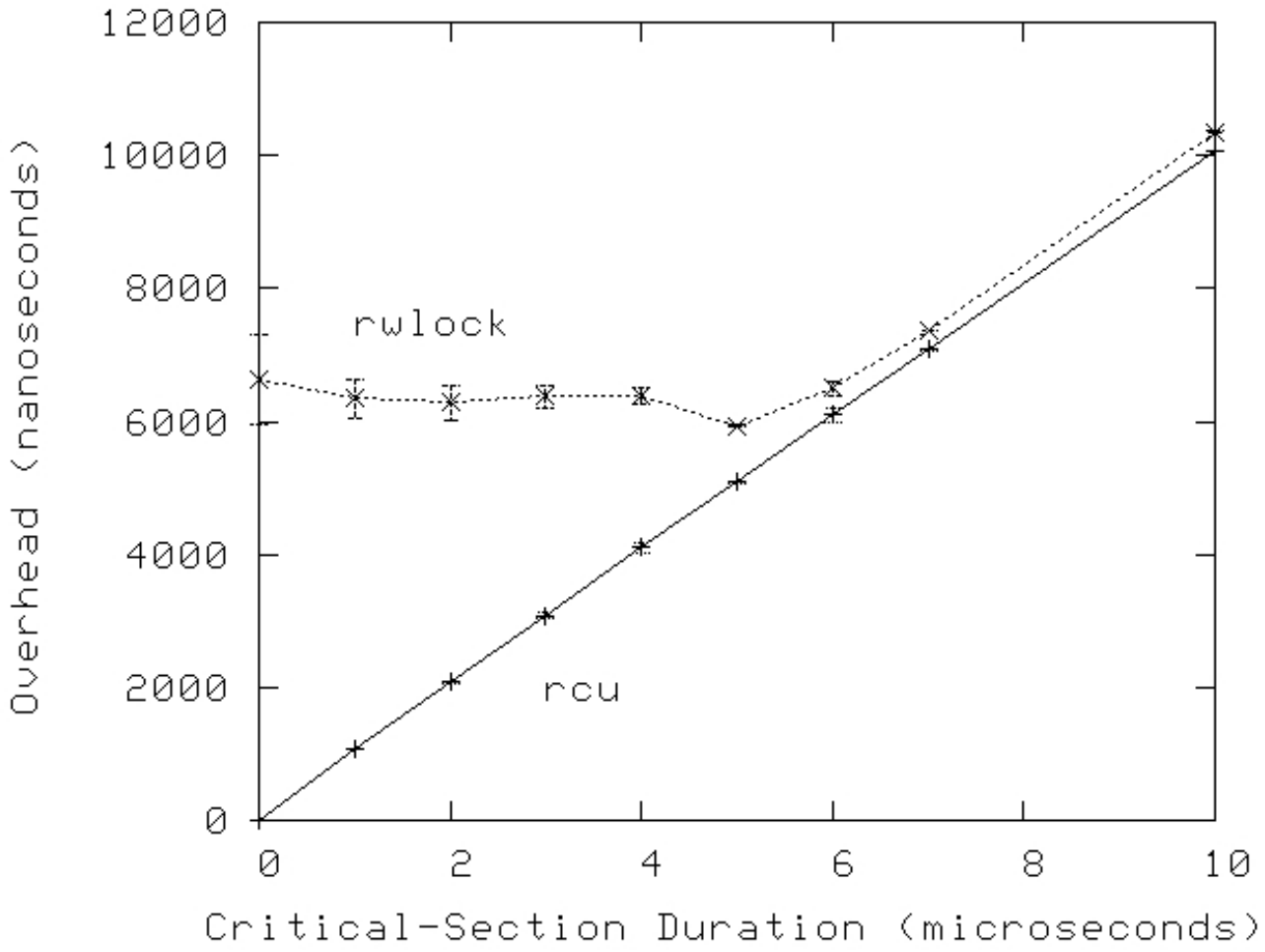
Beyond Locks: More About RCU

Overhead of RCU on *preemptible* 16-CPU Intel x86 at 3GHz



Beyond Locks: More About RCU

Total execution time of rwlock and RCU vs execution time in the critical section(s)



Beyond Locks: More About RCU

Online Resources

From the RCU wizard: Paul McKenney, IBM

http://www.rdrop.com/users/paulmck/RCU

11. Kernel Design

- Interrupts and Exceptions
- Low-Level Synchronization
- Low-Level Input/Output
- Devices and Driver Model
- File Systems and Persistent Storage
- Memory Management
- Process Management and Scheduling
- Operating System Trends
- Alternative Operating System Designs

Hardware Support: Memory-Mapped I/O

External Remapping of Memory Addresses

- Builds on the chipset rather than on the MMU
  - Address translation + redirection to device memory or registers
- Unified mechanism to
  - Transfer data: just load/store values from/to a memory location
  - Operate the device: reading/writing through specific memory addresses actually sends a command to a device
    Example: *strobe* registers (writing anything triggers an event)
- Supports Direct Memory Access (DMA) block transfers
  - Operated by the DMA controller, not the processor
  - Choose between *coherent* (a.k.a. synchronous) or *streaming* (a.k.a. non-coherent or asynchronous) DMA mapping

Hardware Support: Port I/O

Old-Fashioned Alternative

- Old interface for x86 and IBM PC architecture
- Rarely supported by modern processor instruction sets
- Low-performance (ordered memory accesses, no DMA)

11. Kernel Design

- Interrupts and Exceptions
- Low-Level Synchronization
- Low-Level Input/Output
- Devices and Driver Model
- File Systems and Persistent Storage
- Memory Management
- Process Management and Scheduling
- Operating System Trends
- Alternative Operating System Designs

Hardware Interface: Device Drivers

Overview

- Abstracted by system calls or kernel processes
- Manage buffering between device and local buffer
- Control devices through memory-mapped I/O (or I/O ports)
- Devices trigger interrupts (end of request, buffer full, etc.)
- Many concurrency challenges (precise synchronization required)
- Multiple layers for portability and reactivity

Hardware Interface: Driver Model in Linux

Low-Level Device Driver

- Automatic configuration: "plug'n'play"
  - Memory mapping
  - Interrupts (IRQ)
- Automatic configuration of device mappings
  - *Device numbers: kernel anchor for driver interaction*
  - Automatic assignment of *major* and *minor* numbers
    - At *discovery-time*: when a driver recognizes the signature of a device (e.g., PCI number)
    - At boot-time or plug-time
  - Hot pluggable devices

Hardware Interface: Driver Model in Linux

Device Special Files

- *Block*-oriented device
  Disks, file systems: `/dev/hda` `/dev/sdb2` `/dev/md1`
- *Character*-oriented device
  Serial ports, console terminals, audio: `/dev/tty0` `/dev/pts/0` `/dev/usb/hiddev0` `/dev/mixer` `/dev/null`
- *Major* and *minor* numbers to (logically) project device drivers to device special files

Hardware Interface: Driver Model in Linux

Low-Level Statistics and Management

- Generic device abstraction: `proc` and `sysfs` pseudo file systems
  - Class (`/sys/class`)
  - Module (parameters, symbols, etc.)
  - Resource management (memory mapping, interrupts, etc.)
  - Bus interface (PCI: `$ lspci`)
  - Power management (sleep modes, battery status, etc.)

Block Device

```
$ cat /sys/class/scsi_device/0:0:0:0/device/block:sda/dev
8:0

$ cat /sys/class/scsi_device/0:0:0:0/device/block:sda/sda3/dev
8:3
```

Hardware Interface: Driver Model in Linux

Kernel Objects and Events

- Main concept: `kobject`
  - Abstraction for devices, drivers, temporary structures, etc.
  - Representation (path) in `sysfs`
  - Type, parent pointer (hierarchy), reference count (garbage collection)
  - Ability to send *uevent*s to publish the state of the kernel object
  - Define which of these *uevent*s are exported to userspace, e.g., to be monitored by low-level daemons
- One application: automatic device node creation: `udev`
  - Userspace tools: `man udev`, `udevd` daemon, `udevadm` command
  - `udevadm info --export-db`
  - `udevadm monitor`
  - ...

Hardware Interface: Driver Model in Linux

Device Driver Examples

- *Device name: application anchor to interact with the driver*
- User level
- Reconfigurable rules
- Hot pluggable devices

Block Device

```
$ cat /sys/class/scsi_device/0:0:0:0/device/uevent
DEVTYPE=scsi_device
DRIVER=sd
PHYSDEVBUS=scsi
PHYSDEVDRIVER=sd
MODALIAS=scsi:t-0x00
$ cat /sys/class/scsi_device/0:0:0:0/device/block:sda/dev
8:0
$ cat /sys/class/scsi_device/0:0:0:0/device/block:sda/sda3/dev
8:3
```

Hardware Interface: Driver Model in Linux

Device Driver Examples

- *Device name: application anchor to interact with the driver*
- User level
- Reconfigurable rules
- Hot pluggable devices

Network Interface

```
$ cat /sys/class/net/eth0/uevent
PHYSDEVPATH=/devices/pci0000:00/0000:00:1c.2/0000:09:00.0
PHYSDEVBUS=pci
PHYSDEVDRIVER=tg3
INTERFACE=eth0
IFINDEX=2
```

Driver Model: Concurrency Challenges

Cost of Abstraction and Concurrency

- Complex kernel control paths

Typical Kernel Control Path: Swap Memory

1. Page fault of user application
2. Exception, switch to kernel mode
3. Lookup for cause of exception, detect access to swapped memory
4. Look for name of swap device (multiple swap devices possible)
5. Call non-blocking kernel I/O operation
6. Retrieve device major and minor numbers
7. Forward call to the driver
8. Retrieve page (possibly swapping another out)
9. Update the kernel and process's page table
10. Switch back to user mode and proceed

Concurrency Challenges

Concurrent Execution of Kernel Control Path

- Modern kernels are multi-threaded for reactivity and performance
  - Other processes
  - Other kernel control paths (interrupts, preemptive kernel)
  - Deferred interrupts (softirq/tasklet mechanism)
  - Real-time deadlines: timers, buffer overflows (e.g., CDROM)
- Shared-memory parallel architectures
  - Amdahl's law: minimize time spent in critical sections
  - Parallel execution of non-conflicting I/O

11. Kernel Design

- Interrupts and Exceptions
- Low-Level Synchronization
- Low-Level Input/Output
- Devices and Driver Model
- File Systems and Persistent Storage
- Memory Management
- Process Management and Scheduling
- Operating System Trends
- Alternative Operating System Designs

File Systems

Virtual File System

- Mounting multiple file systems under a common tree
  - `$ man mount`
- Superset API for the features found in modern file systems
  - Software layer below POSIX I/O system calls
  - Full support of UNIX file systems
  - Integration of pseudo file systems: `/proc`, `/sys`, `/dev`, `/dev/shm`, etc.
  - Support foreign and legacy file systems: FAT, NTFS, ISO9660, etc.

Modern File Systems: EXT3 and NTFS

Features

- Transparent defragmentation
- Unbounded file name and size
- Sparse bitmap blocks (avoid waste of resources)
- Support large disks and minimize down-time with *journaling*
  - Maximal protection: support logging of all data and meta-data blocks
  - Minimal overhead: logging of meta-data blocks only (traditional method on SGI's XFS and IBM's JFS)
- Atomic (transactional) file operations
- Access control Lists (ACL)

Modern File Systems: EXT3 and NTFS

Features

- Transparent defragmentation
- Unbounded file name and size
- Sparse bitmap blocks (avoid waste of resources)
- Support large disks and minimize down-time with *journaling*
  - Maximal protection: support logging of all data and meta-data blocks
  - Minimal overhead: logging of meta-data blocks only (traditional method on SGI's XFS and IBM's JFS)
- Atomic (transactional) file operations
- Access control Lists (ACL)

Notes About Linux EXT3

- Compatible with EXT2
- Journalization through a specific block device
- Use a hidden file for the log records

Modern File Systems: EXT3 and NTFS

Features

- Transparent defragmentation
- Unbounded file name and size
- Sparse bitmap blocks (avoid waste of resources)
- Support large disks and minimize down-time with *journaling*
  - Maximal protection: support logging of all data and meta-data blocks
  - Minimal overhead: logging of meta-data blocks only (traditional method on SGI's XFS and IBM's JFS)
- Atomic (transactional) file operations
- Access control Lists (ACL)

Notes About Windows NTFS

- Optimization for small files: "resident" data
- Direct integration of compression and encryption

Disk Operation

Disk Structure

- Plates, tracks, cylinders, sectors
- Multiple R/W heads
- Quantitative analysis
  - Moderate peak bandwidth in continuous data transfers
    E.g., up to 3Gb/s on SATA (Serial ATA), 6Gb/s on SAS (Serial Attached SCSI)
    Plus a read (and possibly write) cache in DRAM memory
  - Very high latency when moving to another track/cylinder
    A few milliseconds on average, slightly faster on SAS

Request Handling Algorithms

- Idea: queue pending requests and select them in a way that minimizes *head movement* and *idle plate rotation*
- Heuristics: variants of the *"elevator" algorithm* (depend on block size, number of heads, etc.)
- Strong influence on process scheduling and preemption: *disk thrashing*

11. Kernel Design

- Interrupts and Exceptions
- Low-Level Synchronization
- Low-Level Input/Output
- Devices and Driver Model
- File Systems and Persistent Storage
- Memory Management
- Process Management and Scheduling
- Operating System Trends
- Alternative Operating System Designs

Hardware Support for Memory Management

Segmentation (Old-Fashioned)

- Hardware to separate types of memory (code, data, static, etc.)
- Supported by x86 but totally unused by Linux/UNIX

Paging

- Hardware memory protection and address translation (MMU)
- *At each context switch*, the kernel reconfigures the page table
  - Implementation: assignment to a control register at each context switch
  - Note: this flushes the TLB (cache for address translation), resulting in a severe performance hit in case of scattered physical memory pages
- Use *large pages* for the kernel and for long-lasting memory regions
  - E.g., file system, data base caches, arrays for numerical computing
- Page *affinity* policy for modern cache-coherent architectures

Kernel Mode Memory Management

Classes of Addressable Memory

- Zone allocator

  `ZONE_DMA`: lower 16MB on x86
  `ZONE_NORMAL`: above 16MB and below 896MB on x86 32bits
  `ZONE_HIGHMEM`: above 896MB and below 4096MB on x86 32bits, empty on 64bits

Allocation of Physical Pages

- `kmalloc()` and `kmap()` vs. `malloc()` and `mmap()`
- User processes: contiguous physical memory pages improves performance (TLB usage), but not mandatory
- Kernel: in general, allocation of *lists of non-contiguous physical memory pages*
  - With lower bounds on the size of each contiguous part
  - Note: operates under a specific critical section (multiple resource allocation)

Adaptive Memory Management

Memory Allocation

- *Slab allocator* (original design: Sun Solaris)
  - Caches for special-purpose pools of memory (of fixed size)
  - Learn from previous (de)allocations and anticipate future requests
  - Optimizations for short-lived memory needs
    - E.g., inode cache, block device buffers, etc.
    - Multipurpose buffers from 2^5 to 2^{22} bytes
    - Many other kernel internal buffers
  - `$ man slabinfo` and `$ cat /proc/slabinfo`

11. Kernel Design

- Interrupts and Exceptions
- Low-Level Synchronization
- Low-Level Input/Output
- Devices and Driver Model
- File Systems and Persistent Storage
- Memory Management
- Process Management and Scheduling
- Operating System Trends
- Alternative Operating System Designs

Low-Level Process Implementation

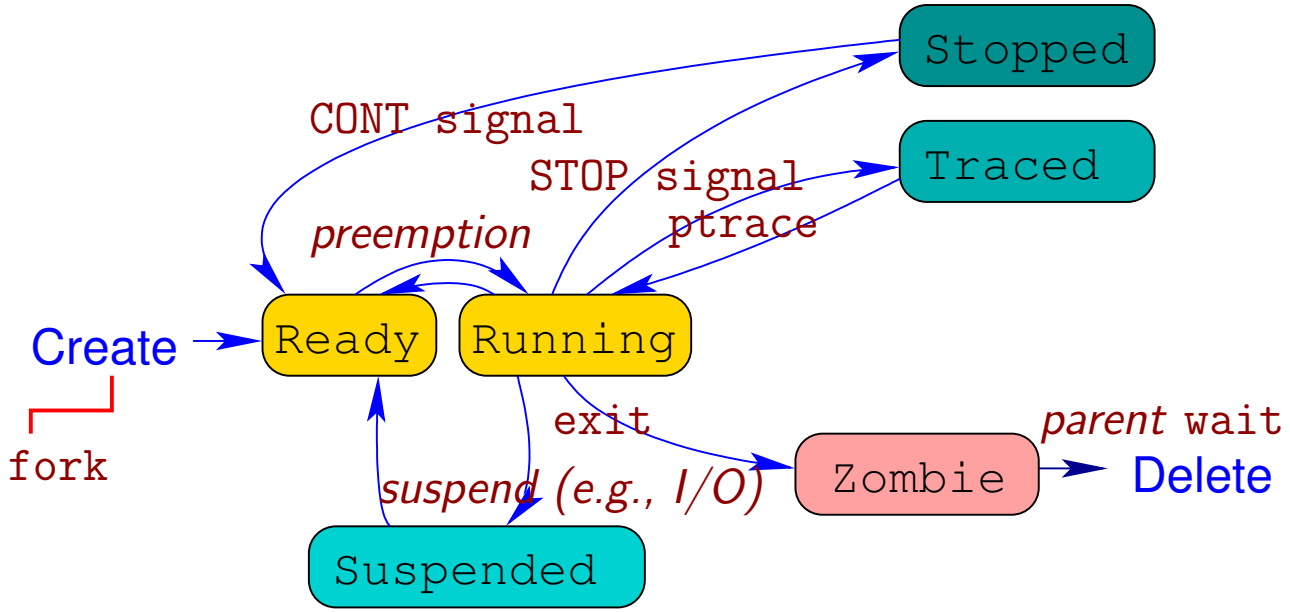
Hardware Context

- Saved and restored by the kernel upon context switches
- Mapped to some hardware thread when running
- Thread *affinity* policy for modern cache-coherent architectures

Heavyweight/Lightweight Processes

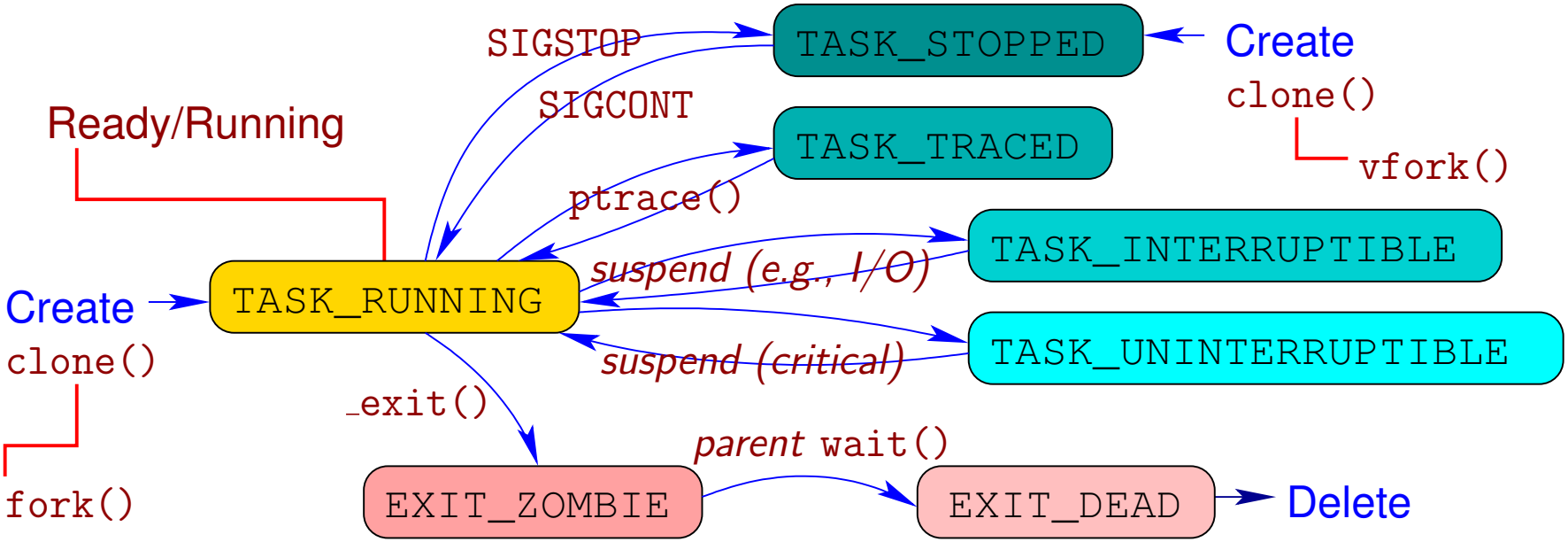
- Implement *one-to-one* model: one *user*-level thread mapped on one *kernel*-level thread
  - Unlike *user*-level threading libraries like the OCaml threads which implement a *many-to-one* model
- Generic `clone()` system call for both threads and processes
  - Setting which attributes are shared/separate
  - Attaching threads of control to a specific execution context

Generic Process States



- `Ready` (runnable) process waits to be scheduled
- `Running` process make progress on a hardware thread
- `Stopped` process awaits a continuation signal
- `Suspended` process awaits a wake-up condition from the kernel
- `Traced` process awaits commands from the debugger
- `Zombie` process retains termination status until parent is notified
- Child created as `Ready` after `fork()`
- Parent is `Stopped` between `vfork()` and child `execve()`

Linux Process States



Notes About Linux

- Context switch does *not* change process state
- Special "non-interruptible" state for critical and real-time I/O

Process Scheduling

Distribute Computations Among Running Processes

- Infamous optimization problem
- Many heuristics... and objective functions
  - Throughput?
  - Reactivity?
  - Deadline satisfaction?
- General (failure to) answer: *time quantum* and *priority*
  - Complex dynamic adaptation heuristic for those parameters
  - `nice()` system call
  - `$ nice` and `$ renice`

Process Scheduling

Scheduling Algorithm

- Process-dependent semantics
  - *Best-effort* processes
  - *Real-time* processes

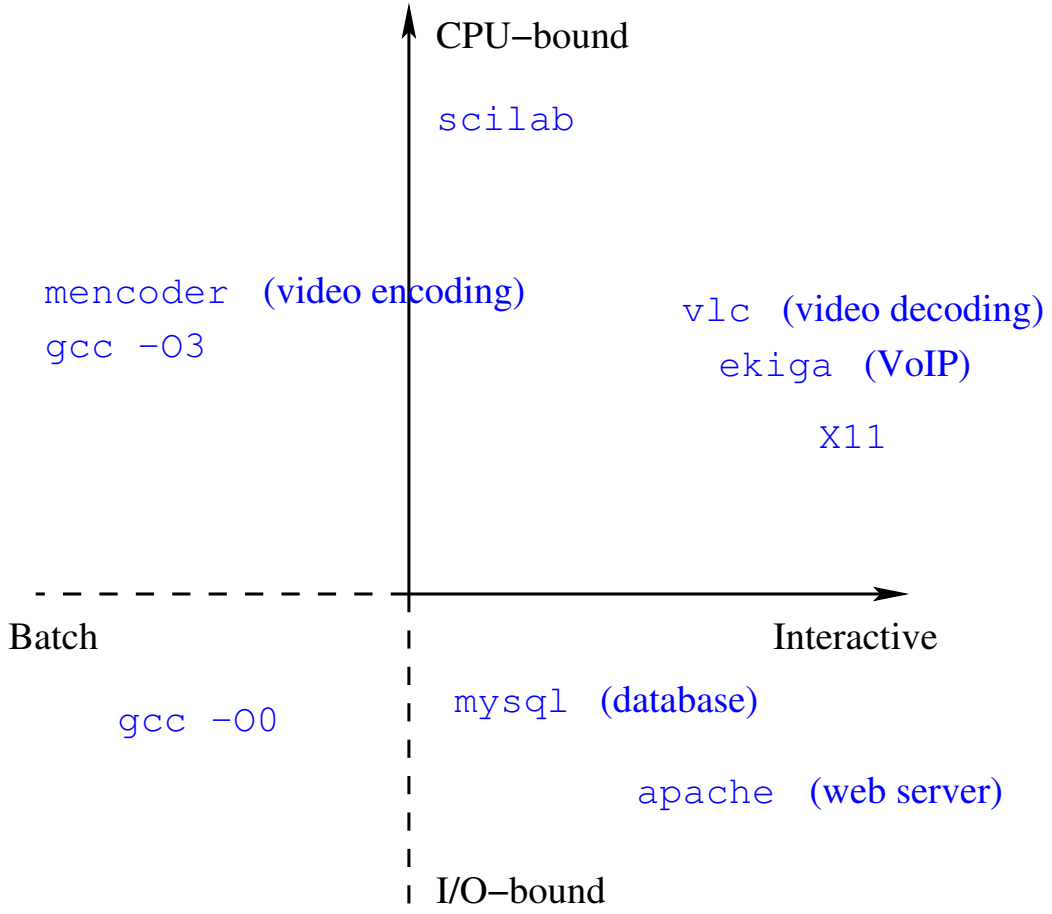
Scheduling Heuristic

- Multiple *scheduling queues*
  - Semantics: split processes according to scheduling algorithm (e.g., preemptive or not)
  - Performance: avoid high-complexity operations on priority queues (minimize context-switch overhead)
- Scheduling *policy*: prediction and adaptation

Scheduling Policy

Classification of Best-Effort Processes

- Two independent features
  - I/O behavior
  - Interactivity



Scheduling Policy

Real-Time Processes

- Challenges
  - Reactivity and low response-time variance
  - Avoid *priority inversion*: priorities + mutual exclusion lead to *priority inversion* (partial answer: *priority inheritance*)
  - Coexistence with normal, time-sharing processes
- `sched_yield()` system call to relinquish the processor voluntarily without entering a suspended state
- Policies: *FIFO* or *round-robin (RR)*

11. Kernel Design

- Interrupts and Exceptions
- Low-Level Synchronization
- Low-Level Input/Output
- Devices and Driver Model
- File Systems and Persistent Storage
- Memory Management
- Process Management and Scheduling
- Operating System Trends
- Alternative Operating System Designs

Operating System Trends

Design for Modularity

- Goals
  - Minimize memory overhead (embedded systems)
  - Handle a variety of hardware devices and software services
  - Incremental compilation of the kernel
- *Kernel modules*
  - Linux kernel modules (`/lib/modules/*.ko`) and Windows kernel DLLs
  - Run specific functions on behalf of the kernel or a process
  - Dynamic (un)loading and configuration of device drivers

Operating System Trends

Design for Maintainability

- Downsizing: *microkernel*
  - Execute most of the OS code in user mode (debug, safety, adaptiveness)
  - The kernel only implements synchronization, communication, scheduling and low-level paging
  - User mode system processes implement memory management, device drivers and system call handlers (through specific access authorizations)

Operating System Trends

Microkernels

- Successes
  - Mach: NeXT, MacOS X
  - Chorus (from INRIA project, secure OS)
  - Model for very small kernels (smart cards, eCos)
- Drawbacks
  - Message passing overhead (across processes and layers)
  - Most of the advantages can be achieved through modularization
  - Diminishing returns on full-size kernels
- Extreme: *exokernel* enforce separation and access control only

Operating System Trends

Miscellaneous Trends

- Bundling of a kernel with a variety of higher level libraries, component systems, development kits, graphical interfaces, network tools, etc.
  - If OS $\neq$ kernel, where are the limits of the OS?
- Scalable performance: better support for NUMA
  - Affinity to a core/processor/node, page and process migration
  - Paging and scheduling aware of physical distribution of memory
  - Linux 2.6 as some of the most sophisticated support (see SGI Altix)
- Tuning of kernel policies
  - Custom process and I/O scheduling, paging, migration...
    E.g., IBM Research's K42 linux-compatible kernel
  - Access control models
    E.g., NSA's SELinux

11. Kernel Design

- Interrupts and Exceptions
- Low-Level Synchronization
- Low-Level Input/Output
- Devices and Driver Model
- File Systems and Persistent Storage
- Memory Management
- Process Management and Scheduling
- Operating System Trends
- Alternative Operating System Designs

Alternative Designs

General-Purpose Systems

- Single-user "file and device managers": CP/M, MSDOS and Windows 3.1
- Unprotected single-user systems: MacOS 1–9, AmigaOS, OS/2, Windows 95
- Non-UNIX multi-user systems: Multics, VMS, OS/360, Windows NT, Windows 2000 and XP
- Modern workstation/server systems: Windows Vista, Solaris, Linux, MacOS X
- Modern embedded systems: SymbianOS, Blackberry, Windows Mobile, Linux, MacOS X

Real-Time Systems

- Examples of RTOS: pSOS+, VxWorks, VRTX, uiTRON, RTAI

We will quickly survey original features of Windows XP and RTOSes

Windows

Prominent operating system?

Quick Figures – 2007

- 90% of desktops
- 66% of servers
- 24% of smartphones
  - Linux 99% of DSL boxes
  - Linux 52% of web servers and 85% supercomputers

Quick Figures – 2011

- 85% of desktops
- 36% of web servers
- 1% of smartphones sold
  - Linux 99% of DSL boxes
  - Linux 56% of web servers and 91% supercomputers
  - Linux 55% of smartphones sold

Windows

Programmer Interface

- *Win32 API* is the default low-level user interface
- Most of POSIX is supported (see also *cygwin*, *mingw*)
- Note: documentation is not fully available to the public

Windows

File Systems and File Names

- Volume-based file system, no unified mount tree, no VFS
  - Historical legacy from CP/M: `A:`, `C:`, etc.
- Use a *swap file* rather than *swap partition* on UNIX
  - Enhanced flexibility and ease of configuration, lower performance
  - Frequent thrashing problems due to kernel control paths with conflicting memory requirements
- Flat *registry* of environment variables and configuration parameters
  - Combines the equivalent of UNIX's `/etc` and environment variables, plus GNOME's `GConf` files in one single associative table
  - Very fragile database: discouraged manual intervention by Microsoft itself in 1998!

Windows

Processes and Threads

- Multiple execution contexts called *subsystems*
- Multiple hardware threads per subsystem, similar to POSIX threads
  - Threads and subsystems are totally distinct objects, unlike Linux, but closer to other UNIX threading models
- Implementation of the *many-to-many* threading model
  - Support the mapping of multiple *user*-level threads to multiple *kernel*-level threads: `fiber` library

Windows

Processes Communications: Ports and Messages

- Cooperation through Mach-like messages
  - Subsystems have *ports* (for rendez-vous and communication)
    - A client subsystem opens a handle to a server subsystem's *connection port*
    - It uses it to send a connection request
    - The server creates *two communication ports* and returns one to the client
    - Both exchange messages, with or without callbacks (asynchronous message handling)
    - Implementation through *virtual memory mappinig* (small messages) or copying
  - Primary usage: *Remote Procedure Calls* (RPC) called *Local Procedure Calls* (LPC)

Windows

Processes Communications: Asynchronous Procedure Call

- Windows does *not* implement signals natively
- More expressive mechanism: *Asynchronous Procedure Call* (APC)
  - Similar to POSIX *message queues with callbacks* (or handlers)
  - APCs are queued (unlike signals)
  - Can be used to simulate signals (more expensive)

Windows

Thread Scheduling

- The Windows scheduler is called the *dispatcher*
- Similar support for real-time thread domains and time-quantum/priority mechanisms in Linux
- Original features (Windows XP)
  - Strong penalization of I/O-bound processes
  - Extend the time-quantum of the foreground subsystem whose window has graphical focus by a factor of 3!

Real-Time Operating System (RTOS)

Time-Dependent Semantics

- Motivations: enforce delay/throughput constraints
- Hypotheses
  - Short-lived processes (or reactions to events)
  - *Predictable* execution time, known at *process execution (launch)* or *reaction time*
- Tradeoffs
  - *Hard* real time: missing deadlines is not tolerable
  - *Soft* real time: missing deadlines is undesirable, but may happen to allow a higher priority task to complete

Real-Time Process Scheduling

Guarantees

- Periodic system: static *schedulability* dominates flexibility

$$\mathbf{T_i} = \text{execution time}, \mathbf{P_i} = \text{execution period}: \sum_i \frac{\mathbf{T_i}}{\mathbf{P_i}} < 1$$

- Aperiodic system: online acceptation/rejection of processes
- Beyond preemption and delay/throughput control, RTOSes may offer *reactivity* and *liveness* guarantees

Constraints

- Real-time scheduling requires static information about processes (e.g., bounds on execution time) and may not be compatible with many services provided by a general-purpose OSes

Trends in Real-Time Systems

Real-Time Features in a General-Purpose OS

- Modern OSes tend to include more and more real-time features
  - Predictable media-processing
  - High-throughput computing (network routing, data bases and web services)
  - Support hard and soft real-time
  - Example: *Xenomai* for Linux

Trends in Real-Time Systems

No-OS Approach

- Real-time operating systems are too complex to model and verify
  - Incremental approach: very simple RTOS with fully verifiable behavior
- Yet, *most critical systems do not use an OS at all*
  - Static code generation of a (reactive) scheduler, tailored to a given set of tasks on a given system configuration
  - Synchronous languages: Lustre (Scade), Signal, Esterel
    $\rightarrow$ main approach for closed systems like flight controllers (Airbus A320–A380)

- See Gérard Berry's lecture-seminars at Collège de France (live, in French)
  `http://www.college-de-france.fr/default/EN/all/inn_tec`